Lua程序设计

书栈(BookStack.CN)

目　录

致谢

简介

第一章 Lua程序设计

- 1.1 类型与值
- 1.2 表达式
- 1.3 语句
- 1.4 函数
- 1.5 迭代器与范型for
- 1.6 协同程序
- 1.7 数据结构
- 1.8 数据文件与持久性
- 1.9 元表与元方法
- 1.10 环境
- 1.11 模块与包
- 1.12 面向对象编程
- 1.13 弱引用table
- 1.14 数学库
- 1.15 table库
- 1.16 字符串库
- 1.17 I／O库
- 1.18 用户自定义类型
- 1.30 练习题

第二章 LuaFramework

- 2.1 HelloWorld
- 2.2 ScriptsFromFile
- 2.3 CallLuaFunction
- 2.4 AccessingLuaVariables
- 2.5 LuaCoroutine
- 2.6 LuaCoroutine2
- 2.7 LuaThread
- 2.8 AccessingArray
- 2.9 Dictionary
- 2.10 Enum
- 2.11 Delegate
- 2.12 GameObject
- 2.13 CustomLoader

2.14 Out
2.15 ProtoBuffer
2.16 Int64
2.17 Inherit
2.18 Bundle
2.19 cjson
2.20 utf8
2.21 String
2.22 Reflection
2.23 List
2.24 Performance
2.25 TestErrorStack
2.26 TestOverload
2.27 代码热更新
2.28 资源热更新
2.29 编写Lua逻辑
2.30 Lua组件
2.31 热更UI
第三章 XLua教程
3.1 HelloWorld
3.2 U3DScripting
3.3 UIEvent
3.4 LuaObjectOrented
3.5 NoGc
3.6 Coroutine
3.7 AsyncTest
3.8 Hotfix
3.9 GenericMethod
3.10 SignatureLoader
3.11 RawObject
3.12 RelmplementInLua
3.13 xLua的配置
3.14 CS调用Lua
3.141 调用基本类型
3.142 调用Table类型
3.143 调用Function
3.15 Lua调用CS
3.151 创建对象

3.152 调用静态成员
3.153 调用实例成员
3.154 调用父类成员
3.155 Ref、Out参数
3.156 调用重载方法
3.157 使用运算符重载
3.158 调用有默认参数方法
3.159 Params参数
3.1510 调用扩展方法
3.1511 调用泛型方法
3.1512 调用枚举类型
3.1513 调用委托
3.1514 调用事件
3.1515 复杂转换
3.1516 调用协同程序
3.16 载入Lua脚本
3.17 UI事件
3.18 热更新

致谢

当前文档 《Lua程序设计》 由 进击的皇虫 使用 书栈(BookStack.CN) 进行构建，生成于 2019-04-08。

书栈(BookStack.CN) 仅提供文档编写、整理、归类等功能，以及对文档内容的生成和导出工具。

文档内容由网友们编写和整理，书栈(BookStack.CN) 难以确认文档内容知识点是否错漏。如果您在阅读文档获取知识的时候，发现文档内容有不恰当的地方，请向我们反馈，让我们共同携手，将知识准确、高效且有效地传递给每一个人。

同时，如果您在日常工作、生活和学习中遇到有价值有营养的知识文档，欢迎分享到 书栈(BookStack.CN) ，为知识的传承献上您的一份力量！

如果当前文档生成时间太久，请到 书栈(BookStack.CN) 获取最新的文档，以跟上知识更新换代的步伐。

内容来源：沈军 https://shenjun-coder.github.io/LuaBook/

文档地址：http://www.bookstack.cn/books/LuaBook

书栈官网：http://www.bookstack.cn

书栈开源：https://github.com/TruthHun

分享，让知识传承更久远！ 感谢知识的创造者，感谢知识的分享者，也感谢每一位阅读到此处的读者，因为我们都将成为知识的传承者。

Lua程序设计

联系邮箱：380921128@qq.com

关于本书

主版本： beta

网站地址： shenjun7792.github.io

网站作者： 沈军

一、什么是热更新？

热更新，是对hot update或者 hot fix的翻译，计算机术语，表示在不停机的前提下对系统进行更改（摘抄一下）：

“hot就是热，机器运行会发烫，hot就是不停机的意思。

热更新，是个很形象的词，机器烫的时候更新，开着更新。

比如Windows不重启的前提下安装补丁

比如Http服务器在不重启的前提下换掉一个文件

那么对于Unity3D来说，何谓热更新？

额……这个真相实在是不想讲出来，因为很多时候，这个词都用错了。

Unity3D是一个客户端工具，用户是否重启客户端，根本是我们不关心的问题。

很多时候我们用着热更新这个词汇，却不需要真的热更新。

只有少部分游戏，游戏资源在玩的过程中边玩边下，不重启的前提下变更了资源。

我们不需要用户不重启客户端就能实现资源代码的更新，我们需要的是用户重启客户端能实现资源代码

的更新。

让我们暂时放过这个我们的需求连词汇都用错了这个基本事实，来总结一下何谓Unity3D热更新，Unity3D热更新就是指：用户重启客户端就能实现客户端资源代码更新的需求或者功能。”

二、为什么要热更新？

热更新，能够缩短用户取得新版客户端的流程，改善用户体验。

没有热更新：

pc用户：

下载客户端->等待下载->安装客户端->等待安装->启动->等待加载->玩

手机用户：

商城下载APP->等待下载->等待安装->启动->等待加载->玩

有了热更新：

pc用户：

启动->等待热更新->等待加载->玩

有独立loader的pc用户：

启动loader->等待热更新->启动游戏->等待加载->玩

手机用户：

启动->等待热更新->等待加载->玩

通过对比就可以看出，有没有热更新对于用户体验的影响还是挺大的，主要就是省去用户自行更新客户端的步骤。

三、如何热更新？Unity3D的热更新的方法比较

3.1、Android 应用的热更新

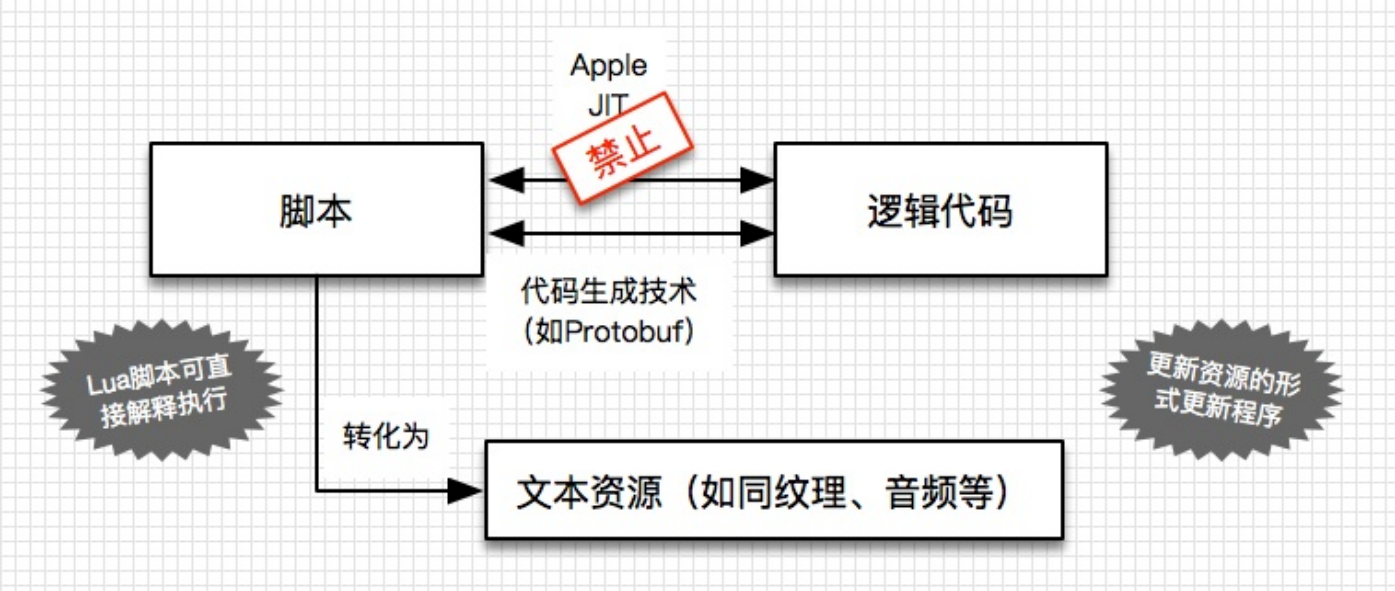
- 将执行代码预编译为assembly dll。

- 将代码作为TextAsset打包进Assetbundle。
- 运行时，使用Reflection机制实现代码的功能。
- 更新相应的Assetbundle即可实现热更新。

3.2、Android 与iOS 热更新的异同

- 苹果官方禁止iOS下的程序热更新；JIT（Just In Time 即时编译技术）在iOS下无效。
- 热更新方案：Unity+Lua插件。

3.3、 使用Lua 插件进行iOS 热更新的原理



3.4、Unity 热更新的注意点

- 需要更新的代码、资源，都必须打包成AssetBundle（建议使用未压缩的格式打包）
- 熟悉Unity的几个重要的路径

```
1.      • Resources（只读）
2.
3.      • StreamingAssets（只读）
4.
5.      • Application.dataPath（只读）
6.
7.      • Application.persistentDataPath（可读写）
```

3.5、重要路径之 之Resources

- Resources文件夹下的资源无论使用与否都会被打包
- 资源会被压缩，转化成二进制
- 打包后文件夹下的资源只读
- 无法动态更改，无法做热更新
- 使用Resources.Load加载

3.6、重要路径之StreamingAssets

- 流数据的缓存目录
- 文件夹下的资源无论使用与否都会被打包
- 资源不会被压缩和加密
- 打包后文件夹下的资源只读，主要存放二进制文件
- 无法做热更新
- WWW类加载（一般用CreateFromFile ，若资源是AssetBundle，依据其打包方式看是否是压缩的来决定）
- 相对路径，具体路径依赖于实际平台
- Application.streamingAssetsPath
- IOS: Application.dataPath + “/Raw” 或Application/xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx/xxx.app/Data/Raw

3.7、重要路径之Application.dataPath

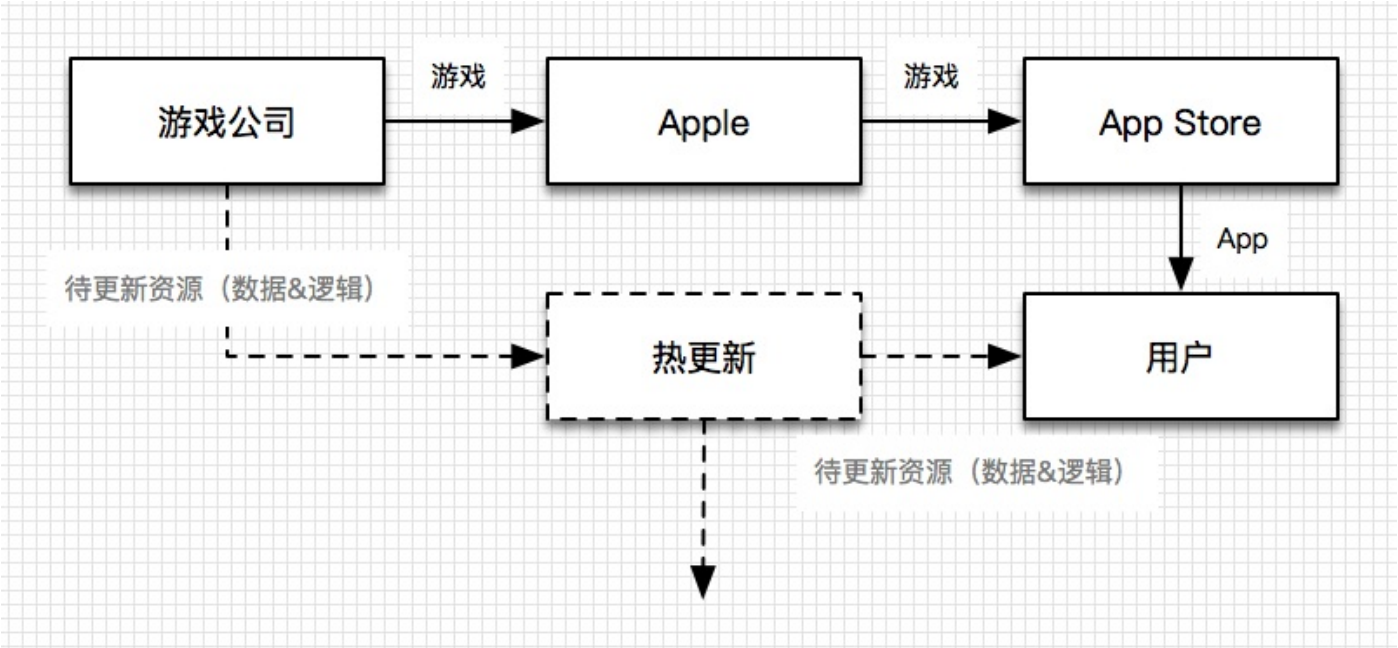
- 游戏的数据文件夹的路径（例如在Editor中的Assets）
- 很少用到
- 无法做热更新
- IOS路径: Application/xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx/xxx.app/Data

3.8、重要路径之Application.persistentDataPath

- 持久化数据存储目录的路径（ 沙盒目录，打包之前不存在 ）
- 文件夹下的资源无论使用与否都会被打包

- 运行时有效，可读写
- 无内容限制，从StreamingAsset中读取二进制文件或从AssetBundle读取文件来写入PersistentDataPath中
- 适合热更新
- IOS路径: Application/xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx/Documents

3.9、使用Lua 插件进行iOS 热更新的总体流程



四、支持Unity iOS 热更新的各种Lua 插件的对比

4.1、uLua(asset store)

- uLua插件原生版本，开山鼻祖
- 不会产生静态代码
- 反射机制，效率低下，速度慢，gcalloc频繁
- 已停止更新维护，不支持Unity5.x，淡出主流

4.2、uLua & cstoLua

- 开发平台成熟稳定，Bug修复迅速
- 开发者众多，资源丰富

- 静态方法，性能优
- 有成功商业产品案例（啪啪三国、超神战队、酷鱼吧捕鱼、绝地战警、这不是刀塔等）
- 都是基于原生版本的改进；未来，两者会合并成一个插件

开源项目地址：

https://github.com/topameng/CsToLua

4.3、sLua

- 静态方法，性能优
- 核心代码简洁
- 资源较少，开发平台不够成熟稳定
- 无成功商业产品案例成功商业产品案例
- 基于原生版本的改进

开源项目地址：

https://github.com/pangweiwei/slua

4.4、C#Light(L#)

- 淡出主流，想要了解的小伙伴点击这里：

https://github.com/lightszero/LSharp

4.5、 uniLua

- c#实现的Lua虚拟机，非完整方案
- 淡出主流

4.6、xLua

xLua是腾讯开源的Unity3D下Lua编程解决方案，自2016年初推广以来，已经应用于十多款腾讯自研游戏，凭借其出色的性能，易用性，扩展性而广受好评。

xLua在功能、性能、易用性都有不少突破，这几方面分别最具代表性的是：

1. Unity3D全平台热补丁技术，可以运行时把C#实现（方法，操作符，属性，事件，构造函数，析构函数，支持泛化）替换成lua实现；

```
2. 自定义struct，枚举在Lua和C#间传递无C# gc alloc；
3. 编辑器下无需生成代码，开发更轻量；
```

官方开源网址：https://github.com/Tencent/xlua

综合来看 肯定是 **xLua** 会更好一些，**uLua & cstoLua** 次之。

```
1. 第二章介绍 uLua & cstoLua 在 Unity 中的框架 LuaFramework。
2. 第三章介绍 xLua 在 Unity 中的框架。
```

五、LuaFramework 实践

熟悉NGUI的小伙伴可以参考这里：

https://github.com/jarjin/SimpleFramework_NGUI

熟悉UGUI的小伙伴可以参考这里：

https://github.com/jarjin/LuaFramework_UGUI

？

Lua 程序设计

特点：

设计目标：能与C语言或其他常用语言相互集成。这样做可以带来很多好处。

1、简单

因为它不需要去做其他语言已经做的很好的方面。

而Lua提供的特性则是C语言不太擅长的：

```
1.  相对于硬件的高层抽象
2.  动态结构
3.  无冗余
4.  简易的测试和调试
```

Lua提供了：

```
1.  安全的运行环境
2.  一套自动内存管理机制：不用管分配内存、释放内存、内存溢出等。
3.  优秀的字符串处理能力
4.  动态大小数据的处理能力：动态类型，支持多态
5.  高级函数和匿名函数：允许更高层的参数化，能使函数变得更加通用
```

2、可扩展性　　语言特性体现了：自动内存管理、高级函数和匿名函数

3、“胶水语言”　　基于组件的开发：可以通过黏合现在的高层组件来创建新的应用程序　　这些组件可以是已编译好的，也可以是静态类型语言（C、C++）编写的。　　Lua则可以成为组织和连接各种组件的胶水。

> 一个项目：窗口、数据结构 很少修改有一部分内容可能会反复修改，则这部分内容可以使用Lua开发。

- 有很多脚本语言，但是Lua提供了一组特性，使它与众不同，成为解决许多问题的首选语言：

```
1.  可扩展性：既可用Lua代码来扩展，又可以用外部的C代码来扩展。
2.  （Lua的大部分基础功能就是通过外部库实现的）。可以集成到C/C++、Java、C#等语言中。
3.
4.  简易性：简单、小巧。概念不多，但每个概念都很有用。易于学习。
5.
```

```
6.      高效：脚本（解释型）语言中最快的语言之一。
7.
8.      可移植性：可运行在任何平台上。编译只依赖于ANSI（ISO）C标准
```

语法规范

- 标识符可以是字母、数字和下划线，但不能以数字开头

```
1.      避免用下划线跟着一个或多个大写字母
2.      一个下划线作为“哑变量”使用
3.      区分大小写
4.      不能是保留字
```

- 单行注释

```
1.          --
```

- 块注释

```
1.          --[[ ... ]]
2.
3.          例如：
4.                  不执行
5.                  --[[
6.                  print(10)
7.                  --]]
8.
9.                  可执行
10.                 ---[[
11.                 print(10)
12.                 --]]
```

- 全局变量

```
1.          不需要声明。只需将一个值赋予一个全局变量就可以创建了。
2.
3.          例如：
4.                  print(b) --> nil          访问一个未初始化的变量不会引发错误，访问结果是
```

```
    一个特殊的值nil。
5.
6.                  b = 10
7.                  print(b) --> 10
8.
9.                  b=nil                   删除一个全局变量
10.                 print(b) --> nil        如果存在一个全局变量，那么它必定具有一个非nil
    的值。
```

Lua环境配置

- 下载**Lua:**

```
1.              www.lua.org
```

- **Mac**终端安装**:**

```
1.              make macosx
2.              sudo make install
3.              lua -v          （测试是否安装成功）
```

- **Windows**下源码编译新建VC++解决方案，在该解决方案下建3个项目，分别是lua库项目，lua编译器项目，lua解释器项目。

```
1.      1). 建lua库项目：Lua53
2.          选择Dll和空项目。
3.          编译模式改成release模式
4.          把源码中src目录下的所有.h文件拷贝到项目中的头文件节点下
5.          把源码中src目录下的除了lua.c和luac.c所有.c文件加入工程下的源文件目录
6.          然后设置项目属性，更改项目编译库类型为 静态库(.lib)
7.          最后编译运行，在项目文件夹release目录下产生了Lua53.lib文件，即lua库生成成功。
8.
9.      2). 建lua编译器项目：Luac
10.         选择控制台应用程序和空项目
11.         编译模式改成release模式
12.         把源码中src目录下的所有.h文件拷贝到项目中的头文件节点下
```

```
        把源码中src目录下的除了lua.c所有.c文件加入工程下的源文件目录
        然后设置项目属性，更改项目编译库类型为 应用程序（.exe）
        最后编译运行，在项目文件夹release目录下产生了Luac.exe文件，即lua编译器文件生成成功。

    3). 建lua解释器项目：Lua
        步骤同建编译器项目，区别是添加src源文件时，不添加luac.c文件。
        编译完成后release目录下多了一个lua.exe文件。即lua解释器文件生成成功。
    4). 把编译好的release目录下的文件拷贝到 C:/Program Files (x86)/Lua5.3 目录下，并修改环境变量，把此路径添加进去。
```

- **sublime 设置:**

方法1:

```
            菜单 Tools -> Build System -> New Build System

            // Mac 配置
            {
             "cmd": ["/usr/local/bin/lua", "$file"],
             "file_regex": "^(...*?):([0-9]*):?([0-9]*)",
             "selector": "source.lua"
            }

            // Windows 配置
            {
                "cmd": ["C:/Program Files (x86)/Lua5.3/luac.exe", "$file"],
                 "file_regex": "^(?:lua:)?[\t ](...*?):([0-9]*):?([0-9]*)",
                 "selector": "source.lua"
            }
```

方法2:

```
    View -> Show Console
    在控制台输入以下代码：
```

```
import urllib.request,os,hashlib; h = 'df21e130d211cfc94d9b0905775a7c0f' + '1e3d39e33b79698005270310898eea76'; pf = 'Package Control.sublime-package'; ipp = sublime.installed_packages_path(); urllib.request.install_opener( urllib.request.build_opener( urllib.request.ProxyHandler()) ); by = urllib.request.urlopen( 'http://packagecontrol.io/' + pf.replace(' ', '%20')).read(); dh = hashlib.sha256(by).hexdigest(); print('Error validating
```

```
    download (got %s instead of %s), please try manual install' % (dh, h)) if dh !=
    h else open(os.path.join( ipp, pf), 'wb' ).write(by)
```

- **Sublime Text2** 可选插件安装:

```
1.          Tools -> Command Palette ...
2.
3.          输入：
4.          install package
5.          JsFormat
6.
7.          install package
8.          LuaJumpDefinition
9.
10.         install package
11.         Tariling Spaces
12.
13.         install package
14.         Terminal
15.
16.         install package
17.         Fix Mac Path
18.
19.         install package
20.         FormatLua
21.
22.         install package
23.         LuaExtended
```

- 运行:

```
1.      command + b
2.      F7
```

- **Sublime Text2** 激活码:

```
1.      -- BEGIN LICENSE --
2.      Michael Barnes
3.      Single User License
4.      EA7E-821385
```

```
    8A353C41 872A0D5C DF9B2950 AFF6F667
    C458EA6D 8EA3C286 98D1D650 131A97AB
    AA919AEC EF20E143 B361B1E7 4C8B7F04
    B085E65E 2F5F5360 8489D422 FB8FC1AA
    93F6323C FD7F7544 3F39C318 D95E6480
    FCCC7561 8A4A1741 68FA4223 ADCEDE07
    200C25BE DBBC4855 C4CFB774 C5EC138C
    0FEC1CEF D9DCECEC D3A5DAD1 01316C36
    -- END LICENSE --
```

?

类型与值

```
1.      • nil（空）
2.      • boolean（布尔）
3.      • number（数字）
4.      • string（字符串）
5.      • table
6.      • function（函数）
7.      • userdata（自定义类型）和 thread（线程）
```

Lua是一种动态类型的语言。在语言中没有类型定义的语法，每个值都“携带”了它自身的类型信息。

函数 `type` 可根据一个值返回其类型名称。 `type` 函数总是返回一个字符串

```
1.
2.                  print(type("Hello wold"))        -- string
3.                  print(type(10.4*3))              -- number
4.                  print(type(print))               -- function
5.                  print(type(type))                -- function
6.                  print(type(true))                -- boolean
7.                  print(type(nil))                 -- nil
8.                  print(type(type(X)))             -- string
```

变量没有预定义的类型，任何变量都可以包含任何类型的值

```
1.                  print(type(a))                  -- nil （a尚未初始化）
2.                  a = 10
3.                  print(type(a))                  -- number
4.                  a = "a string!!"
5.                  print(type(a))                  -- string
6.                  a = print                       -- 是的，这是合法的！
7.                  a(type(a))                      -- function
```

- nil（空）

```
1.              “无效值“
2.              将nil赋予一个全局变量等同于删除它。
```

- boolean（布尔）

```
1.              有两个可选值：false和true
2.
3.              然后boolean却不是一个条件值的唯一表示方式。
4.              在Lua中任何值都可以表示一个条件。Lua将值false和nil视为“假”，而除此之外的其他值视为“真”。
5.              请注意，Lua在条件测试中，将数字零和空字符串也都视为“真”。
```

- number（数字）

number类型用于表示实数。Lua没有整数类型。Lua中的数字可以表示任何32位整数，而不会产生四舍五入的错误。

以下是一些合法的数字常量：

```
1.          4     0.4     4.57e-3     0.3e12     5e+20
```

- string（字符串）

```
1.      不可变性。
2.      需要以一对匹配的单引号或双引号来界定：
3.
4.          a = "a line"
5.          b = 'another line'
```

可以包含类似于C语言中的转义序列

转义符	说明
\a	响铃
\b	退格
\f	提供表格
\n	换行
\r	回车

\t	水平tab
\v	垂直tab
\	反斜杠
\"	双引号
\'	单引号

另外，还可以用一对匹配的双方括号来界定一个字母字符串，就像写“块注释”那样。以这种形式书写的字符串可以延伸多行，Lua不会解释其中的转义序列。

```
1.          例如：
2.
3.                  page = [[
4.                  <html>
5.                  <head>
6.                  <title>An HTML Page</title>
7.                  </head>
8.                  <body>
9.                  <a href="http://www.lua.org">Lua</a>
10.                 </body>
11.                 </html>
12.                 ]]
13.
14.                 write(page)
```

有时字符串中可能需要包含这样的内容： `a=b[c[i]]` 。或者，可能需要包含已经被注释掉的代码。

为了应付这种情况，需要在两个左方括号间加上任意数量的等号，就像 `[===[` 。经过这样修改后，字面字符串只有在遇到一个内嵌有相同数量等号的双右方括号时才会结束（就前例而言，即 `]===]` ）。如果一组左右方括号中的等号数量不等，那么Lua会忽略它。通过选择适当数量的等号，就可以在不加转义的情况下，直接嵌入任意的字符串内容了。

这套机制同样适用于注释。例如，以“ `--[=[` ”开始的一个块注释将延伸至“ `]=]` ”结束。如此便简化了注释那些“已经包含了注释块”的代码。

Lua提供了运行时的数字与字符串的自动转换。

1、在一个字符串上应用算术操作时，Lua会尝试将这个字符串转换成一个数字：

```
1.                  print("10" + 1)             -- 11
2.                  print("10 + 1")             -- 10 + 1
```

```
print("-5.3e-10" * "2")         -- -1.06e-09
print("hello" + 1)              -- 错误（不能转换“hello”）
10=="10"                        -- false
```

> 显式地将一个字符串转换成数字，可以使用函数 `tonumber` 。当这个字符串的内容不能表示一个正确的数字时， `tonumber` 将返回 `nil` 。

2、相反，在Lua期望一个字符串但却得到一个数字时，它也会将数字转换成字符串

```
print(10 .. 20)                 -- 1020
```

在Lua中，“ `..` ”是字符串连接操作符。当直接在一个数字后面输入它的时候，必须要用一个空格来分隔它们。不然，Lua会将第一个点理解为一个小数点。

```
print(tostring(10)=="10")       -- true
```

`tostring` 函数可将一个数字转换成字符串

3、在Lua 5.1中，可以在字符串前放置操作符“ `#` ”来获取该字符串的长度

```
a = "hello"
print(#a)                   -- 5
print(#"good\0bye")         -- 8
```

- table

`table` 类型实现了“关联数组”。“关联数组”是一种具有特殊索引方式的数组。不仅可以通过整数来索引它，还可以使用字符串或其他类型的值（除了 `nil` ）来索引它。

`table` 没有固定的大小，可以动态地添加任意数量的元素到一个 `table` 中。

`table` 是Lua中主要的（事实上也是仅有的）数据结构机制，具有强大的功能。

在Lua中，table既不是“值”也不是“变量”，而是“对象”。

`table` 的创建是通过“构造表达式”完成的，最简单的构造表达式就是 `{ }` 。

```
a = { }             -- 创建一个table，并将它的引用存储到a
k = "x"
a[k] = 10           -- 新条目，key = "x"，value = 10
a[20] = "great"     -- 新条目，key = 20，value = "great"
```

```
print(a["x"])       -- 10
k = 20
print(a[k])         -- “great"
a["x"] = a["x"] + 1 -- 递增条目"x"
print(a["x"])       -- 11
```

`table` 永远是“匿名的”，一个持有 `table` 的变量与 `table` 自身之间没有固定的关联性。

```
a = { }
a["x"] = 10
b = a               -- b与a引用了同一个table
print(b["x"])       -- 10
b["x"] = 20
print(a["x"])       -- 20
a = nil             -- 现在只有b还在引用table
b = nil             -- 再也没有对table的引用了
-- 当一个程序再也没有对一个table的引用时，Lua的垃圾收集器最终会删除该table，并复用它的内存。
```

所有 `table` 都可以用不同类型的索引来访问 `value` （值），当需要容纳新条目时， `table` 会自动增长。

```
a = { }

-- 创建1000个新条目
for i=1, 1000 do a[i] = i*2 end
print(a[9])         -- 18
a["x"] = 10
print(a["x"])       -- 10
print(a["y"])       -- nil 该元素没有初始化
-- 可以将nil赋予table的某个元素来删除该元素。
```

Lua对于诸如 `a["name"]` 的写法提供了一种更简便的“语法糖（syntactic sugar）”，可以直接输入 `a.name` 。

```
a.x = 10            -- 等同于a["x"] = 10
print(a.x)          -- 等同于print(a["x"])
print(a.y)          -- 等同于print(a["y"])

-- 对于Lua来说，这两种形式是等价的，可供自由选择使用。
a.x                 -- 等同于a["x"]
```

```
a[x]                -- 以变量x的值来索引table

----------
a = { }
x = "y"
a[x] = 10           -- 将10放入字段“y”
print(a[x])         -- 10 字段“y”的值
print(a.x)          -- nil 字段“x”（未定义）的值
print(a.y)          -- 10 字段“y”的值
```

若要表示一个传统的数组或线性表，只需以整数作为 `key` 来使用 `table` 即可。这里不需要（也没有必要）声明一个大小值，直接初始化元素就可以了

```
-- 读取10行内容，并存储到一个table中

a = { }
for i=1,10 do
    a[i] = io.read()
end
```

虽然可以用任何值作为一个 `table` 的索引，也可以用任何数字作为数组索引的起始值。但就Lua的习惯而言，数组通常以1作为索引的起始值。并且还有不少机制依赖于这个惯例。

在Lua5.1中，长度操作符“ `#` ”用于返回一个数组或线性表的最后一个索引值（或为其大小）。

```
for i=1,#a do
    print(a[i])
end

print(a[#a])            -- 打印列表a的最后一个值
a[#a] = nil             -- 删除最后一个值
a[#a+1] = v             -- 将v添加到列表末尾

-- 读取一个文件的前10行
a = { }
for i=1,10 do
    a[#a+1] = io.read()
end
```

当对索引的实际类型不是很确定时，可以明确地使用一个显式转换

```
i = 10; j = "10"; k = "+10"
```

```
                a = { }
                a[i] = "one value"
                a[j] = "another value"
                a[k] = "yet another value"
                print(a[j])  --> another value
                print(a[k]) --> yet another value
                print(a[tonumber(j)]) --> one value
                print(a[tonumber(k)]) --> one value
```

- function（函数）

Lua既可以调用以自身Lua语言编写的函数，又可以调用以C语言编写的函数。Lua所有的标准库都是用C语言写的，标准库中包括对字符串的操作、table的操作、I/O、操作系统的功能调用、数学函数和调试函数。同样，应用程序也可以用C语言来定义其他函数。

```
    function function_name( )
        -- body
    end
```

- userdata（自定义类型）和 thread（线程）

由于userdata类型可以将任意的C语言数据存储到Lua变量中。在Lua中，这种类型没有太多的预定义操作，只能进行赋值和相等性测试。userdata用于表示一种由应用程序或C语言库所创建的新类型，例如标准的I/O库就用userdata来表示文件。稍后将在CAPI中详细讨论这种类型。

?

表达式

- 算术操作符
- 关系操作符
- 逻辑操作符
- 字符串连接
- 优先级
- table构造式

表达式用于表示值。Lua的表达式中可以包含数字常量、字面字符串、变量、一元和二元操作符及函数调用。另外有别于传统的是，表达式中还可以包括函数定义和 `table` 构造式。

• 算术操作符

+（加法）、-（减法）、*（乘法）、／（除法）、^（指数）、%（取模）、-（负号）

例如：

```
x^0.5 将计算x的平方根
x^(-1/3) 将计算x立方根的倒数
```

取模操作符是根据以下规则定义的：

```
a%b == a - floor(a/b)*b

例如：
x%1 的结果就是x的小数部分
x - x%1 的结果就是其整数部分
x - x%0.01 则是x精确到小数点后两位的结果

x = math.pi
print(x - x%0.01)   --> 3.14
```

• 关系操作符

```
<、>、<=、>=、==、~=
所有这些操作符的运算结果都是true或false。

操作符==用于相等性测试，操作符～=用于不等性测试。
这两个操作符可以应用于任意两个值。

如果两个值具有不同的类型，Lua就认为它们是不相等的。
否则，Lua会根据它们的类型来比较。

nil只与其自身相等。

对于table、userdata和函数，Lua是作引用比较的。
也就是说，只有当它们引用同一个对象时，才认为它们相等。
```

```
a = {}; a.x = 1; a.y = 0
b = {}; b.x = 1; b.y = 0
c = a
其结果是a==c，但a~=b。
```

> Lua会在遇到字符串和数字的大小比较时引发一个错误，例如 `2<"15"` 就会导致这种错误。

- ## 逻辑操作符

```
and、or、not
与条件控制语句一样，所有的逻辑操作符将false和nil视为假，而将其他的任何东西视为真。

对于操作符and来说，如果它的第一个操作数为假，就返回第一个操作数；不然返回第二个操作数。

对于操作符or来说，如果它的第一个操作数为真，就返回第一个操作数；不然返回第二个操作数。
```

```
print(4 and 5)        --> 5
print(nil and 13)     --> nil
print(false and 13)   --> false
print(4 or 5)         --> 4
print(false or 5)     --> 5
```

```
and和or都是用“短路求值”，也就是说，它们只会在需要时才去评估第二个操作数。
短路求值可以确保像（type(v)=="table" and v.tag == "h1"）这样的表达式不会导致运行时
```

```
    错误。
3.
4.      有一种常用的Lua习惯写法“x = x or v”，它等价于：
5.      if not x then x = v end
6.
7.      另外，还有一种习惯写法是“（a and b）or c”，这类似于C语言中的表达式a?b:c，但前提是b不为
    假。
8.      例如，为了选出数字x和y中的较大者，可以使用以下语句：
9.      max = (x > y) and x or y
```

```
1.      操作符not永远只返回true或false：
2.          print(not nil)        --> true
3.          print(not false)      --> true
4.          print(not 0)          --> false
5.          print(not not nil)    --> false
```

- 字符串连接

要在Lua中连接两个字符串，可以使用操作符“ `..` ”（两个点）。如果其任意一个操作数是数字的话，Lua会将这个数字转换成一个字符串：

```
1.      print("Hello" .. "World")    --> Hello World
2.      print(0 .. 1)                --> 01
```

请记住，Lua中的字符串是不可变的值。连接操作符只会创建一个新字符串，而不会对其原操作数进行任何修改：

```
1.      a = "Hello"
2.      print(a .. "World")    --> Hello World
3.      print(a)               --> Hello
```

- 优先级

```
1.  |优先级|
2.  |:--|
3.  |`^`|
4.  |`not`、`#`、`-`(一元)|
```

```
|`*`、`/`、`%`|
|`+`、`-`|
|`..`|
| `<`、`>`、`<=`、`>=`、 `~=`、 `==`|
|`and`|
|`or`|
```

在二元操作符中，除了指数操作符“ `^` ”和连接操作符“ `..` ”是“右结合”的，所有其他操作符都是“左结合”的。因此，下例中左边的表达式等价于右边的表达式：

```
    a + i < b/2 + 1          <-->       (a+i) < ((b/2) + 1)
    5 + x^2 * 8              <-->       5 + ((x^2) * 8)
    a < y and y <= z         <-->       (a < y) and (y <= z)
    -x^2                     <-->       -(x^2)
    x^y^z                    <-->       x^(y^z)
```

- table构造式

构造式是用于创建和初始化 `table` 的表达式。这是Lua特有的一种表达式，并且也是Lua中最有用、最通用的机制之一。

最简单的构造式就是一个空构造式 `{}` ，用于创建一个空 `table` 。构造式还可以用于初始化数组。元素之间可以用逗号也可以用分号，最后一个元素后面写逗号也是合法的。

```
    days = {"Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday",
"Saturday"}
```

会将 `days[1]` 初始化为字符串“ `Sunday` ”、 `days[2]` 初始化为“ `Monday` ”，以此类推。

Lua还提供了一种特殊的语法用于初始化记录风格的 `table` ：

```
    a = {x=10, y=20}
```

以上这行代码等价于这些语句：

```
    a = {}; a.x = 10; a.y = 20
```

无论使用哪种方式来创建 `table` ，都可以在 `table` 创建之后添加或删除其中的某些字段：

```
w = {x=0, y=0, label="console"}
x = {math.sin(0), math.sin(1), math.sin(2)}
w[1] = "another field"        -- 添加key 1到table w
x.f = w                       -- 添加key "f"到table x
print(w["x"])                 --> 0
print(w[1])                   --> another field
print(x.f[1])                 --> another field
w.x = nil                     -- 删除字段"x"
```

每当Lua评估一个构造式时，都会先创建一个新 `table` ，然后初始化它。

```
-- 用table实现链表：

list = nil
for line in io.lines() do            -- 从标准输入读取每行的内容
    list = {next=list, value=line}   -- 按相反的次序存储到一个链表中
end
```

```
-- 遍历该链表

local l = list
while l do
    print(l.value)
    l = l.next
end
```

将记录风格的初始化与列表风格的初始化混合在一个构造式中使用：

```
polyline = {
    color = "blue", thickness = 2, npoints = 4,
    {x = 0, y = 0},
    {x = -1, y = 0},
    {x = -10, y = 1},
    {x = 0, y = 1}
}
```

上例演示了如何通过嵌套的构造式来表示复杂的数据结构。每个 `polyline[i]` 元素都是一

个 `table` ，表示一条记录：

```
print(polyline[2].x)        --> -1
print(polyline[4].y)        --> 1
```

这两种风格的构造式各有其限制。例如，不能使用负数的索引，也不能用运算符作为记录的字段名。

为了满足这些要求，Lua还提供了一种更通用的格式。这种格式允许在方括号之间，显式地用一个表达式来初始化索引值：

```
opnames = { ["+"] = "add", ["-"] = "sub", ["*"] = "mul", ["/"] = "div" }

i = 20; s = "-"
a = {[i+0] = s, [i+1]=s..s, [i+2]=s..s..s}

print(opnames[s])           --> sub
print(a[22])                --> ---
```

```
构造式{x=0, y=0}等价于{["x"]=0, ["y"]=0}
构造式{"r", "g", "b"}等价于{[1]="r", [2]="g", [3]="b"}
```

对于某些情况如果真的需要以0作为一个数组的起始索引的话，通过这种语法也可以轻松做到：

```
days = {[0]="Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday",
"Saturday"}
```

现在第一个值“ `Sunday` ”的索引就为0了。这个索引0并不影响其他元素，“ `Monday` ”照常索引为1，因为它是构造式中列表风格中的第一个值，后续其他值的索引依次递增。但无论是否使用这种初始化语法，都不推荐在**Lua**中以**0**作为数组的起始索引。大多数内建函数都假设数组起始于索引**1**，若遇到以索引**0**开始的数组，它们就无法进行正确地处理了。

?

语句

```
1.      • 赋值
2.      • 局部变量与块（block）
3.      • 控制结构
4.          • if then else
5.          • while
6.          • repeat
7.          • 数字型for
8.          • 泛型for
9.      • break与return
```

Lua支持的常规语句基本上与C语言或Pascal语言中所支持的那些语句差不多。这些语句包括赋值、控制结构和过程调用。另外，Lua还支持一些不太常见的语句，例如多重赋值和局部变量声明。

• 赋值

```
1.      a = "hello" .. "world"
2.      t = { n = 0 }
3.      t.n = t.n + 1
```

Lua允许“多重赋值”，也就是一下子将多个值赋予多个变量。每个值或每个变量之间以逗号分隔。

```
1.      a, b = 10, 2*x    -- a为10, b为2*x
```

在多重赋值中，Lua先对等号右边的所有元素求值，然后才执行赋值。这样便可以用一句多重赋值来交互两个变量了。

```
1.      x, y = y, x                     -- 交换x与x
2.      a[i], a[j] = a[j], a[i]         -- 交换a[i]与a[j]
```

Lua总是会将等号右边值的个数调整到与左边变量的个数相一致。规则是：若值的个数少于变量的个数，那么多余的变量会被赋予 `nil` ；若值的个数更多的话，那么多余的值会被“静悄悄”丢弃掉：

```
1.      a, b, c = 0, 1
2.      print(a, b, c)             --> 0 1 nil
```

```
a, b = a+1, b+1, b+2      -- 其中b+2会被忽略
print(a, b)               --> 1 2
a, b, c = 0
print(a, b, c)            --> 0 nil nil
```

- 局部变量与块（block）

相对于全局变量，Lua还提供了局部变量。通过 `local` 语句来创建局部变量：

```
x = 10                    -- 全局变量
local i = 1               -- 程序块中的局部变量

while i <= x do
    local x = i*2         -- while循环体中的局部变量
    print(x)              --> 2, 4, 6, 8, ...
    i = i + 1
end

if i > 20 then
    local x               -- then中的局部变量
    x = 20
    print(x+2)            -- 如果测试成功会打印22
else
    print(x)              --> 10 （全局变量）
end

print(x)                  --> 10 （全局变量）
```

在交互模式中每行输入内容自身就形成了一个程序块。为了解决这个问题，可以显式地界定一个块，只需将这些内容放入一对关键字 `do-end` 中即可。每当输入了 `do` 时，Lua就不会单独地执行后面每行的内容，而是直至遇到一个相应的 `end` 时，才会执行整个块的内容。

如果需要更严格地控制某些局部变量的作用域时，这些 `do` 块也会有所帮助：

```
do
    local a = 1
    local b = 2
end             -- a和b的作用域至此结束
```

- ## 控制结构

Lua提供了一组传统的、小巧的控制结构，包括用于条件执行的 `if` ，用于迭代的 `while` 、 `repeat` 和 `for` 。所有的控制结构都有一个显式的终止符： `if` 、 `for` 和 `while` 以 `end` 作为结尾， `repeat` 以 `until` 作为结尾。

控制结构中的条件表达式可以是任何值，Lua将所有不是 `false` 和 `nil` 的值视为“真”。

if then else

```
    if a < 0 then a = 0 end

    if a < b then return a else return b end

    if op == "+" then
        r = a + b
    elseif op == "-" then
        r = a - b
    elseif op == "*" then
        r = a * b
    elseif op == "\" then
        r = a / b
    else
        error("invalid operation")
    end
```

> Lua不支持 `switch` 语句。

while

```
    local i = 1
    while a[i] do
        print(a[i])
        i = i + 1
    end
```

repeat

一条 `repeat-until` 语句重复执行其循环体直到条件为真时结束。循环体至少会执行一次。

```
repeat
    line = io.read()
until line ~= ""
print(line)
```

> 与其他大多数语言不同的是，在Lua中，一个声明在循环体中的局部变量的作用域包括了条件测试：

```
local sqr = x/2
repeat
    sqr = (sqr + x/sqr)/2
    local error = math.abs(sqr^2 - x)
until error < x/10000        -- 在此仍可以访问error
```

数字型 for

`for` 语句有两种形式：数字型 `for` 和泛型 `for` 。

数字型 `for` 的语法如下：

```
for var=exp1, exp2,exp3 do
    <执行体>
end
```

`var` 从 `exp1` 变化到 `exp2` ，每次变化都以 `exp3` 作为步长（ `step` ）递增 `var` ，并执行一次“执行体”。第三个表达式是可选的，若不指定的话，Lua会将步长默认为1。

```
for i=1, f(x) do print(i) end
for i=10, 1, -1 do print(i) end
```

如果不想给循环设置上限的话，可以使用常量 `math.huge` ：

```
for i=1, math.huge do
    if(0.3*i^3 - 20*i^2 - 500 >= 0) then
        print(i)
        break
    end
end
```

首先， `for` 的3个表达式是在循环开始前一次性求值的。例如，上例中的 `f(x)` 只会执行一次。其次，控制变量会被自动地声明为 `for` 语句的局部变量，并且仅在循环体内可见。

泛型for

泛型 `for` 循环通过一个迭代器函数来遍历所有值：

```
1.      -- 打印数组a的所有值
2.      for i,v in ipairs(a) do print(v) end
```

Lua的基础库提供了 `ipairs` ，这是一个用于遍历数组的迭代器函数。在每次循环中， `i` 会被赋予一个索引值，同时 `v` 被赋予一个对应于该索引的数组元素值。

```
1.      -- 打印table t中所有的key
2.      for k in pairs(t) do print(k) end
```

> 迭代table元素的（pairs）、迭代数组元素的（ipairs）、迭代字符串中单词的（string.gmatch）等。

泛型**for**循环与数字型**for**循环有两个相同点：

```
1.      循环变量是循环体的局部变量；
2.      绝不应该对循环变量作任何赋值。
```

```
1.      -- 例：
2.
3.      days = {"Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday",
    "Saturday"}
4.
5.      -- 需要创建如下talbe
6.      revDays = {["Sunday"] = 1, ["Monday"] = 2, ["Tuesday"] = 3, ["Wednesday"] =
    4, ["Thursday"] = 5, ["Friday"] = 6, ["Saturday"] = 7}
7.
8.      -- 则可以按如下实现
9.      revDays = {}
10.     for k, v in pairs(day) do
11.         revDays[v] = k
12.     end
```

- break与return

`break` 和 `return` 语句用于跳出当前的块。 `break` 语句用于结束一个循环，它只会跳出包含它的那个内部循环（ `for` 、 `repeat` 或 `while` ），而不会改变外层的循环。在执行了 `break` 后，程序会在那个被跳出的循环之后继续执行。

`return` 语句用于从一个函数中返回结果，或者用于简单地结束一个函数的执行。任何函数的结尾处都有一句隐式的 `return` 。所以如果一个函数，它没有值需要返回，那么就无须在其结尾处添加 `return` 语句。 由于语法构造的原因， `break` 或 `return` 只能是一个块的最后一条语句。或者是 `end` 、 `else` 或 `until` 前的一条语句。

例如，准备调试一个函数，但又不想执行该函数的内容。在这种情况下，可以使用一个显式的 `do` 块来包住一条 `return` 语句：

```
1.      funtion foo()
2.      return      --<< 语法错误
3.          -- 在以下个块中return就是最后一条语句
4.          do return end       -- OK
5.          <其他语句>
6.      end
```

?

函数

- 多重返回值
- 变长参数
- 具名实参
- 深入函数
    - closure（闭合函数）
    - 非全局的函数
    - 正确的尾调用

一个函数若只有一个参数，并且此参数是一个字面字符串或 `table` 构造式，那么圆括号便是可有可无的。

```
-- 例如：

print "Hello World"     <-->    print("Hello World")
dofile 'a.lua'          <-->    dofile('a.lua')
print [[a multi-line    <-->    print([[a multi-line
  message]]                       message]]
f{x=10, y=20}           <-->    f({x=10, y=20})
type{}                  <-->    type({})
```

Lua为面向对象式的调用也提供了一种特殊的语法—-冒号操作符。

```
o.foo(o, x)      -- 另一种写法是 o::foo(x)
```

冒号操作符使调用 `o.foo` 时将 `o` 隐含地作为函数第一个参数。

> 一个Lua程序既可以使用以Lua编写的函数，又可以调用以C语言编写的函数，调用时没有任何区别。

调用函数时提供的实参数量可以与形参数量不同。Lua会自动调整实参的数量，以匹配参数表的要求。

```
-- 假设一个函数如下：
function f(a, b) return a or b end

-- 在以下几种调用中，实参与形参的对应关系为：
-- 调用         形参
```

```
f(3)          a=3, b=nil
f(3, 4)       a=3, b=4
f(3, 4, 5)    a=3, b=4 (5被丢弃了)
```

```
-- 默认实参的应用

function incCount(n)
    n = n or 1
    count = count + n
end
```

- 多重返回值

Lua允许函数返回多个结果。例如，用于字符串中定位一个模式的函数 `string.find` 。该函数若在字符串中找到了指定的模式，将返回匹配的起始字符和结尾字符的索引。

```
startIndex, endIndex = string.find("hello Lua users", "Lua")
print(startIndex, endIndex)        --> 7   9
```

以Lua编写的函数同样可以返回多个结果，只需在 `return` 关键字后列出所有的返回值即可。

```
-- 查找数组中的最大元素，并返回该元素的位置：

function maximum(a)
    local index = 1         -- 最大值的索引
    local max = a[index]    -- 最大值
    for i,val in ipairs(a) do
        if val > max then
            max = val; index = i
        end
    end
    return max, index
end

print(maximum{8, 10, 23, 12, 5})    --> 23   3
```

Lua会调整一个函数的返回值数量以适应不同的调用情况。若将函数调用作为一条单独语句时，Lua会丢弃函数的所有返回值。若将函数作为表达式的一部分来调用时，Lua只保留函数的第一个返回

值。只有当一个函数调用是一系列表达式中的最后一个元素（或仅有一个元素）时，才能获得它的所有返回值。　　这里所谓的“一系列表达式”在Lua中表现为4种情况：多重赋值、函数调用时传入的实参列表、 `table` 的构造式和 `return` 语句。

```
function foo0() end                    -- 无返回值
function foo1() return "a" end         -- 返回1个结果
function foo2() return "a", "b" end    -- 返回2个结果
```

```
x,y = foo2()                           -- x="a", y="b"
x = foo2()                             -- x="a", "b"被丢弃
x,y,z = 10, foo2()                     -- x=10, y="a", z="b"

x,y = foo0()                           -- x=nil, y=nil
x,y = foo1()                           -- x="a", y=nil
x,y,z = foo2()                         -- x="a", y="b", z=nil

-- 函数调用不是表达式最后一个元素，将只产生一个值：
x,y = foo2(), 20                       -- x="a", y=20
x,y = foo0(), 20, 30                   -- x=nil, y=20, 30被丢弃

-- 函数调用作为另一个函数调用的最后一个（或仅有的）实参时，第一个函数的所有返回值都将作为实参传入第二个函数。如print：

print(foo0())            -->
print(foo1())            --> a
print(foo2())            --> a  b
print(foo2(), 1)         --> a  1
print(foo2() .. "x")     --> ax
```

`table` 构造式可以完整地接收一个函数调用的所有结果：

```
t = {foo0()}             -- t = {}  (一个空的table)
t = {foo1()}             -- t = {"a"}
t = {foo2()}             -- t = {"a", "b"}
```

不过，这种行为只有当一个函数调用作为最后一个元素时才会发生，而在其他位置上的函数调用总是产生一个结果值：

```
t = {foo0(), foo2(), 4}        -- t[1] = nil, t[2] = "a", t[3] = 4
```

最后一种情况是 `return` 语句，诸如 `return f()` 这样的语句将返回 `f` 的所有返回值：

```
function foo(i)
    if i==0 then return foo0();
    elseif i==1 then return foo1();
    elseif i==2 then return foo2()
    end
end

print(foo(1))        -- a
print(foo(2))        -- a  b
print(foo(0))        -- (无返回值)
print(foo(3))        -- (无返回值)
```

也可以将一个函数调用放入一对圆括号中，从而迫使它只返回一个结果：

```
print((foo0()))      -- nil
print((foo1()))      -- a
print((foo2()))      -- a
```

请注意 `return` 语句后面的内容是不需要圆括号的。例如 `return (f(x))` ，将只返回一个值，而无关乎 `f` 返回了几个值。

关于多重返回值还要介绍一个特殊函数 – `unpack` 。它接受一个数组作为参数，并从下标1开始返回该数组的所有元素：

```
print(unpack{10, 20, 30})            --> 10  20  30
a,b = unpack{10, 20, 30}             --> a=10, b=20, 30被丢弃
```

> Lua5.1及之前的版本中， `unpack` 作为全局函数使用，可以直接
>
> ```
> unpack(arg)
> ```
>
> 而5.2之后， `unpack` 被移到了 `table` 下面，于是直接 `unpack` 就会导致报错，新的调用应该为：
>
> ```
> table.unpack(arg)
> ```
>
> 同时修改的还有另外一个：arg以前 `...` 可以在函数内直接使用 `arg` 来处理，但是5.1之后，就需要自己手动变换成 `arg` 了

```
local arg = {…}
print(arg[1])
```

提供一种兼容的方法

```
function test(…)
    if arg ~= nil then
        arg = {…}
    end
    if unpack != nil then        – 5.1及之前的版本
        print(unpack(arg))
    else                         – 之后的版本
        local arg = {…}
        print(table.unpack(arg))
    end
end
```

```
f = string.find
a = {"hello", "ll"}
```

`f(unpack(a))` 将返回3和4，这与直接调用 `string.find("hello", "ll")` 所返回的结果一摸一样。 虽然这个预定义函数 `unpack` 是用C语言直接编写的，但是仍可以在Lua中通过递归实现一样效果：

```
function unpack(t, i)
    i = i or 1
    if t[i] then
        return t[i], unpack(t, i+1)
    end
end
```

- 变长参数

Lua中的函数可以接受不同数量的实参。

```
1.      -- 这个函数返回了所有参数的总和：
2.
3.      function add( ... )
4.          local s = 0
5.          for i, v in ipairs( ... ) do  -- 表达式{...}表示一个由所有变长参数构成的数
    组。
6.              s = s + v
7.          end
8.          return s
9.      end
10.
11.     print(add(3, 4, 10, 25, 12))        --> 54
```

参数中的3个点（ `...` ）表示该函数可接受不同数量的实参。 Lua提供了专门用于格式化文本（ `string.format` ）和输出文本（ `io.write` ）的函数。

```
1.      function fwrite(fmt, ...)
2.      return io.write(string.format(fmt, ...))
```

注意在3个点前有一个固定参数 `fmt` 。具有变长参数的函数同样也可以拥有任意数量的固定参数，但固定参数必须放在变长参数之前。 变长参数中可能会包含一些故意传入的 `nil` ，那么此时就需要用函数 `select` 来访问变长参数了。调用 `select` 时，必须传入一个固定实参 `selector` (选择开关)和一系列变长参数。如果 `selector` 为数字 `n` ，那么 `select` 返回它的第 `n` 个可变实参；否则， `selector` 只能为字符串“ `#` ”，这样 `select` 会返回变长参数的总数。

```
1.      for i=1, select('#', ...) do
2.          local arg = select(i, ...)        -- 得到第i个参数
3.          <循环体>
4.      end
```

特别需要指出的是， `select('#', ...)` 会返回所有变长参数的总数，其中包括 `nil` 。

- 具名实参

```
-- 无效的演示代码
rename(old="temp.lua", new="temp1.lua")
```

Lua并不直接支持这种语法，但可以通过一种细微的改变来获得相同的效果。主要是将所有实参组织到一个 `table` 中，并将这个 `table` 作为唯一的实参传给函数。另外，还需要用到一种Lua中特殊的函数调用语法，就是当实参只有一个 `table` 构造式时，函数调用中的圆括号是可有可无的：

```
rename{old="temp.lua", new="temp1.lua"}
```

另一方面，将 `rename` 改为只接受一个参数，并从这个参数中获取实际的参数：

```
function rename(arg)
    return os.rename(arg.old, arg.new)
end
```

若一个函数拥有大量的参数，而其中大部分参数是可选的话，这种参数传递风格会特别有用。例如在一个GUI库中，一个用于创建新窗口的函数可能会具有许多的参数，而其中大部分都是可选的，那么最好使用具名实参：

```
w = Window{ x=0, y=0, width=300, height=200,
            title="Lua", background="blue", border=true
          }
```

`Window` 函数可以根据要求检查一些必填参数，或者为某些参数添加默认值。假设“ `_Window` ”才是真正用于创建新窗口的函数，它要求所有参数以正确的次序传入，那么 `Window` 函数可以这么写：

```
function Window(options)
    -- 检查必要的参数
    if type(options.title) ~= "string" then
        error("no tile")
    elseif type(options.width) ~= "number" then
        error("no width")
    end

    -- 其他参数都是可选的
    _Window(options.title,
            options.x or 0,            -- 默认值
            options.y or 0,            -- 默认值
            options.width, options.height,
```

```
                options.backgournd or "white",         -- 默认值
                options.border                         -- 默认值为false(nil)
        )
    end
```

- 深入函数

函数可以存储到变量中（无论全局变量还是局部变量）或 `table` 中，可以作为实参传递给其他函数，还可以作为其他函数的返回值。

一个函数可以嵌套在另一个函数中，内部的函数可以访问外部函数中的变量。

函数与其他值一样都是匿名的。当讨论一个函数名时（例如 `print` ），实际上是在讨论一个持有某函数的变量。

```
    a = {p = print}
    a.p("Hello World")          --> Hello World
    print = math.sin            --> 'print'现在引用了正弦函数
    a.p(print(1))               --> 0.841470
    sin = a.p                   --> 'sin'现在引用了print函数
    sin(10, 20)                 --> 10  20
```

```
    -- Lua中最常见的是函数编写方式，诸如：
    function foo(x) return 2*x end
```

只是一种所谓的“语法糖”而已。也就是说，这只是以下代码的一种简化书写形式：

```
    foo = function(x) return 2*x end
```

因此，一个函数定义实际就是一条语句（更准确地说是一条赋值语句），这条语句创建了一种类型为“函数”的值，并将这个值赋予一个变量。 可以将表达式 `“function(x)<body>end”` 视为一种函数的构造式，就像table的构造式 `{}` 一样。 将这种函数构造式的结果称为一个“匿名函数”。虽然一般情况下，会将函数赋予全局变量，即给予其一个名称。但在某些特殊情况下，仍会需要用到匿名函数。

`table` 库提供了一个函数 `table.sort` ，它接受一个 `table` 并对其中的元素排序。像这种函数就必须支持各种各样可能的排序准则，例如升序还是降序、按数字顺序还是按字符顺序或者按 `table` 中 `key` 的顺序等。 `sort` 函数并没有提供所有这些排序准则，而是提供了一个可选的参数，所谓“次序函数”。这个函数接受两个元素，并返回在有序情况下第一个元素是否应排在第二个元

素之前。举例来说，假设有一个 `table` 内容如下：

假设有一个table内容如下：

```
network = {
    {name = "grauna", IP = "210.26.30.34"},
    {name = "arraial", IP = "210.26.30.23"},
    {name = "lua", IP = "210.26.23.12"},
    {name = "derain", IP = "210.26.23.20"}
}
```

如果想以 `name` 字段、按反向的字符顺序来对这个 `table` 排序的话，只需这么写：

```
table.sort(network, function(a,b) return (a.name > b.name) end)
```

```
for i, v in ipairs( network ) do
    print( v["name"] )
end
```

可见匿名函数在这条语句中就显示出了极好的便捷性。

像 `sort` 这样的函数，接受另一个函数作为实参的，称其是一个“高阶函数”。高阶函数是一种强大的编程机制，应用匿名函数来创建高阶函数所需的实参则可以带来更大的灵活性。

• closure（闭合函数）

若将一个函数写在另一个函数之内，那么这个位于内部的函数便可以访问外部函数中的局部变量，这项特征称之为“词法域”。　　先来看一个简单的例子。假设有一个学生姓名的列表和一个对应于每个姓名的年级列表，需要根据每个学生的年级来对他们的姓名进行排序（由高到低）。可以这么做：

```
name = {"Peter", "Paul", "Mary"}
grades = {Mary = 10, Paul = 7, Peter = 8}
table.sort(names, function(n1, n2)
    return grades[n1] > grades[n2]          -- 比较年级
end)
```

现在假设要单独创建一个函数来做这项工作：

```
function sortbygrade(names, grades)
    table.sort(names, function(n1, n2) return grades[n1] > grades[n2] end)
```

```
3.      end
```

在上例中有一点很有趣，传递给 `sort` 的匿名函数可以访问参数 `grades` ，而 `grades` 是外部函数 `sortbygrade` 的局部变量。在这个匿名函数内部， `grades` 既不是全局变量也不是局部变量，将其称为一个“非局部的变量”。

为什么在Lua中允许这种访问呢？原因在于函数是“第一类值”。考虑以下代码：

```
1.      function newCounter()
2.          local i = 0
3.          return function() i = i + 1 return i; end
4.      end
5.
6.      c1 = newCounter()
7.      print(c1())         -->  1
8.      print(c2())         -->  2
```

在这段代码中，匿名函数访问了一个“非局部的变量” `i` ，该变量用于保持一个计数器。 初看上去，由于创建变量i的函数（ `newCounter` ）已经返回，所以之后每次调用匿名函数时， `i` 都应是已超出了作用范围的。但其实不然，Lua会以 `closure` 的概念来正确地处理这种情况。简单地讲，一个 `closure` 就是一个函数加上该函数所需访问的所有“非局部的变量”。如果再次调用 `newCounter` ，那么它会创建一个新的局部变量 `i` ，从而也将得到一个新的 `closure` ：

```
1.      c2 = newCounter()
2.      print(c2())         -->  1
3.      print(c1())         -->  3
4.      print(c2())         -->  2
```

因此 `c1` 和 `c2` 是同一个函数所创建的两个不同的 `closure` ，它们各自拥有局部变量 `i` 的独立实例。

从技术上讲，Lua中只有 `closure` ，而不存在“函数”。因为，函数本身就是一种特殊的 `closure` 。不过只要不会引起混淆，仍将采用术语“函数”来指代 `closure` 。

`closure` 在另一种情况中也非常有用。例如在Lua中函数是存储在普通变量中的，因此可以轻易地重新定义某些函数，甚至是重新定义那些预定义的函数。这也正是Lua相当灵活的原因之一。通常当重新定义一个函数的时候，需要在新的实现中调用原来的那个函数。举例来说，假设要重新定义函数 `sin` ，使其参数能使用角度来代替原来的弧度。那么这个新函数就必须得转换它的实参，并调用原来的 `sin` 函数完成真正的计算。这段代码可能是这样的：

```
1.      oldSin = math.sin
2.      math.sin = function(x)
```

```
        return oldSin(x*math.pi/180)
    end
```

还有一种更彻底的做法是这样的：

```
    do
        local oldSin = math.sin
        local k = math.pi/180
        math.sin=function(x)
            return oldSin(x*k)
        end
    end
```

将老版本的 `sin` 保存到了一个私有变量中，现在只有通过新版本的 `sin` 才能访问到它了。

可以使用同样的技术来创建一个安全的运行环境，即所谓的“沙盒”。当执行一些未受信任的代码时就需要一个安全的运行环境，例如在服务器中执行那些从Internet上接收到的代码。举例来说，如果要限制一个程序访问文件的话，只需使用 `closure` 来重新定义函数 `io.open` 就可以了。

```
    do
        local oldOpen = io.open
        local access_OK = function(filename, mode)
            <检查访问权限>
        end
        io.open = function(filename, mode)
            if access_OK(filename, mode) then
                return oldOpen(filename, mode)
            else
                return nil, "access denied"
            end
        end
    end
```

这个示例的精彩之处在于，经过重新定义后，一个程序就只能通过新的受限版本来调用原来那个未受限的 `open` 函数了。示例将原来不安全的版本保存到了 `closure` 的一个私有变量中，从而使得外部再也无法直接访问到原来的版本了。通过这种技术，可以在Lua的语言层面上就构建出一个安全的运行环境，且不失简易性和灵活性。相对于提供一套大而全的解决方案，Lua提供的则是一套“元机制”，因此可以根据特定的安全需要来创建一个安全的运行环境。

- 非全局的函数

由于函数是一种“第一类值”，因此一个显而易见的推论就是，函数不仅可以存储在全局变量中，还可以存储在 `table` 的字段中和局部变量中。前面讲到了几个将函数存储在 `table` 字段中的示例，大部分Lua库也采用了这种机制（例如 `io.read` 、 `math.sin` ）。若要在Lua中创建这种函数，只需将常规的函数语法与 `table` 语法结合起来使用即可：

```
Lib = {}
Lib.foo = function(x, y) return x + y end
Lib.goo = function(x, y) return x - y end
```

当然，还可以使用构造式：

```
Lib = {
    foo = function(x, y) return x + y end,
    goo = function(x, y) return x - y end
}
```

除了这些之外，Lua还提供了另一种语法来定义这类函数：

```
Lib = {}
function Lib.foo(x, y) return x + y end
function Lib.goo(x, y) return x - y end
```

只要将一个函数存储到一个局部变量中，即得到了一个“局部函数”，也就是说该函数只能在某个特定的作用域中使用。对于“程序包”而言，这种函数定义是非常有用的。因为Lua是将每个程序块作为一个函数来处理的，所以在一个程序块中声明的函数就是局部函数，这些局部函数只在该程序块中可见。词法域确保了程序包中的其他函数可以使用这些局部函数：

```
local f = function(<参数>)
    <函数体>
end

local g = function(<参数>)
    <一些代码>
    f()
    <一些代码>
end
```

对于这种局部函数的定义，Lua还支持一种特殊的“语法糖”：

```
local function f(<参数>)
    <函数体>
```

```
3.      end
```

在定义递归的局部函数时，还有一个特别之处需要注意。像下面这种采用了基本函数定义语法的代码多数是错误的：

```
1.      local fact = function(n)
2.          if n == 0 then return 1
3.          else return n * fact(n-1)         -- 错误
4.          end
5.      end
```

当Lua编译到函数体中调用 `fact(n-1)` 的地方时，由于局部的 `fact` 尚未定义完毕，因此这句表达式其实是调用了一个全局的 `fact` ，而非此函数自身。为了解决这个问题，可以先定义一个局部变量，然后再定义函数本身：

```
1.      local fact
2.      fact = function(n)
3.          if n == 0 then return 1
4.          else return n * fact(n-1)
5.          end
6.      end
```

现在函数中的 `fact` 调用就表示了局部变量。即使在函数定义时，这个局部变量的值尚未完成定义，但之后在函数执行时， `fact` 则肯定已经拥有了正确的值。

当Lua展开局部函数定义的“语法糖”时，并不是使用基本函数定义语法。而是对于局部函数定义：

```
1.      local funciton foo(<参数>) <函数体> end
```

Lua将其展开为：

```
1.      local foo
2.      foo = function(<参数>) <函数体> end
```

因此，使用这种语法来定义递归函数不会产生错误：

```
1.      local function fact(n)
2.          if n == 0 then return 1
```

```
        else return n * fact(n-1)
        end
    end
```

当然，这个技巧对于间接递归的函数而言是无效的。在间接递归的情况中，必须使用一个明确的前向声明：

```
    local f, g        -- 前向声明

    function g()
        <一些代码> f() <一些代码>
    end

    function f()
        <一些代码> g() <一些代码>
    end
```

> 注意⚠，别把第二个函数定义写为“ `local function f` ”。如果那样的话，Lua会创建一个全新的局部变量 `f` ，而将原来声明的 `f` （函数 `g` 中所引用的那个）置于未定义的状态。

- ## 正确的尾调用

Lua中的函数还有一个有趣的特征，那就是Lua支持“尾调用消除”。 所谓“尾调用”就是一种类似于 `goto` 的函数调用。当一个函数调用是另一个函数的最后一个动作时，该调用才算一条“尾调用”。

举例来说，以下代码中对 `g` 的调用就是一条“尾调用”：

```
    function f(x) return g(x) end
```

也就是说，当 `f` 调用完 `g` 之后就再无其他事情可做了。因此在这种情况中，程序就不需要返回那个“尾调用”所在的函数了。 所以在“尾调用”之后，程序也不需要保存任何关于该函数的栈信息了。当 `g` 返回时，执行控制权可以直接返回到调用 `f` 的那个点上。有一些语言实现（例如Lua解释器）可以得益于这个特点，使得在进行“尾调用”时不耗费任何栈空间。将这种实现称为支持“尾调用消除”。

由于“尾调用”不会耗费栈空间，所以一个程序可以拥有无数嵌套的“尾调用”。

举例来说，在调用以下函数时，传入任何数字作为参数都不会造成栈溢出：

```
1.      function foo(n)
2.          if n > 0 then return foo(n-1) end
3.      end
```

有一点需要注意的是，当想要受益于“尾调用消除”时，务必要确定当前的调用是一条“尾调用”。判断的准则就是“一个函数在调用完另一个函数之后，是否就无其他事情需要做了”。有一些看似是“尾调用”的代码，其实都违背了这条准则。

举例来说，在下面的代码中，对 `g` 的调用就不是一条“尾调用”：

```
1.      function f(x) g(x) end
```

这个示例的问题在于，当调用完 `g` 后， `f` 并不能立即返回，它还需要丢弃 `g` 返回的临时结果。类似地，以下所有调用也都不符合上述准则：

```
1.      return g(x) + 1                  -- 必须做一次加法
2.      return x or g(x)               -- 必须调整为一个返回值
3.      return (g(x))                   -- 必须调整为一个返回值
```

在**Lua**中，只有“ `return <func>(<args>)` ”这样的调用形式才算是一条“尾调用”。Lua会在调用前对 `<func>` 及其参数求值，所以它们可以是任意复杂的表达式。

举例来说，下面的调用就是一条“尾调用”：

```
1.      return x[i].foo(x[j] + a * b, i + j)
```

在之前提到了，一条“尾调用”就好比是一条 `goto` 语句。因此，在Lua中“尾调用”的一大应用就是编写“状态机”。这种程序通常以一个函数来表示一个的状态，改变状态就是 `goto` （或调用）到另一个特定的函数。举一个简单的迷宫游戏的例子来说明这个问题。

例如，一个迷宫有几间房间，每间房间中最多有东南西北4扇门。用户在每一步移动中都需要输入一个移动的方向。如果在某个方向上有门，那么用户可以进入相应的房间；不然，程序就打印一条警告。游戏目标就是让用户从最初的房间走到最终的房间。

这个游戏就是一种典型的状态机，其中当前房间就是一个状态。可以将迷宫中的每间房间实现为一个函数，并使用“尾调用”来实现从一间房间移动到另一间房间。在以下代码中，实现一个具有4间房间的迷宫：

```
function room1()
    local move = io.read()
    if move == "south" then return room3()
    elseif move == "east" then return room2()
    else
        print("invalid move")
        return room1()            -- stay in the same room
    end
end

function room2()
    local move = io.read()
    if move == "south" then return room4()
    elseif move == "west" then return room1()
    else
        print("invalid move")
        return room2()            -- stay in the same room
    end
end

function room3()
    local move = io.read()
    if move == "north" then return room1()
    elseif move == "east" then return room4()
    else
        print("invalid move")
        return room3()            -- stay in the same room
    end
end

function room4()
    print("congratulations!")
end
```

通过调用初始房间来开始这个游戏：

```
room1()
```

若没有“尾调用消除”的话，每次用户的移动都会创建一个新的栈层，移动若干步之后就有可能会导致栈溢出。而“尾调用消除”则对用户移动的次数没有任何限制。这是因为每次移动实际上都只是完成一条 `goto` 语句到另一个函数，而非传统的函数调用。

对于这个简单的游戏而言，或许会觉得将程序设计为数据驱动的会更好一点，其中将房间和移动记录在一些 `table` 中。不过，如果游戏中的每间房间都有各自特殊的情况的话，采用这种状态机的设计则更为合适。

？

迭代器与范型for

本张将介绍如何编写适用于泛型 `for` 的迭代器（Iterator）。先从简单的迭代器入手，然后将学习如何利用泛型 `for` 的各种能力来编写更简单、更有效的迭代器。

- 迭代器与closure
- 泛型for的语义
- 无状态的迭代器
- 具有复杂状态的迭代器
- 真正的迭代器

• 迭代器与closure

所谓“迭代器”就是一种可以遍历一种集合中所有元素的机制。在Lua中，通常将迭代器表示为函数。每调用一次函数，即返回集合中的“下一个”元素。

每个迭代器都需要在每次成功调用之间保持一些状态，这样才能知道它所在的位置及如何步进到下一个位置。 `closure` 对于这类任务提供了极佳的支持，一个 `closure` 就是一种可以访问其外部嵌套环境中的局部变量的函数。对于 `closure` 而言，这些变量就可用于在成功调用之间保持状态值，从而使 `closure` 可以记住它在一次遍历中所在的位置。当然，为了创建一个新的 `closure` ，还必须创建它的这些“非局部的变量”。因此一个 `closure` 结构通常涉及到两个函数： `closure` 本身和一个用于创建该 `closure` 的工厂函数。

```
function values( t )
    local i = 0;
    return function( )
        i = i + 1;
        return t[i];
    end
end

t = { 10, 20, 30 }
iter = values(t)
while true do
    local element = iter();
    if element == nil then break end
    print( element )
```

```
    end

    for element in values(t) do
        print( element )
    end
```

作为示例，来为列表编写一个简单的迭代器。与 `ipairs` 不同的是该迭代器并不是返回每个元素的索引，而是返回元素的值：

```
    function values(t)
        local i = 0
        return function() i = i + 1; return t[i] end
    end
```

在本例中， `values` 就是一个工厂。每当调用这个工厂时，它就创建一个新的 `closure` （即迭代器本身）。这个 `closure` 将它的状态保存在其外部变量 `t` 和 `i` 中。每当调用这个迭代器时，它就从列表 `t` 中返回下一个值。直到最后一个元素返回后，迭代器就会返回 `nil` ，以此表示迭代的结束。

可以在一个 `while` 循环中使用这个迭代器：

```
    t = {10, 20, 30}
    iter = values(t)        -- 创建迭代器
    while true do
        local element = iter()
        if element == nil then break end
        print(element)
    end
```

然而使用泛型 `for` 则更为简单。接下来会发现，它正是为这种迭代而设计的：

```
    t = {10, 20, 30}
    for element in values(t) do
        print(element)
    end
```

泛型 `for` 为一次迭代循环做了所有的簿记工作。它在内部保存了迭代器函数，因此不再需要 `iter` 变量。它在每次新迭代时调用迭代器，并在迭代器返回 `nil` 时结束循环。在下一节中将会看到泛型 `for` 能够做更多的事情。

下面是一个更高级的示例，展现了一个可以遍历当前输入文件中所有单词的迭代器 – `allwords` 。为了完成这样的遍历，需要保持两个值：当前行的内容（变量 `line` ）及在该行中所处的位置（变量 `pos` ）。有了这些信息，就可以不断产生下一个单词了。这个迭代器函数的主要部分就是 `string.find` 的调用。此调用会在当前行中，以当前位置作为起始来搜索一个单词。使用模式（ `pattern` ）‘ `%w+` ’来描述一个“单词”，它用于匹配一个或多个的文字 / 数字字符。如果 `string.find` 找到了一个单词，迭代器就会将当前位置更新为该单词之后的第一个字符，并返回该单词。否则，它就读取新的一行并反复这个搜索过程。若没有剩余的行，则返回 `nil` ，以此表示迭代的结束。 尽管迭代器本身具有复杂性，但 `allwords` 的使用还是很简明易懂的：

```
for word in allwords() do
    print(word)
end
```

对于迭代器而言，一种常见的情况就是：编写迭代器本身或许不太容易，但使用它们却是很容易的。这也不会成为一个大问题，因为通常使用Lua编程的最终用户不会去定义迭代器，而只是使用那些程序提供的迭代器。

```
function allwords()
    local line = io.read()    – 当前行
    local pos = 1             – 一行中的当前位置
    return function()         – 迭代器函数
        while line do         – 若为有效的行内容就进入循环
            local s, e = string.find(line, "%w+", pos)
            if s then         – 是否找到一个单词
                pos = e + 1   – 该单词的下一个位置
                return string.sub(line, s, e)    – 返回该单词
            else
                line = io.read()    – 没有找到单词，尝试下一行
                pos = 1             – 在第一个位置上重新开始
            end
        end
        return nil                  – 没有其余行了，遍历结束
    end
end
```

- 泛型for的语义

上述的那些迭代器都有一个缺点，就是需要为每个新的循环都创建一个新的 `closure` 。对于大多

数情况而言，这或许不会有什么问题。例如在之前的 `allwords` 迭代器中，创建一个 `closure` 的代价相对于读取整个文件的代价而言，几乎可以忽略不计。但是，在另外一些情况下，这样的开销就不太容易接受了。因此，在这类情况中，希望能通过泛型 `for` 的自身来保存迭代器状态。在本节中会详细说明泛型 `for` 的这种保存状态的机制。

泛型 `for` 在循环过程内部保存了迭代器函数。实际上它保存着3个值：一个迭代器函数、一个恒定状态和一个控制变量。接下来将对此进行详细说明。

泛型 `for` 的语法如下：

```
for <var-list> in <exp-list> do
    <body>
end
```

其中， `<var-list>` 是一个或多个变量名的列表，以逗号分隔； `<exp-list>` 是一个或多个表达式的列表，同样以逗号分隔。通常表达式列表只有一个元素，即一句对迭代器工厂的调用。

例如，如下代码：

```
for k, v in pairs(t) do print(k, v) end
```

其中变量列表是“ `k, v` ”，表达式列表只有一个元素 `pairs(t)` 。一般来说变量列表中也只有一个变量，例如下面这个循环：

```
for line in io.lines() do
    io.write(line, "\n")
end
```

> 变量列表的第一个元素称为“控制变量”。在循环过程中该值决不会为 `nil` ，因为当它为 `nil` 时循环就结束了。

`for` 做的第一件事情是对 `in` 后面的表达式求值。这些表达式应该返回3个值供 `for` 保存：迭代器函数、恒定状态和控制变量的初值。这里有点类似于多重赋值，即只有最后一个表达式才会产生多个结果，并且只会保留前3个值，多余的值会被丢弃；而不足的话，将以 `nil` 补足。

在初始化步骤之后， `for` 会以恒定状态和控制变量来调用迭代器函数。然后 `for` 将迭代器函数的返回值赋予变量列表中的变量。如果第一个返回值为 `nil` ，那么循环终止。否则， `for` 执行它的循环体，随后再次调用迭代器函数，并重复这个过程。

> 更明确地说，以下语句：
>
> ```
> for var_1, …, var_n in <explist> do <block> end
> ```

等价于以下代码：

```
do
    local _f, _s, _var = <explist>
    while true do
        local var_1, …, var_n = _f(_s, _var)
        _var = var_1
        if _var == nil then break end
        <block>
    end
end
```

因此，假设迭代器函数为 `f` ，恒定状态为 `s` ，控制变量的初值为 a_0 。那么在循环过程中控制变量的值依次为 a_1 = f(s, a_0)、 a_2 = f(s, a_1)，以此类推，直至 a_i 为 `nil` 结束循环。如果 `for` 还有其他变量，那么它们也会在每次调用 `f` 后获得额外的值。

- 无状态的迭代器

所谓“无状态的迭代器”，正如其名所暗示的那样，就是一种自身不保存任何状态的迭代器。因此，我们可以在多个循环中使用同一个无状态的迭代器，避免创建新的 `closure` 开销。

在每次迭代中， `for` 循环都会用恒定状态和控制变量来调用迭代器函数。一个无状态的迭代器可以根据这两个值来为下次迭代生成下一个元素。这类迭代器的一个典型例子就是 `ipairs` ，它可以用来迭代一个数组的所有元素：

```
a = {"one", "two", "three"}
for i, v in ipairs(a) do
    print(i, v)
end
```

在这里，迭代的状态就是需要遍历的 `table` （一个恒定状态，它不会在循环中改变）及当前的索引值（控制变量）。 `ipairs` （工厂）和迭代器都非常简单，在Lua中就可以编写出来：

```
local function iter(a, i)
    i = i + 1
```

```
        local v = a[i]
        if v then
            return i, v
        end
    end

    function ipairs(a)
        return iter, a, 0
    end
```

在Lua调用 `for` 循环中的 `ipairs(a)` 时，它会获得3个值：迭代器函数 `iter` 、恒定状态 `a` 和控制变量的初值0。然后Lua调用 `iter(a, 0)` ，得到1， `a[1]` 。在第二次迭代中，继续调用 `iter(a,1)` ，得到2， `a[2]` ，以此类推，直至得到第一个nil元素为止。

函数 **`pairs`** 与 **`ipairs`** 类似，也是用于遍历一个 **`table`** 中的所有元素。不同的是，它的迭代器函数是**Lua**中的一个基本函数 **`next`** 。

```
    function pairs(t)
        return next, t, nil
    end
```

> 在调用 `next(t,k)` 时， `k` 是 `table t` 的一个 `key` 。此调用会以 `table` 中的任意次序返回一组值：此 `table` 的下一个 `key` ，及这个 `key` 所对应的值。而调用 `next(t,nil)` 时，返回 `table` 的第一组值。若没有下一组值时， `next` 返回 `nil` 。

有些用户喜欢不通过 `pairs` 调用而直接使用 `next` ：

```
    for k, v in next, t do
        <loop body>
    end
```

记住，Lua会自动将 `for` 循环中表达式列表的结果调整为3个值。因此上例中得到了 `next` 、 `t` 和 `nil` ，这也正与调用 `pairs(t)` 的结果完全一致。

关于无状态迭代器的另一个有趣例子是一种可以遍历链表的迭代器。

```
    local function getnext(list, node)
        if not node then
            return list
        else
            return node.next
        end
```

```
    end

    function traverse(list)
        return getnext, list, nil
    end
```

这里使用了一个技巧就是将链表的头节点作为恒定状态（ `traverse` 返回的第二个值），而将当前节点作为控制变量。第一次调用迭代器函数 `getnext` 时， `node` 为 `nil` ，因此函数返回 `list` 作为第一个结点。在后续调用中 `node` 不再为 `nil` 了，所以迭代器如期望的那样返回 `node.next` 。

对于此迭代器的使用则非常简单：

```
    list = nil
    for line in io.lines() do
        list = {val = line, next = list}
    end

    for node in traverse(list) do
        print(node.val)
    end
```

```
    -- 链表的实现

    node = {}
    head = node

    -- 初始化
    function init(v)
        node.val = v
    end

    -- 在尾部插入
    function push_back(v)
        node.next = {val = v}
        node = node.next
    end

    init(10)
    push_back(8)
    push_back(6)
```

```
        -- 迭代器函数
        local function getnext(list, node)
            if node then
                return node.next
            else
                return list
            end
        end

        function traverse( list )
            return getnext, list, nil
        end

        for v in traverse(head) do
            print( v.val )
        end
```

- 具有复杂状态的迭代器

通常，迭代器需要保存许多状态，可是泛型 `for` 却只提供的一个恒定状态和一个控制变量用于状态的保存。一个最简单的解决方法就是使用 `closure` 。或者还可以将迭代器所需的所有状态打包为一个 `table` ，保存在恒定状态中。一个迭代器通过这个 `table` 就可以保存任意多的数据了。此外，它还能在循环过程中改变这些数据。虽然，在循环过程中恒定状态总是同一个 `table` (故称之“恒定”)，但这个 `table` 的内容却可以发生改变。由于这种迭代器可以在恒定状态中保存所有数据，所以它们通常可以忽略泛型 `for` 提供的第二个参数（控制变量）。

作为该技术的一个示例，将重写 `allwords` 迭代器，这个迭代器可以遍历当前输入文件中的所有单词。这次将它的状态保存到一个 `table` 中，这个 `table` 具有两个字段： `line` 和 `pos` 。

迭代的起始函数比较简单，它只需返回迭代器函数和初始状态：

```
1.      local iterator        -- 在后面定义
2.
3.      function  allwords()
4.          local state = {line = io.read(), pos = 1}
5.          return iterator, state
6.      end
```

`iterator` 函数才开始真正的工作：

```
1.      function iterator(state)
2.          while state.line do          -- 若为有效的行内容就进入循环
3.              -- 搜索下一个单词
4.              local s, e = string.find(state.line, "%w+", state.pos)
5.              if s then              -- 找到了一个单词？
6.                  -- 更新下一个位置（到这个单词之后）
7.                  state.pos = e + 1
8.                  return string.sub(state.line, s, e)
9.              else          -- 没有找到单词
10.                 state.line = io.read()         -- 尝试下一行...
11.                 state.pos = 1          -- 从第一个位置开始
12.             end
13.         end
14.         return nil              -- 没有更多行了，结束循环
15.     end
```

尽可能地尝试编写无状态的迭代器，那些迭代器将所有状态保存在 `for` 变量中，不需要在开始一个循环时创建任何新的对象。如果迭代器无法套用这个模型，那么就应该尝试使用 `closure` 。 `closure` 显得更加优雅一点，通常一个基于 `closure` 实现的迭代器会比一个使用 `table` 的迭代器更为高效。这是因为，首先创建一个 `closure` 就比创建一个 `table` 更廉价，其次访问“非局部的变量”也比访问 `table` 字段更快。以后会看到另一种使用协同程序（ `coroutine` ）编写迭代器的方式，这种方式是功能最强的，但稍微有一点开销。

• 真正的迭代器

“迭代器”这个名称多少有点误导的成分。因为迭代器并没有做实际的迭代，真正做迭代的是 `for` 循环。而迭代器只是为每次迭代提供一些成功后的返回值。或许，更准确地应称其为“生成器”。不过“迭代器”这个名称已在其他语言中被广泛使用，例如Java。

还有一种创建迭代器的方式就是，在迭代器中做实际的迭代操作。当使用这种迭代器时，就不需要写一个循环了。相反，需要一个描述了在每次迭代时需要做什么的参数，并以此参数来调用迭代器。更确切地说，迭代器接受一个函数作为参数，并在其内部的循环中调用这个函数。

在此列举一个更具体的例子，就使用这种风格来再次重写 `allwords` 迭代器：

```
1.      function allwords(f)
2.          for line in io.lines() do
3.              for word in string.gmatch(line, "%w+") do
```

```
                f(word)         -- call the function
            end
        end
    end
```

使用这个迭代器时，需要传入一个描述循环体的函数。例如，只想打印每个单词，那么可以使用 `print` ：

```
    allwords(print)
```

通常，还可以使用一个匿名函数作为循环体。例如，以下代码计算了单词“ `hello` ”在输入文件中出现的次数：

```
    local count = 0
    allwords(function(w)
        if w == "hello" then count = count + 1 end
    end)
    print(count)
```

同样的任务若采用之前的迭代器风格也不是很困难的：

```
    local count = 0
    for w in allwords() do
        if w == "hello" then count = count + 1 end
    end
    print(count)
```

“真正的迭代器”在老版本的Lua中曾非常流行，那时语言还没有 `for` 语句。那它们比之生成器风格的迭代器又如何呢？这两种风格都有大致相同的开销，即每次迭代都有一次函数调用。编写真正的迭代器相对比较容易。不过，生成器风格的迭代器则更具灵活性。这种灵活性体现在两方面，首先它允许两个或多个并行的迭代过程，其次它允许在迭代体中使用 `break` 和 `return` 语句。对于真正的迭代器来说， `return` 语句只能从匿名函数中返回，而并不能从做迭代的函数中返回。综上所述，我一般更喜爱生成器。

?

协同程序

协同程序与线程（ `thread` ）差不多，也就是一条执行序列，拥有自己独立的栈、局部变量和指令指针，同时又与其他协同程序共享全局变量和其他大部分东西。从概念上讲线程与协同程序的主要区别在于，一个具有多个线程的程序可以同时运行几个线程，而协同程序却需要彼此协作地运行。就是说，一个具有多个协同程序的程序在任意时刻只能运行一个协同程序，并且正在运行的协同程序只会在其显式地要求挂起（**suspend**）时，它的执行才会暂停。

1. • 协同程序基础
2. • 管道与过滤器
3. • 以协同程序实现迭代器
4. • 非抢先式的多线程

• 协同程序基础

Lua将所有关于协同程序的函数放置在一个名为“ `coroutine` ”的 `table` 中。

函数 `create` 用于创建新的协同程序，它只有一个参数，就是一个函数。该函数的代码就是协同程序所需执行的内容。 `create` 会返回一个 `thread` 类型的值，用以表示新的协同程序。通常 `create` 的参数是一个匿名函数。

```
1.  co = coroutine.create( function () print( "hi" ) end )
2.  print( co )    -- thread: 0x7fe2f1506218
```

一个协同程序可以处于4种不同的状态：挂起（ `suspended` ）、运行（ `running` ）、死亡（ `dead` ）和正常（ `normal` ）。当创建一个协同程序时，它处于挂起状态。也就是说，协同程序不会在创建它时自动执行其内容。可以通过函数 `status` 来检查协同程序的状态：

```
1.  print( coroutine.status( co ) )    -- suspended
```

函数 `coroutine.resume` 用于启动或再次启动一个协同程序的执行，并将其状态由挂起改为运行：

```
1.  coroutine.resume( co )             -- hi
```

在本例中，协同程序的内容只是简单地打印了“ `hi` ”后便终止了，然后它就处于死亡状态，也就

再也无法返回了：

```
print( coroutine.status( co ) )    -- dead
```

到目前为止，协同程序看上去还只是像一种复杂的函数调用方法。其实协同程序的真正强大之处在于函数 `yield` 的使用上，该函数可以让一个运行中的协同程序挂起，而之后可以再恢复它的运行。

```
co = coroutine.create( function ()
    for i = 1, 10 do
        print( "co", i )
        coroutine.yield()
    end
end )
```

现在当唤醒这个协同程序时，它就会开始执行，直到第一个 `yield` ：

```
coroutine.resume( co )    -- co 1
```

如果此时检查其状态，会发现协同程序处于挂起状态，因此可以再次恢复其运行：

```
print( coroutine.status( co ) )    -- suspended
```

从协同程序的角度看，所有在它挂起时发生的活动都发生在 `yield` 调用中。当恢复协同程序的执行时，对于 `yield` 的调用才最终返回。然后协同程序继续它的执行，直到下一个 `yield` 调用或执行结束：

```
coroutine.resume( co )    -- co 2
coroutine.resume( co )    -- co 3
...
coroutine.resume( co )    -- co 10
coroutine.resume( co )    -- 什么都不打印
```

在最后一次调用 `resume` 时，协同程序的内容已经执行完毕，并已经返回。因此，这时协同程序处于死亡状态。如果试图再次恢复它的执行， `resume` 将返回 `false` 及一条错误消息：

```
print(coroutine.resume( co ))    -- false cannot resume dead coroutine
```

> 请注意， `resume` 是在保护模式中运行的。因此，如果在一个协同程序的执行中发生任何错误，Lua是不会显示错误消息的，而是将执行权返回给 `resume` 调用。

当一个协同程序A唤醒一个协同程序B时，协同程序A就处于一个特殊状态，既不是挂起状态（无法

继续A的执行），也不是运行状态（是B在运行）。所以将这时的状态称为“正常”状态。

Lua的协同程序还具有一项有用的机制，就是可以通过一对 `resume-yield` 来交换数据。在第一次调用 `resume` 时，并没有对应的 `yield` 在等待它，因此所有传递给 `resume` 的额外参数都将视为协同程序主函数的参数：

```
1.      co = coroutine.create( function ( a, b, c )
2.          print( "co", a, b, c )
3.      end )
4.      coroutine.resume( co, 1, 2, 3 )    -- co 1 2 3
```

在 `resume` 调用返回的内容中，第一个值为 `true` 则表示没有错误，而后面所有的值都是对应 `yield` 传入的参数：

```
1.      co = coroutine.create( function ( a, b )
2.          coroutine.yield( a + b, a - b )
3.      end )
4.
5.      print( coroutine.resume( co, 20, 10 ) )    -- true 30 10
```

与此对应的是， `yield` 返回的额外值就是对应 `resume` 传入的参数：

```
1.      co = coroutine.create( function ()
2.          print( "co", coroutine.yield() )
3.      end )
4.
5.      print(coroutine.resume( co, "a" ))      -- true
6.      print(coroutine.resume( co, 4, 5, 6 ))  -- co 4 5 6  -- true
7.      print(coroutine.resume( co, 4, 5))      -- false cannot resume dead
    coroutine
```

最后，当一个协同程序结束时，它的主函数所返回的值都将作为对应 `resume` 的返回值：

```
1.      co = coroutine.create( function ()
2.          return 6, 7
3.      end )
4.      print( coroutine.resume( co ) )        -- true 6 7
```

- 管道与过滤器

```
1.  ---------------------------------------- 管道与过滤器
2.
3.      function receive( prod )
4.          local status, value = coroutine.resume( prod )
5.          return value
6.      end
7.
8.      function  send( x )
9.          coroutine.yield( x )
10.     end
11.
12.     function producer( )
13.         return coroutine.create( function ( )
14.             while true do
15.                 local x = io.read()         -- 产生新值
16.                 send(x)
17.             end
18.         end )
19.     end
20.
21.     function filter( prod )
22.         return coroutine.create( function ( )
23.             for line = 1, math.huge do
24.                 local x = recevie(prod)         -- 获取新值
25.                 x = string.format( "%5d %s", line, x )
26.                 send(x)                         -- 将新值发送给消费者
27.             end
28.         end )
29.     end
30.
31.     function consumer( prod )
32.         while true do
33.             local x = receive(prod)         -- 获取新值
34.             io.write( x, "\n" )             -- 消费新值
35.         end
36.     end
37.
38.     -- 运行代码
39.     consumer(filter(producer()))
```

- 以协同程序实现迭代器

将循环迭代器视为“生产者-消费者”模式的一种特例，一个迭代器会产出一些内容，而循环体会消费这些内容。因此，这样看来协同程序似乎也适用于实现迭代器。的确，协同程序为实现这类任务提供了一项有用的工具。那就是先前提到的，协同程序可以一改传统的调用者与被调用者之间的关系。有了这个特性，在编写迭代器时，就无须顾及如何在每次成功的迭代调用之间保存状态信息了。

为了说明这类应用，下面来写一个迭代器，使其可以遍历某个数组的所有排列组合形式。若直接编写这种迭代器可能不太容易，但若编写一个递归函数来产生所有的排列组合则不会很困难。想法很简单，只要将每个数组元素都依次放在最后一个位置，然后递归地生成其余元素的排列。代码如下：

```
1.  function permgen(a, n)
2.      n = n or #a          -- 默认n为a的大小
3.      if n <= 1 then
4.          printResult(a)
5.      else
6.          for i = 1, n do
7.              -- 将第i个元素放到数组末尾
8.              a[n], a[i] = a[i], a[n]
9.              -- 生成其余元素的排列
10.             permgen(a, n - 1)
11.             -- 恢复第i个元素
12.             a[n], a[i] = a[i], a[n]
13.         end
14.     end
15. end
```

然后，还需要定义其中调用到的打印函数 `printResult` ，并以适当的参数来调用 `permgen` ：

```
1.  function printResult(a)
2.      for i = 1, #a do
3.          io.write(a[i], " ")
4.      end
5.      io.write("\n")
6.  end
7.
8.  permgen({1, 2, 3, 4})
9.
10. --> 2 3 4 1
11. --> 3 2 4 1
12. --> 3 4 2 1
13. ...
```

```
14.     --> 2 1 3 4
15.     --> 1 2 3 4
```

当生成函数完成后，将其转换为一个迭代器就非常容易了。首先，将 `printResult` 改为 `yield` ：

```
1.  function permgen(a, n)
2.      n = n or #a
3.      if n <= 1 then
4.          coroutine.yield(a)
5.      else
6.          for i = 1, n do
7.              -- 将第i个元素放到数组末尾
8.              a[n], a[i] = a[i], a[n]
9.              -- 生成其余元素的排列
10.             permgen(a, n - 1)
11.             -- 恢复第i个元素
12.             a[n], a[i] = a[i], a[n]
13.         end
14.     end
15. end
```

然后，定义一个工厂函数，用于将生成函数放到一个协同程序中运行，并创建迭代器函数。迭代器只是简单地唤醒协同程序，让其产生下一种排列：

```
1.  function permutations(a)
2.      local co = coroutine.create(function() permgen(a) end)
3.      return function()  -- 迭代器
4.          local code, res = coroutine.resume(co)
5.          return res
6.      end
7.  end
```

有了上面的函数，在 `for` 语句中遍历一个数组的所有排列就非常简单了：

```
1.  for p in permutations{"a", "b", "c"} do
2.      printResult(p)
3.  end
4.
5.  --> b c a
6.  --> c a b
```

```
7.      --> a c b
8.      --> b a c
9.      --> a b c
```

`permutations` 函数使用了一种在Lua中比较常见的模式，就是将一条唤醒协同程序的调用包装在一个函数中。由于这种模式比较常见，所以Lua专门提供了一个函数 `coroutine.wrap` 来完成这个功能。类似于 `create` ， `wrap` 创建了一个新的协同程序。但不同的是， `wrap` 并不是返回协同程序本身，而是返回一个函数。每当调用这个函数，即可唤醒一次协同程序。但这个函数与 `resume` 的不同之处在于，它不会返回错误代码。当遇到错误时，它会引发错误。若使用 `wrap` ，可以这么写 `permutations` ：

```
1.      function permutations(a)
2.          return coroutine.wrap(function() permgen(a) end)
3.      end
```

通常， `coroutine.wrap` 比 `coroutine.create` 更易于使用。它提供了一个对于协同程序编程实际所需的功能，即一个可以唤醒协同程序的函数。但也缺乏灵活性。无法检查 `wrap` 所创建的协同程序的状态，此外，也无法检测出运行时的错误。

• 非抢先式的多线程

协同程序提供了一种协作式的多线程。每个程序都等于是一个线程。一对 `yield-resume` 可以将执行权在不同线程之间切换。然后，协同程序与常规的多线程的不同之处在于，协同程序是非抢先式的。就是说，当一个协同程序运行时，是无法从外部停止它的。只有当协同程序显式地要求挂起时（调用 `yield` ），它才会停止。对于有些应用而言，这没有问题，而对于另外一些应用则可能无法接受这种情况。当不存在抢先时，编程会简单许多。无须为同步的 `bug` 而抓狂，在程序中所有线程间的同步都是显式的，只需确保一个协同程序在它的临界区域之外调用 `yield` 即可。

对于非抢先式的多线程来说，只要有一个线程调用了一个阻塞的（ `blocking` ）操作，整个程序在该操作完成前，都会停止下来。对于大多数应用程序来说，这种行为是无法接受的。这也导致了许多程序员放弃协同程序，转而使用传统的多线程。接下来会用一个有趣的方法来解决这个问题。

先假设一个典型的多线程使用情况：希望通过 `HTTP` 下载几个远程的文件。当然，若要下载几个远程文件，就必须先知道如何下载一个远程文件。在本例中，将使用 `Diego Nehab` 开发的 `LuaSocket` 。为了下载一个文件，必须先打开一个到该站点的连接，然后发送下载文件的请求，并接收文件（数据块），最后关闭连接。在Lua中可以按以下步骤来完成这项任务。首先，加载 `LuaSocket` 库。

```
1.      require "socket"
```

然后，定义主机和下载的文件。本例，将从 `World Wide Consortium` （环球网协会）下载《 `HTML 3.2参考规范》` 。

```
1.      host = "www.w3.org"
2.      file = "/TR/REC-html32.html"
```

接下来，打开一个 `TCP` 连接，连接到该站点的 `80` 端口。

```
1.      c = assert(socket.connect(host, 80))
```

这步操作将返回一个连接对象，可以用它来发送文件请求。

```
1.      c:send("GET" .. file .. "HTTP/1.0\r\n\r\n")
```

下一步，按1K的字节块来接收文件，并将每块写到标准输出：

```
1.      while true do
2.          local s, status, partial = c:receive(2^10)
3.          io.write(s or partial)
4.          if status == "closed" then break end
5.      end
```

在正常情况下 `receive` 函数会返回一个字符串。若发生错误，则会返回 `nil` ，并且附加错误代码（ `status` ）及出错前读取到的内容（ `partial` ）。当主机关闭连接时，就将其余接收到的内容打印出来，然后退出接收循环。

下载完文件后，关闭连接。

```
1.      c:close()
```

现在已经掌握了如何下载一个文件，那么再回到下载几个文件的问题上。最繁琐的做法是逐个地下载文件。因为，这种顺序的做法太慢了，它只能在下载完一个文件后才开始读取该文件。当接收一个远程文件时，程序将大部分的时间花费在等待数据接收上。更明确地说，是将时间用在了对 `receive` 阻塞调用上。因此，如果一个程序可以同时下载所有文件的话，那么它的运行速度就可以快很多了。当一个连接没有可用数据时，程序便可以从其他连接处读取数据。很明显协同程序提供了一种简便的方式来构建这种并发下载的结构。可以为每个下载任务创建一个新的线程，只要一个线程无可用数据，它就可以将控制权转让给一个简单的调度程序，而这个调度程序则会去调用其他的下载线程。

在以协同程序来重写程序前，先将前面的下载代码重新写为一个函数。代码如下：

```
1.      function download(host, file)
```

```
        local c = assert(socket.connect(host, 80))
        local count = 0          -- 记录接收到的字节数
        c:send("GET " .. file .. "HTTP/1.0\r\n\r\n")
        while true do
            local s, status, partial = receive(c)
            count = count + #(s or partial)
            if status == "closed" then break end
        end
        c:close()
        print(file, count)
    end
```

由于对远程文件的内容并不感兴趣，所以不需要将文件内容写到标准输出中，只需计算并打印出文件大小即可。在上述代码中，还使用了一个辅助函数 `receive` 来从连接接收数据。在顺序下载的方法中， `receive` 的代码可以是这样的：

```
    function receive(connection)
        return connection:receive(2^10)
    end
```

而在并发的实现中，这个函数在接收数据时绝对不能阻塞。因此，它需要在没有足够的可用数据时挂起执行。新代码如下：

```
    function receive(connection)
        connection.settimeout(0)     -- 使recevie调用不会阻塞
        local s, status, partial = connection:receive(2^20)
        if status == "timeout" then
            coroutine.yield(connection)
        end
        return s or partial, status
    end
```

对 `settimeout(0)` 的调用可使以后所有对此连接进行的操作不会阻塞。若一个操作返回的 `status` 为“ `timeout` （超时）”，就表示该操作在返回时还未完成。此时，线程就会挂起执行。而以非假的参数来调用 `yield` ，可以告诉调度程序线程仍在执行任务中。注意，即使在超时的情况下，连接也是会返回已经读取到的内容，即记录在 `partial` 变量中的值。

以下这段代码展示了调度程序及一些辅助代码。 `table threads` 为调度程序保存着所有正在运行中的线程。函数 `get` 确保每个下载任务都在一个独立的线程中执行。调度程序本身主要就是一个循环，它遍历所有的线程，逐个唤醒它们的执行。并且当线程完成任务时，将该线程从列表中删除。在所有线程都完成运行后，停止循环。

```
    threads = {}       -- 用于记录所有正在运行的线程

    function get(host, file)
        -- 创建协同程序
        local co = coroutine.create(function()
            download(host, file)
        end)
        -- 将其插入记录表中
        table.insert(threads, co)
    end

    function dispatch()
        local i = 1
        while true do
            if threads[i] == nil then                    -- 还有线程吗？
                if threads[1] == nil then break end      -- 列表是否为空？
                i = 1                                    -- 重新开始循环
            end
            local status, res = coroutine.resume(threads[i])
            if not res then                              -- 线程是否已经完成了任务？
                table.remove(threads, i)
            else
                i = i + 1
            end
        end
    end
```

最后，主程序需要创建所有的线程，并调用调度程序。例如，若要下载W3C站点上的4个文件，主程序如下：

```
    host = "www.w3.org"

    get(host, "/TR/html401/html40.txt")
    get(host, "/TR/2002/REC-xhtml1-20020801/xhtml1.pdf")
    get(host, "/TR/REC-html32.html")
    get(host, "/TR/2000/REC-DOM-Level-2-Core-20001113/DOM2-Core.txt")

    dispatch()         -- 主循环
```

通过协同程序，计算机只需要6秒便可下载完成这4个文件。但若使用顺序下载的话，则需要多耗费

两倍的时间（15秒左右）。

除了速度有所提高外，上面这个实现还不够完美。只要有一个线程在读取数据，就不会有问题。但若所有线程都没有数据可读，调度程序就会执行一个“忙碌等待（ `Busy Wait` ）”，不断地从一个线程切换到另一个线程，仅仅是为了检测是否还有数据可读。这样便导致了这个协同程序的实现会比顺序下载多耗费将近30倍的CPU时间。

为了避免这样的情况，可以使用 `LuaSocket` 中的 `select` 函数。这个函数可以用于等待一组 `socket` 的状态改变，在等待时程序陷入阻塞（ `block` ）状态。若要在当前实现中应用这个函数，只需要修改调度程序即可，新版本如下：

```
1.      function dispatch()
2.          local i = 1
3.          local connections = {}
4.          while true do
5.              if threads[i] == nil then                    -- 还有线程吗？
6.                  if threads[1] == nil then break end
7.                  i = 1                                    -- 重新开始循环
8.                  connections = {}
9.              end
10.             local status, res = coroutine.resume(threads[i])
11.             if not res then                              -- 线程是否已经完成了任
    务？
12.                 table.remove(threads, i)
13.             else                    -- 超时
14.                 i = i + 1
15.                 connections[#connections + 1] = res
16.                 if #connections == #threads then         -- 所有线程都阻塞了吗？
17.                     socket.select(connections)
18.                 end
19.             end
20.         end
21.     end
```

新的调度程序将所有超时的连接收集到一个名为 `connections` 的 `table` 中。记住， `receive` 会将超时的连接通过 `yield` 传递，也就是 `resume` 会返回它们。如果所有的连接都超时了，调度程序就调用 `select` 来等待这些连接的状态发生变化。这个最终版本的实现与上一个使用协同程序的实现一样快，另外由于它不会有“忙碌等待”，所以只比顺序下载耗费CPU资源略多而已。

？

数据结构

1. - 数组
2. - 矩阵与多维数组
3. - 链表
4. - 队列与双向队列
5. - 集合与无序组（bag）
6. - 字符串缓冲
7. - 图

Lua中的 `table` 不是一种简单的数据结构，它可以作为其他数据结构的基础。其他语言提供的数据结构，如数组、记录、线性表、队列、集合等，在Lua中都可以通过 `table` 来表示。此外，用Lua的 `table` 来实现这些结构的效率高。

在C和Pascal这样的传统语言中，尽管可以使用Lua的 `table` 来实现数组和列表，但通常以数组和列表（记录+指针）来实现大多数的数据结构。因为 `table` 本身就比数组和列表的功能强大得多。由此许多算法都可以忽略一些细节问题，从而简化它们的实现。例如，在Lua中很少编写搜索算法，这是因为 `table` 本身就提供了直接访问任意类型的功能。

高效地使用 `table` 是问题的关键。接下来将演示如何通过 `table` 来实现一些传统的数据结构，并且还将给出一些使用这些结构的例子。首先从数组和列表开始，不是因为需要它们来作为其他结构的基础，而是因为大多数程序员都比较熟悉它们了。在前面关于语言的章节中也曾提到过这方面的内容，不过为了完整性起见，本章将更详细地进行讨论。

- 数组

使用整数来索引 `table` 即可在Lua中实现数组。因此，数组没有一个固定的大小，可以根据需要增长。通常，当初始化一个数组时，也就间接地定义了它的大小。

例如，在执行了以下代码后，任何对字段范围1～1000之外的访问都会返回一个 `nil` ，而不是0：

```
a = {}                  -- 新建一个数组
for i = 1, 1000 do
    a[i] = 0
end
```

长度操作符（ `#` ）依赖于这个事实来计算数组的大小：

```
print(#a)            --> 1000
```

可以使用0、1或其他任意值来作为数组的起始索引：

```
-- 使用索引值-5～5来创建一个数组
a = {}
for i = -5, 5 do
    a[i] = 0
end
```

> 然而，在Lua中的习惯一般是以1作为数组的起始索引。Lua库和长度操作符都遵循这个约定。如果你的数组不是从1开始的，那就无法使用这些功能了。

通过 `table` 的构造式，可以在一句表达式中创建并初始化数组：

```
squares = {1, 4, 9, 16, 25, 36, 49, 64, 81}
```

这种构造式可以根据要求变得更长。

• 矩阵与多维数组

在Lua中，有两种方式来表示矩阵。第一种是使用一个“数组的数组”，也就是说，一个 `table` 中的每个元素是另一个 `table` 。例如，使用以下代码来创建N×M的零矩阵：

```
mt = {}                -- 创建矩阵
for i = 1, N do
    mt[i] = {}             -- 创建一个新行
    for j = 1, M do
        mt[i][j] = 0
    end
end
```

由于在Lua中 `table` 是一种对象，因此在创建矩阵时，必须显式地创建每一行。从一方面看，这的确比在C和Pascal中直接声明一个多维数组烦琐；但从另一方面看，它也给予了更多的灵活性。例如，创建一个三角形矩阵，只需将前例中的循环 `for j = 1, M do … end` 改为 `for j = 1, i do …end` 。这种修改同时可以使三角形矩阵只使用原先一半的内存。

在Lua中表示矩阵的第二种方式是将两个索引合并为一个索引。如果两个索引是整数，可以将第一个索引乘以一个适当的常量，并加上第二个索引。以下代码就使用这种方法来创建N×M：

```
mt = {}        -- 创建矩阵
for i = 1, N do
    for j = 1, M do
        mt[(i-1)*M + j] = 0
    end
end
```

如果索引是字符串，那么可以把索引拼接起来，中间使用一个字符来分隔。例如，使用字符串 `s` 和 `t` 来索引一个矩阵，可以通过代码 `m[s..":"..t]` 。其中， `s` 和 `t` 都不能包含冒号，否则像 `("a:", "b")` 或 `("a", ":b")` 这样的索引会使最终索引变成“ `a::b` ”。如果无法保证这点的话，可以使用例如' `\0` '这样的控制字符来分隔两个索引。

通常应用程序会用到一种特殊的矩阵，称为“稀疏矩阵”，这种矩阵中的大多数元素为0或 `nil` 。例如，可以通过稀疏矩阵来表示一个图（graph）。当矩阵的 `m` ， `n` 位置上有一个值 `x` ，即表示图中的结点 `m` 和 `n` 是相连的，其权重（cost）为 `x` ；若图中这些节点不相连的话，则矩阵 `m` ， `n` 位置上的值为 `nil` 。若要表示一个具有1万个节点的图，其中每个结点大约会与其他5个结点相连，那么就需要一个能包含1亿个元素的矩阵，但是其中大约只有5万个元素不为 `nil` 。许多数据结构的书籍都会讨论到这个大小问题，如何才能实现这种稀疏矩阵而不浪费400MB内存。当在Lua中编程时，则无须用到这些技术。因为，数组是以 `table` 来表示的，它们本身就是稀疏的。在第一种表示（ `table` 的 `table` ）中，需要1万个 `table` ，每个 `table` 包含5个元素，总共5万个条目。在第二种表示中，需要一个 `table` ，其中包含5万个条目。无论哪种表示方式，都只需要为非 `nil` 的元素付出空间。

虽然对稀疏矩阵使用长度操作符不是一种语法错误。但也不能进行该操作，因为在有效条目之间存在“空洞（ `nil` 值）”。在大多数对稀疏矩阵的操作中，由于存在许多空条目，遍历矩阵是非常低效的。所以，一般使用 `pairs` 且只遍历那些非 `nil` 的元素。

例如，要将矩阵的一行与一个常量相乘，可以使用以下代码：

```
function mult(a, rowindex, k)
    local row = a[rowindex]
    for i, v in pairs(row) do
        row[i] = v * k
    end
end
```

注意， `table` 中的 `key` 是无序的，所以使用 `pairs` 的迭代并不保证会按递增次序来访问元素。对于一些任务而言（像上面这个例子），这没有问题；但对于另一些任务而言，或许就需要采用另外的方法了，比如链表。

- 链表

由于 `table` 是动态的实体，所以在Lua中实现链表是很方便的。每个结点以一个 `table` 来表示，一个“链接”只是结点 `table` 中的一个字段，该字段包含了对其他 `table` 的引用。

例如，要实现一个基础的列表，其中每个结点具有两个字段： `next` 和 `value` ，先创建一个用作列表头结点的变量：

```
1.      list = nil
```

在表头插入一个元素，元素值为 `v` ：

```
1.      list = {next = list, value = v}
```

遍历此列表：

```
1.      local l = list
2.      while l do
3.          <访问l.value>
4.          l = l.next
5.      end
```

至于其他类型的列表，例如双向链表或环形表，都可以使用相同的方法实现。然而，在Lua中很少需要这类结构，因为通常存在着一些更简单的方式来表示数据。例如，可以通过一个（几乎无限大的）数组来表示一个栈。

- 队列与双向队列

在Lua中实现队列的一种简单方法是使用 `table` 库的函数 `insert` 和 `remove` 。这两个函数可以在一个数组的任意位置插入或删除元素，并且根据操作要求移动后续元素。不过对于较大的结构，移动的开销是很大的。一种更高效的实现是使用两个索引，分别用于首尾的两个元素：

```
1.      function ListNew()
2.          return {first = 0, last = -1}
3.      end
```

为了避免污染全局名称空间，将在一个 `table` 内部定义所有的队列操作，这个 `table` 且称

为 `List` 。这样，将上例重新写为：

```
List = {}
function List.new()
    return {first = 0, last = -1}
end
```

现在就可以在常量时间内完成在两端插入或删除元素了。

```
function List.pushfirst(list, value)
    local first = list.first - 1
    list.first = first
    list[first] = value
end

function List.pushlast(list, value)
    local last = list.last + 1
    list.last = last
    list[last] = value
end

function List.popfirst(list)
    local first = list.first
    if first > list.last then error("list is empty") end
    local value = list[first]
    list[first] = nil              -- 为了允许垃圾收集
    list.first = first + 1
    return value
end

function List.poplast(list)
    local last = list.last
    if list.first > last then error("list is empty") end
    local value = list[last]
    list[last] = nil               -- 为了允许垃圾收集
    list.last = last - 1
    return value
end
```

如果希望该结构能严格地遵循队列的操作规范，那么只调用 `pushlist` 和 `poplist` 就可以了，这样 `first` 和 `last` 都会不断地增长。然而，在Lua中使用 `table` 来表示数组，即可以在1～20的范围内索引，也可以在16777216～16777236的范围内索引。因为Lua使用双精度来表示数字，程序

可以以每秒1百万次的速度进行插入操作，如此运行200年都不会发生溢出问题。

- ## 集合与无序组（bag）

假设列一份程序代码中的所有标识符，并且过滤掉其中所有的保留字。一些C程序员会倾向于使用字符串的数组来表示保留字集合，然后搜索这个数组来查看一个单词是否属于该集合。为了提高搜索的速度，他们可能还会使用二叉树来表示该集合。

在Lua中有一种高效且简单的方式来表示这类集合，就是将集合元素作为索引放入一个 `table` 中。那么对于任意值都无须搜索 `table` ，只需用该值来索引 `table` ，并查看结果是否为 `nil` 。在当前假设的示例中，可如下：

```
1.      reserved = {["while"] = true, ["end"] = true, ["function"] = true,
    ["local"] = true, }
2.
3.      for w in allwords() do
4.          if not reserved[w] then
5.              <对'w'作任意处理>          -- 'w'不是保留字
6.          end
7.      end
```

若要使初始化过程变得更清晰，可以借助一个辅助函数来创建集合：

```
1.      function Set(list)
2.          local set = {}
3.          for _, l in ipairs(list) do set[l] = true end
4.          return set
5.      end
6.
7.      reserved = Set {"while", "end", "function", "local"}
```

包，有时也称为“多重集合（Multiset）”，与普通的集合的不同之处在于其每个元素可以出现多次。在Lua中包的表示类似于上面的集合表示，只不过包需要将一个计数器与 `table` 的 `key` 关联。若要插入一个元素，则需要递增其计数器：

```
1.      function insert(bag, element)
2.          bag[element] = (bag[element] or 0) + 1
3.      end
```

若要删除一个元素，则需要递减其计数器：

```
1.      function remove(bag, element)
2.          local count = bag[element]
3.          bag[element] = (count and count > 1) and count - 1 or nil
4.      end
```

只有当计数器已存在或大于0时，才保留它。

• 字符串缓冲

假设正在编写一段关于字符串的代码，例如正在逐行地读取一个文件。典型的读取代码是这样的：

```
1.      local buff = ""
2.      for line in io.lines() do
3.          buff = buff .. line .. "\n"
4.      end
```

这段代码看似可以正常工作，但如果面对较大的文件时，它却会导致极大的性能开销。例如，用这段代码来读取一个350KB大小的文件就需要将近1分钟的时间。

为什么会这样呢？为了搞清楚执行期间到底发生了什么，来设想一下当前正在处于读取循环的中间。假设每行有20个字节，已读了2500行，那么buff现在就是一个50KB大小的字符串。当Lua作字符串连接 `buff .. line .. "\n"` 时，就创建了一个长50020字节的新字符串，并从buff中复制了50000字节到这个新字符串。这样，对于后面的每一行，Lua都需要移动50KB甚至更多的内存。在读取了100行（仅2KB）以后，Lua就已经移动了至少5MB的内存。此外，这个算法还具有二次复杂度。最后，当Lua完成了350KB的读取后，它已至少移动了50GB的数据。

这个问题不是Lua所特有的，在其他语言中，只要字符串是不可变的（immutable）值，也会有类似的问题。Java就是最有名的例证。

在继续讲解前，需要注明一下这种情况并不是一种常见的问题。对于较小的字符串，上述循环可以很好地工作。当需要读取整个文件时，Lua提供了 `io.read("*all")` 选项，这样便可以一次性读取整个文件。不过，Java也提供了 `StringBuffer` 来解决这个问题。在Lua中，我们可以将一个 `table` 作为字符串缓冲。其关键是使用函数 `table.concat` ，它会将给定列表中的所有字符串连接起来，并返回连接的结果。使用 `concat` 来重写上述循环：

```
1.      local t = {}
2.      for line in io.lines() do
3.          t[#t + 1] = line .. "\n"
4.      end
5.      local s = table.concat(t)
```

先前代码读取同样的文件需要1分钟，而这个实现只需花小于0.5秒的时间。

`concat` 函数还有第二个可选的参数，可以指定一个插在字符串间的分隔符。有了这个分隔符参数，就不必在每行后插入一个“ `\n` ”了。

```
1.      local t = {}
2.      for line in io.lines() do
3.          t[#t + 1] = line
4.      end
5.      s = table.concat(t, "\n") .. "\n"
```

虽然 `concat` 函数会在字符串之间插入分隔符，但还需要在结尾处添加一个换行。为此上例在最后一次连接时复制了整个结果字符串，而这时的字符串也已经相当长了。没有直接的选项让 `concat` 插入这个额外的分隔符，不过可以“欺骗” `concat` ，只需在t后面添加一个空字符串：

```
1.      t[#t + 1] = ""
2.      s = table.concat(t, "\n")
```

`concat` 在空字符串前插入了这个额外的换行符，位于结果字符串的末尾。

从内部来看， `concat` 和 `io.read("*all")` 都使用了一个相同的算法来连接许多小的字符串。标准库中的其他几个函数也使用这个算法来创建较大的字符串。下面来看一下它是如何工作的。

一开始循环采用了一种线性的方法来连接字符串，把较小的字符串逐个地连接起来，然后每次都将连接结果存入一个累加器。而新的算法可以避免这么做，它采用了一种二分的方法（binary approach）。它只在某些情况下将几个较小的字符串连接起来，然后再将结果字符串与更大的字符串进行连接。其算法的核心是一个栈，已创建的大字符串位于栈的底部，而较小的字符串则通过栈顶进入。对栈中元素的处理类似于著名的“汉诺塔（Tower of Hanoi）”。栈中的任意字符串都比下面的字符串短。如果压入的新字符串比下面已存在的字符串长，就将两者连接。然后，再将连接后的新字符串与更下面的字符串比较，如果是新建字符串更长的话，则再次连接它们。这样的连接一直向下延续应用，直到遇到一个更大的字符串或者到达了栈底为止。

```
1.      function addString(stack, s)
2.          stack[#stack + 1] = s             -- 将's'压入栈
3.          for i = #stack - 1, i, -1 do
4.              if #stack[i] > #stack[i + 1] then
5.                  break
6.              end
7.              stack[i] = stack[i] .. stack[i + 1]
8.              stack[i + 1] = nil
9.          end
```

```
10.     end
```

为了获取栈缓冲中的最终内容，只需连接其中所有的字符串就可以了。

- 图

就像其他编程语言一样，Lua允许程序员写出多种图的实现，每种实现都有其所适用的算法。接下来将介绍一种简单的面向对象的实现，其中结点表示为对象及边（arc）表示为结点间的引用。

每个结点表示为一个 `table` ，这个 `table` 有两个字段： `name` （结点的名称）和 `adj` （与此结点邻接的结点集合）。由于从一个文本文件中读取图数据，所以需要一种通过一个结点名来找到该结点的方法。因此，使用了一个额外的 `table` 来将名称对应到结点。函数 `name2node` 可以根据给定的名称返回对应的结点：

```
1.      local function name2node(graph, name)
2.          if not graph[name] then
3.              -- 结点不存在，创建一个新的
4.              graph[name] = {name = name, adj = {}}
5.          end
6.          return graph[name]
7.      end
```

以下这个函数用于构造一个图。它逐行地读取一个文件，文件中的每行都有两个结点名称，表示了在两个结点之间有一条边，边的方向从第一个结点到第二个结点。函数对于每行都使用 `string.match` 来切分一行中的两个名称，然后根据名称查找结点（如果需要还会创建结点），最后连接结点。

```
1.      function readgraph()
2.          local graph = {}
3.          for line in io.lines() do
4.              -- 切分行中的两个名称
5.              local namefrom, nameto = string.match(line, "(%S+)%s+(%S+)")
6.              -- 查找相应的结点
7.              local from = name2node(graph, namefrom)
8.              local to = name2node(graph, nameto)
9.              -- 将'to'添加到'from'的邻接集合
10.             from.adj[to] = true
11.         end
12.         return graph
13.     end
```

以下这个函数展示了一个使用图的算法。函数 `findpath` 采用深度优先的遍历，在两个结点间搜索一条路径。它的第一个参数是当前结点，第二个参数是其目标结点，第三个参数用于保存从起点到当前结点的路径，最后一个参数是所有已访问过结点的集合（用于避免回路）。注意，该算法直接对结点进行操作，而不是它们的名称。例如， `visited` 是一个结点集合，而不是结点名称的集合。 `path` 也一样是一个结点的列表。

```
function findpath(curr, to, path, visited)
    path = path or {}
    visited = visited or {}
    if visited[curr] then          -- 结点是否已访问过？
        return nil                     -- 这里没有路径
    end
    visited[curr] = true          -- 将结点标记为已访问过
    path[#path + 1] = curr          -- 将其加到路径中
    if curr == to then
        return path
    end
    -- 尝试所有的邻接结点
    for node in pairs(curr.adj) do
        local p = findpath(node, to, path, visited)
        if p then return p end
    end
    path[#path] = nil              -- 从路径中删除结点
end
```

为了测试上述代码，首先编写一个函数来打印一条路径，然后再编写一些代码来使其运行。

```
function printpath(path)
    for i = 1, #path do
        print(path[i].name)
    end
end

g = readgraph()
a = name2node(g, "a")
b = name2node(g, "b")
p = findpath(a, b)
if p then printpath(p) end
```

？

数据文件与持久性

当涉及到数据文件的处理时，人们往往会认为写数据比读数据简单得多。当写一个文件时，对写的内容拥有完全的控制权。但是当读一个文件时，却无从得知会读到什么内容。一个强健的程序除了需要处理一个合法文件中所包含的所有类型的数据，还应能很好地处理坏损的文件。因此，编写一个强健的输入程序总是比较困难的。

在本章中，我们将看到如何使用Lua来避免程序中所有有关数据读取的代码，只需将数据按一种适当的格式书写就可以了。

数据文件

可以借由table构造式来定义一种文件格式。只需在写数据时做一点额外的工作，读取数据就会变得相当容易。这项技术也就是将数据作为Lua代码来输出，当运行这些代码时，程序也就读取了数据。而table的构造式可以使这些输出代码看上去更像是一个普通的数据文件。

下面通过一个示例来更清楚地理解这种做法。如果数据文件是一种预定义的格式，例如CSV（Comma-Separated Values，逗号分隔值）或XML，那么可以选择的做法很少。不过，如果是为了应用而创建数据文件的话，那么就可以使用Lua的构造式作为格式。在这种格式中，每条数据记录表示为一个Lua构造式。这样，原来以这种形式书写的数据文件：

```
1.      Donald E. Knuth, Literate Programming, CSLI, 1992
2.      Jon Bentley, More Programming Pearls, Addison-Wesley, 1990
```

现在可以改为：

```
1.      Entry{ "Donald E. Knuth",
2.             "Literate Programming",
3.             "CSLI",
4.             1992}
5.
6.      Entry{ "Jon Bentley",
7.             "More Programming Pearls",
8.             "Addison-Wesley",
9.             1990}
```

记住，`Entry{<code>}`与`Entry({<code>})`是完全等价的，都是以一个table作为参数来调用函数Entry。因此，上面这段数据也是一个Lua程序。为了读取该文件，我们只需定义一个合适的Entry，然后运行此程序就可以了。例如，以下程序计算了数据文件中条目的数量：

```
local count = 0
function Entry(_) count = count + 1 end
dofile("data")
print("number of entries: " .. count)
```

下一个程序则可用于收集数据文件中所有作者的姓名，然后打印出这些姓名（不需要与文件中的次序相同）：

```
local authors = {}            -- 作者姓名的集合
function Entry(b) authors[b[1]] = true end
dofile("data")
for name in pairs(authors) do print(name) end
```

可以看到这些代码片段都采用了事件驱动的做法。Entry函数作为一个回调函数，在dofile时为数据文件中的每个条目所调用。

若文件不是非常大，可以使用名值对来表示每个字段：

```
Entry{
    author = "Donald E. Knuth",
    title = "Literate Programming",
    publicsher = "CSLI",
    year = 1992
}

Entry{
    author = "Jon Bentley",
    title = "More Programming Pearls",
    publicsher = "Addison-Wesley",
    year = 1990
}
```

这种格式就是“自描述的数据（self-describing data）”格式，其中每项数据都伴随一个表示其含义的简短描述。自描述的数据比CSV或其他紧缩格式更具可读性。当需要修改时，也易于手工编辑，可以在基本格式中作出一个细小的改动，而不需要同时改变数据文件。例如，如果要新增一个字段，只需修改读取程序中的一小块就可以了，内容就是当该字段不存在时提供一个默认值。

使用名值对格式后，那个收集作者姓名的程序改为：

```
local authors = {}         -- 作者姓名的集合
function Entry(b) authors[b.author] = true end
```

```
    dofile("data")
    for name in pairs(authors) do print(name) end
```

现在字段的次序就不重要了，即使有些条目没有作者字段，也只需要修改Entry函数：

```
    function Entry(b)
        if b.author then authers[b.author] = true end
    end
```

Lua不仅运行速度快，而且编译速度也快。例如，上面这个用于列出作者的程序在处理2MB数据时，只需不到1秒钟的时间。这不是偶然的结果，自从Lua创建之初就把数据描述作为Lua的主要应用之一来考虑的，开发人员为能较快地编译大型程序投入了很多的努力。

串行化（Serialization）

通常需要串行化一些数据，也就是将数据转换为一个字节流或字符流。然后就可以将其存储到一个文件中，或者通过网络连接发送出去了。串行化后的数据可以用Lua代码来表示，这样当运行这些代码时，存储的数据就可以在读取程序中得到重构了。

如果想要恢复一个全局变量的值，那么串行化的结果或许可以是“ `varname = <exp>` ”，其中 `<exp>` 是一段用于创建该值的代码。例如，对于一个数字值，方法如下：

```
    function serialize(o)
        if type(o) == "number" then
            io.write(o)
        else
            <其他情况>
        end
    end
```

对于一个字符串值，方法如下：

```
    if type(o) == "string" then
        io.write("'", o, "'")
```

然而，如果字符串中包含特殊字符（例如引号、换行），那么最终代码就不是一段有效的Lua程序了。

也可以使用另一种字符串字面表示方法，如下所示：

```
    if type(o) == "string" then
        io.write("[[", o "]]")
```

注意，如果有用户故意使其字符串为 “]]..os.execute('rm *')..[[”，那么最终保存下来的结果将变成：

```
    varname = [[]]..os.execute('rm *')..[[]]
```

若加载这个“数据”将会出现不可估量的后果。

可以使用一种简单且安全的方法来扩住一个字符串，那就是以“%q”来使用string.format函数。这样它就会用双引号来括住字符串，并且正确地转移其中的双引号和换行符等其他特殊字符。

```
    a = 'a "problematic"\\string'
    print(string.format("%q", a))        --> "a \"problematic\" \\string"
```

通过使用这个特性，serialize函数可以改为：

```
    function serialize(o)
        if type(o) == "number" then
            io.write(o)
        elseif type(o) == "string" then
            io.write(string.format("%q", o))
        else
            <其他情况>
        end
    end
```

Lua5.1还提供了另一种可以以一种安全的方法来括住任意字符串的方法。这是一种新的标记方式 `[=[…]=]` ，用于长字符串。然而，这种新方式主要是为手写的代码提供方便的，通过它就不需要改变任何字符串的内容了。在自动生成的代码中，要转移那些问题字符，还是使用string.format与"%q"选项更为方便。

如果仍然在自动生成的代码中使用长字符串标记的话，那么就需要注意两个细节问题。　　首先，必须使用正确数量的等号。这个正确的数量应比字符串中出现的最长的等号序列还大1。由于，在字符串中出现长序列的等号是很有可能的，并且其他序列也不会产生一个错误的字符串结尾的标记，所以要注意等号序列。　　第二个细节是，Lua总是会忽略所有长字符串开头的换行符。一种避免这个问题的简单方法就是，在字符串起始处添加一个换行符。

以下这个quote函数就是根据上面提到的两个注意点编写的处理函数。

```
    function quote(s)
        -- 查找最长的等号序列
        local n = -1
        for w in string.gmatch(s, "]=*") do
            n = math.max(n, #w - 1)
        end

        -- 产生'n' + 1个等号
        local eq = string.rep("=", n + 1)

        -- 生成长字符串的字面表示
        return string.format(" [%s[\n%s]%s] ", eq, s, eq)
    end
```

它可以接收任意字符串，并放回其格式化为长字符串的结果。对 `string.gmatch` 的调用会创建一个迭代器，通过该迭代器就可以遍历字符串s中所有出现模式 `']=*'` 的地方。在每处出现等号的地方，循环就会更新n，使其保持为当前所遇到的最大等号数量。在循环结束后使用 `string.rep` 将等号重复n+1遍，也就是生成一个等号序列字符串，其长度出现有字符串中的最长等号序列还多1。最后， `string.format` 将s嵌入一对具有正确数量等号的方括号对中，并在方括号外添加一些额外的空格，以及在s开头插入一个换行符。

保存无环的table

下一个任务是保存table。保存table有几种方法，选用哪种方法取决于对table的结构作出了哪些限制性的假设。没有一种算法适用于所有的情况。简单的table不仅需要更简单的算法，而且需要更完美地输出结果。

第一个算法如下：

```
    function serialize(o)
        if type(o) == "number" then
            io.write(o)
        elseif type(o) == "string" then
            io.write(string.format("%q", o)) then
        elseif type(o) == "table" then
            io.write("{\n")
            for k, v in pairs(o) do
                io.write(" ", k, " = ")
                serialize(v)
                io.write(",\n")
```

```
            end
            io.write("}\n")
        else
            error("cannot serialize a " .. type(o))
        end
    end
```

尽管这个函数很简单，但却可以完成基本的保存工作。只要table的结构是一个树结构，它甚至还能处理嵌套的table（table中的table）。可以作为一个练习，尝试在输出格式中缩进那些嵌套的table。

上例函数假设了一个table中的所有key都是合法的标识符。但如果一个table的key为数字或者非法的Lua标识符，那么就会出现问题。一个简单的解决方法是将这行：

```
    io.write(" ", k, " = ")
```

改为：

```
    io.write(" ["); serialize(k); io.write("] = ")
```

这样，便增强了这个函数的强健性，但却损失了结果文件的美观性。对于调用：

```
    serialize{a=12, b='Lua', key='another "one"'}
```

第一个版本的serialize会输出：

```
    {
        a = 12,
        b = "Lua",
        key = "another \"one\""
    }
```

而第二个版本则输出：

```
    {
        ["a"] = 12,
        ["b"] = "Lua",
        ["key"] = "another \"one\"",
    }
```

可以测试每种需要方括号的情况，从而改善结果的美观性。

保存有环的table

若要处理具有任意拓扑结构（带环的table或共享子table）的table，就需要采用另外一种方法了，table构造式是无法表示这类table的。所以为了表示“环”，则需要引入名称，接下来这个保存函数要求将待保存的值及其名称一起作为参数传入。此外，还必须持有一份所有已保持过的table的名称记录，以此来检测环并复用其中的table。使用一个额外的table用作此项纪录，这个table以其他table作为key，并以其他table的名称作为value。代码如下：

```
1.  function basicSerialize(o)
2.      if type(o) == "number" then
3.          return tostring(o)
4.      else             -- assume it is a string
5.          return string.format("%q", o)
6.      end
7.  end
8.
9.  function save(name, value, saved)
10.     saved = saved or {}         -- 初始值
11.     io.write(name, " = ")
12.     if type(value) == "number" or type(value) == "string" then
13.         io.write(basicSerialize(value), "\n")
14.     elseif type(value) == "table" then
15.         if saved[value] then         -- 该value是否已保存过？
16.             io.write(saved[value], "\n")         -- 使用先前的名字
17.         else
18.             saved[value] = name         -- 为下次使用保持名字
19.             io.write("{}\n")             -- 创建一个新的table
20.             for k, v in pairs(value) do
21.                 k = basicSerialize(k)
22.                 local fname = string.format("%s[%s]", name, k)
23.                 save(fname, v, saved)
24.             end
25.         end
26.     else
27.         error("cannot save a " .. type(value))
28.     end
29. end
```

假设准备保存的table的key只为字符串或数字。函数basicSerialize用于串行化这些基本类型，返回串行化的结果。而另一个函数save则完成真正的工作。saved参数是一个table，用于记录已

保存过的table。假设有一个table如下所示：

```
a = {x=1, y=2; {3, 4, 5}}
a[2] = a         -- 环
a.z = a[1]         -- 共享子table
```

然后，调用save("a", a)将它保存为：

```
a = {}
a[1] = {}
a[1][1] = 3
a[1][2] = 4
a[1][3] = 5
a[2] = a
a["x"] = 2
a["y"] = 1
a["z"] = a[1]
```

这些赋值语句的实际顺序可能会有所不同，这取决于一个table的遍历顺序。不过，该算法可以保证在一句新的定义中所用到的变量都已经定义过了。

如果想以共享的方式来保存几个table中的共同部分，只需在调用saved时使用相同的saved参数。例如，假设有两个table：

```
a = {{"one", "two"}, 3}
b = {k = a[1]}
```

如果以独立的方式保存它们，那么结果中不会有共同部分：

```
save("a", a)
save("b", b)

--> a = {}
--> a[1] = {}
--> a[1][1] = "one"
--> a[1][2] = "two"
--> a[2] = 3
--> b = {}
--> b["k"] = {}
--> b["k"][1] = "one"
--> b["k"][2] = "two"
```

然而，当使用同一个saved table来调用save时，串行化结果就会共享共同部分：

```
local t = {}
save("a", a, t)
save("b", b, t)

--> a = {}
--> a[1] = {}
--> a[1][1] = "one"
--> a[1][2] = "two"
--> a[2] = 3
--> b = {}
--> b["k"] = a[1]
```

在Lua中，还有一些其他比较常见的方法。有的在保存一个值时无须给出一个全局名称（而是通过一段代码来构造一个局部值，并返回这个值），有的则可以处理函数（通过构造一个辅助table，来将函数与它的名称关联起来）等。Lua赋予了构建这些机制的能力。

?

元表（metatable）与元方法（meatmethod）

通常，Lua中的每个值都有一套预定义的操作集合。例如，可以将数字相加，可以连接字符串，还可以在table中插入一对key-value等。但是我们无法将两个table相加，无法对函数作比较，也无法调用一个字符串。

可以通过元表来修改一个值的行为，使其在面对一个非预定义的操作时执行一个指定的操作。例如，假设a和b都是table，通过元表可以定义如何计算表达式a+b。当Lua试图将两个table相加时，它会先检查两者之一是否有元表，然后检查该元表中是否有一个叫__add的字段。如果Lua找到了该字段，就调用该字段对应的值。这个值也就是所谓的“元方法”，它应该是一个函数，在本例中，这个函数用于计算table的和。

Lua中的每个值都有一个元表。table和userdata可以有各自独立的元表，而其他类型的值则共享其类型所属的单一元表。**Lua**在创建新的**table**时不会创建元表：

```
1.      t = {}
2.      print(getmetatable(t))        --> nil
```

可以使用setmetatable来设置或修改任何table的元表：

```
1.      t1 = {}
2.      setmetatable(t, t1)
3.      assert(getmetatable(t) == t1)
```

任何table都可以作为任何值的元表，而一组相关的table也可以共享一个通用的元表，此元表描述了它们的共同的行为。一个table甚至可以作为它自己的元表，用于描述其特有的行为。总之，任何搭配形式都是合法的。

在**Lua**代码中，只能设置**table**的元表。若要设置其他类型的值的元表，则必须通过C代码来完成。在第20章中，将会看到标准的字符串程序库为所有的字符串都设置了一个元表，而其他类型在默认情况中都没有元表。

```
1.      print(getmetatable("hi")         --> table:0x80772e0
2.      print(getmetatable(10))          --> nil
```

算术类的元方法

在本节中，会引入一个简单的示例，以说明如何使用元表。假设用table来表示集合，并且有一些

函数用来计算集合的并集和交集等。为了保持名称空间的整齐，则将这些函数存入一个名为Set的table中。

```
    Set = {}

    -- 根据参数列表中的值创建一个新的集合
    function Set.new(l)
        local set = {}
        for _, v in ipairs(l) do set[v] = true end
        return set
    end

    function Set.union(a, b)
        local res = Set.new{}
        for k in pairs(a) do res[k] = true end
        for k in pairs(b) do res[k] = true end
        return res
    end

    function Set.intersection(a, b)
        local res = Set.new{}
        for k in pairs(a) do
            res[k] = b[k]
        end
        return res
    end
```

为了帮助检查此示例，还定义了一个用于打印集合的函数：

```
    function Set.tostring(set)
        local l = {}        -- 用于存放集合中所有元素的列表
        for e in pairs(set) do
            l[#l + 1] = e
        end
        return "{" .. table.concat(l, ", " .. "}"
    end

    function Set.print(s)
        print(Set.tostring(s))
    end
```

假设使用加号（ `+` ）来计算两个集合的并集，那么就需要让所有用于表示集合的table共享一个

元表，并且在该元表中定义如何执行一个加法操作。第一步是创建一个常规的table，准备用作集合的元表：

```
local mt = {}        -- 集合的元表
```

下一步是修改Set.new函数。这个函数是用于创建集合的，在新版本中只加了一行，即将mt设置为当前所创建table的元表：

```
function Set.new(l)        -- 第2版
    local set = {}
    setmetatable(set, mt)
    for _, v in ipairs(l) do set[v] = true end
    return set
end
```

在此之后，所有由Set.new创建的集合都具有一个相同的元表：

```
s1 = Set.new{10, 20, 30, 50}
s2 = Set.new{30, 1}
print(getmetatable(s1))    --> table: 00672B60
print(getmetatable(s2))    --> table: 00672B60
```

最后，将元方法加入元表中。在本例中，这个元方法就是用于描述如何完成加法的 `__add` 字段。

```
mt.__add = Set.union
```

此后只要Lua试图将两个集合相加，它就会调用Set.union函数，并将两个操作数作为参数传入。可以使用加号来求集合的并集：

```
s3 = s1 + s2
Set.print(s3)        -->  {1, 10, 20, 30, 50}
```

类似地，还可以使用乘号来求集合的交集：

```
mt.__mul = Set.intersection
Set.print((s1 + s2)*s1)    --> {10, 20, 30, 50}
```

在元表中，每种算术操作符都有对应的字段名。除了上述的 **`add`** 和 `mul` 外，还有 **`sub`** （减法）、 `div` （除法）、 **`unm`** （相反数）、 `mod` （取模）和 **`pow`** （乘幂）。此外，还可以定义 `concat` 字段，用于描述连接操作符的行为。

当两个集合相加时，可以使用任意一个集合的元表。然而，当一个表达式中混合了具有不同元表的值时，例如：

```
1.      s = Set.new{1, 2, 3}
2.      s = s + 8
```

Lua会按照如下步骤来查找元表：如果第一个值有元表，并且元表中有 `add` 字段，那么**Lua**就以这个字段为元方法，而与第二个值无关；反之，如果第二个值有元表并含有 `add` 字段，Lua就以此字段为元方法；如果两个值都没有元方法，Lua就引发一个错误。因此，上例会调用Set.union，而表达式10+s和“hello”+s也是一样的。

Lua可以包含这些混合类型，但实现需要注意如果执行了s=s+8，那么在Set.union内部就会发生错误：

```
1.      bad argument $1 to 'pairs' (table expected, got number)
```

如果想要得到更清楚的错误消息，则必须在实际操作前显式地检查操作数的类型：

```
1.      function Set.union(a, b)
2.          if getmetatable(a) ~= mt or getmetatable(b) ~= mt then
3.              error("attempt to 'add' a set with a non-set value", 2)
4.          end
5.          <与前例相同的内容>
```

注意，error的第二个参数（上例中的2）用于指示哪个函数调用造成了该错误消息。

关系类的元方法

元表还可以指定关系操作符的含义，元方法为 `eq`（等于）、 `lt`（小于）和 `__le`（小于等于）。而其他3个关系操作符则没有单独的元方法，Lua会将 `a~=b` 转化为 `not(a==b)`，将 `a>b` 转化为 `b<a`，将 `a>=b` 转化为 `b<=a`。

在Lua4.0之前，所有的顺序操作符都被转化为一种操作符（小于），例如， `a<=b` 转化为 `not(b<a)`。不过，这种转化遇到“部分有序（partial order）”就会发生错误。所谓“部分有序”是指，对于一种类型而言，并不是所有的值都能排序的。例如，大多数计算机中的浮点数就不是完全可以排序的。因为存在着一种叫“Not a Number(NaN)”的值。IEEE754是一份当前所有浮点数硬件都采用的事实标准，其中将NaN视为一种未定义的值，例如0/0的结果就是NaN。标准规定了任何涉及NaN的比较都应返回false（假）。这意味着 `NaN<=x` 永远为假，但是 `x<NaN` 也为假。因此，前面提到的将 `a<=b` 转化为 `not(b<a)` 就不合法了。

在上面的集合示例中，也存在着类似的问题。在集合操作中 `<=` 通常表示集合间的包含关系： `a<=b` 通常意味着a是b的一个子集。根据这样的表示，仍有可能得到 `a<=b` 和 `b<a` 同时为假的情况。因此需要分别为 `le` （小于等于）和 `lt` （小于）提供实现：

```
mt.__le = function(a, b)        -- 集合包含
              for k in pairs(a) do
                  if not b[k] then return false end
              end
              return true
          end

    mt.__lt = function(a, b)
                  return a<=b and not (b<=a)
              end
```

最后，还可以定义集合的相等性判断：

```
    mt.__eq = function(a, b)
                  return a <= b and b <= a
              end
```

有了这些定义后，就可以比较集合了：

```
    s1 = Set.new{2, 4}
    s2 = Set.new{4, 10, 2}
    print(s1 <= s2)             -- true
    print(s1 < s2)             -- true
    print(s1 >= s1)             -- true
    print(s1 > s1)             -- false
    print(s1 == s2 * s1)    -- true
```

与算术类的元方法不同的是，关系类的元方法不能应用于混合的类型。对于混合类型而言，关系类元方法的行为就模拟这些操作符在Lua中普通的行为。如果试图将一个字符串与一个数字作顺序性比较，Lua会引发一个错误。同样，如果试图比较两个具有不同元方法的对象，Lua也会引发一个错误。

等于比较永远不会引发错误。但是如果两个对象拥有不同的元方法，那么等于操作不会调用任何一个元方法，而是直接返回false。这种行为模拟了Lua的普通行为。在Lua的普通行为中，字符串总是不等于数字的，与它们的值无关。另外，只有当两个比较对象共享一个元方法时，Lua才调用这个等于比较的元方法。

库定义的元方法

各种程序库在元表中定义它们自己的字段是很普通的方法。到目前为止介绍的所有元方法都只针对于Lua的核心，也就是一个虚拟机（virtual machine）。它会检测一个操作中的值是否有元表，这些元表中是否定义了关于此操作的元方法。从另一方面说，由于元表也是一种常规的table，所以任何人、任何函数都可以使用它们。

函数 `tostring` 就是一个典型的实例。在前面已介绍过 `tostring` 了，它能将各种类型的值表示为一种简单的文本格式：

```
print({})            --> table: 0x8062ac0
```

函数 `print` 总是调用 `tostring` 来格式化其输出。当格式化任意值时， `tostring` 会检查该值是否有一个 `__tostring` 的元方法。如果有这个元方法， `tostring` 就用该值作为参数来调用这个元方法。接下来由这个元方法完成实现的工作，它返回的结果也就是 `tostring` 的结果。

在集合的示例中，已定义了一个将集合表示为字符串的函数。接下来要做的就是设置元表的 `__tostring` 字段：

```
mt.__tostring = Set.tostring
```

此后只要调用 `print` 来打印集合， `print` 就会调用 `tostring` 函数，进而调用到 `Set.tostring` ：

```
s1 = Set.new{10, 4, 5}
print(s1)                --> {4, 5, 10}
```

函数 `setmetatable` 和 `getmetatable` 也会用到元表中的一个字段，用于保护元表。假设想要保护集合的元表，使用户既不能看也不能修改集合的元表。那么就需要用到字段 `__metatable` 。当设置了该字段时， `getmetatable` 就会返回这个字段的值，而 `setmetatable` 则会引发一个错误：

```
mt.__metatable = "not your business"

s1 = Set.new{}
print(getmetatable(s1))        --> not your business
setmetatable(s1, {})
stdin:1: cannot change protected metatable
```

table访问的元方法

算术类和关系类元算符的元方法都为各种错误情况定义了行为，它们不会改变语言的常规行为。但是Lua还提供了一种可以改变table行为的方法。有两种可以改变的table行为：查询table及修改table中不存在的字段。

- __index元方法

当访问一个table中不存在的字段时，得到的结果为nil。这是对的，但并非完全正确。实际上，这些访问会促使解释器去查找一个叫__index的元方法。如果没有这个元方法，那么访问结果如前述的为nil。否则，就由这个元方法来提供最终结果。

下面将介绍一个有关继承的典型示例。假设要创建一些描述窗口的table，每个table中必须描述一些窗口参数，例如位置、大小及主题颜色等。所有这些参数都有默认值，因此希望在创建窗口对象时可以仅指定那些不同于默认值的参数。第一种方法是使用一个构造式，在其中填写那些不存在的字段。第二种方法是让新窗口从一个原型窗口处继承所有不存在的字段。首先，声明一个原型和一个构造函数，构造函数创建新的窗口，并使它们共享同一个元表：

```
1.      Window = {}          -- 创建一个名字空间
2.      -- 使用默认值来创建一个原型
3.      Window.prototype = {x=0, y=0, width=100, height=100}
4.      Window.mt = {}          -- 创建元表
5.      -- 声明构造函数
6.      function Window.new(o)
7.          setmetatable(o, Window.mt)
8.          return o
9.      end
```

现在，来定义 `__index` 元方法：

```
1.      Window.mt.__index = function(table, key)
2.          return Window.prototype[key]
3.      end
```

在这段代码之后，创建一个新窗口，并查询一个它没有的字段：

```
1.      w = Window.new{x=10, y=20}
2.      print(w.width)            --> 100
```

若Lua检测到 `w` 中没有某字段，但在其元表中却有一个 `index` 字段，那么**Lua**就会以 `w(table)` 和“ `width` ”（不存在的**key**）来调用这个 `index` 元方法。随后元方法用这个key来索引原型table，并返回结果。

在Lua中，将 `index` 元方法用于继承是很普通的方法，因此**Lua**还提供了一种更便捷的方式来实

现此功能。 `__index` 元方法不必一定是一个函数，它还可以是一个table。当它是一个函数时，Lua以table和不存在的key作为参数来调用该函数，这就如同上述内容。而当它是一个table时，Lua就以相同的方式来重新访问这个table。因此，前例中 `__index` 的声明可以简单地写为：

```
1.      Window.mt.__index = Window.prototype
```

现在，当Lua查找到元表的 **`__index`** 字段时，发现 `__index` 字段的值是一个table，那么Lua就会在Window.prototype中继续查找。也就是说，Lua会在这个table中重复这个访问过程，类似于执行这样的代码：

```
1.      Window.prototype["width"]
```

然后由这次访问给出想要的结果。

将一个table作为 **`__index`** 元方法是一种快捷的、实现单一继承的方式。虽然将函数作为 `__index` 来实现相同功能的开销较大，但函数更加灵活。可以通过函数来实现多重继承、缓存及其他一些功能。

如果不想在访问一个table时涉及到它的 `__index` 元方法，可以使用函数 `rawget` 。调用 `rawget(t, i)` 就是对table t进行了一个“原始的(raw)”访问，也就是一次不考虑元表的简单访问。一次原始访问并不会加速代码执行，但有时会用到它。

- __newindex元方法

`__newindex` 元方法与 **`__index`** 类似，不同之处在于前者用于**table**的更新，而后者用于**table**的查询。当对一个table中不存在的索引赋值时，解释器就会查找 `__newindex` 元方法。如果有这个元方法，解释器就调用它，而不是执行赋值。如果这个元方法是一个table，解释器就在此table中执行赋值，而不是对原来的table。此外，还有一个原始函数允许绕过元方法：调用 `rawset(t,k,v)` 就可以不涉及任何元方法而直接设置table t中与key k相关联的value v。

组合使用 **`__index`** 和 `__newindex` 元方法就可以实现出Lua中的一些强大功能，例如，只读的table、具有默认值的table和面向对象编程中的继承。

- 具有默认值的table

常规table中的任何字段默认都是nil。通过元表就可以很容易地修改这个默认值：

```
1.      function setDefault(t, d)
2.          local mt = {__index = function() return d end}
3.          setmetatable(t, mt)
4.      end
```

```
5.
6.      tab = {x=10, y=20}
7.      print(tab.x, tab.z)           --> 10 nil
8.      setDefault(tab, 0)
9.      print(tab.x, tab.z)           --> 10 0
```

在调用 `setDefault` 后，任何对tab中存在字段的访问都将调用它的 `__index` 元方法，而这个元方法会返回0（这个元方法中d的值）。

`setDefault` 函数为所有需要默认值的table创建了一个新的元表。如果准备创建很多需要默认值的table，这种方法的开销或许就比较大了。由于在元表中默认值d是与元方法关联在一起的，所以 `setDefault` 无法为所有table都使用同一个元表。若要让具有不同默认值的table都使用同一个元表，那么就需要将每个元表的默认值都存放到table本身中。可以使用额外的字段来保持默认值。如果不担心名字冲突的话，可以使用“ `_` ”这样的key作为这个额外的字段：

```
1.      local mt = {__index = function(t) return t.___ end}
2.      function setDefault(t, d)
3.          t.___ = d
4.          setmetatable(t, mt)
5.      end
```

如果担心名称冲突，那么要确保这个特殊key的唯一性也很容易。只需创建一个新的table，并用它作为key即可：

```
1.      local key = {}              -- 唯一的key
2.      local mt = {__index = function(t) return t[key] end}
3.      function setDefault(t, d)
4.          t[key] = d
5.          setmetatable(t, mt)
6.      end
```

还有一种方法可以将table与其默认值关联起来：使用一个独立的table，它的key为各种table，value就是各种table的默认值。不过，为了正确地实现这种做法，我们还需要一种特殊性质的table，就是“ `弱引用table(Weak Table)` ”。在这里我们就不使用它了。将在后续章节中详细讨论。

- 跟踪table的访问

`index` 和 `newindex` 都是在table中没有所需访问的index时才发挥作用的。因此，只有将一个table保持为空，才有可能捕捉到所有对它的访问。为了监视一个table的所有访问，就应该为

真正的table创建一个代理。这个代理就是一个空的table，其中 `__index` 和 `__newindex` 元方法可用于跟踪所有的访问，并将访问重定向到原来的table上。假设，我们想跟踪table t的访问。那么可以这么做：

```
t = {}                  -- 原来的table(在其他地方创建的)

-- 保持对原table的一个私有访问
local _t = t

-- 创建代理
t = {}

-- 创建元表
local mt = {
    __index = function(t, k)
        print("*access to element " .. tostring(k))
        return _t[k]        -- 访问原来的table
    end,

    __newindex = function(t, k, v)
        print("*update of element " .. tostring(k) .. " to " ..
tostring(v))
        _t[k] = v           -- 更新原来的table
}
setmetatable(t, mt)
```

这段代码跟踪了所有对t的访问：

```
>t[2] = "hello"
*update of element 2 to hello
>print(t[2])
*access to element 2
hello
```

但上例中的方法存在一个问题，就是无法遍历原来的table。函数pairs只能操作代理table，而无法访问原来的table。

如果想要同时监视几个table，无须为每个table创建不同的元表。相反，只要以某种形式将每个代理与原来table关联起来，并且所有代理都共享一个公共的元表。这个问题与上节所讨论的将table与其默认值相关联的问题类似。例如将原来的table保存在代理table的一个特殊的字段中。代码如下：

```lua
    local index = {}            -- 创建私有索引

    local mt = {
        __index = function(t, k)
            print("*access to element " .. tostring(k))
            return t[index][k]    -- 访问原来的table
        end,

        __newindex = function(t, k, v)
            print("*update of element " .. tostring(k) .. " to " ..
tostring(v))
            t[index][k] = v     -- 更新原来的table
        end
    }

    function track(t)
        local proxy = {}
        proxy[index] = t
        setmetatable(proxy, mt)
        return proxy
    end
```

现在，若要监视table t，唯一要做的就是执行: `t = track(t)` 。

- 只读的table

通过代理的概念，可以很容易地实现出只读的table。只需跟踪所有对table的更新操作，并引发一个错误就可以了。由于无须跟踪查询访问，所以对于 `index` 元方法可以直接使用原**table**来代替函数。这也更简单，并且在重定向所有查询到原**table**时效率也更高。不过，这种做法要求为每个只读代理创建一个新的元表，其中 `index` 指向原来的table。

```lua
    function readOnly(t)
        local proxy = {}
        local mt = {              -- 创建元表
            __index = t,
            __nexindex = function(t, k, v)
                error("attempt to update a read-only table", 2)
            end
        }
        setmetatable(proxy, mt)
        return proxy
```

```
11.     end
```

下面是一个使用的示例，创建了一个表示星期的只读table:

```
1.      days = readOnly{"Sunday", "Monday", "Tuesday", "Wednesday", "Thursday",
    "Friday", "Saturday"}
2.
3.      print(days[1])               -- Sunday
4.      days[2] = "Noday"
5.      stdin:1: attempt to update a read-only table
```

?

环境

Lua将其所有的全局变量保存在一个常规的table中，这个table称为“环境（environment）”。这种组织结构的优点在于，其一，不需要再为全局变量创造一种新的数据结构，因此简化了Lua的内部实现。另一个优点是，可以像其他table一样操作这个table。为了便于实施这种操作，Lua将环境table自身保存在一个全局变量 `_G` 中。例如，以下代码打印了当前环境中所有全局变量的名称：

```
1.      for n in pairs(_G) do print(n) end
```

在本章中，将看到几种关于环境操作的实用技术。

具有动态名字的全局变量

对于访问和设置全局变量，通常赋值操作就可以了。不过，有时也会用到一些元编程（meta-programming）的形式。例如，当操作一个全局变量时，而它的名称却存储在另一个变量中，或者需要通过运行时的计算才能得到。为了获取这个变量的值，许多程序员都试图写出这样的代码：

```
1.      value = loadstring("return " .. varname)()
```

如果varname是x，那么连接操作的结果就是字符串“ `return x` ”。这段代码就执行了这个字符串，并得到了x的值。然而，在这段代码中包含了一个新程序块的创建和编译。因此可以使用以下代码来完成相同的效果，但效率却比上例高出一个数量级：

```
1.      value = _G[varname]
```

正因为环境是一个常规的table，才可以使用一个key（变量名）去直接索引它。类似地，还可以动态地计算出一个名称，然后将一个值赋予具有该名称的全局变量：

```
1.      _G[varname] = value
```

不过注意，有些程序员对于该技能的运用就有些过度了，他们写出的 `_G["a"] = _G["var1"]` ，其实就是简单的一句 `a=var1` 。

上面问题的一般化形式是，允许使用动态的字段名，如“ `io.read` ”或“ `a.b.c.d` ”。如果直接写 `_G["io.read"]` 则不会从 `table io` 中得到字段 `read` 。但可以写一个函数 `getfield` 来实现这个效果，即通过调用 `getfield("io.read")` 返回所要求的结果。这个函数是一个循环，从 `_G` 开始逐个字段地深入求值：

```
function getfield(f)
    local v = _G        -- 从全局变量的table开始
    for w in string.gmatch(f, "[%w_]+") do
        v = v[w]
    end
    return v
end
```

依靠 `string` 库中的 `gmatch` 来遍历f中所有的单词。

与之对应的设置字段的函数则稍显复杂。像 `a.b.c.d = v` 这样的赋值等价于以下代码：

```
local temp = a.b.c
temp.d = v
```

也就是说，必须一直检索到最后一个名称，然后分别进行操作。下面这个函数 `setfield` 就完成了这项任务，并且创建路径中间那些不存在的 `table` 。

```
function setfield(f, v)
    local t = _G            -- 从全局变量的table开始
    for w, d in string.gmatch(f, "([%w_]+)(%.?)" do
        if d == "." then    -- 是最后一个字段吗？
            t[w] = t[w] or {}    -- 如果不存在就创建table
            t = t[w]        -- 获取该table
        else                -- 最后的字段
            t[w] = v        -- 完成赋值
        end
    end
end
```

上例中用到了一种字符串模式（pattern），通过这种模式就可以将字段名捕获到变量w中，并将一个可选的句号捕获到d中。通过调用上面这个函数：

```
setfield("t.x.y", 10)
```

便创建了两个 `table` ：全局 `t` 和 `t.x` ，并将10赋予 `t.x.y` ：

```
print(t.x.y)                --> 10
print(getfield("t.x.y"))    --> 10
```

全局变量声明

Lua中的全局变量不需要声明就可以使用。对于小型程序来说较为方便，但在大型程序中，一处简单的笔误就有可能造成难以发现的错误。不过这种性能可以改变。由于Lua将全局变量存放在一个普通的 `table` 中，则可以通过元表来改变其访问全局变量时的行为。

一种方法是简单地检测所有对全局 `table` 中不存在 `key` 的访问：

```
1.      setmetatable(_G, {
2.          __newindex = function(_, n)
3.              error("attemp to write to undeclared variable " .. n, 2)
4.          end,
5.          __index = function(_, n)
6.              error("attempt to read undeclared variable " .. n, 2)
7.          end
8.      })
```

执行过这段代码后，所有对全局 `table` 中不存在 `key` 的访问都将引发一个错误：

```
1.      >print(a)
2.      stdin:1: attempt to read undeclared variable a
```

但是该如何声明一个新的变量呢？其一是使用rawset，它可以绕过元表：

```
1.      function declare(name, initval)
2.          rawset(_G, name, initval or false)
3.      end
```

另外一种更简单的方法就是只允许在主程序块中对全局变量进行赋值，那么当声明以下变量时：

```
1.      a = 1
```

就只需检查此赋值是否在主程序块中。这可以使用 `debug` 库，调用 `debug.getinfo(2, "S")` 将返回一个 `table` ，其中的字段 `what` 表示了调用元方法的函数是主程序块还是普通的Lua函数，又或是C函数。可以通过该函数将 `__newindex` 元方法重写为：

```
1.      __newindex = function(t, n, v)
2.          local w = debug.getinfo(2, "S").what
3.          if w ~= "main" and w ~= "C" then
4.              error("attempt to write to undeclared variable " .. n, 2)
5.          end
```

```
        rawset(t, n, v)
    end
```

这个新版本还可以接受来自C代码的赋值，因为一般C代码都知道自己是做什么的。

为了测试一个变量是否存在，就不能简单地将它与 `nil` 比较。因为如果它为 `nil` ，访问就会抛出一个错误。这时同样可以使用 `rawget` 来绕过元方法：

```
    if rawget(_G, var) == nil then
        -- 'var'没有声明
        ...
    end
```

正如前面提到的，不允许全局变量具有 `nil` 值，因为具有 `nil` 的全局变量都会被自动地认为是未声明的。但要纠正这个问题并不难，只需引入一个辅助 `table` 用于保存已声明变量的名称。一旦调用了元方法，元方法就检查该 `table` ，以确定变量是否已声明过，代码如下所示：

```
    local declaredNames = {}

    setmetatable(_G, {
        __newindex = function(t, n, v)
            if not declaredNames[n] then
                local w = debug.getinfo(2, "S").what
                if w ~= "main" and w ~= "C" then
                    error("attempt to write to undeclared variable " .. n, 2)
                end
                declaredNames[n] = true
            end
            rawset(t, n, v)                    -- 完成实际的设置
        end,

        __index = function(_n, n)
            if not declaredNames[n] then
                error("attempt to read undeclared variable " .. n, 2)
            else
                return nil
            end
        end,
    })
```

此时，即使是 `x = nil` 这样的赋值也可以起到声明全局变量的作用。　　上述两种方法所导致的开销基本可以忽略不计。在第一种方法中，完全没有涉及到元方法的调用。第二种方法虽然会使程序调

用到元方法，但只有当程序访问一个为 `nil` 的变量时才会发生。

有些Lua发行版本中包含一个叫 `strick.lua` 的模块，它实现了对全局变量的检查。究其本质就是使用了上述的技术。推荐在编写Lua代码时使用它，可以养成良好的习惯。

非全局的环境

关于“环境”的一大问题在于它是全局的，任何对它的修改都会影响程序的所有部分。例如，若安装一个元表用于控制全局变量的访问，那么整个程序都必须遵循这个规范。当使用某个库时，没有先声明就使用了全局变量，那么这个程序就无法运行了。

Lua5对这个问题进行了改进，它允许每个函数拥有一个自己的环境来查找全局变量。第一次听到这项机制可能会感觉不理解，毕竟一个用于记录全局变量的 `table` 本身也应该是全局的。在后续章节会看到几种基于该机制的技术，它们能使全局变量访问任何地方。

可以通过函数 `setfenv` 来改变一个函数的环境。该函数的参数是一个函数和一个新的环境 `table` 。第一个参数除了可以指定为函数本身，还可以指定为一个数字，以表示当前函数调用栈中的层数。数字1表示当前函数，数字2表示调用当前函数的函数，以此类推。

第一次天真地试用 `setfenv` 可能会带来糟糕的结果，如下代码：

```
1.      a = 1                          -- 创建一个全局变量
2.      -- 将当前环境改为一个新的空table
3.      setfenv(1, {})
4.      print(a)
```

会导致：

```
1.      setin:5: attempt to call global 'print' (a nil value)
```

一旦改变了环境，所有的全局访问就都会使用新的 `table` 。如果新 `table` 是空的，那么就会丢失所有的全局变量，包括 `_G` 。所以应该先将一些有用的值录入其中，例如原来的环境：

```
1.      a = 1                        -- 创建一个全局变量
2.      setfenv(1, {g = _G})     -- 改变当前的环境
3.      g.print(a)                    --> nil
4.      g.print(g.a)                --> 1
```

此时访问“全局的” `g` 就会得到原来的环境，这个环境中包含了字段 `print` 。 可以使用名字 `_G` 来代替 `g` ，从而重写前例：

```
setfenv(1, {_G = _G})
_G.print(a)                --> nil
_G.print(_G.a)             --> 1
```

对于Lua来说， `_G` 只是一个普通的名字。当Lua创建最初的全局 `table` 时，只是将这个 `table` 赋予了全局变量 `_G` ，Lua不会在意这个变量 `_G` 的当前值。 `setfenv` 不会在新环境中设置这个变量。但如果希望在新环境中引用最初的全局 `table` ，一般使用 `_G` 这个名称即可，如上例。

另一种组装新环境的方法是使用继承：

```
a = 1
local newgt = {}           -- 创建新环境
setmetatable(newgt, {__index = _G})
setfenv(1, newgt)           -- 设置它
print(a)                   --> 1
```

在这段代码中，新环境从原环境继承了 `print` 和 `a` 。然而，任何赋值都发生在新的 `table` 中。若误改了一个全局变量也没什么，仍然能通过 `_G` 来修改原来的全局变量：

```
-- 继续前面的代码
a = 10
print(a)                   --> 10
print(_G.a)                   --> 1
_G.a = 20
print(_G.a)                   --> 20
```

每个函数及某些 `closure` 都有一个继承的环境。下面这段代码就演示了这种机制：

```
function factory()
    return function()
        return a            -- "全局的"a
    end
end

a = 3

f1 = factory()
f2 = factory()

print(f1())                   --> 3
```

```
13.     print(f2())                  --> 3
14.
15.     setfenv(f1, {a = 10})
16.     print(f1())                  --> 10
17.     print(f2())                  --> 3
```

`factory` 函数创建了一个简单的 `closure` ，这个 `closure` 返回了它的全局 `a` 的值。每次调用 `factory` 都会创建一个新的 `closure` 和一个属于该 `closure` 的环境。每个新创建的函数都继承了创建它的函数的环境。因此，上例中的 `closure` 都共享一个全局环境。在这个环境中 `a` 为3，当调用 `setfenv(f1, {a=10})` 时，就改变了f1的环境，在新环境中a为10。这期间f2的环境并未受到影响。

由于函数继承了创建其函数的环境。所以一个程序块若改变了它自己的环境，那么后续由它创建的函数都将共享这个新环境。这项机制对于创建名称空间是很有用的。

?

模块与包

通常，Lua不会设置规则（policy）。相反，Lua会提供许多强有力的机制来使开发者有能力实现出最适合的规则。然而，这种方法对于模块就不可行了。模块系统的一个主要目标是允许以不同的形式来共享代码。但若没有一项公共的规则就无法实现这样的共享。

Lua从5.1开始，为模块和包（`package`）定义了一系列的规则。这些规则不需要语言引入额外的技能，程序员可以使用他们早已熟知的`table`、函数、元表和环境来实现这些规则。然而，有两个重要的函数可以很容易通过这些规则，它们是`require`（用于使用模块）和`module`（用于创建模块）。程序员完全可以使用不同的规则来重新实现这两个函数。但是，新的实现可能会使程序无法使用外部模块，或者编写的模块无法被外部程序所使用。

从用户观点来看，一个模块就是一个程序库，可以通过`require`来加载。然后便得到了一个全局变量，表示一个`table`。这个`table`就像是一个名称空间，其内容就是模块中导出的所有东西，例如函数和常量。一个规范的模块还应使`require`返回这个`table`。

使用table来实现模块的优点在于，可以像操作普通table那样来操作模块，并且能利用Lua现有的功能来实现各种额外的功能。在大多数语言中，模块不是“`第一类值（first-class value）`”，所以那些语言需要为模块实现一套专门的机制。在Lua中，可以轻易地实现所有这些功能。

例如，一个用户要调用一个模块中的函数。其中最简单的方法是：

```
require "mod"
mod.foo()
```

如果希望使用较短的模块名称，则可以为模块设置一个局部名称：

```
local m = require "mod"
m.foo()
```

还可以为个别函数提供不同的名称：

```
require "mod"
local f = mod.foo
f()
```

上述这些方法，都不需要来自于语言的显式支持，只需使用语言现有的内容。

require函数

Lua提供了一个名为 `require` 的高层函数用来加载模块，但这个函数只假设了关于模块的基本概念。对于 `require` 而言，一个模块就是一段定义了一些值的代码。

要加载一个模块，只需简单地调用 `require "<模块名>"` 。该调用会返回一个由模块函数组成的 `table` ，并且还会定义一个包含该 `table` 的全局变量。然而，这些行为都是由模块完成的，而非 `require` 。所以，有些模块会选择返回其他值，或者具有其他的效果。

即使知道某些用到的模块可能已加载了，但只要用到 `require` 就是一个良好的编程习惯。可以将标准库排除在此规则之外，因为Lua总是会预先加载它们。不过，有些用户还是喜欢为标准库中的模块使用显式的 `require` ：

```
1.      local m = require "io"
2.      m.write("hello world\n")
```

以下代码详细说明了 `require` 的行为：

```
1.      function require(name)
2.          if not packag.loaded[name] then         -- 模块是否已加载？
3.              local loader = findloader(name)
4.              if loader == nil then
5.                  error("unable to load module " .. name)
6.              end
7.              package.loaded[name] = true         -- 将模块标记为已加载
8.              local res = loader(name)            -- 初始化模块
9.              if res ~= nil then
10.                 package.loaded[name] = res
11.             end
12.         end
13.         return package.loaded[name]
14.     end
```

首先，它在 `table package.loaded` 中检查模块是否已加载。如果是的话， `require` 就返回相应的值。因此，只要一个模块已加载过，后续的 `require` 调用都将返回同一个值，不会再次加载它。

如果模块尚未加载， `require` 就试着为该模块找一个加载器（ `loader` ），会先在 `table package.preload` 中查询传入的模块名。如果在其中找到了一个函数，就以该函数作为模块的加载器。通过这个 `preload table` ，就有了一种通用的方法来处理各种不同的情况。通常这个 `table` 中不会找到有关指定模块的条目，那么 `require` 就会尝试从Lua文件或C程序库中加载模块。

如果 `require` 为指定模块找到了一个Lua文件，它就通过 `loadfile` 来加载该文件。而如果找

到的是一个C程序库，就通过 `loadlib` 来加载。注意， `loadfile` 和 `loadlib` 都只是加载了代码，并没有运行它们。为了运行代码， `require` 会以模块名作为参数来调用这些代码。如果加载器有返回值， `require` 就将这个返回值存储到 `table package.loaded` 中，以此作为将来对同一模块调用的返回值。如果加载器没有返回值， `require` 就会返回 `table package.loaded` 中的值。在本章后面会看到，一个模块还可以将返回给 `require` 的值直接放入 `package.loaded` 中。

上述代码中还有一个重要的细节，就是在调用加载器前， `require` 先将 `true` 赋予了 `package.loaded` 中的对应字段，以此将模块标记为已加载。这是因为如果一个模块要求加载另一个模块，而后者又要递归地加载前者。那么后者的 `require` 调用就会马上返回，从而避免了无限循环。

若要强制使 `require` 对用一个库加载两次的话，可以简单地删除 `package.loaded` 中的模块条目。例如，在成功地 `require "foo"` 后， `package.loaded["foo"]` 就不为 `nil` 了。下面代码就可以再次加载该模块：

```
1.      package.loaded["foo"] = nil
2.      require "foo"
```

在搜索一个文件时， `require` 所使用的路径与传统的路径有所不同。大部分程序所使用的路径就是一连串目录，指定了某个文件的具体位置。然而，ANSIC却没有任何关于目录的概念。所以， `require` 采用的路径是一连串的模式（ `pattern` ），其中每项都是一种将模块名转换为文件名的方式。进一步说，这种路径中的每项都是一个文件名，每项中还包含一个可选的问号。 `require` 会用模块名来替换每个“?”，然后根据替换的结果来检查是否存在这样一个文件。如果不存在，就会尝试下一项。路径中的每项以分号隔开。例如，假设路径为：

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

那么，调用 `require "sql"` 就会试着打开以下文件：

```
1.      sql
2.      sql.lua
3.      c:\windows\sql
4.      /usr/local/lua/sql/sql.lua
```

`require` 函数只处理了分号（作为各项之间的分隔符）和问号。其他例如目录分隔符或文件扩展名，都由路径自己定义。

`require` 用于搜索Lua文件的路径存放在变量 `package.path` 中。当Lua启动后，便以环境变量 `LUA_PATH` 的值来初始化这个变量。如果没有找到该环境变量，则使用一个编译时定义的默认路径来初始化。在使用 `LUA_PATH` 时，Lua会将其中所有的子串“;;”替换成默认路径。例如，假设 `LUA_PATH` 为“ `mydir/?.lua;;` ”，那么最终路径就是“ `mydir/?.lua` ”，并紧随默认路径。

如果 `require` 无法找到与模块名相符的Lua文件，它就会找C程序库。这类搜索会从变量 `package.cpath` （相对于 `package.path` ）获取路径。而这个变量则是通过环境变量 `LUA_CPATH` （相对于 `LUA_PATH` ）来初始化。在UNIX中，它的值一般是这样的：

```
1.      ./?.so;/usr/local/lib/lua/5.1/?.so
```

注意，文件的扩展名是由路径定义的（例如，上例中使用的.so）。而在Windows中，此路径通常可以是这样的：

```
1.      .\?.dll;C:\Program Files\Lua501\dll\?.dll
```

当找到一个C程序库后， `require` 就会通过 `package.loadlib` 来加载它， `loadlib` 在前面章节中已讨论过。C程序库与Lua程序块是不同的，它没有定义一个单一的主函数，而是导出了几个C函数。具有良好行为的C程序库应该导出一个名为“ `luaopen_<模块名>` ”的函数。 `require` 会在链接完程序库后，尝试调用这个函数。将在后续章节中讨论如何编写C程序库。

一般通过模块的名称来使用它们。但有时必须将一个模块改名，以避免冲突。一种典型的情况是，在测试中需要加载同一模块的不同版本。对于一个Lua模块来说，其内部名称不是固定的，可以轻易地编辑它以改变其名称。但是却无法编辑一个二进制数据模块中 `luaopen*函数的名称` 。为了允许这种重命名， *`require`* 用到了一个小技巧：如果一个模块名中包含了连字符， *`require`* 就会用连字符后的内容来创建 *`luaopen`* *`*函数名`* 。例如，若一个模块名为 `a-b` ， `require` 就认为它的 `open` 函数名为 `luaopen_b` ，而不是 `luaopen_a-b` 。因此，如果要使用的两个模块名都为 `mod` ，那么可以将其中一个重命名为 `v1-mod` （或者 `-mod` ，或其他类似形式）。当调用 `m1 = require "v1-mod"` 时， `require` 会找到改名后的文件 `v1-mod` ，并将其中的函数 `luaopen_mod` 作为 `open` 函数。

编写模块的基本方法

在Lua中创建一个模块最简单的方法是：创建一个 `table` ，并将所有需要导出的函数放入其中，最后返回这个 `table` 。以下代码演示这种方法。注意，将 `inv` 声明为程序块的局部变量，就是将其定义成一个私有的名称。

```
1.      complex = {}
2.
3.      function complex.new(r, i) return {r=r, i=i} end
4.
5.      -- 定义一个常量'i'
6.      complex.i = complex.new(0, 1)
7.
```

```
    function complex.add(c1, c2)
        return complex.new(c1.r  c2.r, c1.i + c2.i)
    end

    function complex.sub(c1, c2)
        return complex.new(c1.r - c2.r, c1.i - c2.i)
    end

    function complex.mul(c1, c2)
        return complex.new(c1.r*c2.r - c1.i*c2.i,
                           c1.r*c2.i + c1.i*c2.r)
    end

    local function inv(c)
        local n = c.r^2 + c.i^2
        return complex.new(c.r/n, -c.i/n)
    end

    function complex.div(c1, c2)
        return complex.mul(c1, inv(c2))
    end

    return complex
```

上例中使用 `table` 编写模块时，没有提供与真正模块完全一致的功能性，首先，必须显式地将模块名放到每个函数定义中。其次，一个函数在调用同一模块中的另一个函数时，必须限定被调用函数的名称。可以使用一个固定的局部名称（例如 `M` ）来定义和调用模块内的函数，然后将这个局部名称赋予模块的最终名称。通过这种方法，可以将上例改写为：

```
    local M = {}
    complex = M                     -- 模块名

    M.i = {r=0, i=1}
    function M.new(r, i) return {r=r, i=i} end

    function M.add(c1, c2)
        return M.new(c1.r + c2.r, c1.i + c2.i)
    end
    <如前>
```

只要一个函数调用了同一模块中另一个函数（或者递归地调用自己），就仍需要一个前缀名称。但至少两个函数之间的连接不再需要依赖模块名，并且也只需在整个模块中的一处写出模块名。实际上，

可以完全避免写模块名，因为 `require` 会将模块名作为参数传给模块：

```
1.      local modname = ...
2.      local M = {}
3.      _G[modname] = M
4.
5.      M.i = {r=0, i=1}
6.      <如前>
```

经过这样的修改，若需要重命名一个模块，只需重命名并定义它的文件就可以了。

另一项小改进与结尾的 `return` 语句有关。若能将所有与模块相关的设置任务集中在模块开头，会更好。消除 `return` 语句的一种方法是，将模块 `table` 直接赋予 `package.loaded` ：

```
1.      local modname = ...
2.      local M = {}
3.      _G[modname] = M
4.      package.loaded[modname] = M
5.      <如前>
```

通过这样的赋值，就不需要在模块结尾返回 `M` 了。注意，如果一个模块无返回值的话， `require` 就会返回 `package.loaded[modname]` 的当前值。

使用环境

创建模块的基本方法的缺点在于，它要求程序员投入一些额外的关注。当访问同一模块中的其他公共实体时，必须限定其名称。并且，只要一个函数的状态从私有改为公有（或从公有改为私有），就必须修改调用。另外，在私有声明中也很容易忘记关键字 `local` 。

“函数环境”是一种有趣的技术，它能够解决所有上述创建模块时遇到的问题。基本想法就是让模块的主程序块有一个独占的环境。这样不仅它的所有函数都可共享这个 `table` ，而且它的所有全局变量也都记录在这个 `table` 中。还可以将所有公有函数声明为全局变量，这样它们就都自动地记录在一个独立的 `table` 中了。模块所要做的就是将这个 `table` 赋予模块名和 `package.loaded` 。以下代码片段演示了这种技术：

```
1.      local modname = ...
2.      local M = {}
3.      _G[modname] = M
4.      package.loaded[modname] = M
5.      setfenv(1, M)
```

此时，当声明函数 `add` 时，它就称为了 `complex.add` ：

```
1.      function add(c1, c2)
2.          return new(c1.r + c2.r, c1.i + c2.i)
3.      end
```

此外，在调用同一模块的其他函数时，也不再需要前缀。例如， `add` 会从其环境中得到 `new` ，也就是 `complex.new` 。

这种方法为模块提供了一种良好的支持，并且只引入了一点额外的工作。此时完全不需要前缀，并且调用一个导出的函数与调用一个私有函数没有什么区别。如果程序员忘记了写local关键字，那么也不会污染全局名称空间。只会将一个私有函数变成了公有而已。

还缺少什么？是的，那就是访问其他模块。当创建了一个空 `table M` 作为环境后，就无法访问前一个环境中全局变量了。以下提出几种重获访问的方法，每种方法各有其优缺点。

最简单的方法是继承，就像之前看到的那样：

```
1.      local modname = ...
2.      local M = {}
3.      _G[modname] = M
4.      package.loaded[modname] = M
5.      setmetatable(M, {__index = _G})
6.      setfenv(1, M)
```

必须先调用 `setmetatable` 再调用 `setfenv` ，因为通过这种方法，模块就能直接访问任何全局标识了，每次访问只需付出很小的开销。这种方法导致了一个后果，即从概念上说，此时的模块中包含了所有的全局变量。例如，某人可以通过你的模块来调用标准的正弦函数： `complex.math.sin(x)` 。

还有一种更便捷的方法来访问其他模块，即声明一个局部变量，用以保存对旧环境的访问：

```
1.      local modname = ...
2.      local M = {}
3.      _G[modname] = M
4.      package.loaded[modname] = M
5.      local _G = _G
6.      setfenv(1, M)
```

此时必须在所有全局变量的名称前加“ `_G.` ”。由于没有涉及到元方法，这种访问会比前面的方法略快。

一种更正规的方法是将那些需要用到的函数或模块声明为局部变量：

```
1.      -- 模块设置
2.      local modname = ...
3.      local M = {}
4.      _G[modname] = M
5.      package.loaded[modname] = M
6.
7.      -- 导入段：
8.      -- 声明这个模块从外界所需的所有东西
9.      local sqrt = math.sqrt
10.     local io = io
11.
12.     -- 在这句之后就不再需要外部访问了
13.     setfenv(1, M)
```

这种技术要求做更多的工作，但是它能清晰地说明模块的依赖性。同时，较之前面的两种方法，它的运行速度也更快。

module函数

读者或许注意到了，前面几个示例中的代码形式。它们都以相同的模式开始：

```
1.      local modname = ...
2.      local M = {}
3.      _G[modname] = M
4.      package.loaded[modname] = M
5.      <setup for external access>
6.      setfenv(1, M)
```

Lua5.1提供了一个新函数 `module` ，它囊括了以上这些功能。在开始编写一个模块时，可以直接用以下代码来取代前面的设置代码：

```
1.      module(...)
```

这句调用会创建一个新的 `table` ，并将其赋予适当的全局变量和 `loaded table` ，最后还会将这个 `table` 设为主程序块的环境。

默认情况下， `module` 不提供外部访问。必须在调用它前，为需要访问的外部函数或模块声明适当的局部变量。也可以通过继承来实现外部访问，只需在调用 `module` 时加一个选

项 `package.seeall` 。这个选项等价于以下代码：

```
setmetatable(M, {__index = _G})
```

因而只需这么做：

```
module(..., package.seeall)
```

在一个模块文件的开头有了这句调用后，后续所有的代码都可以像普通的Lua代码那样编写了。不需要限定模块名和外部名字，同样也不需要返回模块 `table` 。要做的只是加上这么一句调用。

`module` 函数还提供了一些额外的功能。虽然大部分模块不需要这些功能，但有些发行模块可能需要一些特殊处理（例如，一个模块中同时包含C函数和Lua函数）。 `module` 在创建模块 `table` 之前，会先检查 `package.loaded` 是否已包含了这个模块，或者是否已存在与模块同名的变量。如果 `module` 由此找到了这个 `table` ，它就会复用该 `table` 作为模块。也就是说，可以用 `module` 来打开一个已创建的模块。如果没有找到模块 `table` ， `module` 就会创建一个模块 `table` 。然后在这个 `table` 中设置一些预定义的变量，包括： `_M` ，包含了模块 `table` 自身（类似于 `_G` ）； `_NAME` ，包含了模块名（传给 `module` 的第一个参数）； `_PACKAGE` ，包含了包（ `package` ）的名称。

子模块与包

Lua支持具有层级性的模块名，可以用一个点来分隔名称中的层级。假设，一个模块名为 `mod.sub` ，那么它就是 `mod` 的一个子模块。因此，可以认为模块 `mod.sub` 会将其所有值都定义在 `table mod.sub` 中，也就是一个存储在 `table mod` 中且 `key` 为 `sub` 的 `table` 。一个“包（Package）”就是一个完整的模块树，它是Lua中发行的单位。

当 `require` 一个模块 `mod.sub` 时， `require` 会用原始的模块名“ `mod.sub` ”作为 `key` 来查询 `table package.loaded` 和 `package.preload` ，其中，模块名中的点在搜索中没有任何含义。

然而，当搜索一个定义子模块的文件时， `require` 会将点转换为另一个字符，通常就是系统的目录分隔符。转换之后 `require` 就像搜索其他名称一样来搜索这个名称。例如，假设路径为：

```
./?.lua;/usr/local/lua/?.lua;/usr/local/lua/?/init.lua
```

并且目录分隔符为“ `/` ”，那么调用 `require "a.b"` 就会尝试打开以下文件：

```
./a/b.lua
/usr/local/lua/a/b.lua
```

```
3.      /usr/local/lua/a/b/init.lua
```

通过这样的加载策略，就可以将一个包中的所有模块组织到一个目录中。例如，一个包中有模块 `p` 、 `p.a` 和 `p.b` ，那么它们对应的文件名就分别为 `p/init.lua` ， `p/a.lua` 和 `p/b.lua` ，它们都是目录 `p` 下的文件。

Lua使用的目录分隔符是编译时配置的，可以是任意的字符串。例如，在没有目录层级的系统中，就可以使用“ `_` ”作为“目录分隔符”。那么 `require "a.b"` 就会搜索到文件 `a_b.lua` 。

C函数名中不能包含点，因此一个用C编写的子模块 `a.b` 无法导出函数 `luaopen_a.b` 。所以， `require` 会将点转换为下划线。例如，一个名为 `a.b` 的C程序库就应将其初始化函数命名为 `luaopen_a_b` 。在此又可以巧用连字符，来实现一些特殊的效果。例如，有一个C程序库 `a` ，现在想将它作为 `mod` 的一个子模块，那么就可以将文件名改为 `mod/-a` 。当执行 `require "mod.-a"` 时， `require` 就会找到改名后的文件 `mod/-a` 及其中的函数 `luaopen_a` 。

作为一项扩展功能， `require` 在加载C子模块时还有一些选项。当 `require` 加载子模块时，无法找到对应的Lua文件或C程序库。它就会再次搜索C路径，不过这次将以包的名称来查找。例如，一个程序 `require` 子模块 `a.b.c` ，当无法找到文件 `a/b/c` 时，再次搜索就会找到文件 `a` 。如果找到了C程序库 `a` ， `require` 就查看该程序库中是否有 `open` 函数 `luaopen_a_b_c` 。这项功能使得一个发行包可以将几个子模块组织到一个单一C程序库中，并且具有各自的 `open` 函数。

`module` 函数也为子模块提供了显式的支持。当我们创建一个子模块时，调用 `module("a.b.c")` ， `module` 就会将环境 `table` 放入变量 `a.b.c` ，也就是“ `table a中的table b中的table c` ”。如果这些中间的 `table` 不存在， `module` 就会创建它们。否则，就复用它们。

从Lua的观点看，同一个包中的子模块除了它们的环境 `table` 是嵌套的之外，它们之间并没有显式的关联性。 `require` 模块 `a` 并不会自动地加载它的任何子模块。同样， `require` 子模块 `a.b` 也并不会自动地加载 `a` 。当然只要愿意，包的实现者完全可以实现这种关联。例如，模块 `a` 的一个子模块在加载时会显式地加载 `a` 。

?

面向对象编程

Lua中的 `table` 就是一种对象，这句话可以从3个方面来证实。首先， `table` 与对象一样可以拥有状态。其次， `table` 也与对象一样拥有一个独立于其值的标识（一个 `self` ）。例如，两个具有相同值的对象（ `table` ）是两个不同的对象。最后， `table` 与对象一样具有独立于创建者和创建地的生命周期。

对象有其自己的操作。同样 `table` 也有这样的操作：

```
1.      Account = {balance = 0}
2.      function Account.withdraw(v)
3.          Account.balance = Account.balance - v
4.      end
```

上面的代码创建了一个新函数，并将该函数存入 `Account` 对象的 `withdraw` 字段中。则可进行如下调用：

```
1.      Account.withdraw(100.00)
```

这种函数就是所谓的“方法（Method）”。不过，在函数中使用全局名称 `Account` 是一个不好的编程习惯。因为这个函数只能针对特定对对象工作，并且，这个特定对象还必须存储在特定的全局变量中。如果改变了对象的名称， `withdraw` 就再也不能工作了：

```
1.      a = Account;Account = nil
2.      a.withdraw(100.00)        -- 错误！
```

这种行为违反了前面提到的对象特性，即对象拥有独立的生命周期。

有一种灵活的方法，即指定一项操作所作用的“接受者”。因此需要一个额外的参数来表示该接受者。这个参数通常称为 `self` 或 `this` ：

```
1.      function Account.withdraw(self, v)
2.          self.balance = self.balance - v
3.      end
```

此时当调用该方法时，必须指定其作用的对象：

```
1.      a1 = Account; Account = nil
2.      ...
3.      a1.withdraw(a1, 100.00)        -- OK
```

通过对 `self` 参数的使用，还可以针对多个对象使用同样的方法：

```
a2 = {balance=0, withdraw=Account.withdraw}
...
a2.withdraw(a2, 260.00)
```

使用 `self` 参数是所有面向对象语言的一个核心。大多数面向对象语言都能对程序员隐藏部分 `self` 参数，从而使得程序员不必显式地声明这个参数。Lua只需使用冒号，则能隐藏该参数。即可将上例重写为：

```
function Account:withdraw(v)
    self.balance = self.balance - v
end
```

调用时可写为：

```
a:withdraw(100.00)
```

冒号的作用是在一个方法定义中添加一个额外的隐藏参数，以及在一个方法调用中添加一个额外的实参。冒号只是一种语法便利，并没有引入任何新的东西。例如，用点语法来定义一个函数，并用冒号语法调用它。反之，只要能正确地处理那个额外参数即可：

```
Account = {balance=0,
           withdraw=function(self, v)
                       self.balance = self.balance - v
                    end
          }

function Account:deposit(v)
    self.balance = self.balance + v
end

Account.deposit(Account, 200.00)
Account:withdraw(100.00)
```

现在的对象已有一个标识、一个状态和状态之上的操作。不过还缺乏一个类（class）系统、继承和私密性（privacy）。首先解决第一个问题，如何创建多个具有类似行为的对象？更准确地说，如何创建多个account账户对象？

类

一个类就像是一个创建对象的模具。有些面向对象语言提供了类的概念，在这些语言中每个对象都是某个特定类的实例。Lua则没有类的概念，每个对象只能自定义行为和形态。不过，要在Lua中模拟类也并不困难，可以参照一些基于原型的语言，例如 `Self` 和 `NewtonScript` 。在这些语言中，对象是没有“类型”的（objects have no classes）。而是每个对象都有一个原型（prototype）。原型也是一种常规的对象，当其他对象（类的实例）遇到一个未知操作时，原型会先查找它。在这种语言中要表示一个类，只需创建一个专用作其他对象（类的实例）的原型。类和原型都是一种组织对象间共享行为的方式。

在Lua中实现原型很简单，使用后续章节所述的继承即可。更准确地说，如果有两个对象a和b，要让b作为a的一个原型，只需输入如下语句：

```
1.      setmetatable(a, {__index = b})
```

在此之后，a就会在b中查找所有它没有的操作。若将b称为是对象a的类，只不过是术语上的一个变化。

回到先前银行账号的示例。为了创建更多与 `Account` 行为类似的账号，可以让这些新对象从 `Account` 行为中继承这些操作。具体做法就是使用 `__index` 元方法。可以应用一项小优化，则无须创建一个额外的 `table` 作为账户对象的元表。而是使用 `Account table` 自身作为元表：

```
1.      function Account:new(o)
2.          o = o or {}         -- 如果用户没有提供table, 则创建一个
3.          setmetatable(o, self)
4.          self.__index = self
5.          return o
6.      end
```

当调用 `Account:new` 时， `self` 就等于 `Account` 。因此可以直接使用 `Account` 来代替 `self` 。不过，当引入类继承时，使用 `self` 则会更为准确。在这段代码之后，创建一个新账户或调用一个方法时会发生什么呢？

```
1.      a = Account:new{balance = 0}
2.      a:deposit(100.00)
```

当创建新账户时，a会将 `Account` （ `Account:new` 调用中的 `self`)作为其元表。而当调用 `a:deposit(100.00)` 时，就是调用了 `a.deposit(a, 100.00)` 。因此冒号只不过是一个“语法糖”。当Lua无法在 `table a` 中找到条目“ `deposit` ”时，它会进一步搜索元表的 `__index` 条目。最终的调用情况为：

```
1.      getmetatable(a).__index.deposit(a, 100.00)
```

`a` 的元表是 `Account` ， `Account.__index` 也是 `Account` 。因此，上面这个表达式可以简化为：

```
1.      Account.deposit(a, 100.00)
```

结果为Lua调用了原来的 `deposit` 函数，但传入 `a` 作为 `self` 参数。因此新账户 `a` 从 `Account` 继承了 `deposit` 函数。同样，它还能从 `Account` 继承所有的字段。

继承不仅可以作用于方法，还可以作用于所有其他在新账户中没有的字段。因此，一个类不仅可以提供方法，还可以为实例中的字段提供默认值。回忆一下，在第一个 `Account` 定义中，有一个 `balance` 字段为0。如果在创建新账户时没有提供 `balance` 的初值，那么它就会继承这个默认值：

```
1.      b = Account:new()
2.      print(b.balance)            --> 0
```

在 `b` 上调用 `deposit` 方法时， `self` 就是 `b` ，就相当于执行了：

```
1.      b.balance = b.balance + v
```

在第一次调用 `deposit` 时，对表达式 `b.balance` 的求值结果为0，然后一个初值被赋予了 `b.balance` 。后续对 `b.balance` 的访问就不会再涉及到 `__index` 元方法了，因为此时 `b` 已有自己的 `balance` 字段。

继承

由于类也是对象，它们也可以从其他类获得方法。这种行为就是一种继承，可以很容易地在Lua中。

假设有一个基类 `Account` ：

```
1.      Account = {}
2.
3.      function Account:new(o)
4.          o = o or {}
5.          setmetatable(o, self)
6.          self.__index = self
7.          return o
```

```
    end

    function Account:deposit(v)
        self.balance = self.balance + v
    end

    function Account:withdraw(v)
        if v > self.balance then error "insufficient funds" end
        self.balance = self.balance - v
    end
```

若想从这个类派生出一个子类 `SpecialAccount` ，以使客户能够透支。则先需要创建一个空的类，从基类继承所有的操作：

```
    SpecialAccount = Account:new()
```

直到现在， `SpecialAccount` 还只是 `Account` 的一个实例。如下所示：

```
    s = SpecialAccount:new{limit=1000.00}
```

`SpecialAccount` 从 `Account` 继承了 `new` ，就像继承其他方法一样。不过这次 `new` 在执行时，它的 `self` 参数表示为 `SpecialAccount` 。因此， `s` 的元表为 `SpecialAccount` ， `SpecialAccount` 中字段 `__index` 的值也是 `SpecialAccount` 。 `s` 继承自 `SpecialAccount` ，而 `SpecialAccount` 又继承自 `Account` 。当执行：

```
    s:deposit(100.00)
```

Lua在 `s` 中找不到 `deposit` 字段时，就会查找 `SpecialAccount` 。如果仍找不到 `deposit` 字段，就查找 `Account` 。最终会在那里找到 `deposit` 的原始实现。

`SpecialAccount` 之所以特殊是因为可以重定义那些从基类继承的方法。编写一个方法的新实现只需：

```
    function SpecialAccount:withdraw(v)
        if v - self.balance >= self:getLimit() then
            error "insufficient funds"
        end
        self.balane = self.balance - v
    end

```

```
function SpecialAccount:getLimit()
    return self.limit or 0
end
```

现在，当调用 `s:withdraw(200.00)` 时，Lua就不会在 `Account` 中查找了。因为Lua会在 `SpecialAccount` 中先找到 `withdraw` 方法。由于 `s.limit` 为1000.00，程序会执行取款，并使 `s` 变成一个负的余额。

Lua中的对象有一个特殊现象，就是无须为指定一种新行为而创建一个新类。如果只有一个对象需要某种特殊的行为，那么可以直接在该对象中实现这个行为。例如，账户 `s` 表示一个特殊的客户，这个客户的透支额度总是其余额的10%。那么可以只修改这个对象：

```
function s:getLimit()
    return self.balance * 0.10
end
```

在这段代码后，调用 `s:withdraw(200.00)` 还是会执行 `SpecialAccount` 的 `withdraw` 。但 `withdraw` 所调用的 `self:getLimit` 则是上面这个定义。

多重继承

由于Lua中的对象不是原生的（Primitive），因此在Lua中进行面向对象编程时有几种方法。上面介绍了一种使用 `__index` 元方法的做法，这是集简易、性能和灵活性于一体的做法。另外还有一些其他的做法，可能更适用于某些特殊的情况。在此将介绍另一种做法，可以在Lua中实现多重继承。

这种做法的关键在于同一个函数作为 **`index`** 元字段。例如，若在一个 **`table`** 的元表中， `index` 字段为一个函数。那么只要Lua在原来的 `table` 中找不到一个 `key` ，就会调用这个函数。基于这点，就可以让 `__index` 函数在其他地方查找缺失的 `key` 。

多重继承意味着一个类可以具有多个基类。因此无法使用一个类中的方法来创建子类，而是需要定义一个特殊的函数来创建。下面的 `createClass` 就是这样的函数，它会创建一个 `table` 表示新类，其中一个参数表示新类的所有基类。创建时它会设置元表中的 **`index`** 元方法，而多重继承正是在这个 `index` 元方法中完成的。虽然是多重继承，但每个对象实例仍属于单个类，并且都在这个类中查找所有的方法。因此，类和基类之间的关系不同于类和实例之间的关系。尤其是一个类不能同时作为其实例和子类的元表。在以下代码中，将类作为其实例的元表，并创建了另一个 `table` 作为类的元表。

```
-- 在table 'plist'中查找'k'
local function search(k, plist)
    for i=1, #plist do
```

```
            local v = plist[i][k]         -- 尝试第i个基类
            if v then return v end
        end
    end

    function createClass(...)
        local c = {}                      -- 新类
        local parents = {...}

        -- 类在其父类列表中的搜索方法
        setmetatable(c, {__index = function(t, k)
            return search(k, parents)
        end})

        -- 将'c'作为其实例的元表
        c.__index = c

        -- 为这个新类定义一个新的构造函数（construction）
        function c:new(o)
            o = o or {}
            setmetatable(o, c)
            return o
        end

        return c                  -- 返回新类
    end
```

接下来是一个使用 `createClass` 的例子。假设有两个类，一个是前面提到的 `Account` 类；另一个是 `Named` 类，它有两个方法 `setname` 和 `getname` ：

```
    Named = {}
    function Named:getname()
        return self.name
    end

    function Named:setname(n)
        self.name = n
    end
```

要创建一个新类 `NamedAccount` ，同时从 `Account` 和 `Named` 派生，那么只需调用 `createClass` ：

```
NamedAccount = createClass(Account, Named)
```

如下要创建并使用实例：

```
account = NamedAccount:new{name = "Paul"}
print(account:getname())        --> Paul
```

现在，来研究最后代码是如何工作的。首先，Lua在 `account` 中无法找到字段“ `getname` ”。因此，就查找 `account` 元表中的 **`index`** 字段，该字段为 **`NamedAccount`** 。由于在 **`NamedAccount`** 也无法提供字段“ **`getname`** ”。因此，**Lua**查找 **`NamedAccount`** 元表中的 `index` 字段。由于这个字段也是一个函数，Lua就调用了它。该函数则先在 `Account` 中查找“ `getname` ”。未找到后，继而查找 `Named` 。最终在 `Named` 中找到了一个非 `nil` 的值，即为搜索的最终结果。

由于这项搜索具有一定的复杂性，则多重继承的性能不如单一继承。有一种改进性能的简单做法是将继承的方法复制到子类中。通过这种技术，类的 `__index` 元方法如下所示：

```
setmetatable(c, {__index = function(t, k)
    local v = search(k, parents)
    t[k] = v                  -- 保存下来，以备下次访问
    return v
end})
```

用了这种技术后，访问继承的方法就能像访问局部方法一样快了。但缺点是当系统运行后就较难修改方法的定义，因为这些修改不会沿着继承体系向下传播。

私密性

许多人认为私密性应成为面向对象语言不可或缺的一部分，每个对象的状态都应该是由它自己掌握。在一些面向对象语言中，例如 `C++` 和 `Java` ，能控制对象中的字段或方法是否在对象之外可见。而对于其他语言，例如 `Smalltalk` ，规定所有的变量都是私有的，但所有的方法却都是公有的。第一个面向对象语言 `Simula` 则不提供任何形式的私密性保护。

Lua在设计对象时，没有提供私密性机制，这具体章节已看到了。一方面这是因为使用了普通的结构（ `table` ）来表示对象，另一方面也反映了Lua某些基本的设计决定。Lua并不打算构建需要许多程序员长期投入的大型程序。相反，Lua定位于开发中小型程序，这些程序通常是一个更大系统的一部分。而参与编程的程序员一般只有一名或几名，甚至还可以是非程序员。因此，Lua尽量避免过多冗余和人为限制。如果不想访问一个对象中的内容，则无须进行操作。

Lua的另外一项设计目标是灵活性。Lua提供给程序员各种元机制，以使他们能模拟许多不同的机

制。虽然在Lua对象的基础设计中没有提供私密性机制。但可以用其他方法来实现对象，从而获得对象的访问限制。这种实现不常用，只做基本的了解，它既探索了Lua中的某些知识又可以成为其他问题的解决方案。

这种做法的基本思想是，通过两个 `table` 来表示一个对象。一个 `table` 用来保存对象的状态；另一个用于对象的操作，或称为“接口”。对象本身是通过第二个 `table` 来访问的，即通过其接口的方法来访问。为了避免未授权的访问，表示状态的 `table` 不保存在其他 `table` 中，而只是保存在方法的 `closure` 中。例如，若使用这种设计来表示一个银行账户，可以调用下面这个工厂函数来创建新的账户对象：

```
1.      function newAccount(initialBalance)
2.          local self = {balance = initialBalance}
3.
4.          local withdraw = function(v)
5.              self.balance = self.balance - v
6.          end
7.
8.          local deposit = function(v)
9.              self.balance = self.balance + v
10.         end
11.
12.         local getBalance = function() return self.balance end
13.
14.         return {
15.             withdraw = withdraw,
16.             deposit = deposit,
17.             getBalance = getBalance
18.         }
19.     end
```

这个函数先创建了一个 `table` ，用于保存对象的内部状态，并将其存储在局部变量 `self` 中。然后再创建对象的方法。最后，函数创建并返回一个供外部使用的对象，其中将方法名与真正的方法实现匹配起来。区别关键在于，这些方法不需要额外的 `self` 参数，因为它们可以直接访问 `self` 变量。由于没有了额外的参数，也就无须使用冒号语法来操作对象。则可以像普通函数那样来调用这些方法：

```
1.      acc1 = newAccount(100.00)
2.      acc1.withdraw(40.00)
3.      print(acc1.getBalance())        --> 60
```

这种设计给予存储在 `self table` 中所有东西完全的私密性。当 `newAccount` 返回后，就无法

直接访问这个 `table` 了。只能通过 `newAccount` 中创建的函数来访问它。上例只将一个成员变量放到了私有 `table` 中，其实可以将一个对象中所有的私有部分都存入这个 `table` 。另外还可以定义私有的方法，它们类似于公有方法，但不放入接口中。例如，该账户可以给那些余额大于某个值的用户额外10%的信用额度，但是又不想让用户访问到这些计算细节。那么可以将这个功能按以下方法实现：

```
    function newAccount(initialBalance)
        local self = {
            balance = initialBalance,
            LIM = 10000.00
        }

        local extra = function()
            if self.balance > self.LIM then
                return self.balance * 0.10
            else
                return 0
            end
        end

        local getBalance = function()
            return self.balance + extra()
        end

        <如前>
```

与前一个示例一样，任何用户都无法直接访问 `extra` 函数。

单一方法（single-method）做法

上述面向对象编程的做法有一种特殊情况，就是当一个对象只有一个方法时，可以不用创建接口 `table` ，但要将这个单独的方法作为对象表示来返回。如果无法理解，请参阅前面章节。前面章节介绍了如何构造一个迭代器函数，那个函数将状态保存为 `closure` 。一个具有状态的迭代器是一个单一方法对象。

单一方法对象还有一种情况，若这个方法是一个调度（ `dispatch` ）方法，它根据某个参数来完成不同的操作。则可以这样来实现一个对象：

```
    function newObject(value)
        return function(action, v)
            if action == "get" then return value
```

```
            elseif action == "set" then value = v
            else error "invalid action"
            end
        end
    end
```

如下所示：

```
    d = newObject(0)
    print(d("get"))                --> 0
    d("set", 10)
    print(d("get"))                --> 10
```

这种非传统的对象实现方式是很高效的。语句 `d("set", 10)` 虽然有些奇特，但只比传统的 `d:set(10)` 多出两个字符。每个对象都用一个 `closure` ，这比都用一个 `table` 更高效。虽然无法实现继承，却拥有了完全的私密性控制。访问一个对象状态只有一个方式，那就是通过它的单一方法。

Tcl/Tk对它的窗口部件使用了类似的做法。在Tk中，一个窗口部件的名称就是一个函数名，通过这个函数就可以完成所有针对该部件的操作。

?

弱引用table

Lua采用了自动内存管理。一个程序只需创建对象，而无须删除对象。通过使用垃圾收集机制，Lua会自动地删除那些已成为垃圾的对象。这减轻了程序员在内存管理方面的负担，更重要的是将程序员从许多内存相关的bug（例如无效指针、内存泄漏）中解放出来。

Lua的垃圾收集器与一些其他的收集器有所不同，它没有环形引用的问题。当用到环形数据结构时，无须作出任何特殊的处理，它们也可以像其他数据一样被正常回收。不过，有时即使是再聪明的收集器也需要帮助。垃圾收集器无法解决所有内存管理的问题。

垃圾收集器只能回收那些它认为是垃圾的东西，它不会回收那些用户认为是垃圾的东西。一个典型的例子就是栈，栈通常由一个数组和一个表示顶部的索引来实现。这个数组的有效部分总是向顶部扩展的，但Lua却不知道。如果弹出一个元素时只是简单地递减顶部索引，那么这个仍留在数组中的对象对于Lua来说就不是垃圾。同理，对于那些存储在全局变量中的对象，即使程序不会再用到它们，但对于Lua来说就不是垃圾。在这两种情况中，都需要由用户来将这些对象变量赋值为 `nil` 。这样才能使它们得以释放。

不过，简单地清楚引用可能还不够。有些情况需要程序和收集器之间进行更多的协作。例如，如果要将一些对象放在一个数组中，这看似简单，好像只需把每个对象插入数组即可。但是，当一个对象处于数组中时，它就无法被回收。这是因为即使当前没有其他地方在使用它，但数组仍引用着它。除非用户告诉Lua这项引用不应该阻碍此对象的回收，否则，Lua是无从得知这个事实的。

弱引用 `table` （ `weak table` ）就是这样一种机制，用户能用它来告诉Lua一个引用不应该阻碍一个对象的回收。所谓“弱引用（ `weak reference` ）”就是一种会被垃圾收集器忽视的对象引用。如果一个对象的所有引用都是弱引用，那么Lua就可以回收这个对象了，并且还可以以某种形式来删除这些弱引用本身。Lua用“ `弱引用table` ”来实现“弱引用”，一个弱引用 `table` 就是一个具有弱引用条目的 `table` 。如果一个对象只被一个弱引用 `table` 所持有，那么最终Lua是会回收这个对象的。

`table` 中有 `key` 和 `value` ，这两者都可以包含任意类型的对象。通常，垃圾收集器不会回收一个可访问 `table` 中作为 `key` 或 `value` 的对象。也就是说，这些 `key` 和 `value` 都是强引用（ `strong reference` ），它们会阻止对其所引用对象的回收。在一个弱引用 `table` 中， `key` 和 `value` 是可以回收的。有3种弱引用 `table` ：具有弱引用 `key` 的 `table` 、具有弱引用 `value` 的 `table` 、同时具有两种弱引用 `table` 。不论是哪种类型的弱引用 `table` ，只要有一个 `key` 或 `value` 被回收了，那么它们所在的整个条目都会从 `table` 中删除。

一个table的弱引用类型是通过其元表中的 `__mode` 字段来决定的。这个字段的值应为一个字符串，如果这个字符串中包含字母‘ `k` ’，那么这个table的key是弱引用的；如果这个字符串中包含字母‘ `v` ’，那么这个 `table` 的 `value` 是弱引用的。下面这个示例虽然是人为制造的，但演示了弱

引用 `table` 的一些基本行为：

```
a = {}
b = {__mode = 'k'}
setmetatable(a, b)     -- 现在'a'的key就是弱引用
key = {}               -- 创建第一个key
a[key] = 1
key = {}               -- 创建第二个key
a[key] = 2
collectgarbage()       -- 强制进行一次垃圾收集
for k, v in pairs(a) do print(v) end
--> 2
```

在本例中，第二句赋值 `key = {}` 会覆盖第一个 `key` 。当收集器运行时，由于没有其他地方在引用第一个 `key` ，因此第一个 `key` 就被回收了，并且 `table` 中的相应条目也被删除了。至于第二个 `key` ，变量 `key` 仍引用着它，因此它没有被回收。

注意，Lua只会回收弱引用 `table` 中的对象。而像数字和布尔这样的“值”是不可回收的。例如，对于一个插入 `table` 的数字 `key` ，收集器是永远不会删除它的。当然，如果一个数字 `key` 所对应的 `value` 被回收了，那么整个条目都会从这个弱引用 `table` 中删除。

字符串在此则显得有些特殊。虽然从实现的角度看，字符串是可回收的。但在有些方面，字符串却与其他可回收的对象不同。其他对象，例如 `table` 和函数都是显式创建的。又如，当Lua对表达式 `{}` 求值时，它就会创建一个新的 `table` 。同样地，求值 `function()…end` 时就会创建一个新函数。然而，当Lua对 `"a".."b"` 求值时，它会创建一个新字符串吗？如果当前系统中已有了一个字符串"ab"，它会复用吗？还是创建一个新的字符串？编译器会在运行程序前先创建这个字符串吗？这些都无关紧要，它们都是实现的细节。从程序员的角度看，字符串就是值，而非对象。因此，字符串就像数字和布尔一样，不会从弱引用 `table` 中删除。

备忘录（memoize）函数

一项通用的编程技术是“用空间换时间”。例如有一种做法就可以提高一些函数的运行速度，记录下函数计算的结果，然后当使用同样的参数再次调用该函数时，便可以复用之前的结果了。

假设有一个普通的服务器，在它收到的请求中包含Lua代码。每当服务器收到一个请求，它就要对代码字符串调用 `loadstring` ，然后再调用编译好的函数。不过， `loadstring` 是一个昂贵的函数，而有些发给服务器的命令具有很高的频率，例如“ `closeconnection()` ”。与其每次收到一条常见命令就调用 `loadstring` ，还不如让服务器用一个辅助的 `table` 记录下所有调用 `loadstring` 的结果。因此，在每次调用 `loadstring` 前，服务器先检查 `table` 中是否已记录了代码字符串编译后的结果。如果没有，才调用 `loadstring` ，并将结果存储到 `table` 中。可以将

这个行为写成一个新函数：

```
local results = {}
funcition mem_loadstring(s)
    local res = results[s]
    if res == nil then        -- 是否已记录过？
        res = assert(loadstring(s))        -- 计算新结果
        results[s] = res        -- 存下以备之后复用
    end
    return res
end
```

这项优化节省的时间非常可观。但，它也可能导致不易察觉的开销。虽然有些命令会重复出现，但还有许多命令只发生一次。例如，`table results`会逐渐地积累服务器收到的所有命令及其编译结果。经过一定的时间后，这种累积会耗费服务器的内存。弱引用的`table`正好可以解决这个问题，如果`results table`具有弱引用的`value`，那么每次垃圾收集都会删除所有在执行时未使用的编译结果。

```
local results = {}
setmetatable(results, {__mode = 'v'})        -- 使用value称为弱引用
funcition mem_loadstring(s)
    <如前>
```

实际上，由于`key`总是字符串，则可以使这个`table`编程完全弱引用。若这么做：

```
setmetatable(results, {__mode = 'kv'})
```

则最终效果完全一样。

“备忘录”技术还可以用于确保某类对象的唯一性。假设一个系统用`table`来表示颜色，其中3个字段`red`、`green`和`blue`都具有相同的取值范围。最简单的颜色生成函数是：

```
function createRGB(r, g, b)
    return {red=r, green=g, blue=b}
end
```

通过备忘录技术，可以复用具有相同颜色的`table`。备忘录`table`的`key`可以根据颜色分量来生成，本例中是将颜色分量以分隔符连接起来：

```
local results = {}
setmetatable(result, {__mode='v'})        -- 使用value成为弱引用
```

```
    function createRGB(r, g, b)
        local key = r .. "-" .. g .. "-" .. b
        local color = results[key]
        if color == nil then
            color = {red=r, green=g, blue=b}
            results[key] = color
        end
        return color
    end
```

这种实现可以使用户通过原始的相等性操作符比较两种颜色。若两种同时存在的颜色相等，那么它们必定是由同一个 `table` 表示的。不过，相同的颜色也可能在不同时间由不同的 `table` 表示，这是因为期间执行过垃圾收集，清除了 `results table` 。只要一种颜色正在使用，就不会被清除出 `results` 。因此，只要一个颜色未被清除，它就可与新颜色进行比较，它的表示也可作为后续调用来复用。

对象属性

关于弱引用 `table` ，还有一项重要的应用是将属性与对象关联起来。有很多情况需要把有些属性绑定到某个对象，例如函数与其名称、 `table` 的默认值及数组的大小等。

当对象是一个 `table` 时，可以通过适当的 `key` 将属性存储在这个 `table` 中。正如先前所看到的，创建唯一性 `key` 的最简单办法是创建一个新对象（通常是一个 `table` ）。不过，若对象不是一个 `table` ，它就无法保存属性了。另外，即使是 `table` ，有时也不想将属性存储在原 `table` 中。例如，想保持属性的私有性，或者不想让属性扰乱 `table` 的遍历就需要用其他办法来关联属性与对象了。显然，使用外部 `table` 来关联它们是一种理想的做法。可以将对象作为 `key` ，对象的属性作为 `value` 。这个外部 `table` 可以保存任意对象的属性，Lua也允许将任何对象作为 `table` 的 `key` 。另外，存储在外部对象中的属性不会干扰其他对象，只要 `table` 本身是私有的，这些属性也会是私有的。

然而，这个看似完美的做法却有一个重大缺陷。当用户将一个对象作为外部 `table` 的 `key` 时，就是引用了它。Lua是无法回收一个作为 `table key` 的对象。如果用这个外部 `table` 来关联函数和函数名，那么这些函数就永远无法回收。用户可以使用弱引用 `table` 来解决这个问题。而本例需要的是弱引用 `key` 。当一个弱引用 `key` 没有其他引用时，Lua就可以回收它。注意，这个 `table` 不能使用弱引用 `value` ，否则“存留的”对象的属性就有可能被回收。

回顾table的默认值

前面章节讨论了如何实现具有非 `nil` 默认值的 `table` 。库定义的元方法一节中注明了还有两

种技术需要弱引用 `table` 的支持。这里将介绍两种用于默认值的技术，它们其实是上述备忘录和对象属性的特殊应用。

第一种做法是使用一个弱引用 `table` ，通过它将每个 `table` 与其默认值关联起来：

```
local defaults = {}
setmetatable(defaults, {__mode='k'})
local mt = {__index=function(t) return defaults[t] end}
function setDefault(t, d)
    defaults[t] = d
    setmetatable(t, mt)
end
```

如果 `defaults` 没有弱引用 `key` ，它就会使所有具有默认值的 `table` 持久存在下去。

第二种做法是对每种不同的默认值使用不同的元表。不过，只要有重复的默认值，就复用同样的元表。这是备忘录的典型应用：

```
local metas = {}
setmetatable(metas, {__mode = 'v'})
function setDefault(t, d)
    local mt = metas[d]
    if mt == nil then
        mt = {__index = function() return d end}
        metas[d] = mt             -- 备忘录
    end
    setmetatable(t, mt)
end
```

这里用到了弱引用 `value` ，这样当 `metas` 中的元表在不使用时就可以被回收了。

这两种默认值的实现，哪种更好呢？一般而言，它们具有类似的复杂度和性能表现。第一种做法需要为每个 `table` 的默认值（ `defaults` 中的一个条目）使用内存。而第二种做法则需要为每种不同的默认值使用一组内存（一个新 `table` ，一个新 `closure` 和 `metas` 中的一个条目）。因此，如果程序中有上千个 `table` 和一些默认值，第二种做法则是首选。但如果只有很少的 `table` 共享几个公用的默认值，那么就应该选择第一种做法。

?

数学库

从本章节开始将介绍标准程序库，这些章节中并不会给出每个函数的完整说明，而只说明标准库中提供了哪些功能。为了说明清楚，讲解过程中会回避某些微妙的选项和行为。用户可以在Lua参考手册中学习更多的知识。

`math`（数学）库由一组标准的数学函数构成，包括三角函数（`sin`、`cos`、`tan`、`asin`、`acos`等）、指数和对数函数（`exp`、`log`、`log10`）、取整函数（`floor`、`ceil`）、`max`和`min`、生成伪随机数的函数（`random`、`randomseed`）以及变量`pi`和`huge`。其中`huge`为Lua可以表示的最大数字。

所有的三角函数都使用弧度单位，可以用函数`deg`和`rad`来转换角度和弧度。如果使用角度单位，可以像这样重新定义三角函数：

```
1.      local sin,asin, ... = math.sin, math.asin, ...
2.      local deg, rad = math.deg, math.rad
3.      math.sin = function(x) return sin(rad(x)) end
4.      math.asin = function(x) return deg(asin(x)) end
5.      ...
```

函数`math.random`用于生成伪随机数，可以用3种方式来调用它。如果在调用它时不提供任何参数，它将返回一个在区间`[0,1)`内均匀分布的伪随机实数。如果提供了一个整数`n`作为参数，它将返回一个在区间`[1, n]`内的伪随机整数。例如，`random(6)`就可以用来模拟一次掷骰子的结果。最后一种方式是在调用它时提供两个整数参数`m`和`n`，这样会得到一个在区间`[m, n]`内的伪随机整数。

函数`randomseed`用于设置伪随机数生成器的种子数，它的唯一参数就是这个种子数。通常在一个程序启动时，用一个固定的种子数来调用它，以此初始化伪随机数生成器。这样每次程序运行时，都会生成相同的伪随机数序列。从调试的角度看，这是一个不错的特性。对于一个游戏来说，则每次都会得到相同的情景。对此通常的解决方法是使用当前时间作为种子数：

```
1.      math.randomseed(os.time())
```

函数`os.time`返回一个表示当前时间的数字，一般这个数字表示从某个时间点开始至今的秒数。

> `math.random`函数使用了C标准库中的`rand`函数，在某些C标准库的实现中，这个函数所产生的数字并不具备统计意义上的均匀分布特性。Lua的某些独立发行的版本中包含了更好的伪随机数生成器。

?

table库

`table` 库是由一些辅助函数构成的，这些函数将 `table` 作为数组来操作。其中，有用于在列表中插入和删除元素的函数，有对数组元素进行排序的函数，还有连接一个数组中所有字符串的函数。

插入和删除

函数table.insert用于将一个元素插入到一个数组的指定位置，它会移动后续元素以空出空间。例如，如果 `t` 是数组 `{10, 20, 30}` ，当调用 `table.insert(t, 1, 15)` 后， `t` 会变为 `{15, 10, 20, 30}` 。但有一种特殊情况，如果在调用 `insert` 时没有指定位置参数，则会将元素添加到数组末尾。下面这段代码逐行地读取了程序的输入，并将所有的行保存在一个数组中：

```
1.      t = {}
2.      for line in io.lines() do
3.          table.insert(t, line)
4.      end
5.      print(#t)              -->(读入的行数)
```

函数 `table.remove` 会删除（并返回）数组指定位置上的元素，并将该位置之后的所有元素前移，以填补空隙。如果在调用这个函数时不指定位置参数，它就会删除数组的最后一个元素。

有了这两个函数，就可以很容易地实现栈、队列和双向队列。可以用 `t={}` 来初始化这种结构， `table.insert(t, x)` 等价于压入操作， `table.remove(t)` 等价于弹出操作。 `table.insert(t,1,x)` 会在结构的另一端（也就是起始处）插入一个新元素，而 `table.remove(t,1)` 会从这一端删除元素。后两个操作不是很高效，因为它们必须移动元素。不过由于 `table` 库中的函数都是用C语言实现的，所以这些循环的花销并不高，对于较小的数组来说使用这种实现较好。

排序

另一个有用的数组函数是 `table.sort` ，我们在此之前已经用到过它。它可以对一个数组进行排序，还可以指定一个可选的次序函数。这个次序函数有两个参数，如果希望第一个参数在排序结果中位于第二个参数值前，就应当返回 `true` 。如果没有提供这个函数， `sort` 就使用默认的小于操作。

一个常见的错误是试图对一个 `table` 的索引进行排序。在 `table` 中，索引是一个无序的集合。如果对它们进行排序，则必须将它们复制到一个数组中，然后对这个数组进行排序。下面演示一个示例，假设要读取一个源文件，并构建一个 `table` 记录每个函数并定义它的行号：

```
    lines = {
        luaH_set = 10,
        luaH_get = 24,
        luaH_present = 48,
    }
```

现在要求按字母次序打印这些函数名。如果使用 `pairs` 来遍历这个 `table` ，就会发现遍历所得的名称是无序的。由于这些名称是 `table` 的 `key` ，因此不能对它们进行直接排序。但是，可以将它们放到一个数组中，然后对这个数组进行排序。正确的做法是先用这些名称创建一个数组，然后对数组排序，最后打印结果：

```
    a = {}
    for n in pairs(lines) do a[#a+1] = n end
    table.sort(a)
    for i,n in ipairs(a) do print(n) end
```

注意，对于Lua来说，数组也是无序的，它们本质上是 `table` 。然而由于用户知道如何计算索引，因此在访问数组时，只要使用有序的索引，就可以顺序地访问数组。这就是为什么必须用 `ipairs` 而不是 `pairs` 来遍历数组的原因。前者使 `key` 为 `1、2、……` 的顺序，而后者采用 `table` 的原始顺序。

另外还有一个更高级的方法，就是写一个迭代器，使它根据 `table key` 的次序来进行遍历。同时，还有一个可选参数 `f` ，用于指定某种特殊次序。以下函数先将 `key` 排序到一个数组中，然后迭代这个数组，且每步都返回原 `table` 中 `key` 和 `value` 。

```
    function pairsByKeys(t, f)
        local a = {}
        for n in pairs(t) do a[#a+1] = n end
        table.sort(a, f)
        local i = 0           -- 迭代器变量
        return function()     -- 迭代器函数
            i = i + 1
            return a[i], t[a[i]]
        end
    end
```

通过这个函数就可以很容易地按字母次序来打印那些函数名了：

```
    for name, line in pairsByKeys(line) do
        print(name, line)
    end
```

连接

在数据结构章节中已经看到过 `table.concat` 。它接受一个字符串数组，并返回这些字符串连接后的结果。它有一个可选参数，用于指定插到字符串之间的分隔符。这个函数另外还接受两个可选参数，用于指定第一个和最后一个要连接的字符串索引。

下面这个函数是 `table.concat` 的一个扩展，它能处理嵌套的字符串数组：

```
1.      function rconcat(l)
2.          if type(l) ~= "table" then return l end
3.          local res = {}
4.          for i=1, #l do
5.              res[i] = rconcat(l[i])
6.          end
7.          return table.concat(res)
8.      end
```

对于数组中的每个元素， `rconcat` 都递归地调用自己，以此来连接所有可能嵌套的字符串数组。最后，它调用 `table.concat` 来连接这些结果部分。

```
1.      print(rconcat{{"a", {" nice"}}, " and", {{" long"}, {" list"}}})
2.      --> a nice and long list
```

?

字符串库

```
1.      --[[ lua function
2.      -- 大小写转换
3.      --print(string.upper(msg))
4.      --print(string.lower(msg))
5.      --print(msg:upper())
6.      --print(msg:lower())
7.
8.      -- 重复字符串
9.      --print(string.rep(msg, 2))
10.     --print(msg:rep(2))
11.
12.     -- 截取子串
13.     print(string.sub(msg, 3, -1)) -- -1表示最后1位字符 -2代表倒数第2个字符
14.
15.     a = "[Hello World!]"
16.     a = a:sub(2, -2)  -- 字符串是不可变的
17.     print(a) -- 表示去掉第1个和最后1个字符
18.
19.     -- 字符与asc码转换
20.     print(string.char(97))        --> a
21.     i = 99;print(string.char(i, i+1, i+2))    --> cde
22.     print(string.byte("abc"))        --> 97
23.     print(string.byte("abc", 2))        --> 98
24.     print(string.byte("abc", -1))        --> 99
25.
26.     -- 格式化字符串
27.     print(string.format("%s, %d", "===", i))
28.
29.     -- 查找子串
30.     i, j = string.find(a, "Hello")  -- i,j分别为找到的起始索引与结束索引, 没找到返回
    nil
31.     print(a:sub(i, j))
32.
33.     -- 匹配查找
34.     data = "Today is 1/29/2018"
35.     -- print(data:match("%d+/%d+/%d+"))
36.
37.     -- 替换字符串
```

```
    print(data:gsub("Today", "YesToday"))  -- 返回值: 替换后的字符串 替换的次数

    count = select(2, data:gsub(" ", " ")) -- 统计字符串中空格的个数
    print(count)

    -- string.gmatch 会返回一个函数，通过这个函数可以遍历到所有出现指定模式的地方
    words = {}
    for w in string.gmatch(data, "%a+") do
        words[#words+1] = w
    end
    print("words : #", #words)
    ]]
```

原始的Lua解释器操作字符串的能力是很有限的。一个程序只能创建字符串常量、连接字符串及获取字符串的长度。它无法提取子串或者检索字符串的内容。在Lua中真正的字符串操作能力来源于字符串库。

字符串库中的所有函数都导出在模块 `string` 中。在Lua5.1中，它还将这些函数导出作为 `string` 类型的方法。这样，假设要将一个字符串转换到大写形式，可以写 `string.upper(s)` ，也可以写 `s:upper()` 。但是为了避免与Lua5.0不兼容，在本书的大多数示例中将采用基于模块的写法。

基础字符串函数

字符串库中有一些函数非常简单。函数 `string.len(s)` 可返回字符串 `s` 的长度。函数 `string.rep(s, n)` （或 `s:rep(n)` ）可返回字符串 `s` 重复 `n` 次的结果。例如，可以用 `string.rep("a", 2^20)` 来创建一个1MB的字符串。函数 `string.lower(s)` 可返回一份 `s` 的副本，其中所有的大写字母都被转换成小写形式，而其他字符则保持不变。 `string.upper` 与之相反，它将小写转换成大写。大小写转换函数有一个典型用途，假设要对一个字符串数组进行排序，并且不区分大小写，可以这样写：

```
    table.sort(a, function(a, b)
        return string.lower(a) < string.lower(b)
    end)
```

函数 `string.upper` 和 `string.lower` 都遵循当前的区域设置（local）。因此，如果在EuropeanLatin-1区域工作，那么表达式 `string.upper("acao")` 的结果就是“ACAO”。

函数 `string.sub(s,i,j)` 可以从字符串 `s` 中提取第 `i` 个到第 `j` 个字符。在Lua中，字符串的第一个字符的索引是1。还可以用负数索引，这样会从字符串的尾部开始计数，索引-1代表字符串

的最后一个字符，-2代表倒数第二个字符，以此类推。这样，调用函数 `string.sub(s,1,j)` 或 `s:sub(1,j)` ，就可以得到字符串 `s` 中长度为 `j` 的前缀。调用 `string.sub(s,j,-1)` 或 `s:sub(j)` ，就可以得到字符串中从第 `j` 个字符开始的一个后缀。调用 `string.sub(s,2,-2)` 可以返回去掉字符串 `s` 的第一个和最后一个字符后的复制。

```
1.      s = "[in brackets]"
2.      print(string.sub(s, 2, -2)        --> in brackets
```

记住，Lua中的字符串是不可变的。和Lua中的所有其他函数一样， `string.sub` 不会改变字符串的值，它只会返回一个新字符串。一种常见的错误是这么写：

```
1.      string.sub(s, 2, -2)      --
```

这里假定 `s` 的值就这样被改变了。如果要改变一个变量的值，就必须赋予它一个新的值：

```
1.      s = string.sub(s, 2, -2)
```

函数 `string.char` 和 `string.byte` 用于转换字符及其内部数值表示。 `string.char` 函数接受零个或多个整数，并将每个整数转换成对应的字符，然后返回一个由这些字符连接而成的字符串。 `string.byte(s,i)` 返回字符串 `s` 中第 `i` 个字符的内部数值表示，它的第二个参数是可选的，调用 `string.byte(s)` 可返回字符串s中第一个字符的内部数值表示。在下例中，假定字符是用ASCII表示的：

```
1.      print(string.char(97))        --> a
2.      i = 99;print(string.char(i, i+1, i+2))    --> cde
3.      print(string.byte("abc"))        --> 97
4.      print(string.byte("abc", 2))        --> 98
5.      print(string.byte("abc", -1))        --> 99
```

最后一行用了一个负数索引来访问字符串的最后一个字符。

在Lua5.1中， `string.byte` 还可以接受可选的第三个参数。调用 `string.byte(s,i,j)` 可以返回索引 `i` 到 `j` 之间（包括 `i` 和 `j` ）的所有字符的内部表示值。

```
1.      print(string.byte("abc", 1, 2))        --> 97 98
```

`j` 的默认值是 `i` ，因此在调用该函数时若不指定这个参数，那么就只返回第 `i` 个字符的值，这就与Lua5.0一样了。还有一种习惯写法是 `{s:byte(1,-1)}` ，这种写法会创建一个 `table` ，其中包含了 `s` 中所有字符的编码。有了这个 `table` ，就可以调用 `string.char(unpack(t))` 来重建原字符串。但是，由于Lua限制了一个函数的返回值数量，因此

这项技术无法应用于较长的字符串。

函数string.format是用于格式化字符串的利器，经常用在输出上。它会根据第一个参数的描述，返回后续其他参数的格式化版本，这第一个参数也称为“格式化字符串”。编写格式化字符串的规则，与标准C语言中printf等函数的规则基本相同：它由常规文本和指示（directive）组成，这些指示控制了每个参数应放到格式化结果的什么位置，及如何放入它们。一个指示由字符‘%’加上一个字母组成，这些字母指定了如何格式化参数，例如‘d’用于十进制数、‘x’用于十六进制数、‘o’用于八进制数、‘f’用于浮点数和‘s’用于字符串等。在字符‘%’和字母之间可以再指定一些其他选项，用于控制格式的细节。例如，指定一个浮点数中有几个十进制数字。

```
1.      print(string.format("pi = %.4f", math.pi))        --> pi = 3.1416
2.      d = 5; m = 11; y = 1990
3.      print(string.format("%02d/%02d/%04d", d, m, y))        --> 05/11/1990
4.      tag, title = "h1", "a title"
5.      print(string.format("<%s>%s<%s>", tag, title, tag))        --> <h1>a
    title<h1>
```

在第一次打印中，%.4f表示一个浮点数的小数点后有4位数字。在第二次打印中，%02d表示一个十进制数字至少有两个数字，如不足两个数字，则用0补足；而指示%2d则表示用空格来补足。关于这些指示的完整描述可以参数Lua参考手册。或者查阅C语言手册，因为Lua是通过调用C标准库来完成实际的工作。

模式匹配（pattern-matching）函数

字符串库中最强大的函数是 `find` 、 `match` 、 `gsub` （global substitution，全局替换）和 `gmatch` （global match，全局匹配），它们都是基于“模式（pattern）”的。

不同于其他脚本语言，Lua既没有使用POSIX（regexp），也没有使用Perl正则表达式来进行模式匹配。其原因主要是考虑到Lua的大小。一个典型的POSIX正则表达式实现需要超过4000行代码，这相当于所有Lua标准库加在一起的大小。而相比之下，Lua采用的模式匹配实现的代码只有500行不到。当然，Lua的模式匹配所能达到的功能不及完整的POSIX实现。但是，Lua的模式匹配仍是一个强大的工具，并且它还具有一些特性，能在进行某些匹配时，比标准POSIX实现更为方便。

- string.find函数

`string.find` 函数用于在一个给定的目标字符串中搜索一个模式。最简单的模式就是一个单词，它只会匹配与自己完全相同的拷贝。例如，模式“ `hello` ”会搜索目标字符串中的子串“ `hello` ”。当 `find` 找到一个模式后，它会返回两个值：匹配到的起始索引和结尾索引。如果没有找到任何匹配，它就返回 `nil` 。

```
s = "hello world"
i, j = string.find(s, "hello")
print(i, j)           --> 1 5
print(string.sub(s, i, j))    --> hello
print(string.fin(s, "world")    --> 7 11
i, j = string.find(s, "l")
print(i, j)           --> 3 3
print(string.find(s, "llll"))        --> nil
```

如果匹配成功，就可以用 `string.find` 的返回值来调用 `string.sub` ，以此提取出目标字符串中匹配于该模式的那部分子串。

`string.find` 函数还具有一个可选的第三个参数，它是一个索引，告诉函数应从目标字符串的哪个位置开始搜索。当处理所有与给定模式相匹配的部分时，这个参数就很有用。可以重复搜索新的匹配，且每次搜索都从上一次找到的位置开始。下面这个示例用字符串中所有换行符的位置创建了一个 `table` ：

```
local t = {}        -- 存储索引的table
local i = 0
while true do
    i = string.find(s, "\n", i+1)        -- 查找下一个换行符
    if i == nil then break end
    t[#t + 1] = i
end
```

接下来将介绍一种更简单的方法来编写这个循环，其中用到了 `string.gmatch` 迭代器。

- string.match函数

从某种意义上说，函数 `string.match` 与 `string.find` 非常相似，它也是用于在一个字符串中搜索一种模式。不同之处在于， `string.match` 返回的是目标字符串中与模式相匹配的那部分子串，而非该模式所在的位置。

```
print(string.match("hello world", "hello"))        --> hello
```

对于固定的模式，例如“ `hello` ”，使用这个函数就没有什么意义了。但当使用变量模式（ `Variable Pattern` ）时，它的特性就显现出来了，如下示例：

```
date = "Today is 17/7/1990"
d = string.match(date, "%d+/%d+/%d+")
```

```
3.      print(d)        --> 17/7/1990
```

在后面将会讨论模式“ `%d+/%d+/%d+` ”的含义及 `string.match` 的高级用法。

- string.gsub函数

`string.gsub` 有3个参数：目标字符串、模式和替换字符串。它的基本用法是将目标字符串中所有出现模式的地方替换为替换字符串（最后一个参数）：

```
1.      s = string.gsub("Lua is cute", "cute", "great")
2.      print(s)            --> Lua s great
3.      s = string.gsub("all lii", "l", "x")
4.      print(s)            --> axx xii
5.      s = string.gsub("Lua is great", "Sol", "Son")
6.      print(s)            --> Lua is great
```

另外还有可选的第四个参数，可以限制替换的次数：

```
1.      s = string.gsub("all lii", "l", "x", 1)
2.      print(s)            --> axl lii
3.      s = string.gsub("all lii", "l", "x", 2)
4.      print(s)            --> axx lii
```

函数 `string.gsub` 还有另一个结果，即实际替换的次数。例如，以下代码就是一种统计字符串中空格数量的简单方法：

```
1.      count = select(2, string.gsub(str, " ", " "))
```

- string.gmatch函数

`string.gmatch` 会返回一个函数，通过这个函数可以遍历到一个字符串中所有出现指定模式的地方。例如，以下示例找出了给定字符串 `s` 中所有的单词：

```
1.      words = {}
2.      for w in string.gmatch(s, "%a+") do
3.          words[#words + 1] = w
4.      end
```

其中模式“ `%a+` ”表示匹配一个或多个字母字符的序列（也就是单词）。在本例中， `for` 循环

会遍历目标字符串中所有的单词，并且将它们储存到列表 `words` 中。

通过 `gmatch` 和 `gsub` 可以模拟出Lua中的 `require` 在寻找模块时所用的搜索策略，如下：

```
1.      function search(modname, path)
2.          modname = string.gsub(modname, "%.", "/")
3.          for c in string.gmatch(path, "[^;]+") do
4.              local fname = string.gsub(c, "?", modname)
5.              local f = io.open(fname)
6.              if f then
7.                  f:close()
8.                  return fname
9.              end
10.         end
11.         return nil        -- 未找到
12.     end
```

首先，用点符号来替换所有的目录分隔符，示例假设目录分隔符未‘ `/` ’。接下去我们就会看到，在模式中“点”具有特殊的含义，因此若要表示一个点必须写为“ `%.` ”。其次，函数遍历路径中的所有组成部分，这里所谓的组成部分是指那些不包含分号的最长子串。对于每个组成部分，都用模块名来替换其中的问号，以此获得最终的文件名。最后，检查该文件是否存在。如果存在，则关闭该文件，并返回它的名称。

模式

可以用字符分类（character class）创建更多有用的模式。字符分类就是模式中的一项，可以与一个特定集合中的任意字符相匹配。例如，分类 `%d` 可匹配任意数字。如下例可以用模式“ `%d%d/%d%d/%d%d%d%d` ”搜索符合“ `dd/mm/yyyy` ”格式的日期：

```
1.      s = "Deadline is 30/05/1999, firm"
2.      date = "%d%d/%d%d/%d%d%d%d"
3.      print(string.sub(s, string.find(s, data)))        --> 30/05/1999
```

下表列出了所有的字符分类：

.	所有字符
%a	字母
%c	控制字符
%d	数字
%l	小写字母

%p	标点符号
%s	空白字符
%u	大写字母
%w	字母和数字字符
%x	十六进制数字
%z	内部表示为0的字符

这些分类的大写形式表示它们的补集，例如，“ `%A` ”表示所有非字母字符：

```
print(string.gsub("hello, up-down!", "%A", "."))
--> hello..up.down.4
```

注意，打印出的“4”不是结果字符串的一部分，而是 `gsub` 的第二个结果，即替换的次数。在后续打印 `gsub` 结果的示例里，将忽略这个计数结果。

在模式中还有一些字符被称为“魔法字符”，它们具有特殊的含义。这些魔法字符有：

```
( ) . % + - * ? [ ] ^ $
```

字符‘ `%` ’作为这些魔法字符的转义符。因此“ `%.` ”表示匹配一个点，“ `%%` ”表示匹配字符‘ `%` ’本身。不仅可以将‘ `%` ’用于魔法字符，还可以用于其他所有非字母和数字的字符。当不确定某个字符是否需要转义时，应该前置一个转义符。

对于Lua来说，模式就是普通的字符串，并像其他字符串一样遵循相同的规则。只有模式函数才会解释它们，此时才会将‘ `%` ’当作转义符来处理。在模式中放入一个引号的方法，与在普通字符串中放入引号的方法相同。也就是说，需要用Lua的转义符‘ `\` ’来对引号进行转义。

在一对方括号内将不同的字符分类或单个字符组合起来，即可创建出属于用户自己的字符分类，这种新的字符分类叫做“ `字符集（char-set）` ”。例如，字符集“ `[%w_]` ”表示同时匹配字母、数字和下划线；字符集“ `[01]` ”表示匹配二进制数字；字符集“ `[%[%]]` ”表示匹配方括号本身。若要统计一段文本中元音的数量，可以这么写：

```
nvow = select(2, string.gsub(text, "[AEIOUaeiou]", ""))
```

在字符集中包含一段字符范围的做法是写出字符范围的第一个字符和最后一个字符，并用横线连接它们。这个方法用的很少，因为大多数常用的字符范围都已预定义好了，例如“ `[0-9]` ”即为“ `%d` ”，“ `0-9a-fA-F` ”则为“ `%x` ”。不过，如果需要查找一个八进制数字，那么可以写“ `[0-7]` ”而不是“ `[01234567]` ”。在一个字符集前加一个‘ `^` ’，就可以得到这个字符集的补集，像上例模式“ `[^0-7]` ”表示所有非八进制数字的字符，而模式“ `[^\n]` ”则表示除了换行符以外的其他字符。对于简单的分类，使用其大写形式也可以得到其补集，“ `%S` ”显然要

比“ `[^%s]` ”简单。

字符分类使用与Lua相同的区域设置。因此，“ `[a-z]` ”可能并不等于“ `%l` ”。在某些区域设置中，后者可能会包括像“ `ç` ”和“ `ã` ”这样的字母。一般情况下，选用后者，因为它更简单、更具移植性、也更高效。

还可以通过修饰符来描述模式中的重复部分和可选部分。Lua的模式提供4种修饰符：

+	重复1次或多次
*	重复0次或多次
-	也是重复0次或多次
?	可选部分（出现0或1次）

“ `+` ”修饰符可匹配属于字符分类的一个或多个字符。它总是获取与模式相匹配的最长序列，例如，模式“ `%a+` ”表示一个或多个字母，即单词：

```
print(string.gsub("one, and two; and three", "%a+", "word"))
--> word, word word; word word
```

模式“ `%d+` ”匹配一个或多个数字（一个整数）：

```
print(string.match("the number 1298 is evn", "%d+"))    --> 1298
```

修饰符“ ` ` ”类似于“ `+` ”，但它还接受出现*0*次的情况。一种典型的用途是匹配一个模式不同部分之间的空格。例如，匹配像“ `()` ”或“ `( )` ”这样的一对空圆括号，可以使用模式“ `%(%s %)` ”。其中“ `%s` ”可匹配*0*个或多个空格。另一个示例是用模式“ `[%a][%w]` ”匹配Lua程序中的标识符，标识符是一个由字母或下划线开始，并伴随0个或多个下划线、字母或数字的序列。

修饰符‘ `-` ’和‘ ` ` ’一样，也是用于匹配属于字符分类的*0*个或多个字符的。不过，它会匹配最短的子串。虽然有时候“ ` ` ”和“ `-` ”没什么差别，但通常它们所表现出来的结果却是截然不同的。例如，当试图用模式“ `[%a][%w]-` ”来查找一个标识符时，只会找到第一个字母，因为“ `[_%w]-` ”总是匹配空串。又如，假设要查找一个C程序中的注释，通常会首先尝试“ `/%.%/` ”，然后由于“ `.` ”会尽可能地扩展，因此程序中的第一个“ `/` ”只会与最后一个“ ` / ` ”相匹配：

```
test = "int x; /* x */ int y; /* y */"
print(string.gsub(test, "/%*.*%*/", "<COMMENT>"))
--> int x; <COMMENT>
```

若使用“ `.-` ”模式，则会以尽可能少的扩展来找到第一个“ `*/` ”。这样就能得到想要的结果了：

```
test = "int x; /* x */ int y; /* y */"
print(string.gsub(test, "/%*.-%*/", "<COMMENT>"))
--> int x; <COMMENT> int y; <COMMENT>
```

最后一个修饰符“ `?` ”可用于匹配一个可选的字符。例如，要在一段文本中寻找一个整数，而这个整数可以包括一个可选的正负号。那么使用模式“ `[+-]?%d+` ”就可以完成这项任务，它可以匹配像“ `-12` ”、“ `23` ”、“ `+1009` ”这样的数字，而“ `[+-]` ”是一个匹配‘ `+` ’和‘ `-` ’号的字符分类，后面的‘ `?` ’说明这个符号是可选的。

与其他系统不同的是，Lua中的修饰符只能应用于一个字符分类，无法对一组分类进行修饰。例如，无法写出匹配一个可选单词的模式。在本章的最后会介绍一些高级技术，可以使用它们来绕开这条限制。

如果模式以一个‘ `^` ’起始，那么它只会匹配目标字符串的开头部分。类似地，如果模式以‘ `$` ’结尾，那么它只会匹配目标字符串的结尾部分。这些标记可用于限定一个模式的查找。例如，下面这行测试：

```
if string.find(s, "^%d") then ...
```

可检查字符串 `s` 是否以一个数字开头，而以下测试：

```
if string.find(s, "^[+-]?%d+$") then ...
```

则可检查这个字符串是否表示一个整数，并且没有多余的前导字符和结尾字符。

在模式中，还可以使用“ `%b` ”，用于匹配成对的字符。它的写法是“ `%b<x><y>` ”，其中 `<x>` 和 `<y>` 是两个不同的字符， `<x>` 作为一个起始字符， `<y>` 作为一个结束字符。例如，模式“ `%b()` ”可匹配以‘ `(` ’开始，并以‘ `)` ’结束的子串：

```
s = "a (enclosed (in) parentheses) line"
print(string.gsub(s, "%b()", ""))        --> a line
```

这种模式的典型用法包括“ `%b()` ”、“ `%b[]` ”、“ `%b{}` ”和“ `%b<>` ”，但实际上可以用任何字符作为分隔符。

捕获（capture）

捕获功能可根据一个模式从目标字符串中抽出匹配于该模式的内容。在指定捕获时，应将模式中需要捕获的部分写到一对圆括号内。

对于具有捕获的模式，函数 `string.match` 会将所有捕获到的值作为单独的结果返回。即它会将目标字符串切成多个捕获到的部分：

```
pair = "name = Anna"
key, value = string.match(pair, "(%a+)%s*=%s(%a+)")
print(key, value)        --> name Anna
```

模式“ `%a+` ”表示一个非空的字母序列，模式“ `%s*` ”表示一个可能为空的空格序列。因此上例中的这个模式表示一个字母序列，伴随着一个空格序列，一个‘ `=` ’，一些空格，以及另一个字母序列。模式中表示两个字母序列的部分都放在一对圆括号中，因此如果发现匹配，就能捕获到它们。下面是一个类似的示例：

```
date = "Today is 17/7/1990"
d, m, y = string.match(date, "(%d+)/(%d+)/(%d+)")
print(d, m, y)        --> 17 7 1990
```

还可以对模式本身使用捕获。在一个模式中，可以有“ `%d` ”这样的项，其中 `d` 是一个只有一位的数字，该项表示只匹配与第 `d` 个捕获相同的内容。有一个典型的实例可以说明它的作用，假设要在一个字符串中寻找一个由单引号或双引号括起来的子串。那么可以用这样的模式“ `["'].-["']` ”，它表示一个引号后面是任意内容及另外一个引号。但是，这种模式在处理像“ `it's all right` ”这样的字符串时就出现问题。要解决这个问题，可以捕获第一个引号，然后用它来指定第二个引号：

```
s = [[then he said: "it's all right"!]]
q, quotedPart = string.match(s, "([\"'])(.-)%1")
print(quotedPart)        --> it's all right
print(q)                 --> "
```

第一个捕获是引号字符本身，第二个捕获是引号中的内容，即与“ `.-` ”相匹配的子串。

又如，匹配Lua中的长字符串：

```
%[(=*)%[(.-)%]%1%]
```

它匹配的内容依次是：一个左方括号、零个或多个等号、另一个左方括号、任意内容（即字符串的内容）、一个右方括号、相同数量的等号及另一个右方括号：

```
p = "%[(=*)%[(.-)%]%1%]"
s = "a = [=[[[ something ]] ]==] ]=]; print(a)"
print(string.match(s, p))        --> =    [[ something ]]    ]==]
```

第一个捕获是等号序列，本例的等号序列中只有一个等号。第二个捕获是字符串的内容。

对于捕获到的值，还可用于 `gsub` 函数的字符串替换。和模式一样，用于替换的字符串中也可以包含“ `%d` ”这样的项。当进行替换时，这些项就对应于捕获到的内容。“ `%0` ”表示整个匹配，并且替换字符串中的“ `%` ”必须被转义为“ `%%` ”。下面这个示例会重复字符串中的每个字符，并且在每个副本之间插入一个减号：

```
print(string.gsub("hello Lua!", "%a", "%0-%0"))
--> h-he-el-ll-lo-o L-Lu-ua-a!
```

下例交换了所有相邻的字符：

```
print(string.gsub("hello Lua", "(.)(.)", "%2%1"))
--> ehll ouLa
```

以下是一个更有用的示例，写一个简单的格式化转换器，它能读取用LaTeX风格书写的命令字符串，例如：

```
\command{some text}
```

并将它转换成XML风格的格式：

```
<command>some text</command>
```

如果不考虑嵌套的命令，那么下面这行代码即可完成这项工作：

```
s = string.gsub(s, "\\(%a+){(.-)}", "<%1>%2</%1>")
```

例如，如果s是一个字符串，其内容为：

```
the \quote{task} is to \em{change} that.
```

调用 `gsub` 后， `s` 会变成：

```
the <quote>task</quote> is to <em>change</em> that.
```

最后，还有一个剔除字符串两端空格的示例：

```
function trim(s)
    return (string.gsub(s, "^%s*(.-)%s*$", "%1"))
```

```
3.      end
```

注意，模式中某些部分的作用，两个定位标记（ `'^'和'$'` ）表示在操作整个字符串；而“ `.-` ”会试图匹配尽可能少的内容，所以首尾两处的“ `%s*` ”便可匹配到两端所有的空格。其次， `gsub` 会返回两个值，并用一对额外的括号来丢弃多余的结果，即丢弃“匹配的总数”。

替换

`string.gsub` 函数的第三个参数不仅是一个字符串，还可以是一个函数或 `table` 。当用一个函数来调用时， `string.gsub` 会在每次找到匹配时调用该函数，调用时的参数就是捕获到的内容，而该函数的返回值则作为要替换的字符串。当用一个 `table` 来调用时， `string.gsub` 会用每次捕获到的内容作为 `key` ，在 `table` 中进行查找，并将对应的 `value` 作为要替换的字符串。如果 `table` 中不包含这个 `key` ，那么 `string.gsub` 不改变这个匹配。

例如，以下函数将完成一次变量展开。它对字符串中所有格式为 `$varname` 的部分，替换为对应全局变量 `varname` 的值：

```
1.      function expand(s)
2.          return (string.gsub(s, "$(%w+)", _G))
3.      end
4.
5.      name = "Lua"; status = "great"
6.      print(expand("$name is $status, isn't it?"))
7.      --> Lua is great, isn't it?
```

对每处与“ `$(%w+)` ”相匹配的地方， `string.gsub` 都会在 `table _G` 中查找捕获到的名称，并用找到的名称替换字符串中的匹配部分。如果 `table` 中没有这个 `key` ，则不尽兴替换。

```
1.      print(expand("$othername is $status, isn't it?"))
2.      --> $othername is great, isn't it?
```

如果不确定所有的变量都有一个对应的字符串值，则可以对它们的值应用 `tostring` 。在这种情况下，可以用一个函数来提供要替换的值：

```
1.      function expand(s)
2.          return (string.gsub(s, "$(%w+)", function(n)
3.              return tostring(_G[n])
4.          end))
5.      end
6.
```

```
7.      print(expand("print = $print; a = $a"))
8.      --> print = function: 0x8050ce0; a = nil
```

现在，对于所有匹配“ `$(%w+)` ”的地方， `string.gsub` 都会调用给定的函数，并传入捕获到的名称。如果函数返回 `nil` ，则不作替换。在本例中不会出现这种情况，因为 `tostring` 不会返回 `nil` 。

最后一个示例则继续回到上一节中提及的格式转换器。本例仍然是将LaTex风格的命令 `(\example{text})` 转换成XML风格 `(<example>text</example>)` ，但是允许嵌套的命令。以下函数用递归的方式完成了这项任务：

```
1.      function toxml(s)
2.          s = string.gsub(s, "\\(%a+)(%b{})", function(tag, body)
3.              body = string.sub(body, 2, -2)          -- 删除花括号
4.              body = toxml(body)                      -- 处理嵌套的命令
5.              return string.format("<%s>%s<%s>", tag, body, tag)
6.              end)
7.          return s
8.      end
9.
10.     print(toxml("\\title{The \\bold{big} example}"))
11.     --> <title>The <bold>big</bold> example</title>
```

- URL 编码

下一个示例是关于URL编码，这是HTTP所使用的一种编码方式，用于在一个URL中传送各种参数。这种编码方式会将特殊字符（如‘ `=` ’、‘ `&` ’、‘ `+` ’）编码为“ `%<xx>` ”的形式，其中 `<xx>` 是字符的十六进制表示。此外，它还会将空格转换为“ `+` ”。例如，它会将字符串“ `a+b = c` ”编码为“ `a%2Bb+%3D+c` ”。最后，它会将每对参数名及其值用“ `=` ”连接起来，并将每对结果 `name=value` 用“ `&` ”连接起来。例如，对于值：

```
1.      name = "a1"; query = "a+b = c"; q="yes or no"
2.  ``` `
3.
4.    会被编码为：
5.
6.  ```lua
7.      "name=a1&query=a%2Bb+%3D+c&q=yes+or+no"
```

现在，假设要对这个URL进行解码。要求对编码中的每个值，以其名称作为 `key` ，保存到一

个 `table` 内。以下函数完成一次基本的解码：

```
function unescape(s)
    s = string.gsub(s, "+", " ")
    s = string.gsub(s, "%%(%x%x)", function(h)
        return string.char(tonumber(h, 16))
    end)
    return s
end
```

第一条语句将字符串中的所有的“ `+` ”改为空格，第二句 `gsub` 则匹配所有以“ `%` ”为前缀的两位十六位数字，并对每处匹配调用一个匿名函数。这个函数会将十六进制数转换成一个数字，然后调用 `string.char` 返回相应的字符。如下所示：

```
print(unescape("a%2Bb+%eD+c"))      --> a+b = c
```

用 `gmatch` 来对 `name=value` 进行解码。由于名称和值都不能包含“ `&` ”和“ `=` ”，所以可以用模式“ `[^&=]+` ”来匹配它们：

```
cgi = {}
function decode(s)
    for name, value in string.gmatch(s, "([^&=]+)=([^&=]+)") do
        name = unescape(name)
        value = unescape(value)
        cgi[name] = value
    end
end
```

调用 `gmatch` 会匹配所有格式为 `name=value` 的部分。对于每组参数，迭代器都会将捕获到的内容作为变量 `name` 和 `value` 的值。循环体只是对两个字符串调用 `unescape` ，然后将结果保存到 `table cgi` 中。

另一方面，编码函数也很容易编写。首先，写一个 `escape` 函数，它会将所有的特殊字符编码为“ `%` ”并伴随该字符的十六进制码。另外，它还会将空格转换为“ `+` ”。

```
function escape(s)
    s = string.gsub(s, "[&=+%%%c]", function(c)
        return string.format("%%%02X", string.byte(c))
    end)
    s = string.gsub(s, " ", "+")
    return s
```

```
    end
```

`encode` 函数会遍历整个待编码的 `table` ，构建出最终的结果字符串：

```
function encode(t)
    local b = {}
    for k,v in pairs(t) do
        b[#b + 1] = (escape(k) .. "=" .. escape(v))
    end
    return table.concat(b, "&")
end

t = {name = "a1", query = "a+b = c", q = "yes or no"}
print(encode(t))        --> q=yes+or+no&query=a%2Bb+%3D+c&name=a1
```

- tab扩展

在Lua中，像“ `()` ”这样的空白捕获具有特殊的含义。这个模式不是代表捕获空内容，而是捕获它在目标字符串中的位置，返回的是一个数字：

```
print(string.match("hello", "()ll()"))      --> 3 5
```

注意，这个示例的结果与调用 `string.find` 得到的结果并不一样，因为第二个空捕获的位置是在匹配之后的。

这里有一个关于空捕获应用的示例，在一个字符串中扩展 `tab` （制表符）：

```
function expandTabs(s, tab)
    tab = tab or 8              -- 制表符的“大小”（默认为8）
    local corr = 0
    s = string.gsub(s, "()\t", function(p)
        local sp = tab - (p - 1 + corr)%tab
        corr = corr - 1 + sp
        return string.rep(" ", sp)
    end)
    return s
end
```

`gsub` 会匹配字符串中所有的 `tab` ，捕获它们的位置。内嵌函数会根据每个 `tab` 的位置，计算出还需多少空格才能达到整数倍 `tab` 的列。它先对位置减一，使其从0开始计数，然后加

上 `corr` 以补偿前面的 `tab` 。并且，它还会更新这个补偿值，用于在一个 `tab` 的修正，减一以去掉当前 `tab` ，再加上要添加的空格数 `sp` 。最后这个函数返回正确数量的空格。

为了完整起见，再看一下如何实现逆向的操作，即将空格转换为 `tab` 。第一种方法可以考虑通过空捕获来对位置进行操作。还有一种更简单的方法即可以在字符串中每8个字符后插入一个标记。无论该标记是否位于空格前，都用 `tab` 替换它。

```
1.      function unexpandTabs(s, tab)
2.          tab = tab or 8
3.          s = expandTabs(s)
4.          local pat = string.rep(".", tab)
5.          s = string.gsub(s, pat, "%0\1")
6.          s = string.gsub(s, " +\1", "\t")
7.          s = string.gsub(s, "\1", "")
8.          return s
9.      end
```

这个函数首先对字符串中所有的 `tab` 进行了扩展。然后计算出一个辅助模式，用于匹配所有的 `tab` 字符序列，并通过这个模式在每个 `tab` 字符后添加一个标记（控制字符“ `\1` ”）。之后，它将所有以此标记结尾的空格序列都替换为 `tab` 。最后，删除剩下的标记，即那些没有位于空格后的标记。

技巧

模式匹配是一种操作字符串的强大工具。通过很少的 `string.gsub` 调用就可以完成许多复杂的操作，但是应该谨慎使用。

模式匹配不能代替传统的分析器。对于某些要求并不严格的程序来说，可以在源代码中做一些有用的匹配操作，但很难构建出高质量的产品。例如，之前曾用一个模式来匹配C代码中的注释。如果C代码中有一个字符串常量含有“ `/*` ”，就会得到一个错误的结果：

```
1.      test = [[char s[] = "a /* here"; /* a tricky string */]]
2.      print(string.gsub(test, "/%*.-%*/", "<COMMENT>"))
3.      --> char s[] = "a <COMMENT>"
```

含有注释标记的字符串并不常见，所以对于自用的程序而言，这个模式足以达到要求。但是，不应该将这个有问题的程序发布出去。

通常，在Lua程序中使用模式匹配时非常有效。一台奔腾333MHz计算机可以在不到十分之一秒的时间内，匹配一个具有20万个字符的文本中所有的单词，但需要注意的是，应该提供尽可能精确的模式，

宽泛的模式会比精确的模式慢许多。一个较极端的示例就是模式“ `(.-)%$` ”，它可以获取一个单元符号中不存在美元符号，这个算法就会试着从字符串起始位置开始匹配此模式，为了搜索美元符号，它会遍历整个字符串。当到达字符串结尾时，这次模式匹配就会失败，但此失败仅意味着从字符串起始位置开始的匹配失败了。然后，这个算法还会从字符串的第二个位置开始第二次搜索，其结果仍是无法匹配这个模式。这样的匹配过程以此类推，从而表现为二次方的时间复杂度。但在一台奔腾333MHz的计算机上，搜索20万个字符需要执行超过3小时的时间。要解决这个问题，只需将模式限定为仅匹配字符串的起始部分，也就是将模式指定为“ `^(.-)%$` ”。这样便告诉了算法，如果不能在起始位置上找到匹配，就停止搜索。有了这种位置限定后，再运行该模式就只需要不到十分之一秒的时间了。

此外还要小心“空模式”的使用，也就是那些会匹配空字符串的模式。例如，如果准备用模式“ `%a*` ”来匹配名称，那么会发现到处都是名称：

```
1.      i, j = string.find(";$% **#$hello13", "%a*")
2.      print(i,j)          --> 1 0
```

在这个示例中， `string.find` 调用会成功地在字符串起始处找到一个空的字母序列。

在模式的开始或结束处使用修饰符“ `-` ”是没有意义的，因此这样总会匹配到空字符串。此修饰符总是需要有些东西在它周围以限制它的扩展范围。同样，使用含有“ `.*` ”的模式也需要注意，因为这种指示可能会扩展到预期的范围之外。

有时用Lua自身来构造一个模式也是很有用的。在前面的空格转换为 `tab` 的程序中，已用到了这个技巧。接下来再看另外一个示例，如何找出一个文本中较长的行，这里的“较长”是指超过70个字符的行。起始，一个长行就是一个具有70个或更多个字符的序列，其中每个字符都不为换行符。可以用字符分类“ `[^\n]` ”来匹配除换行符以外的其他单个字符。因此，可以将这个匹配耽搁字符的模式重复70此来匹配一个长行。在这里，可以用 `string.rep` 代替手写来创建这个模式：

```
1.      pattern = string.rep("[^\n]", 70) .. "[^\n]*"
```

另外还有一个示例，假设要进行一次与大小写无关的查找。一种方法是将模式中的任何字母 `x` 用分类“ `[xX]` ”来替换，也就是一种同时包含原字母大小写形式的分类。可以用一个函数来自动地做这种转换：

```
1.      function nocase(s)
2.          s = string.gsub(s, "%a", function(c)
3.              return "[" .. string.lower(c) .. string.upper(c) .. "]"
4.          end)
5.          return s
6.      end
7.
8.      print(nocase("Hi there!"))
```

```
9.      --> [hH][iI] [tT][hH][eE][rR][eE]!
```

有时要将所有出现 `s1` 的地方替换为 `s2` ，而不管其中是否包含魔法字符。如果字符串 `s1` 和 `s2` 是字面常量，那么可以在编写字符串时对魔法字符使用适当的转义。但如果字符串是一个变量值，那么就需要用另一个 `gsub` 来做转义了：

```
1.      s1 = string.gsub(s1, "(%w)", "%%%1")
2.      s2 = string.gsub(s2, "%%", "%%%%")
```

在搜索字符串时，要对所有非字母和非数字的字符进行转义，而在替换字符串时，只对“ `%` ”进行转义。

关于模式匹配还有一项有用的技术，就是在进行实际工作前，对目标字符串进行预处理。假设，要将一个字符串中位于一对双引号内的部分改为大写，但又允许其中包含转义的引号（ `"\""` ）：

```
1.      follows a typical string: "This is \"great\"!".
```

处理这种情况的做法是对文本进行预处理，将所有可能导致歧义的部分编码为另一种形式。例如，可以将 `“\""` 编码为 `“\1”` 。不过，若原文中已含有 `“\1”` ，就会出错。一种可以避免此类问题的做法是将所有 `“\x”` 序列编码为 `"\ddd"` 形式，其中 `ddd` 是字符 `x` 的十六进制表示形式：

```
1.      function code(s)
2.          return (string.gsub(s, "\\(.)", function(x)
3.              return string.format("\\%03d", string.byte(x))
4.          end))
5.      end
```

现在，编码后的字符串中的 `“\ddd”` 序列都来源于编码，因为原字符串中所有的 `“\ddd”` 都已被Lua翻译成对应字符了。这样解码就很简单了：

```
1.      function decode(s)
2.          return (string.gsub(s, "\\(%d%d%d)", function(d)
3.              return "\\" .. string.char(d)
4.          end))
5.      end
```

现在便完成了这项任务。由于编码后的字符串中不包含任何转义的引号（ `“\""` ），就可以直接用 `“.-*”` 来查找位于一对引号中的内容：

```
1.      s = [[follows a typical string: "This is \"great\"!".]]
2.      s = code(s)
```

```
3.      s = string.gsub(s, '".-"', string.upper)
4.      s = decode(s)
5.      print(s)        --> follows a typical string: "THIS IS \"GREAT\"!".
```

或者，更紧凑的写法：

```
1.      print(decode(string.gsub(code(s), '".-"', stirng.upper)))
```

?

I/O库

I/O库为文件操作提供了两种不同的模型，简单模型（ `simple model` ）和完整模型（ `complete model` ）。简单模型假设有一个当前输入文件和一个当前输出文件，它的I/O操作均作用于这些文件。完整模型则使用显式的文件句柄。它采用了面向对象的风格，并将所有的操作定义为文件句柄上的方法。

在本书前面的章节示例中涉及到的简单操作都使用简单模型并且十分方便。但是对于更多的高级文件操作，例如同时读取多个文件，它就无法做到了。对于这些高级操作，需要使用完整模型。

简单I/O模型

简单模型的所有操作都作用于两个当前文件。I/O库将当前输入文件初始化为进程标准输入（ `stdin` ），将当前输出文件初始化为进程标准输出（ `stdout` ）。在执行 `io.read()` 操作时，就会从标准输入中读取一行。

用函数 `io.input` 和 `io.output` 可以改变这两个当前文件。 `io.input(filename)` 调用会以只读模式打开指定的文件，并将其设为当前输入文件。之后除非再次调用 `io.input` ，否则所有的输入都将来源于这个文件。在输出方面， `io.output` 也可以完成类似的工作。如果出现错误，这两个函数都会引发（raise）错误。如果想直接处理这些错误，则必须使用完整模型中的 `io.open` 。

通常 `write` 比 `read` 简单些，首先看一下 `write` 。函数 `io.write` 接受任意数量的字符串参数，并将它们写入当前输出文件。它也可以接受数字参数，数字参数会根据常规的转换规则转换为字符串。如果想要完全地控制这种转换，则应该使用函数 `string.format` ：

```
1.      >io.write("sin (3) = ", math.sin(3), "\n")
2.      --> sin (3) = 0.14112000805987
3.      >io.write(string.format("sin (3) = %.4f\n", math.sin(3)))
4.      --> sin (3) = 0.1411
```

在实际操作中应当避免写出 `io.write(a..b..c)` 这样的代码，而是应该调用 `io.write(a,b,c)` ，它能达到与 `io.write(a..b..c)` 相同的效果，并且可以避免连接操作，因此效率更高。

无论使用 `print` 还是 `io.write` 都有一个原则。即在随意编写（quick-and-dirty）的程序中，或者为调试目的而编写的代码中，提倡使用 `print` ；而在其他需要完全控制输出的地方使用 `write` 。

```
1.      >print("hello", "Lua");print("Hi")
```

```
2.      --> hello Lua
3.      --> Hi
4.
5.      >io.write("hello", "Lua");io.write("Hi", "\n")
6.      --> helloLuaHi
```

`write` 与 `print` 有几点不同。首先， `write` 在输出时不会添加像制表符或回车这样的额外字符。其次， `write` 使用当前输出文件，而 `print` 总是使用标准输出。最后， `print` 会自动调用其参数的 `tostring()` 方法，因此它还能显示 `table` 、函数和 `nil` 。

函数 `io.read` 从当前输入文件中读取字符串，它的参数决定了要读取的数据：

`"all"`	读取整个文件
`"line"`	读取下一行
`"*number"`	读取一个整数
`<num>`	读取一个不超过 `<num>` 个字符的字符串

调用 `io.read("*all")` 会读取当前输入文件的所有内容，以当前位置作为开始。如果当前位置处于文件的末尾，或者文件为空，那么该调用会返回一个空字符串。

由于Lua可以高效地处理长字符串，因此在Lua中一种简单的、编写过滤器的技术就可以将整个文件读到一个字符串中，然后处理这个字符串（通常使用 `gsub` ），最后把这个字符串写到输出：

```
1.      t = io.read("*all")        -- 读取整个文件
2.      t = string.gsub(t, ...)    -- 做相关的处理
3.      io.write(t)                -- 写输出
```

下面是一个完整的示例，这段代码使用MIME quoted-printable编码方式对文件内容进行编码。在这种编码方式中，非ASCII字符被编码为 `=<xx>` 的形式，其中 `<xx>` 是这个字符的十六进制数字代码。此外，为了保持编码的一致性，字符“ `=` ”也需要被编码：

```
1.      t = io.read("*all")
2.      t = string.gsub(t, "([\128-\255=])"), function(c)
3.              return string.format("=%02X", string.byte(c))
4.          end)
5.      io.write(t)
```

`gsub` 中使用的模式可以捕获所有编码为 `128～255` 的字符及等号字符。

调用 `io.read("line")` 会返回当前文件的下一行，但不包括换行字符。当到达文件末尾时，该调用会返回 `nil` ，以表示无后续行可返回。它也是 `read` 的默认模式。通常，我只在需要逐行处理

的算法中使用这种模式。另外，建议使用 `all` 一次性读取整个文件，或者像后面介绍的按块来读取。

作为使用该模式的一个简单示例，下面这个程序将当前输入中的内容复制到当前输出中，并对每行进行编号：

```
for count = 1, math.huge do
    local line = io.read()
    if line == nil then break end
    io.write(string.format("%6d ", count), line, "\n")
end
```

如果只为了迭代文件中的所有行，那么 `io.lines` 迭代器更为合适。例如，下面这个程序可以对文件中的所有行进行排序：

```
local lines = {}
-- 读取table 'lines'中所有行
for line in io.lines() do lines[#lines + 1] = line end
-- 排序
table.sort(lines)
-- 输出所有行
for _, l in ipairs(lines) do io.write(l "\n") end
```

调用 `io.read("number")` 会从当前输入文件中读取一个数字。此时， `read` 会返回一个数字，而不是字符串。当一个程序需要从文件中读取大量数字时，应当避免生成中间的字符串过渡形式，这样可以提高程序的性能。 `number` 选项会忽略数字前面所有的空格，并且能处理像 `-3` 、 `+5.2` 、 `1000` 及 `-3.4e-23` 这样的数字格式。如果无法在当前文件位置读到一个数字， `read` 会返回 `nil` 。

在调用 `read` 时可以指定多个选项，函数会根据每个选项参数返回相应地内容。假设，有一个文件，其中每行有3个数字：

```
6.0    -3.23    15e12
4.3    234      1000001
...
```

现在要打印出每一行中最大的数字。可以用一次 `read` 函数调用来读取每行的3个数字：

```
while true do
    local n1, n2, n3 = io.read("*number", "*number", "*number")
    if not nil then break end
```

```
4.          print(math.max(n1, n2, n3))
5.      end
```

对于这类问题，还可以采用“ *all ”读取整个文件，然后再用 gmatch 来提取其中内容：

```
1.      local pat = "(%S+)%s+(%S+)%s+(%S+)%s+"
2.      for n1, n2, n3 in string.gmatch(io.read("*all"), pat) do
3.          print(math.max(tonumber(n1), tonumber(n2), tonumber(n3)))
4.      end
```

除了以上这些基本的读取模式，还可以用一个数字 n 作为 read 的参数。此时， read 会试着从输入文件中读取 n 个字符。如果读不到任何字符，它会返回 nil 。否则会返回一个最多 n 个字符的字符串。下面这个示例演示了这种读取模式，它是一种将数据从 stdin 复制到 stdout 的高效方法：

```
1.      while true do
2.          local block = io.read(2^13)          -- 缓冲大小为8K
3.          if not block then break end
4.          io.write(block)
5.      end
```

io.read(0) 是一种特殊情况，它用于检查是否到达了文件末尾。如果还有数据可以读取，它会返回一个空字符串，否则返回 nil 。

完整I/O模型

若要作更多的I/O控制，可以使用完整模型。这个模型是基于文件句柄的，它等价于C语言中的流（FILE *），表示一个具有当前位置的打开文件。

要打开一个文件，可以使用 io.open 函数。它仿照了C语言中的 fopen 函数，并且具有两个参数一个是要打开的文件名，另一个是模式（Mode）字符串。模式字符串可以是：“ r ”表示读取、“ w ”表示写入（同时会删除文件原来的内容）及“ a ”表示追加，另外还有一个可选的“ b ”表示打开二进制文件。 open 函数会返回表示文件的新句柄。若发生错误，则返回 nil ，及一条错误消息和一个错误代码。

```
1.      print(io.open("non-existent-file", "r"))
2.      --> nil  non-existent-file: No such file or directory  2
3.
4.      print(io.open("/etc/passwd", "w"))
5.      --> nil  /etc/passwd: Permission denied  13
```

错误代码的解释依赖于系统。

一个错误检查的典型做法是：

```
1.      local f = assert(io.open(filename, mode))
```

如果打开失败，错误消息就会成为 `assert` 的第二个参数，然后 `assert` 会显示这个消息。

当打开一个文件后，就可以用 `read/write` 方法读写文件了。这与 `read/write` 函数相似，但是需要用冒号语法，将它们作为文件句柄的方法来调用。例如，要打开一个文件，并读取其所有内容，那么这么做：

```
1.      local f = assert(io.open(filename, "r"))
2.      local t = f:read("*all")
3.      f:close()
```

I/O库提供了3个预定义C语言流的句柄： `io.stdin` 、 `io.stdout` 和 `io.stderr` 。这样，就可以将信息直接写到错误流：

```
1.      io.stderr:write(message)
```

用户可以混合使用完整模式和简单模式。通过不指定参数调用 `io.input()` ，可以得到当前输入文件的句柄。而通过 `io.input(handle)` ，可以设置当前输入文件的句柄。例如，要临时改变当前输入文件，可以这么做：

```
1.      local temp = io.input()      -- 保存当前文件
2.      io.input("newinput")         -- 打开一个新的当前文件
3.      <对新的输入文件做一些操作>
4.      io.input():close()           -- 关闭当前文件
5.      io.input(temp)               -- 恢复原来的输入文件
```

- 性能小诀窍

通常在Lua中，一次性读取整个文件比逐行地读取要快一些。但必须处理一些大文件（几十或几百兆字节）时，就无法一次性地读取所有的内容。如果希望以最高性能来处理这种大文件，那么最快的方法就是用足够大的块（例如，8KB大小的块）来读取文件。为了避免在行中间断开，只需在读一个块时再加上一行：

```
1.      local lines, rest = f:read(BUFSIZE, "*line")
```

变量 `rest` 包含了被块所断开的那行的剩余部分。这样就可以将块与行的剩余部分连接起来，从而得到了一个总是起止于行边界上的块。

下面这个示例运行此技术实现了 `wc` ， `wc` 是一个用于统计文件中字符数、单词数和行数的程序：

```
1.      local BUFSIZE = 2^13            -- 8K
2.      local f = io.input(arg[1])      -- 打开输入文件
3.      local cc, lc, wc = 0, 0, 0      -- 字符、行、单词的计数
4.      while true do
5.         local lines, rest = f:read(BUFSIZE, "*line")
6.         if not lines then break end
7.         if rest then lines = lines .. rest .. "\n" end
8.         cc = cc + #lines
9.         -- 统计块中的单词数
10.        local _, t = string.gsub(lines, "%S+", "")
11.        wc = wc + t
12.        -- 统计块中的换行字符数量
13.        _, t = string.gsub(lines, "\n", "\n")
14.        lc = lc + t
15.     end
16.     print(lc, wc, cc)
```

- 二进制文件

简单模式中的函数 `io.input` 和 `io.output` 总是以文本方式打开文件（默认行为）。在UNIX中，二进制文件和文本文件是没有差别的。但在其他一些系统中，特别是在Windows中，必须用特殊的标识来打开二进制文件。在处理二进制文件时， `io.open` 的模式字符串中必须带有字母“ `b` ”。

在Lua中，二进制数据的处理与文本处理类似。Lua中的字符串可能包含任意字节，库中几乎所有函数都能处理任意字节。只要模式字符串中不包含值为零的字节，甚至还可以对二进制数据作模式匹配。如果确实需要在模式字符串中包含值为零的字节，可以用转义字符 `%z` 来表示。

通常在读取二进制数据时，使用 `*all` 模式来读取整个文件，或者使用 `<num>` 模式来读取 `n` 个字节。下面是一个简单的示例程序，它会把DOS格式的文本文件转换为UNIX格式。它并没有使用标准的I/O文件（ `stdin` 和 `stdout` ），因为这些文件都是以文本方式打开的。它假设输入文件和输出文件的名称分别由程序的参数指定：

```
1.      local inp = assert(io.open(arg[1], "rb"))
2.      local out = assert(io.open(arg[2], "wb"))
3.
```

```
    local data = inp:read("*all")
    data = string.gsub(data, "\r\n", "\n")
    out:write(data)

    assert(out:close())
```

可以用以下命令行来调用这个程序：

```
    >lua prog.lua file.dos file.unix
```

下面是另外一个示例，它打印了在一个二进制文件中找到的所有字符串：

```
    local f = assert(io.open(arg[1], "rb"))
    local data = f:read("*all")
    local validchars = "[%w%p%s]"
    local pattern = string.rep(validchars, 6) .. "+%z"
    for w in string.gmatch(data, pattern) do
        print(w)
    end
```

这个程序假定字符串是一个以0结尾，并包含至少6个有效字符的序列。所谓“有效字符”是指被模式 `validchars` 所认可的任意字符。在这个示例中，这个模式包含了数字、字母、标点符号和空格字符。然后，通过 `string.rep` 和连接操作创建了一个新的模式，这个新模式可用于捕获6个或更多的 `validchars` 。而其结尾的 `%z` 用于匹配字符串末尾的零字节。

下面是最后一个示例，它打印了一个二进制文件的内容：

```
    local f = assert(io.open(arg[1], "rb"))
    local block = 16
    while true do
        local bytes = f:read(block)
        if not bytes then break end
        for _, b in ipairs{string.byte(bytes, 1, -1)} do
            io.write(string.format("%02X ", b))
        end
        io.write(string.rep("   ", block - string.len(bytes)))
        io.write(" ", string.gsub(bytes, "%c", "."), "\n")
    end
```

同样，程序的第一个参数是输入文件名，而结果则被输出到标准输出。这个程序以16字节作为一块读取文件。对于每个块，它先输出每个字节的十六进制表示。然后，将整个块作为文本输出，而块中的

控制字符都会替换为点符号。

以下是在UNIX系统上将这个程序应用于自身后的结果。

```
6C 6F 63 61 6C 20 66 20 3D 20 61 73 73 65 72 74 local f = assert
28 69 6F 2E 6F 70 65 6E 28 61 72 5B 31 5D 2C 2C (io.open(arg[1],
6C 6F 63 61 6C 20 66 20 3D 20 61 73 73 65 72 74  "rb")).local bl
6C 6F 63 61 6C 20 66 20 3D 20 61 73 73 65 72 74  ock = 16.while t
6C 6F 63 61 6C 20 66 20 3D 20 61 73 73 65 72 74  rue do. local b
6C 6F 63 61 6C 20 66 20 3D 20 61 73 73 65 72 74  ng.gsub(bytes, "
6C 6F 63 61 6C 20 66 20 3D 20 61 73 73 65 72 74  %c", "."), "\n"
6C 6F 63 61 6C 20                                .end.
```

- 其他文件操作

函数 `tmpfile` 返回一个临时文件的句柄，这个句柄是以读/写方式打开。这个文件会在程序结束时自动删除。函数 `flush` 会将缓冲中的数据写入文件。它与 `write` 函数一样，将其作为一个函数调用时， `io.flush()` 会刷新当前输出文件；而将其作为一个方法调用时， `f:flush()` 会刷新某个特定的文件 `f` 。

函数 `seek` 可以获取和设置一个文件的当前位置。它的一般形式是 `f:seek(whence, offset)` ，其中 `whence` 参数是一个字符串，指定了如何解释 `offset` 参数。它的有效值包括：“ `set` ”， `offset` 解释为相对于文件起始的偏移量。函数的返回值与 `whence` 无关，它总是返回文件的当前位置，即相对于文件起始处的偏移字节数。

`whence` 参数的默认值是“ `cur` ”， `offset` 的默认值是0。因此，调用 `file:seek()` 不会改变文件的当前位置，并会返回当前的文件位置。调用 `file:seek("set")` 会将当前位置设置到文件的起始处（并返回0）。调用 `file:seek("end")` 会将当前位置设置到文件的末尾，并返回文件的大小。下面这个函数可以不改变文件的当前位置而获取文件的大小：

```lua
function fsize(file)
    local current = file:seek()      -- 获取当前位置
    local size = file:seek("end")    -- 获取文件大小
    file:seek("set", current)        -- 恢复位置
    return size
end
```

如果发生错误，所有这些函数都会返回 `nil` 和一条错误消息。

?

用户自定义类型

本章将介绍如何用C语言编写新的类型来扩展Lua。下面将从一个小示例入手，使用元表和其他机制来扩展它。

这个示例实现了一种很简单的类型–布尔数组。选用这个示例是因为它不涉及到复杂的算法，从而可以使读者专注于API的问题。不过，这个示例本身还是具有实用价值的。当然，可以在Lua中用 `table` 来实现布尔数组。但C语言实现可以将每个布尔值存储在一个 `bit` 中，从而将内存用量减少到不足 `table` 方法的 `3%` 。

这个实现需要以下定义：

```
1.      #include <limits.h>
2.
3.      #define BITS_PER_WORD (CHAR_BIT* sizeof(unsigned int))
4.      #define I_WORD(i)    ((unsigned int)(i)/BITS_PER_WORD)
5.      #define I_BIT(i)    (1 << ((unsigned int)(i) % BITS_PER_WORD))
```

`BITS_PER_WORD` 是一个无符号整型的 `bit` 数量。宏 `I_WORD` 根据给定的索引来计算对应的 `bit` 位所存放的 `word` （字），`I_BIT` 计算出一个掩码，用于访问这个 `word` 中的正确 `bit` 。

可以使用以下结构来表示数组：

```
1.      typedef struct NumArray {
2.          int size;
3.          unsigned int values[1];    /* 可变部分 */
4.      } NumArray
```

这里。由于C89标准不允许分配0长度的数组，所以声明了数组 `values` 需要有一个元素来作为占位符。接下来会在分配数组时定义实际的大小。下面这个表达式可以计算出具有 `n` 个元素的数组大小：

```
1.      sizeof(NumArray) + I_WORD(n - 1)*sizeof(unsigned int)
```

注意，这里无须对 `I_WORD` 加1，因为原来的结构中已经包含了一个元素的空间。

userdata

首先要面临的问题是如何在Lua中表示这个NumArray结构。Lua为此提供了一种基本类型 `userdata` 。 `userdata` 提供了一块原始的内存区域，可以用来存储任何东西。并且，在Lua中 `userdata` 没有任何预定义的操作。

函数 `lua_newuserdata` 会根据指定的大小分配一块内存，并将对应的userdata压入栈中，最后返回这个内存块的地址：

```
1.      void *lua_newuserdata(lua_State *L, size_t size);
```

如果由于某些原因，需要通过其他机制来分配内存。那么可以创建只有一个指针大小的 `userdata` ，然后将指向真正内存块的指针存入其中。在下一章中就有这样的例子。

以下函数就用 `lua_newuserdata` 创建了一个新的布尔数组：

```
1.      static int newarray(lua_State *L) {
2.          int i, n;
3.          size_t nbytes;
4.          NumArray *a;
5.
6.          n = luaL_checkint(L, 1);
7.          luaL_argcheck(L, n >= 1, 1, "invalid size")
8.          nbytes = sizeof(NumArray) + I_WORD(n - 1)*sizeof(unsigned int);
9.          a = (NumArray *)lua_newuserdata(L, nbytes);
10.
11.         a->size = n;
12.         for(i = 0; i <= I_WORD(n-1); i++)
13.             a->values[i] = 0;         /* 初始化数组 */
14.         return 1;         /* 新的userdata已在栈上 */
15.     }
```

其中，宏 `luaL_checkint` 只是在调用 `luaL_checkinteger` 后进行了一个类型转换。只要在Lua中注册好 `newarray` ，就可以通过语句 `a=array.new(1000)` 来创建一个新数组。

可以通过这样的调用 `array.set(array, index, value)` ，在数组中存储元素。后面的内容会介绍如何使用元表来实现更传统的语法 `array[index]=value` 。无论哪种写法，底层函数都是相同的。

下面将遵循Lua惯例，假设索引从1开始。

```
1.      static int setarray(lua_State *L)
2.          NumArray *a = (NumArray *)lua_touserdata(L, 1);
3.          int index = luaL_checkint(L, 2) - 1;
```

```
        luaL_checkany(L, 3);

        luaL_argcheck(L, a != NULL, 1, "'array' expected");

        luaL_argcheck(L, 0 <= index && index < a->size, 2, "index out of
    range");

        if(lua_toboolean(L, 3))
            a->values[I_WORD(index)] |= I_BIT(index);    /* 设置bit */
        else
            a->values[I_WORD(index)] &= ~I_BIT(index);    /* 重置bit */
        return 0;
```

由于Lua中任何值都可以转换为布尔，所以这里对第3个参数使用 `luaL_checkany` ，它只确保了在这个参数位置上有一个值。如果用错误的参数调用了 `setarray` ，就会得到这样的错误消息：

```
    array.set(0, 11, 0)
    --> stdin:1: bad argument #1 to 'set' ('array' expected)
    array.set(a, 0)
    --> stdin:1: bad argument #3 to 'set' (value expected)
```

下一个函数用于检索元素：

```
    static int getarray(lua_State *L) {
        NumArray *a = (NumArray *)lua_touserdata(L, 1);
        int index = luaL_checkint(L, 2) - 1;

        luaL_argcheck(L, a != NULL, 1, "'array' expected");

        lua_pushboolean(L, a->values[I_WORD(index)] & I_BIT(index));
        return 1;
    }
```

下面还定义了一个函数用于检索一个数组的大小：

```
    static int getsize(lua_State *L) {
        NumArray *a = (NumArray *)lua_touserdata(L, 1);
        luaL_argcheck(L, a != NULL, 1, "'array' expected");
        lua_pushinteger(L, a->size);
        return 1;
    }
```

最后，需要一些代码来初始化这个库：

```
static const struct luaL_Reg arraylib [] = {
    {"new", newarray},
    {"set", setarray},
    {"get", getarray},
    {"size", getsize},
    {NULL, NULL}
};

int luaopen_array(lua_State *L) {
    luaL_register(L, "array", arraylib);
    return 1;
}
```

同样其中用到了辅助库的 `luaL_register` ，它会根据给定的名称（本例中为“ `array` ”）创建一个 `table` ，并用数组 `arraylib` 中指定的名称/函数对来填充它。

在打开库后，就可以在Lua中使用这个新类型了：

```
a = array.new(1000)
print(a)                --> userdata: 0x8064d48
print(array.size(a))    --> 1000
for i = 1, 1000 do
    array.set(a, i, i%5 == 0)
end
print(array.get(a, 10))        --> true
```

元表

当前的实现有一个重大的安全漏洞，假定用户写了这样的语句 `array.set(io.stdin, 1, false)` 。 `io.stdin` 的值是一个 `userdata` ，是一个文件流指针（FILE *）。由于这是一个 `userdata` ， `array.set` 会认为它是一个合法的参数，结果就使内存遭到破坏（如果幸运的话，可能会得到一个索引超出范围的错误）。但对于有些Lua库来说，这种行为是不可接受的。问题的原因不在于如何使用一个C程序库，而在于程序库不应破坏C数据或在Lua中导致核心转储（Core Dump）。

一种辨别不同类型的 `userdata` 的方法是，为每种类型创建一个唯一的元表。每当创建了一个 `userdata` 后，就用相应的元表来标记它。而每当得到一个 `userdata` 后，就检查它是否拥有正确的元表。由于Lua代码不能改变 `userdata` 的元表，因此也就无法欺骗代码了。

另外还需要有个地方来存储这个新的元表，然后才能用它来创建新的 `userdata` ，并检查给定的 `userdata` 是否具有正确的类型。在前面已提到过，有三个候选地可用于存储元表：注册表、环境或程序库中函数的 `upvalue` 。在Lua中，通常习惯是将所有新的C类型注册到注册表中，以一个类型名作为 `key` ，元表作为 `value` 。由于注册表中还有其他的内容，所以必须小心地选择类型名，以避免与 `key` 冲突。在示例中，将使用“ `LuaBook.array` ”作为其新类型的名称。

通常，辅助库中提供了一些函数来帮助实现这些内容。可以使用的辅助库函数有：

```
1.      int luaL_newmetatable(lua_State *L, const char *tname);
2.      void luaL_getmetatable(lua_State *L, const char *tname);
3.      void *luaL_checkudata(lua_State *L, int index, const char *tname);
```

`luaL_newmetatable` 函数会创建一个新的table用作元表，并将其压入栈顶，然后将这个 `table` 与注册表中的指定名称关联起来。 `luaL_getmetatable` 函数可以在注册表中检索与 `tname` 关联的元表。 `luaL_checkudata` 可以检查栈中指定位置上是否为一个 `userdata` ，并且是否具有与给定名称相匹配的元表。如果该对象不是一个 `userdata` ，或者它不具有正确的元表，就会引发一个错误；否则，它就返回这个 `userdata` 的地址。

现在可以修改前面的实现了。第一步是修改打开库的函数。新版本必须为数组创建一个元表：

```
1.      int luaopen_array(lua_State *L) {
2.          luaL_newmetatable(L, "LuaBook.array");
3.          luaL_register(L, "array", arraylib);
4.          return 1;
5.      }
```

下一步是修改 `newarray` ，使其能为所有新建的数组设置这个元表：

```
1.      static int newarray(lua_State *L) {
2.          <如前>
3.
4.          luaL_getmetatable(L, "LuaBook.array");
5.          lua_setmetatable(L, -2);
6.
7.          return 1;          /* 新的userdata已在栈中 */
8.      }
```

`lua_setmetatable` 函数会从栈中弹出一个 `table` ，并将其设为指定索引上对象的元表。在本例中，这个对象就是一个新建的 `userdata` 。

最后， `setarray` 、 `getarray` 和 `getsize` 必须检查其第一个参数是否为一个合法的数组。为了简化这样的任务，定义了以下宏：

```
#define checkarray(L) \
    (NumArray *)luaL_checkudata(L, 1, "LuaBook.array")
```

使用这个宏 `getsize` 同样简单：

```
static int getsize(lua_State *L)
    NumArray *a = checkarray(L);
    lua_pushinteger(L, a->size);
    return 1;
```

由于 `setarray` 和 `getarray` 使用了一段相同的代码来检查第二个索引参数，所以将这段公共部分单独组成以下函数：

```
static unsigned int *getindex(lua_State *L, unsigned int *mask)
{
    NumArray *a = checkarray(L);
    int index = luaL_checkint(L, 2) - 1;

    luaL_argcheck(L, 0 <= index && index < a->size, 2,
             "index out of range");

    /* 返回元素地址 */
    *mask = I_BIT(index);
    return &a->values[I_WORD(index)];
}
```

使用了 `getindex` 的 `setarray` 和 `getarray` 也同样简单明了：

```
static int getarray(lua_State *L) {
    unsigned int mask;
    unsigned int *entry = getindex(L, &mask);
    luaL_checkany(L, 3);
    if(lua_toboolean(L, 3))
        *entry |= mask;
    else
        *entry &= ~mask;
    return 0;
}

static int getarray(lua_State *L) {
    unsigned int mask;
```

```
14.         unsigned int *entry = getindex(L, &mask);
15.         lua_pushboolean(L, *entry & mask);
16.         return 1;
17.     }
```

现在，如果试图这样调用 `array.get(io.stdin, 10)` ，就会得到一个正确的错误消息：

```
1.      error: bad argument #1 to 'get' ('array' expected)
```

面向对象的访问

下一步是将这种类型变换成一个对象，然后就可以用普通的面向对象语法来操作它的实例了。例如：

```
1.      a = array.new(1000)
2.      print(a:size())          --> 1000
3.      a:set(10, true)
4.      print(a:get(10))        --> true
```

注意， `a:size()` 等价于 `a.size(a)` 。因此，必须使表达式 `a.size` 返回前面定义的函数 `getsize` 。实现这点的关键是使用 `__index` 元方法。对于 `table` 而言，Lua会在找不到指定 `key` 时调用这个元方法。对于 `userdata` ，则会在每次访问时都调用它，因为 `userdata` 根本没有 `key` 。

假设，运行了以下代码：

```
1.      local metaarray = getmetatable(array.new(1))
2.      metaarray.__index = metaarray
3.      metaarray.set = array.set
4.      metaarray.get = array.get
5.      metaarray.size = array.size
```

第一行创建了一个数组，并将它的元表赋予了 `metaarray` 。然后将 `metaarray.index` 设为 `metaarray` 。当对 `a.size` 求值时，由于 `a` 是一个 `userdata` ，所以Lua无法在对象 `a` 中找到 `key "size"` 。因此，Lua会尝试通过 `a` 的元表的 `__index` 字段来查找这个值，而这个字段也就是 `metaarray` 自身。由于 `metaarray.size` 为 `array.size` ，因此 `a.size(a)` 的结果就是 `array.size(a)` 。

其实，在C中也可以达到相同的效果，甚至还可以做得更好。现在的数组是一种具有操作的对象，可以无须在 `table array` 中保存这些操作。程序库只要导出一个用于创建新数组的函数new就可以

了，所有其他操作都可作为对象的方法。C代码同样可以直接注册这些方法。

操作 `getsize` 、 `getarray` 和 `setarray` 无须作任何改变，唯一需要改变的是注册它们的方式。现在，需要修改打开程序库的函数。首先，需要设置两个独立的函数列表，一个用于常规的函数，另一个用于方法：

```
1.      static const struct luaL_Reg arraylib_f[] = {
2.          {"new", newarray},
3.          {NULL, NULL}
4.      };
5.
6.      static const struct luaL_Reg arraylib_m[] = {
7.          {"set", setarray},
8.          {"get", getarray},
9.          {"size", getsize},
10.         {NULL, NULL}
11.     };
```

新的打开函数 `luaopen_array` 必须创建元表，并将它赋予 `__index` 字段，然后在元表中注册所有的方法，最后创建并填充 `array table` ：

```
1.      int luaopen_array(lua_State *L) {
2.          luaL_newmetatable(L, "LuaBook.array");
3.
4.          /* 元表.__index = 元表 */
5.          lua_pushvalue(L, -1);    /* 复制元表 */
6.
7.          luaL_register(L, NULL, arraylib_m);
8.
9.          luaL_register(L, "array", arraylib_f);
10.         return 1;
11.     }
```

其中用到了 `luaL_register` 的另一个特性。在第一次调用中，以 `NULL` 作为库名， `luaL_register` 不会创建任何用于存储函数的 `table` ，而是以栈顶的 `table` 作为存储函数的 `table` 。在本例中，栈顶 `table` 就是元表本身，因此 `luaL_register` 会将所有的方法放入其中。第二次调用 `luaL_register` 则提供了一个库名，它就根据此名（ `array` ）创建了一个新 `table` ，并将指定的函数注册在这个 `table` 中（也就是本例中唯一的 `new` 函数）。

最后，给这个数组类型添加一个 `__tostring` 方法。使 `print(a)` 可以打印出“ `array` ”以及数组的大小，就像“ `array(1000)` ”。这个函数如下：

```
int array2string(lua_State *L) {
    NumArray *a = checkarray(L);
    lua_pushfstring(L, "array(%d)", a->size);
    return 1;
}
```

`lua_pushfstring` 可以在栈顶创建并格式化一个字符串。另外，还需要将 `array2string` 加到列表 `arraylib_m` 中，从而将这个函数加入数组对象的元表。

```
static const struct luaL_Reg arraylib_m[] = {
    {"__tostring", array2string},
    <其他方法>
};
```

数组访问

另一种面向对象写法是使用常规的数组访问写法。相对于 `a:get(i)` ，可以简单地写为 `a[i]` 。对于上面的示例，很容易可以做到这点。由于函数 `setarray` 和 `getarray` 所接受的参数次序暗合相关元方法的参数次序，因此在Lua代码中可以快速地将这些元方法定义为：

```
local metaarray = getmetatable(array.new(1))
metaarray.__index = array.get
metaarray.__newindex = array.set
metaarray.__len = array.size
```

必须在第一个数组实现上运行这段代码，而不能应用于那个为面向对象访问而修改的版本。使用这些标准语法很简单：

```
a = array.new(1000)
a[10] = true            -- setarray
print(a[10])            -- getarray        --> true
print(#a)               -- getsize         --> 1000
```

如果还要更完美，可以在C代码中注册这些方法。为此，需要再次修改初始化函数：

```
static const struct luaL_Reg arraylib_f [] = {
    {"new", newarray},
    {NULL, NULL}
};
```

```
5.
6.      static const struct luaL_Reg arraylib_m [] = {
7.          {"__newindex", setarray},
8.          {"__index", getarray},
9.          {"__len", getsize},
10.         {"__tostring", array2string},
11.         {NULL, NULL}
12.     };
13.
14.     int luaopen_array(lua_State *L) {
15.         luaL_newmetatable(L, "LuaBook.array");
16.         luaL_register(L NULL, arraylib_m);
17.         luaL_register(L, "array", arraylib_f);
18.         return 1;
19.     }
```

在这个新版本中，仍只有一个公共函数 `new` 。所有其他函数都作为特定操作的元方法。

轻量级userdata（light userdata）

到现在为止所使用的 `userdata` 都称为“完全 `userdata(full userdata)` ”。Lua还提供另一种“轻量级 `userdata(light userdata)` ”。

轻量级 `userdata` 是一种表示C指针的值（即 `void *` ）。由于它是一个值，所以不用创建它。要将一个轻量级 `userdata` 放入栈中，只需调用 `lua_pushlightuserdata` 即可：

```
1.      void lua_pushlightuserdata(lua_State *L, void *p);
```

尽管两种 `userdata` 在名称上差不多，但它们之间还是存在很大不同的。轻量级 `userdata` 不是缓冲，只是一个指针而已。它也没有元表，就像数字一样，轻量级 `userdata` 无须受垃圾收集器的管理。

有时会将轻量级 `userdata` 当作一种廉价的完全 `userdata` 来使用。但这种用法并没有太大意义。首先，使用轻量级 `userdata` 时用户必须自己管理内存，因为轻量级 `userdata` 不属于垃圾收集的范畴。其次，不要被“完全”二字所迷惑，完全 `userdata` 的开销并不比轻量级 `userdata` 大多少。它们只为分配内存增加了一些 `malloc` 的开销。

轻量级 `userdata` 的真正用途是相等性判断。一个完全 `userdata` 是一个对象，它只与自身相等。而一个轻量级 `userdata` 则表示了一个C指针的值。因此，它与所有表示同一个指针的轻量级 `userdata` 相等。可以将轻量级 `userdata` 用于查找Lua中的C对象。

以下是一种比较典型的情况，假设正在实现一种Lua与某个窗口系统的绑定。在这种绑定中，用完全 `userdata` 表示窗口。每个 `userdata` 可以包含整个窗口的数据结构，也可以只包含一个指向系统所创建窗口的指针。当在一个窗口中发生了一个事件时（例如单击鼠标），系统要调用对应于该窗口的回调函数。而窗口是通过其地址来识别的。为了调用Lua中实现的回调函数，必须先找到表示指定窗口的 `userdata` 。若要寻找这个 `userdata` ，可以用一个 `table` 来保存窗口的信息，它的 `key` 是表示窗口地址的轻量级 `userdata` ，而 `value` 则是表示窗口本身的完全 `userdata` 。当得到一个窗口地址时，就可以把它作为一个轻量级 `userdata` 压入栈中，并用这个 `userdata` 来索引 `table` 。从而得到那个表示窗口本身的完全 `userdata` ，并由此调用回调函数。

?

练习题

1、实现冒泡排序。

LuaFramework

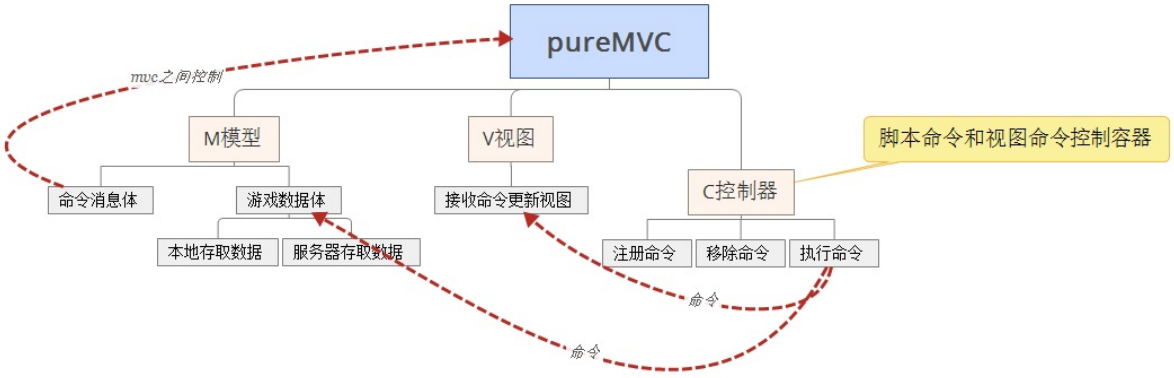
Lua源码目录介绍：

- **3rd:** 里面是第三方的一些插件lua、实例源码文件，比如: `cjson` 、 `pbc` 、 `pblua` 、 `sproto` 等。
- **Common:** 公用的 `lua` 文件目录，如 `define.lua` 文件，一些变量声明，全局配置等， `functions.lua` 常用函数库，通讯的 `protocal.lua` 协议文件。
- **Controller:** 控制器目录，它不依赖于某一个 `Lua` 面板，它是独立存活在 `Luavm` 中的一个操作类，操作数据、控制面板显示而已。
- **Logic:** 目录里面存放的是一些管理器类，比如 `GameManager` 游戏管理器、 `NetworkManager` 网络管理器，如果你有新的管理器可以放到里面。
- **System:** 这个目录是 `cstolua` 的系统目录，里面存放都是一些常用的 `lua` 类，为了优化 `lua` 调用速度，用 `lua` 重写的 `unity` 常用类。
- **View:** 这是面板的视图层，里面都是一些被 `Unity` 调用的面板的变量，走的是 `Unity GameObject` 的生命周期的事件调用。

lua和C#的交互两个工具类：

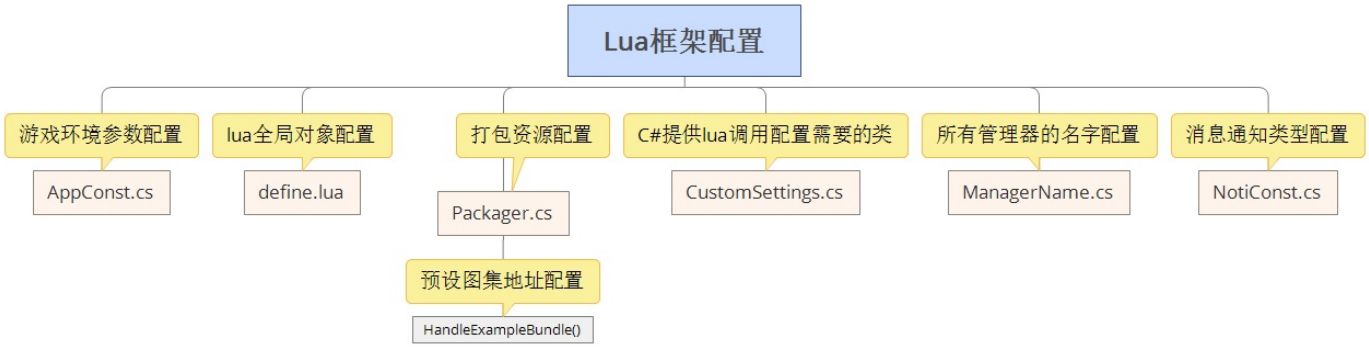
```
1.      LuaHelper.cs静态类    //lua调用C#
2.      Util.cs               //静态方法CallMethod C#调用lua
```

分析图：

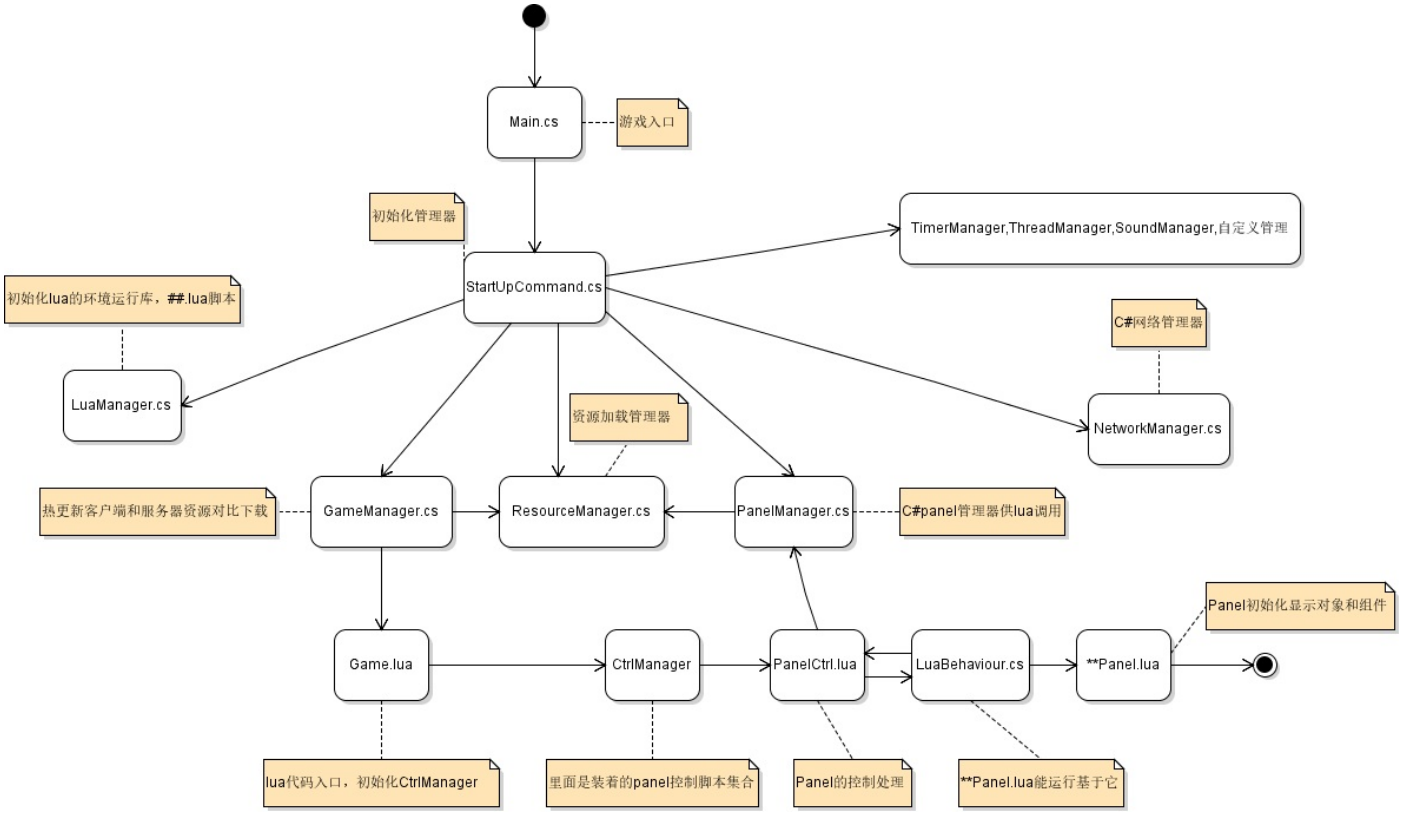

pureMVC
mvc之间控制
M模型
V视图
脚本命令和视图命令控制容器
命令消息体
游戏数据体
接收命令更新视图
C控制器
本地存取数据
服务器存取数据
注册命令
移除命令
执行命令
命令
命令

)

配置文件：


Lua框架配置
游戏环境参数配置
lua全局对象配置
打包资源配置
C#提供lua调用配置需要的类
所有管理器的名字配置
消息通知类型配置
AppConst.cs
define.lua
Packager.cs
CustomSettings.cs
ManagerName.cs
NotiConst.cs
预设图集地址配置
HandleExampleBundle()

启动流程：


Main.cs
游戏入口
初始化管理器
StartUpCommand.cs
TimerManager,ThreadManager,SoundManager,自定义管理
初始化lua的环境运行库，##.lua脚本
LuaManager.cs
C#网络管理器
NetworkManager.cs
资源加载管理器
热更新客户端和服务器资源对比下载
GameManager.cs
ResourceManager.cs
PanelManager.cs
C#panel管理器供lua调用
Panel初始化显示对象和组件
Game.lua
CtrlManager
PanelCtrl.lua
LuaBehaviour.cs
**Panel.lua
lua代码入口，初始化CtrlManager
里面是装着的panel控制脚本集合
Panel的控制处理
**Panel.lua能运行基于它

?

HelloWorld.cs

```
1.      using UnityEngine;
2.      using LuaInterface;
3.      using System;
4.
5.      public class HelloWorld : MonoBehaviour
6.      {
7.          /*
8.              这个类及其的简单，就是教你怎么启动luastate，然后将字符串lua代码执行进lua虚拟机里面，
9.              因为并没有作用域，所以当DoString的时候，就直接打印出字符串来了。然后将luastate析构掉，置空完毕。
10.         */
11.         void Awake()
12.         {
13.             LuaState lua = new LuaState();
14.             lua.Start();
15.             string hello =
16.                 @"
17.                     print('hello tolua#')
18.                 ";
19.
20.             lua.DoString(hello, "HelloWorld.cs");
21.             lua.CheckTop();
22.             lua.Dispose();
23.             lua = null;
24.         }
25.     }
```

?

ScriptsFromFile.cs

```
using UnityEngine;
using System.Collections;
using LuaInterface;
using System;
using System.IO;

/*
一、require

  1.功能：载入文件并执行代码块，对于相同的文件只执行一次

  2.调用：require(“filename”)

  注：寻找文件的路径在package.path中，print(package.path)即可得到。

二、dofile

  1.功能：载入文件并执行代码块，对于相同的文件每次都会执行

  2.调用：dofile("filename")

  3.错误处理：如果代码块中有错误则会引发错误

  4.优点：对简单任务而言，非常便捷

  5.缺点：每次载入文件时都会执行程序块

  6.定位：内置操作，辅助函数
*/

//展示searchpath 使用，require 与 dofile 区别
public class ScriptsFromFile : MonoBehaviour
{
    LuaState lua = null;
    private string strLog = "";

    void Start ()
    {
#if UNITY_5
```

```
40.             Application.logMessageReceived += Log;  // 收到日志消息时触发的事件
41.     #else
42.             Application.RegisterLogCallback(Log);
43.     #endif
44.             lua = new LuaState();
45.             lua.Start();
46.             //如果移动了ToLua目录，自己手动修复吧，只是例子就不做配置了
47.             string fullPath = Application.dataPath +
    "\\LuaFramework/ToLua/Examples/02_ScriptsFromFile";
48.             lua.AddSearchPath(fullPath);
49.         }
50.
51.         void Log(string msg, string stackTrace, LogType type)
52.         {
53.             strLog += msg;
54.             strLog += "\r\n";
55.         }
56.
57.         void OnGUI()
58.         {
59.             GUI.Label(new Rect(100, Screen.height / 2 - 100, 600, 400),
    strLog);
60.
61.             if (GUI.Button(new Rect(50, 50, 120, 45), "DoFile"))
62.             {
63.                 strLog = "";
64.                 lua.DoFile("ScriptsFromFile.lua");
65.             }
66.             else if (GUI.Button(new Rect(50, 150, 120, 45), "Require"))
67.             {
68.                 strLog = "";
69.                 lua.Require("ScriptsFromFile");
70.             }
71.
72.             lua.Collect();
73.             lua.CheckTop();
74.         }
75.
76.         void OnApplicationQuit()
77.         {
78.             lua.Dispose();
79.             lua = null;
```

```
80.
81.     #if UNITY_5
82.             Application.logMessageReceived -= Log;
83.     #else
84.             Application.RegisterLogCallback(null);
85.     #endif
86.         }
87.     }
```

ScriptsFromFile.lua

```
1.     print("This is a script from a utf8 file")
2.     print("tolua: 你好! こんにちは! 안녕하세요!")
```

?

调用Lua方法

CallLuaFunction.cs

```
1.      //#define TEST_GC
2.      using UnityEngine;
3.      using System.Collections;
4.      using LuaInterface;
5.      using System;
6.
7.      public class CallLuaFunction : MonoBehaviour
8.      {
9.          private string script =
10.             @"  function luaFunc(num)
11.                     return num + 1
12.                 end
13.
14.                 test = {}
15.                 test.luaFunc = luaFunc
16.             ";
17.
18.         LuaFunction func = null;
19.         LuaState lua = null;
20.         string tips = null;
21.
22.         void Start ()
23.         {
24.     #if !TEST_GC
25.         #if UNITY_5
26.             Application.logMessageReceived += ShowTips;
27.         #else
28.             Application.RegisterLogCallback(ShowTips);
29.         #endif
30.     #endif
31.             lua = new LuaState();
32.             lua.Start();
33.             lua.DoString(script);
34.
35.             //Get the function object
36.             func = lua.GetFunction("test.luaFunc");
37.
```

```
38.             if (func != null)
39.             {
40.                 //有gc alloc
41.                 object[] r = func.Call(123456);
42.                 Debugger.Log("generic call return: {0}", r[0]);
43.
44.                 // no gc alloc
45.                 int num = CallFunc();
46.                 Debugger.Log("expansion call return: {0}", num);
47.             }
48.
49.             lua.CheckTop();
50.         }
51.
52.         void ShowTips(string msg, string stackTrace, LogType type)
53.         {
54.             tips += msg;
55.             tips += "\r\n";
56.         }
57.
58.     #if !TEST_GC
59.         void OnGUI()
60.         {
61.             GUI.Label(new Rect(Screen.width / 2 - 200, Screen.height / 2 - 150,
    400, 300), tips);
62.         }
63.     #endif
64.
65.         void OnDestroy()
66.         {
67.             if (func != null)
68.             {
69.                 func.Dispose();
70.                 func = null;
71.             }
72.
73.             lua.Dispose();
74.             lua = null;
75.
76.     #if !TEST_GC
77.         #if UNITY_5
78.             Application.logMessageReceived -= ShowTips;
```

```
79.         #else
80.             Application.RegisterLogCallback(null);
81.         #endif
82.     #endif
83.         }
84.
85.         int CallFunc()
86.         {
87.             func.BeginPCall();
88.             func.Push(123456);
89.             func.PCall();
90.             int num = (int)func.CheckNumber();
91.             func.EndPCall();
92.             return num;
93.         }
94.
95.         //在profiler中查看gc alloc
96.     #if TEST_GC
97.         void Update ()
98.         {
99.             func.Call(123456);
100.            //CallFunc();
101.        }
102.    #endif
103.    }
```

?

访问Lua变量

AccessingLuaVariables.cs

```
1.      using UnityEngine;
2.      using System.Collections.Generic;
3.      using LuaInterface;
4.
5.      // 访问Lua变量
6.      public class AccessingLuaVariables : MonoBehaviour
7.      {
8.          private string script =
9.              @"
10.                 print('Objs2Spawn is: '..Objs2Spawn)
11.                 var2read = 42
12.                 varTable = {1,2,3,4,5}
13.                 varTable.default = 1
14.                 varTable.map = {}
15.                 varTable.map.name = 'map'
16.
17.                 meta = {name = 'meta'}
18.                 setmetatable(varTable, meta)
19.
20.                 function TestFunc(strs)
21.                     print('get func by variable')
22.                 end
23.             ";
24.
25.         void Start ()
26.         {
27.             LuaState lua = new LuaState();
28.             lua.Start();
29.             lua["Objs2Spawn"] = 5;
30.             lua.DoString(script);
31.
32.             //通过LuaState访问
33.             Debugger.Log("Read var from lua: {0}", lua["var2read"]);
34.             Debugger.Log("Read table var from lua: {0}",
    lua["varTable.default"]);
35.
36.             LuaFunction func = lua["TestFunc"] as LuaFunction;
```

```
37.             func.Call();
38.             func.Dispose();
39.
40.             //cache成LuaTable进行访问
41.             LuaTable table = lua.GetTable("varTable");
42.             Debugger.Log("Read varTable from lua, default: {0} name: {1}",
    table["default"], table["map.name"]);
43.             table["map.name"] = "new";
44.             Debugger.Log("Modify varTable name: {0}", table["map.name"]);
45.
46.             table.AddTable("newmap");
47.             LuaTable table1 = (LuaTable)table["newmap"];
48.             table1["name"] = "table1";
49.             Debugger.Log("varTable.newmap name: {0}", table1["name"]);
50.             table1.Dispose();
51.
52.             table1 = table.GetMetaTable();
53.
54.             if (table1 != null)
55.             {
56.                 Debugger.Log("varTable metatable name: {0}", table1["name"]);
57.             }
58.
59.             object[] list = table.ToArray();
60.
61.             for (int i = 0; i < list.Length; i++)
62.             {
63.                 Debugger.Log("varTable[{0}], is {1}", i, list[i]);
64.             }
65.
66.             table.Dispose();
67.             lua.CheckTop();
68.             lua.Dispose();
69.         }
70.     }
```

?

Lua协同

TestCoroutine.cs

```
1.  using UnityEngine;
2.  using System;
3.  using System.Collections;
4.  using LuaInterface;
5.
6.  //例子5和6展示的两套协同系统勿交叉使用，此为推荐方案
7.  public class TestCoroutine : MonoBehaviour
8.  {
9.      public TextAsset luaFile = null;
10.     private LuaState lua = null;
11.     private LuaLooper looper = null;
12.
13.     void Awake ()
14.     {
15.         lua  = new LuaState();
16.         lua.Start();
17.         LuaBinder.Bind(lua);
18.         looper = gameObject.AddComponent<LuaLooper>();
19.         looper.luaState = lua;
20.
21.         lua.DoString(luaFile.text, "TestCoroutine.cs");
22.         LuaFunction f = lua.GetFunction("TestCortinue");
23.         f.Call();
24.         f.Dispose();
25.         f = null;
26.     }
27.
28.     void OnDestroy()
29.     {
30.         looper.Destroy();
31.         lua.Dispose();
32.         lua = null;
33.     }
34.
35.     void OnGUI()
36.     {
37.         if (GUI.Button(new Rect(50, 50, 120, 45), "Start Counter"))
```

```
            {
                LuaFunction func = lua.GetFunction("StartDelay");
                func.Call();
                func.Dispose();
            }
            else if (GUI.Button(new Rect(50, 150, 120, 45), "Stop Counter"))
            {
                LuaFunction func = lua.GetFunction("StopDelay");
                func.Call();
                func.Dispose();
            }
            else if (GUI.Button(new Rect(50, 250, 120, 45), "GC"))
            {
                lua.DoString("collectgarbage('collect')", "TestCoroutine.cs");
                Resources.UnloadUnusedAssets();
            }
        }
    }
```

TestLuaCoroutine.lua.bytes

```
    function fib(n)
        local a, b = 0, 1
        while n > 0 do
            a, b = b, a + b
            n = n - 1
        end

        return a
    end

    function CoFunc()
        print('Coroutine started')
        local i = 0
        for i = 0, 10, 1 do
            print(fib(i))
            coroutine.wait(0.1)
        end
        print("current frameCount: "..Time.frameCount)
        coroutine.step()
```

```
20.         print("yield frameCount: "..Time.frameCount)
21.
22.         local www = UnityEngine.WWW("http://www.baidu.com")
23.         coroutine.www(www)
24.         local s = tolua.tolstring(www.bytes)
25.         print(s:sub(1, 128))
26.         print('Coroutine ended')
27.     end
28.
29.     function TestCortinue()
30.         coroutine.start(CoFunc)
31.     end
32.
33.     local coDelay = nil
34.
35.     function Delay()
36.         local c = 1
37.
38.         while true do
39.             coroutine.wait(1)
40.             print("Count: "..c)
41.             c = c + 1
42.         end
43.     end
44.
45.     function StartDelay()
46.         coDelay = coroutine.start(Delay)
47.     end
48.
49.     function StopDelay()
50.         coroutine.stop(coDelay)
51.     end
```

?

Lua协同

TestCoroutine2.cs

```
1.      using UnityEngine;
2.      using System.Collections;
3.      using LuaInterface;
4.
5.      //两套协同勿交叉使用，类unity原生，大量使用效率低
6.      public class TestCoroutine2 : LuaClient
7.      {
8.          string script =
9.          @"
10.             function CoExample()
11.                 WaitForSeconds(1)
12.                 print('WaitForSeconds end time: '.. UnityEngine.Time.time)
13.                 WaitForFixedUpdate()
14.                 print('WaitForFixedUpdate end frameCount:
    '..UnityEngine.Time.frameCount)
15.                 WaitForEndOfFrame()
16.                 print('WaitForEndOfFrame end frameCount:
    '..UnityEngine.Time.frameCount)
17.                 Yield(null)
18.                 print('yield null end frameCount:
    '..UnityEngine.Time.frameCount)
19.                 Yield(0)
20.                 print('yield(0) end frameCime: '..UnityEngine.Time.frameCount)
21.                 local www = UnityEngine.WWW('http://www.baidu.com')
22.                 Yield(www)
23.                 print('yield(www) end time: '.. UnityEngine.Time.time)
24.                 local s = tolua.tolstring(www.bytes)
25.                 print(s:sub(1, 128))
26.                 print('coroutine over')
27.             end
28.
29.             function TestCo()
30.                 StartCoroutine(CoExample)
31.             end
32.
33.             local coDelay = nil
34.
```

```
            function Delay()
                local c = 1

                while true do
                    WaitForSeconds(1)
                    print('Count: '..c)
                    c = c + 1
                end
            end

            function StartDelay()
                coDelay = StartCoroutine(Delay)
            end

            function StopDelay()
                StopCoroutine(coDelay)
                coDelay = nil
            end
        ";

        protected override void OnLoadFinished()
        {
            base.OnLoadFinished();

            luaState.DoString(script, "TestCoroutine2.cs");
            LuaFunction func = luaState.GetFunction("TestCo");
            func.Call();
            func.Dispose();
            func = null;
        }

        //屏蔽，例子不需要运行
        protected override void CallMain() { }

        bool beStart = false;

        void OnGUI()
        {
            if (GUI.Button(new Rect(50, 50, 120, 45), "Start Counter"))
            {
                if (!beStart)
                {
```

```
77.                     beStart = true;
78.                     LuaFunction func = luaState.GetFunction("StartDelay");
79.                     func.Call();
80.                     func.Dispose();
81.                 }
82.             }
83.             else if (GUI.Button(new Rect(50, 150, 120, 45), "Stop Counter"))
84.             {
85.                 if (beStart)
86.                 {
87.                     beStart = false;
88.                     LuaFunction func = luaState.GetFunction("StopDelay");
89.                     func.Call();
90.                     func.Dispose();
91.                 }
92.             }
93.         }
94.     }
```

?

Lua线程

TestLuaThread.cs

```
using UnityEngine;
using System.Collections;
using LuaInterface;

public class TestLuaThread : MonoBehaviour
{
    string script =
        @"
            function fib(n)
                local a, b = 0, 1
                while n > 0 do
                    a, b = b, a + b
                    n = n - 1
                end

                return a
            end

            function CoFunc(len)
                print('Coroutine started')
                local i = 0
                for i = 0, len, 1 do
                    local flag = coroutine.yield(fib(i))
                    if not flag then
                        break
                    end
                end
                print('Coroutine ended')
            end

            function Test()
                local co = coroutine.create(CoFunc)
                return co
            end
        ";

    LuaState state = null;
```

```
38.         LuaThread thread = null;
39.
40.         void Start ()
41.         {
42.             state = new LuaState();
43.             state.Start();
44.             state.LogGC = true;
45.             state.DoString(script);
46.
47.             LuaFunction func = state.GetFunction("Test");
48.             func.BeginPCall();
49.             func.PCall();
50.             thread = func.CheckLuaThread();
51.             thread.name = "LuaThread";
52.             func.EndPCall();
53.             func.Dispose();
54.             func = null;
55.
56.             thread.Resume(10);
57.         }
58.
59.         void OnDestroy()
60.         {
61.             if (thread != null)
62.             {
63.                 thread.Dispose();
64.                 thread = null;
65.             }
66.
67.             state.Dispose();
68.             state = null;
69.         }
70.
71.         void Update()
72.         {
73.             state.CheckTop();
74.             state.Collect();
75.         }
76.
77.         void OnGUI()
78.         {
79.             if (GUI.Button(new Rect(10, 10, 120, 40), "Resume Thead"))
```

```
80.             {
81.                 if (thread != null && thread.Resume(true) ==
    (int)LuaThreadStatus.LUA_YIELD)
82.                 {
83.                     object[] objs = thread.GetResult();
84.                     Debugger.Log("lua yield: " + objs[0]);
85.                 }
86.             }
87.             else if (GUI.Button(new Rect(10, 60, 120, 40), "Close Thread"))
88.             {
89.                 if (thread != null)
90.                 {
91.                     thread.Dispose();
92.                     thread = null;
93.                 }
94.             }
95.         }
96.     }
```

?

访问数组

AccessingArray.cs

```
1.  using UnityEngine;
2.  using LuaInterface;
3.
4.  public class AccessingArray : MonoBehaviour
5.  {
6.      private string script =
7.          @"
8.              function TestArray(array)
9.                  local len = array.Length
10.
11.                 for i = 0, len - 1 do
12.                     print('Array: '..tostring(array[i]))
13.                 end
14.
15.                 local t = array:ToTable()
16.
17.                 for i = 1, #t do
18.                     print('table: '.. tostring(t[i]))
19.                 end
20.
21.                 local iter = array:GetEnumerator()
22.
23.                 while iter:MoveNext() do
24.                     print('iter: '..iter.Current)
25.                 end
26.
27.                 local pos = array:BinarySearch(3)
28.                 print('array BinarySearch: pos: '..pos..' value:
    '..array[pos])
29.
30.                 pos = array:IndexOf(4)
31.                 print('array indexof bbb pos is: '..pos)
32.
33.                 return 1, '123', true
34.             end
35.         ";
36.
```

```
37.         LuaState lua = null;
38.         LuaFunction func = null;
39.
40.         void Start()
41.         {
42.             lua = new LuaState();
43.             lua.Start();
44.             lua.DoString(script, "AccessingArray.cs");
45.
46.             int[] array = { 1, 2, 3, 4, 5};
47.             func = lua.GetFunction("TestArray");
48.
49.             func.BeginPCall();
50.             func.Push(array);
51.             func.PCall();
52.             double arg1 = func.CheckNumber();
53.             string arg2 = func.CheckString();
54.             bool arg3 = func.CheckBoolean();
55.             Debugger.Log("return is {0} {1} {2}", arg1, arg2, arg3);
56.             func.EndPCall();
57.
58.             //转换一下类型，避免可变参数拆成多个参数传递
59.             object[] objs = func.Call((object)array);
60.
61.             if (objs != null)
62.             {
63.                 Debugger.Log("return is {0} {1} {2}", objs[0], objs[1],
    objs[2]);
64.             }
65.
66.             lua.CheckTop();
67.         }
68.
69.         void OnApplicationQuit()
70.         {
71.             func.Dispose();
72.             lua.Dispose();
73.         }
74.     }
```

?

Lua枚举

AccessingEnum.cs

```
using UnityEngine;
using System;
using LuaInterface;

public class AccessingEnum : MonoBehaviour
{
    string script =
        @"
            space = nil

            function TestEnum(e)
                print('Enum is:'..tostring(e))

                if space:ToInt() == 0 then
                    print('enum ToInt() is ok')
                end

                if not space:Equals(0) then
                    print('enum compare int is ok')
                end

                if space == e then
                    print('enum compare enum is ok')
                end

                local s = UnityEngine.Space.IntToEnum(0)

                if space == s then
                    print('IntToEnum change type is ok')
                end
            end

            function ChangeLightType(light, type)
                print('change light type to Directional')
                light.type = UnityEngine.LightType.Directional
            end
        ";
```

```
38.
39.         void Start ()
40.         {
41.             LuaState state = new LuaState();
42.             state.Start();
43.             LuaBinder.Bind(state);
44.
45.             state.DoString(script);
46.             state["space"] = Space.World;
47.
48.             LuaFunction func = state.GetFunction("TestEnum");
49.             func.BeginPCall();
50.             func.Push(Space.World);
51.             func.PCall();
52.             func.EndPCall();
53.             func.Dispose();
54.             func = null;
55.
56.             GameObject go = GameObject.Find("/Light");
57.             Light light = go.GetComponent<Light>();
58.             func = state.GetFunction("ChangeLightType");
59.             func.BeginPCall();
60.             func.Push(light);
61.             func.Push(LightType.Directional);
62.             func.PCall();
63.             func.EndPCall();
64.             func.Dispose();
65.             func = null;
66.
67.             state.CheckTop();
68.             state.Dispose();
69.             state = null;
70.         }
71.     }
```

?

Lua委托

TestDelegate.cs

```
using UnityEngine;
using System;
using System.Collections.Generic;
using LuaInterface;


public class TestDelegate: MonoBehaviour
{
    private string script =
    @"
            function DoClick1(go)
                print('click1 gameObject is '..go.name)
            end

            function DoClick2(go)
                print('click2 gameObject is '..go.name)
            end

            function AddClick1(listener)
                if listener.onClick then
                    listener.onClick = listener.onClick + DoClick1
                else
                    listener.onClick = DoClick1
                end
            end

            function AddClick2(listener)
                if listener.onClick then
                    listener.onClick = listener.onClick + DoClick2
                else
                    listener.onClick = DoClick2
                end
            end

            function SetClick1(listener)
                if listener.onClick then
                    listener.onClick:Destroy()
```

```
38.                     end
39.
40.                     listener.onClick = DoClick1
41.                 end
42.
43.                 function RemoveClick1(listener)
44.                     if listener.onClick then
45.                         listener.onClick = listener.onClick - DoClick1
46.                     else
47.                         print('empty delegate')
48.                     end
49.                 end
50.
51.                 function RemoveClick2(listener)
52.                     if listener.onClick then
53.                         listener.onClick = listener.onClick - DoClick2
54.                     else
55.                         print('empty delegate')
56.                     end
57.                 end
58.
59.                 --测试重载问题
60.                 function TestOverride(listener)
61.                     listener:SetOnFinished(TestEventListener.OnClick(DoClick1))
62.
    listener:SetOnFinished(TestEventListener.VoidDelegate(DoClick2))
63.                 end
64.
65.                 function TestEvent()
66.                     print('this is a event')
67.                 end
68.
69.                 function AddEvent(listener)
70.                     listener.onClickEvent = listener.onClickEvent + TestEvent
71.                 end
72.
73.                 function RemoveEvent(listener)
74.                     listener.onClickEvent = listener.onClickEvent - TestEvent
75.                 end
76.
77.                 local t = {name = 'byself'}
78.
```

```
                function t:TestSelffunc()
                    print('callback with self: '..self.name)
                end

                function AddSelfClick(listener)
                    if listener.onClick then
                        listener.onClick = listener.onClick +
    TestEventListener.OnClick(t.TestSelffunc, t)
                    else
                        listener.onClick =
    TestEventListener.OnClick(t.TestSelffunc, t)
                    end
                end

                function RemoveSelfClick(listener)
                    if listener.onClick then
                        listener.onClick = listener.onClick -
    TestEventListener.OnClick(t.TestSelffunc, t)
                    else
                        print('empty delegate')
                    end
                end
        ";

        LuaState state = null;
        TestEventListener listener = null;

        LuaFunction SetClick1 = null;
        LuaFunction AddClick1 = null;
        LuaFunction AddClick2 = null;
        LuaFunction RemoveClick1 = null;
        LuaFunction RemoveClick2 = null;
        LuaFunction TestOverride = null;
        LuaFunction RemoveEvent = null;
        LuaFunction AddEvent = null;
        LuaFunction AddSelfClick = null;
        LuaFunction RemoveSelfClick = null;

        //需要删除的转LuaFunction为委托，不需要删除的直接加或者等于即可
        void Awake()
        {
            state = new LuaState();
```

```
118.             state.Start();
119.             LuaBinder.Bind(state);
120.             Bind(state);
121.
122.             state.LogGC = true;
123.             state.DoString(script);
124.             GameObject go = new GameObject("TestGo");
125.             listener =
    (TestEventListener)go.AddComponent(typeof(TestEventListener));
126.
127.             SetClick1 = state.GetFunction("SetClick1");
128.             AddClick1 = state.GetFunction("AddClick1");
129.             AddClick2 = state.GetFunction("AddClick2");
130.             RemoveClick1 = state.GetFunction("RemoveClick1");
131.             RemoveClick2 = state.GetFunction("RemoveClick2");
132.             TestOverride = state.GetFunction("TestOverride");
133.             AddEvent = state.GetFunction("AddEvent");
134.             RemoveEvent = state.GetFunction("RemoveEvent");
135.
136.             AddSelfClick = state.GetFunction("AddSelfClick");
137.             RemoveSelfClick = state.GetFunction("RemoveSelfClick");
138.         }
139.
140.         void Bind(LuaState L)
141.         {
142.             L.BeginModule(null);
143.             TestEventListenerWrap.Register(state);
144.             L.EndModule();
145.
146.             DelegateFactory.dict.Add(typeof(TestEventListener.OnClick),
    TestEventListener_OnClick);
147.             DelegateFactory.dict.Add(typeof(TestEventListener.VoidDelegate),
    TestEventListener_VoidDelegate);
148.         }
149.
150.         void CallLuaFunction(LuaFunction func)
151.         {
152.             func.BeginPCall();
153.             func.Push(listener);
154.             func.PCall();
155.             func.EndPCall();
156.         }
```

```
157.
158.         //自动生成代码后拷贝过来
159.         class TestEventListener_OnClick_Event : LuaDelegate
160.         {
161.             public TestEventListener_OnClick_Event(LuaFunction func) :
     base(func) { }
162.
163.             public void Call(UnityEngine.GameObject param0)
164.             {
165.                 func.BeginPCall();
166.                 func.Push(param0);
167.                 func.PCall();
168.                 func.EndPCall();
169.             }
170.         }
171.
172.         public static Delegate TestEventListener_OnClick(LuaFunction func,
     LuaTable self, bool flag)
173.         {
174.             if (func == null)
175.             {
176.                 TestEventListener.OnClick fn = delegate { };
177.                 return fn;
178.             }
179.
180.             TestEventListener_OnClick_Event target = new
     TestEventListener_OnClick_Event(func);
181.             TestEventListener.OnClick d = target.Call;
182.             target.method = d.Method;
183.             return d;
184.         }
185.
186.         class TestEventListener_VoidDelegate_Event : LuaDelegate
187.         {
188.             public TestEventListener_VoidDelegate_Event(LuaFunction func) :
     base(func) { }
189.
190.             public void Call(UnityEngine.GameObject param0)
191.             {
192.                 func.BeginPCall();
193.                 func.Push(param0);
194.                 func.PCall();
```

```
                func.EndPCall();
            }
        }

        public static Delegate TestEventListener_VoidDelegate(LuaFunction func,
    LuaTable self, bool flag)
        {
            if (func == null)
            {
                TestEventListener.VoidDelegate fn = delegate { };
                return fn;
            }

            TestEventListener_VoidDelegate_Event target = new
    TestEventListener_VoidDelegate_Event(func);
            TestEventListener.VoidDelegate d = target.Call;
            target.method = d.Method;
            return d;
        }

        void OnGUI()
        {
            if (GUI.Button(new Rect(10, 10, 120, 40), " = OnClick1"))
            {
                CallLuaFunction(SetClick1);
            }
            else if (GUI.Button(new Rect(10, 60, 120, 40), " + Click1"))
            {
                CallLuaFunction(AddClick1);
            }
            else if (GUI.Button(new Rect(10, 110, 120, 40), " + Click2"))
            {
                CallLuaFunction(AddClick2);
            }
            else if (GUI.Button(new Rect(10, 160, 120, 40), " - Click1"))
            {
                CallLuaFunction(RemoveClick1);
            }
            else if (GUI.Button(new Rect(10, 210, 120, 40), " - Click2"))
            {
                CallLuaFunction(RemoveClick2);
            }
```

```
235.             else if (GUI.Button(new Rect(10, 260, 120, 40), "+ Click1 in C#"))
236.             {
237.                 LuaFunction func = state.GetFunction("DoClick1");
238.                 TestEventListener.OnClick onClick =
    (TestEventListener.OnClick)DelegateFactory.CreateDelegate(typeof(TestEventListene
     func);
239.                 listener.onClick += onClick;
240.             }
241.             else if (GUI.Button(new Rect(10, 310, 120, 40), " - Click1 in C#"))
242.             {
243.                 LuaFunction func = state.GetFunction("DoClick1");
244.                 listener.onClick =
    (TestEventListener.OnClick)DelegateFactory.RemoveDelegate(listener.onClick,
    func);
245.                 func.Dispose();
246.                 func = null;
247.             }
248.             else if (GUI.Button(new Rect(10, 360, 120, 40), "OnClick"))
249.             {
250.                 if (listener.onClick != null)
251.                 {
252.                     listener.onClick(gameObject);
253.                 }
254.                 else
255.                 {
256.                     Debug.Log("empty delegate!!");
257.                 }
258.             }
259.             else if (GUI.Button(new Rect(10, 410, 120, 40), "Override"))
260.             {
261.                 CallLuaFunction(TestOverride);
262.             }
263.             else if (GUI.Button(new Rect(10, 460, 120, 40), "Force GC"))
264.             {
265.                 //自动gc log: collect lua reference name , id xxx in thread
266.                 state.LuaGC(LuaGCOptions.LUA_GCCOLLECT, 0);
267.                 GC.Collect();
268.             }
269.             else if (GUI.Button(new Rect(10, 510, 120, 40), "event +"))
270.             {
271.                 CallLuaFunction(AddEvent);
272.             }
```

```
            else if (GUI.Button(new Rect(10, 560, 120, 40), "event -"))
            {
                CallLuaFunction(RemoveEvent);
            }
            else if (GUI.Button(new Rect(10, 610, 120, 40), "event call"))
            {
                listener.OnClickEvent(gameObject);
            }
            else if (GUI.Button(new Rect(200, 10, 120, 40), "+self call"))
            {
                CallLuaFunction(AddSelfClick);
            }
            else if (GUI.Button(new Rect(200, 60, 120, 40), "-self call"))
            {
                CallLuaFunction(RemoveSelfClick);
            }
        }

        void Update()
        {
            state.Collect();
            state.CheckTop();
        }

        void SafeRelease(ref LuaFunction luaRef)
        {
            if (luaRef != null)
            {
                luaRef.Dispose();
                luaRef = null;
            }
        }

        void OnDestroy()
        {
            SafeRelease(ref AddClick1);
            SafeRelease(ref AddClick2);
            SafeRelease(ref RemoveClick1);
            SafeRelease(ref RemoveClick2);
            SafeRelease(ref SetClick1);
            SafeRelease(ref TestOverride);
            state.Dispose();
```

```
315.             state = null;
316.         }
317.     }
```

TestEventListener.cs

```
1.  using UnityEngine;
2.  using System;
3.  using System.Collections;
4.  using LuaInterface;
5.
6.  public class TestEventListener : MonoBehaviour
7.  {
8.      public delegate void VoidDelegate(GameObject go);
9.      public delegate void OnClick(GameObject go);
10.     public OnClick onClick = delegate { };
11.
12.     public event OnClick onClickEvent = delegate { };
13.
14.     public Func<bool> TestFunc = null;
15.
16.     public void SetOnFinished(OnClick click)
17.     {
18.         Debugger.Log("SetOnFinished OnClick");
19.     }
20.
21.     public void SetOnFinished(VoidDelegate click)
22.     {
23.         Debugger.Log("SetOnFinished VoidDelegate");
24.     }
25.
26.     [NoToLuaAttribute]
27.     public void OnClickEvent(GameObject go)
28.     {
29.         onClickEvent(go);
30.     }
31. }
```

TestEventListenerWrap.cs

```
1.      //this source code was auto-generated by tolua#, do not modify it
2.      using System;
3.      using LuaInterface;
4.
5.      public class TestEventListenerWrap
6.      {
7.          public static void Register(LuaState L)
8.          {
9.              L.BeginClass(typeof(TestEventListener),
    typeof(UnityEngine.MonoBehaviour));
10.             L.RegFunction("SetOnFinished", SetOnFinished);
11.             L.RegFunction("__eq", op_Equality);
12.             L.RegFunction("__tostring", Lua_ToString);
13.             L.RegVar("onClick", get_onClick, set_onClick);
14.             L.RegVar("onClickEvent", get_onClickEvent, set_onClickEvent);
15.             L.RegFunction("OnClick", TestEventListener_OnClick);
16.             L.RegFunction("VoidDelegate", TestEventListener_VoidDelegate);
17.             L.EndClass();
18.         }
19.
20.         [MonoPInvokeCallbackAttribute(typeof(LuaCSFunction))]
21.         static int SetOnFinished(IntPtr L)
22.         {
23.             try
24.             {
25.                 int count = LuaDLL.lua_gettop(L);
26.
27.                 if (count == 2 && TypeChecker.CheckTypes(L, 1,
    typeof(TestEventListener), typeof(TestEventListener.VoidDelegate)))
28.                 {
29.                     TestEventListener obj =
    (TestEventListener)ToLua.ToObject(L, 1);
30.                     TestEventListener.VoidDelegate arg0 = null;
31.                     LuaTypes funcType2 = LuaDLL.lua_type(L, 2);
32.
33.                     if (funcType2 != LuaTypes.LUA_TFUNCTION)
34.                     {
35.                         arg0 =
    (TestEventListener.VoidDelegate)ToLua.ToObject(L, 2);
36.                     }
```

```
37.                     else
38.                     {
39.                         LuaFunction func = ToLua.ToLuaFunction(L, 2);
40.                         arg0 =
    DelegateFactory.CreateDelegate(typeof(TestEventListener.VoidDelegate), func) as
    TestEventListener.VoidDelegate;
41.                     }
42.
43.                     obj.SetOnFinished(arg0);
44.                     return 0;
45.                 }
46.                 else if (count == 2 && TypeChecker.CheckTypes(L, 1,
    typeof(TestEventListener), typeof(TestEventListener.OnClick)))
47.                 {
48.                     TestEventListener obj =
    (TestEventListener)ToLua.ToObject(L, 1);
49.                     TestEventListener.OnClick arg0 = null;
50.                     LuaTypes funcType2 = LuaDLL.lua_type(L, 2);
51.
52.                     if (funcType2 != LuaTypes.LUA_TFUNCTION)
53.                     {
54.                          arg0 = (TestEventListener.OnClick)ToLua.ToObject(L,
    2);
55.                     }
56.                     else
57.                     {
58.                         LuaFunction func = ToLua.ToLuaFunction(L, 2);
59.                         arg0 =
    DelegateFactory.CreateDelegate(typeof(TestEventListener.OnClick), func) as
    TestEventListener.OnClick;
60.                     }
61.
62.                     obj.SetOnFinished(arg0);
63.                     return 0;
64.                 }
65.                 else
66.                 {
67.                     return LuaDLL.luaL_throw(L, "invalid arguments to method:
    TestEventListener.SetOnFinished");
68.                 }
69.             }
70.             catch(Exception e)
```

```
71.             {
72.                 return LuaDLL.toluaL_exception(L, e);
73.             }
74.         }
75.
76.         [MonoPInvokeCallbackAttribute(typeof(LuaCSFunction))]
77.         static int op_Equality(IntPtr L)
78.         {
79.             try
80.             {
81.                 ToLua.CheckArgsCount(L, 2);
82.                 UnityEngine.Object arg0 = (UnityEngine.Object)ToLua.ToObject(L,
    1);
83.                 UnityEngine.Object arg1 = (UnityEngine.Object)ToLua.ToObject(L,
    2);
84.                 bool o = arg0 == arg1;
85.                 LuaDLL.lua_pushboolean(L, o);
86.                 return 1;
87.             }
88.             catch(Exception e)
89.             {
90.                 return LuaDLL.toluaL_exception(L, e);
91.             }
92.         }
93.
94.         [MonoPInvokeCallbackAttribute(typeof(LuaCSFunction))]
95.         static int Lua_ToString(IntPtr L)
96.         {
97.             object obj = ToLua.ToObject(L, 1);
98.
99.             if (obj != null)
100.             {
101.                 LuaDLL.lua_pushstring(L, obj.ToString());
102.             }
103.             else
104.             {
105.                 LuaDLL.lua_pushnil(L);
106.             }
107.
108.             return 1;
109.         }
110.
```

```
111.         [MonoPInvokeCallbackAttribute(typeof(LuaCSFunction))]
112.         static int get_onClick(IntPtr L)
113.         {
114.             object o = null;
115.
116.             try
117.             {
118.                 o = ToLua.ToObject(L, 1);
119.                 TestEventListener obj = (TestEventListener)o;
120.                 TestEventListener.OnClick ret = obj.onClick;
121.                 ToLua.Push(L, ret);
122.                 return 1;
123.             }
124.             catch(Exception e)
125.             {
126.                 return LuaDLL.toluaL_exception(L, e, o == null ? "attempt to
    index onClick on a nil value" : e.Message);
127.             }
128.         }
129.
130.         [MonoPInvokeCallbackAttribute(typeof(LuaCSFunction))]
131.         static int get_onClickEvent(IntPtr L)
132.         {
133.             ToLua.Push(L, new EventObject("TestEventListener.onClickEvent"));
134.             return 1;
135.         }
136.
137.         [MonoPInvokeCallbackAttribute(typeof(LuaCSFunction))]
138.         static int set_onClick(IntPtr L)
139.         {
140.             object o = null;
141.
142.             try
143.             {
144.                 o = ToLua.ToObject(L, 1);
145.                 TestEventListener obj = (TestEventListener)o;
146.                 TestEventListener.OnClick arg0 = null;
147.                 LuaTypes funcType2 = LuaDLL.lua_type(L, 2);
148.
149.                 if (funcType2 != LuaTypes.LUA_TFUNCTION)
150.                 {
151.                     arg0 = (TestEventListener.OnClick)ToLua.CheckObject(L, 2,
```

```
    typeof(TestEventListener.OnClick));
152.                 }
153.                 else
154.                 {
155.                     LuaFunction func = ToLua.ToLuaFunction(L, 2);
156.                     arg0 =
    DelegateFactory.CreateDelegate(typeof(TestEventListener.OnClick), func) as
    TestEventListener.OnClick;
157.                 }
158.
159.                 obj.onClick = arg0;
160.                 return 0;
161.             }
162.             catch(Exception e)
163.             {
164.                 return LuaDLL.toluaL_exception(L, e, o == null ? "attempt to
    index onClick on a nil value" : e.Message);
165.             }
166.         }
167.
168.         [MonoPInvokeCallbackAttribute(typeof(LuaCSFunction))]
169.         static int set_onClickEvent(IntPtr L)
170.         {
171.             try
172.             {
173.                 TestEventListener obj = (TestEventListener)ToLua.CheckObject(L,
    1, typeof(TestEventListener));
174.                 EventObject arg0 = null;
175.
176.                 if (LuaDLL.lua_isuserdata(L, 2) != 0)
177.                 {
178.                     arg0 = (EventObject)ToLua.ToObject(L, 2);
179.                 }
180.                 else
181.                 {
182.                     return LuaDLL.luaL_throw(L, "The event
    'TestEventListener.onClickEvent' can only appear on the left hand side of += or
    -= when used outside of the type 'TestEventListener'");
183.                 }
184.
185.                 if (arg0.op == EventOp.Add)
186.                 {
```

```
187.                     TestEventListener.OnClick ev =
    (TestEventListener.OnClick)DelegateFactory.CreateDelegate(typeof(TestEventListene
     arg0.func);
188.                     obj.onClickEvent += ev;
189.                 }
190.                 else if (arg0.op == EventOp.Sub)
191.                 {
192.                     TestEventListener.OnClick ev =
    (TestEventListener.OnClick)LuaMisc.GetEventHandler(obj,
    typeof(TestEventListener), "onClickEvent");
193.                     Delegate[] ds = ev.GetInvocationList();
194.                     LuaState state = LuaState.Get(L);
195.
196.                     for (int i = 0; i < ds.Length; i++)
197.                     {
198.                         ev = (TestEventListener.OnClick)ds[i];
199.                         LuaDelegate ld = ev.Target as LuaDelegate;
200.
201.                         if (ld != null && ld.func == arg0.func)
202.                         {
203.                             obj.onClickEvent -= ev;
204.                             state.DelayDispose(ld.func);
205.                             break;
206.                         }
207.                     }
208.
209.                     arg0.func.Dispose();
210.                 }
211.
212.                 return 0;
213.             }
214.             catch(Exception e)
215.             {
216.                 return LuaDLL.toluaL_exception(L, e);
217.             }
218.         }
219.
220.         [MonoPInvokeCallbackAttribute(typeof(LuaCSFunction))]
221.         static int TestEventListener_OnClick(IntPtr L)
222.         {
223.             try
224.             {
```

```
225.                 LuaFunction func = ToLua.CheckLuaFunction(L, 1);
226.                 Delegate arg1 =
    DelegateFactory.CreateDelegate(typeof(TestEventListener.OnClick), func);
227.                 ToLua.Push(L, arg1);
228.                 return 1;
229.             }
230.             catch(Exception e)
231.             {
232.                 return LuaDLL.toluaL_exception(L, e);
233.             }
234.         }
235.
236.         [MonoPInvokeCallbackAttribute(typeof(LuaCSFunction))]
237.         static int TestEventListener_VoidDelegate(IntPtr L)
238.         {
239.             try
240.             {
241.                 LuaFunction func = ToLua.CheckLuaFunction(L, 1);
242.                 Delegate arg1 =
    DelegateFactory.CreateDelegate(typeof(TestEventListener.VoidDelegate), func);
243.                 ToLua.Push(L, arg1);
244.                 return 1;
245.             }
246.             catch(Exception e)
247.             {
248.                 return LuaDLL.toluaL_exception(L, e);
249.             }
250.         }
251.     }
```

?

TestGameObject.cs

```
using UnityEngine;
using System.Collections;
using LuaInterface;

public class TestGameObject: MonoBehaviour
{
    private string script =
        @"
            local Color = UnityEngine.Color
            local GameObject = UnityEngine.GameObject
            local ParticleSystem = UnityEngine.ParticleSystem

            function OnComplete()
                print('OnComplete CallBack')
            end

            local go = GameObject('go', typeof(UnityEngine.Camera))
            go:AddComponent(typeof(ParticleSystem))
            local node = go.transform
            node.position = Vector3.one
            print('gameObject is: '..tostring(go))
            --go.transform:DOPath({Vector3.zero, Vector3.one * 10}, 1,
DG.Tweening.PathType.Linear, DG.Tweening.PathMode.Full3D, 10, nil)
            --go.transform:DORotate(Vector3(0,0,360), 2,
DG.Tweening.RotateMode.FastBeyond360):OnComplete(OnComplete)
            GameObject.Destroy(go, 2)
            print('delay destroy gameobject is: '..go.name)
        ";

    LuaState lua = null;

    void Start()
    {
        lua = new LuaState();
        lua.LogGC = true;
        lua.Start();
        LuaBinder.Bind(lua);
        lua.DoString(script, "TestGameObject.cs");
    }
```

```
38.
39.         void Update()
40.         {
41.             lua.CheckTop();
42.             lua.Collect();
43.         }
44.
45.         void OnApplicationQuit()
46.         {
47.             lua.Dispose();
48.             lua = null;
49.         }
50.     }
```

?

TestCustomLoader.cs

```
1.      using UnityEngine;
2.      using System.IO;
3.      using LuaInterface;
4.
5.      //use menu Lua->Copy lua files to Resources. 之后才能发布到手机
6.      public class TestCustomLoader : LuaClient
7.      {
8.          string tips = "Test custom loader";
9.
10.         protected override LuaFileUtils InitLoader()
11.         {
12.             return new LuaResLoader();
13.         }
14.
15.         protected override void CallMain()
16.         {
17.             LuaFunction func = luaState.GetFunction("Test");
18.             func.Call();
19.             func.Dispose();
20.         }
21.
22.         protected override void StartMain()
23.         {
24.             luaState.DoFile("TestLoader.lua");
25.             CallMain();
26.         }
27.
28.         new void Awake()
29.         {
30.     #if UNITY_5
31.             Application.logMessageReceived += Logger;
32.     #else
33.             Application.RegisterLogCallback(Logger);
34.     #endif
35.             base.Awake();
36.         }
37.
38.         new void OnApplicationQuit()
39.         {
```

```
            base.OnApplicationQuit();

    #if UNITY_5
            Application.logMessageReceived -= Logger;
    #else
            Application.RegisterLogCallback(null);
    #endif
        }

        void Logger(string msg, string stackTrace, LogType type)
        {
            tips += msg;
            tips += "\r\n";
        }

        void OnGUI()
        {
            GUI.Label(new Rect(Screen.width / 2 - 200, Screen.height / 2 - 200, 400, 400), tips);
        }
    }
```

TestLoader.lua.bytes

```
    print("This is a script from a utf8 file")
    print("tolua: 你好! こんにちは! 안녕하세요!")

    function Test()
        print("this is lua file load by Resource.Load")
    end
```

?

TestOutArg.cs

```
1.      using UnityEngine;
2.      using System.Collections;
3.      using LuaInterface;
4.      using System;
5.
6.      public class TestOutArg : MonoBehaviour
7.      {
8.          string script =
9.              @"
10.                 print('start')
11.                 local box = UnityEngine.BoxCollider
12.
13.                 function TestPick(ray)
14.                     local _layer = 2 ^ LayerMask.NameToLayer('Default')
15.                     local flag, hit = UnityEngine.Physics.Raycast(ray, nil,
    5000, _layer)
16.                     --local flag, hit = UnityEngine.Physics.Raycast(ray,
    RaycastHit.out, 5000, _layer)
17.
18.                     if flag then
19.                         print('pick from lua, point: '..tostring(hit.point))
20.                     end
21.                 end
22.             ";
23.
24.         LuaState state = null;
25.         LuaFunction func = null;
26.
27.         void Start ()
28.         {
29.             new LuaResLoader();
30.             state = new LuaState();
31.             LuaBinder.Bind(state);
32.             state.Start();
33.             state.DoString(script, "TestOutArg.cs");
34.
35.             func = state.GetFunction("TestPick");
36.         }
37.
```

```
38.         void Update()
39.         {
40.             if (Input.GetMouseButtonDown(0))
41.             {
42.                 Camera camera = Camera.main;
43.                 Ray ray = camera.ScreenPointToRay(Input.mousePosition);
44.                 RaycastHit hit;
45.                 bool flag = Physics.Raycast(ray, out hit, 5000, 1 <<
    LayerMask.NameToLayer("Default"));
46.
47.                 if (flag)
48.                 {
49.                     Debugger.Log("pick from c#, point: [{0}, {1}, {2}]",
    hit.point.x, hit.point.y, hit.point.z);
50.                 }
51.
52.                 func.BeginPCall();
53.                 func.Push(ray);
54.                 func.PCall();
55.                 func.EndPCall();
56.             }
57.
58.             state.CheckTop();
59.             state.Collect();
60.         }
61.
62.         void OnDestroy()
63.         {
64.             func.Dispose();
65.             func = null;
66.
67.             state.Dispose();
68.             state = null;
69.         }
70.     }
```

?

TestProtoBuffer.cs

```
1.  //#define USE_PROTOBUF_NET
2.  using UnityEngine;
3.  using System.Collections;
4.  using LuaInterface;
5.  using System;
6.  using System.IO;
7.
8.  #if USE_PROTOBUF_NET
9.  using ProtoBuf;
10.
11. [ProtoContract]
12. class Person
13. {
14.     [ProtoMember(1, IsRequired = true)]
15.     public int id { get; set; }
16.
17.     [ProtoMember(2, IsRequired = true)]
18.     public string name { get; set; }
19.
20.     [ProtoMember(3, IsRequired = false)]
21.     public string email { get; set; }
22. }
23.
24. #endif
25.
26. public class TestProtoBuffer : LuaClient
27. {
28.     private string script = @"
29.         local person_pb = require 'Protol.person_pb'
30.
31.         function Decoder()
32.             local msg = person_pb.Person()
33.             msg:ParseFromString(TestProtol.data)
34.             print('person_pb decoder: '..tostring(msg))
35.         end
36.
37.         function Encoder()
38.             local msg = person_pb.Person()
39.             msg.id = 1024
```

```
                msg.name = 'foo'
                msg.email = 'bar'
                local pb_data = msg:SerializeToString()
                TestProtol.data = pb_data
            end
            ";

        private string tips = "";

        //实际应用如Socket.Send(LuaStringBuffer buffer)函数发送协议, 在lua中调用Socket.Send(pb_data)
        //读取协议 Socket.PeekMsgPacket() {return MsgPacket}; lua 中, 取协议字节流MsgPack.data 为 LuaStringBuffer类型
        //msg = Socket.PeekMsgPacket()
        //pb_data = msg.data
        new void Awake()
        {
    #if UNITY_5
            Application.logMessageReceived += ShowTips;
    #else
            Application.RegisterLogCallback(ShowTips);
    #endif
            base.Awake();
        }

        protected override LuaFileUtils InitLoader()
        {
            return new LuaResLoader();
        }

        protected override void Bind()
        {
            base.Bind();

            luaState.BeginModule(null);
            TestProtolWrap.Register(luaState);
            luaState.EndModule();
        }

        //屏蔽, 例子不需要运行
        protected override void CallMain() { }

```

```
80.         protected override void OnLoadFinished()
81.         {
82.             base.OnLoadFinished();
83.             luaState.DoString(script, "TestProtoBuffer.cs");
84.
85.     #if !USE_PROTOBUF_NET
86.             LuaFunction func = luaState.GetFunction("Encoder");
87.             func.Call();
88.             func.Dispose();
89.
90.             func = luaState.GetFunction("Decoder");
91.             func.Call();
92.             func.Dispose();
93.             func = null;
94.     #else
95.             Person data = new Person();
96.             data.id = 2048;
97.             data.name = "foo";
98.             data.email = "bar";
99.             MemoryStream stream = new MemoryStream();
100.            Serializer.Serialize<Person>(stream, data);
101.            byte[] buffer = stream.ToArray();
102.
103.            TestProtol.data = new LuaByteBuffer(buffer);
104.
105.            LuaFunction func = luaState.GetFunction("Decoder");
106.            func.Call();
107.            func.Dispose();
108.            func = null;
109.
110.            func = luaState.GetFunction("Encoder");
111.            func.Call();
112.            func.Dispose();
113.            func = null;
114.
115.            stream = new MemoryStream(TestProtol.data.buffer);
116.            data = Serializer.Deserialize<Person>(stream);
117.            Debugger.Log("Decoder from lua fixed64 is: {0}", data.id);
118.    #endif
119.        }
120.
121.        void ShowTips(string msg, string stackTrace, LogType type)
```

```
        {
            tips = tips + msg + "\r\n";
        }

        void OnGUI()
        {
            GUI.Label(new Rect(Screen.width / 2 - 200, Screen.height / 2 - 100, 400, 300), tips);
        }

        new void OnApplicationQuit()
        {
            base.Destroy();
    #if UNITY_5
            Application.logMessageReceived -= ShowTips;
    #else
            Application.RegisterLogCallback(null);
    #endif
        }
    }
```

TestProtol.cs

```
    using System;
    using LuaInterface;

    public static class TestProtol
    {
        [LuaByteBufferAttribute]
        public static byte[] data;
    }
```

TestProtolWrap.cs

```
    //this source code was auto-generated by tolua#, do not modify it
    using System;
    using LuaInterface;

```

```
5.      public class TestProtolWrap
6.      {
7.          public static void Register(LuaState L)
8.          {
9.              L.BeginStaticLibs("TestProtol");
10.             L.RegVar("data", get_data, set_data);
11.             L.EndStaticLibs();
12.         }
13.
14.         [MonoPInvokeCallbackAttribute(typeof(LuaCSFunction))]
15.         static int get_data(IntPtr L)
16.         {
17.             try
18.             {
19.                 LuaDLL.tolua_pushlstring(L, TestProtol.data,
    TestProtol.data.Length);
20.                 return 1;
21.             }
22.             catch(Exception e)
23.             {
24.                 return LuaDLL.toluaL_exception(L, e);
25.             }
26.         }
27.
28.         [MonoPInvokeCallbackAttribute(typeof(LuaCSFunction))]
29.         static int set_data(IntPtr L)
30.         {
31.             try
32.             {
33.                 byte[] arg0 = ToLua.CheckByteBuffer(L, 2);
34.                 TestProtol.data = arg0;
35.                 return 0;
36.             }
37.             catch(Exception e)
38.             {
39.                 return LuaDLL.toluaL_exception(L, e);
40.             }
41.         }
42.     }
```

person.proto

```
1.      message Person {
2.        required int32 id = 1;
3.        required string name = 2;
4.        optional string email = 3;
5.
6.        extensions 10 to max;
7.      }
8.
9.      message Phone {
10.         extend Person { repeated Phone phones = 10;}
11.         enum PHONE_TYPE{
12.             MOBILE = 1;
13.             HOME = 2;
14.         }
15.         optional string num = 1;
16.         optional PHONE_TYPE type = 2;
17.     }
```

?

TestInt64.cs

```
1.  using UnityEngine;
2.  using System.Collections;
3.  using System;
4.  using LuaInterface;
5.  using System.Collections.Generic;
6.
7.
8.  public class TestInt64 : MonoBehaviour
9.  {
10.     private string tips = "";
11.
12.     string script =
13.         @"
14.             function TestInt64(x)
15.                 x = x + 789
16.                 assert(tostring(x) == '92233720368547758 07')
17.                 local low, high = int64.tonum2(x)
18.                 print('x value is: '..tostring(x)..' low is: '.. low .. '
    high is: '..high.. ' type is: '.. tolua.typename(x))
19.                 local y = int64.new(1,2)
20.                 local z = int64.new(1,2)
21.
22.                 if y == z then
23.                     print('int64 equals is ok, value: '..int64.tostring(y))
24.                 end
25.
26.                 x = int64.new(123)
27.
28.                 if int64.equals(x, 123) then
29.                     print('int64 equals to number ok')
30.                 else
31.                     print('int64 equals to number failed')
32.                 end
33.
34.                 x = int64.new('78962871035984074')
35.                 print('int64 is: '..tostring(x))
36.
37.                 local str = tostring(int64.new(3605690779, 30459971))
38.                 local n2 = int64.new(str)
```

```
                    local l, h = int64.tonum2(n2)
                    print(str..':'..tostring(n2)..' low:'..l..' high:'..h)

                    print('----------------------------uint64-----------------
    -----------')
                    x = uint64.new('18446744073709551615')
                    print('uint64 max is: '..tostring(x))
                    l, h = uint64.tonum2(x)
                    str = tostring(uint64.new(l, h))
                    print(str..':'..tostring(x)..' low:'..l..' high:'..h)

                    return y
                end
            ";


        void Start()
        {
    #if UNITY_5
            Application.logMessageReceived += ShowTips;
    #else
            Application.RegisterLogCallback(ShowTips);
    #endif
            new LuaResLoader();
            LuaState lua = new LuaState();
            lua.Start();
            lua.DoString(script, "TestInt64.cs");

            LuaFunction func = lua.GetFunction("TestInt64");
            func.BeginPCall();
            func.Push(9223372036854775807 - 789);
            func.PCall();
            long n64 = func.CheckLong();
            Debugger.Log("int64 return from lua is: {0}", n64);
            func.EndPCall();
            func.Dispose();
            func = null;

            lua.CheckTop();
            lua.Dispose();
            lua = null;
        }
```

```
80.
81.         void ShowTips(string msg, string stackTrace, LogType type)
82.         {
83.             tips += msg;
84.             tips += "\r\n";
85.         }
86.
87.         void OnDestroy()
88.         {
89.     #if UNITY_5
90.             Application.logMessageReceived -= ShowTips;
91.     #else
92.             Application.RegisterLogCallback(null);
93.     #endif
94.         }
95.
96.         void OnGUI()
97.         {
98.             GUI.Label(new Rect(Screen.width / 2 - 200, Screen.height / 2 - 150,
    400, 300), tips);
99.         }
100.    }
```

?

TestInherit.cs

```
using UnityEngine;
using System.Collections;
using LuaInterface;

public class TestInherit : MonoBehaviour
{
    private string script =
    @"  LuaTransform =
        {
        }

        function LuaTransform.Extend(u)
            local t = {}
            local _position = u.position
            tolua.setpeer(u, t)

            t.__index = t
            local get = tolua.initget(t)
            local set = tolua.initset(t)

            local _base = u.base

            --重写同名属性获取
            get.position = function(self)
                return _position
            end

            --重写同名属性设置
            set.position = function(self, v)
                if _position ~= v then
                    _position = v
                    _base.position = v
                end
            end

            --重写同名函数
            function t:Translate(...)
                print('child Translate')
                _base:Translate(...)
```

```
                end

                return u
            end


            --既保证支持继承函数，又支持go.transform == transform 这样的比较
            function Test(node)
                local v = Vector3.one
                local transform = LuaTransform.Extend(node)
                --local transform = node

                local t = os.clock()
                for i = 1, 200000 do
                    transform.position = transform.position
                end
                print('LuaTransform get set cost', os.clock() - t)

                transform:Translate(1,1,1)

                local child = transform:FindChild('child')
                print('child is: ', tostring(child))

                if child.parent == transform then
                    print('LuaTransform compare to userdata transform is ok')
                end

                transform.xyz = 123
                transform.xyz = 456
                print('extern field xyz is: '.. transform.xyz)
            end
            ";

        LuaState lua = null;

        void Start ()
        {
            lua = new LuaState();
            lua.Start();
            LuaBinder.Bind(lua);
            lua.DoString(script, "TestInherit.cs");

```

```
82.             float time = Time.realtimeSinceStartup;
83.
84.             for (int i = 0; i < 200000; i++)
85.             {
86.                 Vector3 v = transform.position;
87.                 transform.position = v;
88.             }
89.
90.             time = Time.realtimeSinceStartup - time;
91.             Debugger.Log("c# Transform get set cost time: " + time);
92.
93.             LuaFunction func = lua.GetFunction("Test");
94.             func.BeginPCall();
95.             func.Push(transform);
96.             func.PCall();
97.             func.EndPCall();
98.
99.             lua.CheckTop();
100.            lua.Dispose();
101.            lua = null;
102.        }
103.    }
```

?

TestABLoader.cs

```
1.  using UnityEngine;
2.  using System.Collections;
3.  using System.Collections.Generic;
4.  using System.IO;
5.  using LuaInterface;
6.  using System;
7.
8.  //click Lua/Build lua bundle
9.  public class TestABLoader : MonoBehaviour
10. {
11.     int bundleCount = int.MaxValue;
12.     string tips = null;
13.
14.     IEnumerator CoLoadBundle(string name, string path)
15.     {
16.         using (WWW www = new WWW(path))
17.         {
18.             if (www == null)
19.             {
20.                 Debugger.LogError(name + " bundle not exists");
21.                 yield break;
22.             }
23.
24.             yield return www;
25.
26.             if (www.error != null)
27.             {
28.                 Debugger.LogError(string.Format("Read {0} failed: {1}",
    path, www.error));
29.                 yield break;
30.             }
31.
32.             --bundleCount;
33.             LuaFileUtils.Instance.AddSearchBundle(name, www.assetBundle);
34.             www.Dispose();
35.         }
36.     }
37.
38.     IEnumerator LoadFinished()
```

```
        {
            while (bundleCount > 0)
            {
                yield return null;
            }

            OnBundleLoad();
        }

        public IEnumerator LoadBundles()
        {
            string streamingPath = Application.streamingAssetsPath.Replace('\\', '/');

    #if UNITY_5
    #if UNITY_ANDROID && !UNITY_EDITOR
            string main = streamingPath + "/" + LuaConst.osDir + "/" + LuaConst.osDir;
    #else
            string main = "file:///" + streamingPath + "/" + LuaConst.osDir + "/" + LuaConst.osDir;
    #endif
            WWW www = new WWW(main);
            yield return www;

            AssetBundleManifest manifest = (AssetBundleManifest)www.assetBundle.LoadAsset("AssetBundleManifest");
            List<string> list = new List<string>(manifest.GetAllAssetBundles());
    #else
            //此处应该配表获取
            List<string> list = new List<string>() { "lua.unity3d", "lua_cjson.unity3d", "lua_system.unity3d", "lua_unityengine.unity3d", "lua_protobuf.unity3d", "lua_misc.unity3d", "lua_socket.unity3d", "lua_system_reflection.unity3d" };
    #endif
            bundleCount = list.Count;

            for (int i = 0; i < list.Count; i++)
            {
                string str = list[i];

```

```
73.     #if UNITY_ANDROID && !UNITY_EDITOR
74.                 string path = streamingPath + "/" + LuaConst.osDir + "/" + str;
75.     #else
76.                 string path = "file:///" + streamingPath + "/" + LuaConst.osDir
    + "/" + str;
77.     #endif
78.                 string name = Path.GetFileNameWithoutExtension(str);
79.                 StartCoroutine(CoLoadBundle(name, path));
80.             }
81.
82.             yield return StartCoroutine(LoadFinished());
83.         }
84.
85.         void Awake()
86.         {
87.     #if UNITY_5
88.             Application.logMessageReceived += ShowTips;
89.     #else
90.             Application.RegisterLogCallback(ShowTips);
91.     #endif
92.             LuaFileUtils file = new LuaFileUtils();
93.             file.beZip = true;
94.             StartCoroutine(LoadBundles());
95.         }
96.
97.         void ShowTips(string msg, string stackTrace, LogType type)
98.         {
99.             tips += msg;
100.            tips += "\r\n";
101.        }
102.
103.        void OnGUI()
104.        {
105.            GUI.Label(new Rect(Screen.width / 2 - 200, Screen.height / 2 - 150,
    400, 300), tips);
106.        }
107.
108.        void OnApplicationQuit()
109.        {
110.    #if UNITY_5
111.            Application.logMessageReceived -= ShowTips;
112.    #else
```

```
            Application.RegisterLogCallback(null);
    #endif
        }

        void OnBundleLoad()
        {
            LuaState state = new LuaState();
            state.Start();
            state.DoString("print('hello tolua#:'..tostring(Vector3.zero))");
            state.DoFile("Main.lua");
            LuaFunction func = state.GetFunction("Main");
            func.Call();
            func.Dispose();
            state.Dispose();
            state = null;
        }
    }
```

?

TestCJson.cs

```
1.      using UnityEngine;
2.      using System.Collections;
3.      using LuaInterface;
4.
5.      public class TestCJson : LuaClient
6.      {
7.          string script = @"
8.          local json = require 'cjson'
9.
10.         function Test(str)
11.             local data = json.decode(str)
12.             print(data.glossary.title)
13.             s = json.encode(data)
14.             print(s)
15.         end
16.     ";
17.         protected override LuaFileUtils InitLoader()
18.         {
19.             return new LuaResLoader();
20.         }
21.
22.         protected override void OpenLibs()
23.         {
24.             base.OpenLibs();
25.             OpenCJson();
26.         }
27.
28.         protected override void OnLoadFinished()
29.         {
30.             base.OnLoadFinished();
31.
32.             TextAsset text = (TextAsset)Resources.Load("jsonexample",
    typeof(TextAsset));
33.             string str = text.ToString();
34.             luaState.DoString(script);
35.             LuaFunction func = luaState.GetFunction("Test");
36.             func.BeginPCall();
37.             func.Push(str);
38.             func.PCall();
```

```
            func.EndPCall();
            func.Dispose();
        }

        //屏蔽，例子不需要运行
        protected override void CallMain() { }
    }
```

?

TestUTF8.cs

```
using UnityEngine;
using LuaInterface;

public class TestUTF8 : LuaClient
{
    string script =
@"
    local utf8 = utf8

    function Test()
        local l1 = utf8.len('你好')
        local l2 = utf8.len('こんにちは')
        print('chinese string len is: '..l1..' japanese len: '..l2)

        local s = '遍历字符串'

        for i in utf8.byte_indices(s) do
            local next = utf8.next(s, i)
            print(s:sub(i, next and next -1))
        end

        local s1 = '天下风云出我辈'
        print('风云 count is: '..utf8.count(s1, '风云'))
        s1 = s1:gsub('风云', '風雲')

        local function replace(s, i, j, repl_char)
            if s:sub(i, j) == '辈' then
                return repl_char
            end
        end

        print(utf8.replace(s1, replace, '輩'))
    end
";

    protected override LuaFileUtils InitLoader()
    {
        return new LuaResLoader();
    }
```

```
40.
41.         //屏蔽，例子不需要运行
42.         protected override void CallMain() { }
43.
44.         protected override void OnLoadFinished()
45.         {
46.             base.OnLoadFinished();
47.             luaState.DoString(script);
48.             LuaFunction func = luaState.GetFunction("Test");
49.             func.Call();
50.             func.Dispose();
51.             func = null;
52.         }
53.     }
```

?

TestString.cs

```
1.  using UnityEngine;
2.  using System.Collections;
3.  using LuaInterface;
4.  using System;
5.  using System.Reflection;
6.  using System.Text;
7.
8.  public class TestString : LuaClient
9.  {
10.     string script =
11. @"
12.     function Test()
13.         local str = System.String.New('男儿当自强')
14.         local index = str:IndexOfAny('儿自')
15.         print('and index is: '..index)
16.         local buffer = str:ToCharArray()
17.         print('str type is: '..type(str)..' buffer[0] is ' .. buffer[0])
18.         local luastr = tolua.tolstring(buffer)
19.         print('lua string is: '..luastr..' type is: '..type(luastr))
20.         luastr = tolua.tolstring(str)
21.         print('lua string is: '..luastr)
22.     end
23. ";
24.
25.     protected override LuaFileUtils InitLoader()
26.     {
27.         return new LuaResLoader();
28.     }
29.
30.     //屏蔽，例子不需要运行
31.     protected override void CallMain() { }
32.
33.     protected override void OnLoadFinished()
34.     {
35.         base.OnLoadFinished();
36.         luaState.DoString(script);
37.         LuaFunction func = luaState.GetFunction("Test");
38.         func.Call();
39.         func.Dispose();
```

```
40.             func = null;
41.         }
42.     }
```

?

TestReflection.cs

```
1.      using UnityEngine;
2.      using System.Collections.Generic;
3.      using LuaInterface;
4.      using System;
5.      using System.Reflection;
6.
7.
8.      public class TestReflection : LuaClient
9.      {
10.         string script =
11.     @"
12.         require 'tolua.reflection'
13.         tolua.loadassembly('Assembly-CSharp')
14.         tolua.loadassembly('mscorlib')
15.         local BindingFlags = require 'System.Reflection.BindingFlags'
16.
17.         function DoClick()
18.             print('do click')
19.         end
20.
21.         function Test()
22.             local t = typeof('TestExport')
23.             local func = tolua.getmethod(t, 'TestReflection')
24.             func:Call()
25.             func:Destroy()
26.             func = nil
27.
28.             local objs = {Vector3.one, Vector3.zero}
29.             local array = tolua.toarray(objs, typeof(Vector3))
30.             local obj = tolua.createinstance(t, array)
31.             --local constructor = tolua.getconstructor(t,
    typeof(Vector3):MakeArrayType())
32.             --local obj = constructor:Call(array)
33.             --constructor:Destroy()
34.
35.             func = tolua.getmethod(t, 'Test',
    typeof('System.Int32'):MakeByRefType())
36.             local r, o = func:Call(obj, 123)
37.             print(r..':'..o)
```

```
            func:Destroy()

            local property = tolua.getproperty(t, 'Number')
            local num = property:Get(obj, null)
            print('object Number: '..num)
            property:Set(obj, 456, null)
            num = property:Get(obj, null)
            property:Destroy()
            print('object Number: '..num)

            local field = tolua.getfield(t, 'field')
            num = field:Get(obj)
            print('object field: '.. num)
            field:Set(obj, 2048)
            num = field:Get(obj)
            field:Destroy()
            print('object field: '.. num)

            field = tolua.getfield(t, 'OnClick')
            local onClick = field:Get(obj)
            onClick = onClick + DoClick
            field:Set(obj, onClick)
            local click = field:Get(obj)
            click:DynamicInvoke()
            field:Destroy()
            click:Destroy()
        end
    ";

        string tips = null;

        protected override LuaFileUtils InitLoader()
        {
    #if UNITY_5
            Application.logMessageReceived += ShowTips;
    #else
            Application.RegisterLogCallback(ShowTips);
    #endif
            return new LuaResLoader();
        }

        //屏蔽，例子不需要运行
```

```
80.         protected override void CallMain() { }
81.
82.         void TestAction()
83.         {
84.             Debugger.Log("Test Action");
85.         }
86.
87.         protected override void OnLoadFinished()
88.         {
89.             base.OnLoadFinished();
90.
91.             /*Type t = typeof(TestExport);
92.             MethodInfo md = t.GetMethod("TestReflection");
93.             md.Invoke(null, null);
94.
95.             Vector3[] array = new Vector3[] { Vector3.zero, Vector3.one };
96.             object obj = Activator.CreateInstance(t, array);
97.             md = t.GetMethod("Test", new Type[] { typeof(int).MakeByRefType()
    });
98.             object o = 123;
99.             object[] args = new object[] { o };
100.            object ret = md.Invoke(obj, args);
101.            Debugger.Log(ret + " : " + args[0]);
102.
103.            PropertyInfo p = t.GetProperty("Number");
104.            int num = (int)p.GetValue(obj, null);
105.            Debugger.Log("object Number: {0}", num);
106.            p.SetValue(obj, 456, null);
107.            num = (int)p.GetValue(obj, null);
108.            Debugger.Log("object Number: {0}", num);
109.
110.            FieldInfo f = t.GetField("field");
111.            num = (int)f.GetValue(obj);
112.            Debugger.Log("object field: {0}", num);
113.            f.SetValue(obj, 2048);
114.            num = (int)f.GetValue(obj);
115.            Debugger.Log("object field: {0}", num);*/
116.
117.            luaState.CheckTop();
118.            luaState.DoString(script, "TestReflection.cs");
119.            LuaFunction func = luaState.GetFunction("Test");
120.            func.Call();
```

```
121.             func.Dispose();
122.             func = null;
123.         }
124.
125.         void ShowTips(string msg, string stackTrace, LogType type)
126.         {
127.             tips += msg;
128.             tips += "\r\n";
129.         }
130.
131.         new void OnApplicationQuit()
132.         {
133.     #if UNITY_5
134.             Application.logMessageReceived += ShowTips;
135.     #else
136.             Application.RegisterLogCallback(ShowTips);
137.     #endif
138.             Destroy();
139.         }
140.
141.         void OnGUI()
142.         {
143.             GUI.Label(new Rect(Screen.width / 2 - 250, Screen.height / 2 - 150,
      500, 300), tips);
144.         }
145.     }
```

?

UseList

```
using UnityEngine;
using System.Collections;
using LuaInterface;
using System.Collections.Generic;
using System;

//需要导出委托类型如下：
//System.Predicate<int>
//System.Action<int>
//System.Comparison<int>
public class UseList : LuaClient
{
    private string script =
        @"
            function Exist2(v)
                return v == 2
            end

            function IsEven(v)
                return v % 2 == 0
            end

            function NotExist(v)
                return false
            end

            function Compare(a, b)
                if a > b then
                    return 1
                elseif a == b then
                    return 0
                else
                    return -1
                end
            end

            function Test(list, list1)
                list:Add(123)
                print('Add result: list[0] is '..list[0])
```

```
40.                     list:AddRange(list1)
41.                     print(string.format('AddRange result: list[1] is %d,
    list[2] is %d', list[1], list[2]))
42.
43.                     local const = list:AsReadOnly()
44.                     print('AsReadOnley:'..const[0])
45.
46.                     index = const:IndexOf(123)
47.
48.                     if index == 0 then
49.                         print('const IndexOf is ok')
50.                     end
51.
52.                     local pos = list:BinarySearch(1)
53.                     print('BinarySearch 1 result is: '..pos)
54.
55.                     if list:Contains(123) then
56.                         print('list Contain 123')
57.                     else
58.                         error('list Contains result fail')
59.                     end
60.
61.                     if list:Exists(Exist2) then
62.                         print('list exists 2')
63.                     else
64.                         error('list exists result fail')
65.                     end
66.
67.                     if list:Find(Exist2) then
68.                         print('list Find is ok')
69.                     else
70.                         print('list Find error')
71.                     end
72.
73.                     local fa = list:FindAll(IsEven)
74.
75.                     if fa.Count == 2 then
76.                         print('FindAll is ok')
77.                     end
78.
79.                     --注意推导后的委托声明必须注册, 这里是System.Predicate<int>
80.                     local index = list:FindIndex(System.Predicate_int(Exist2))
```

```
81.
82.                     if index == 2 then
83.                         print('FindIndex is ok')
84.                     end
85.
86.                     index = list:FindLastIndex(System.Predicate_int(Exist2))
87.
88.                     if index == 2 then
89.                         print('FindLastIndex is ok')
90.                     end
91.
92.                     index = list:IndexOf(123)
93.
94.                     if index == 0 then
95.                         print('IndexOf is ok')
96.                     end
97.
98.                     index = list:LastIndexOf(123)
99.
100.                    if index == 0 then
101.                        print('LastIndexOf is ok')
102.                    end
103.
104.                    list:Remove(123)
105.
106.                    if list[0] ~= 123 then
107.                        print('Remove is ok')
108.                    end
109.
110.                    list:Insert(0, 123)
111.
112.                    if list[0] == 123 then
113.                        print('Insert is ok')
114.                    end
115.
116.                    list:RemoveAt(0)
117.
118.                    if list[0] ~= 123 then
119.                        print('RemoveAt is ok')
120.                    end
121.
122.                    list:Insert(0, 123)
```

```
                    list:ForEach(function(v) print('foreach: '..v) end)
                    local count = list.Count

                    list:Sort(System.Comparison_int(Compare))
                    print('--------------sort list over----------------------')

                    for i = 0, count - 1 do
                        print('for:'..list[i])
                    end

                    list:Clear()
                    print('list Clear not count is '..list.Count)
                end
            ";


        protected override LuaFileUtils InitLoader()
        {
            return new LuaResLoader();
        }

        //屏蔽，例子不需要运行
        protected override void CallMain() { }

        protected override void OnLoadFinished()
        {
            base.OnLoadFinished();
            luaState.DoString(script, "UseList.cs");
            List<int> list1 = new List<int>();
            list1.Add(1);
            list1.Add(2);
            list1.Add(4);

            LuaFunction func = luaState.GetFunction("Test");
            func.BeginPCall();
            func.Push(new List<int>());
            func.Push(list1);
            func.PCall();
            func.EndPCall();
            func.Dispose();
            func = null;
        }
```

```
165.     }
```

?

TestPerformance.cs

```
1.  using System;
2.  using UnityEngine;
3.  using System.Collections.Generic;
4.  using LuaInterface;
5.
6.  public class TestPerformance : MonoBehaviour
7.  {
8.      LuaState state = null;
9.      private string tips = "";
10.
11.     void Start ()
12.     {
13. #if UNITY_5
14.         Application.logMessageReceived += ShowTips;
15. #else
16.         Application.RegisterLogCallback(ShowTips);
17. #endif
18.         new LuaResLoader();
19.         state = new LuaState();
20.         state.Start();
21.         LuaBinder.Bind(state);
22.         state.DoFile("TestPerf.lua");
23.         state.LuaGC(LuaGCOptions.LUA_GCCOLLECT);
24.         state.LogGC = false;
25.
26.         Debug.Log(typeof(List<int>).BaseType);
27.     }
28.
29.     void ShowTips(string msg, string stackTrace, LogType type)
30.     {
31.         tips += msg;
32.         tips += "\r\n";
33.     }
34.
35.     void OnApplicationQuit()
36.     {
37. #if UNITY_5
38.         Application.logMessageReceived -= ShowTips;
39. #else
```

```
40.             Application.RegisterLogCallback(null);
41.     #endif
42.             state.Dispose();
43.             state = null;
44.         }
45.
46.         void OnGUI()
47.         {
48.             GUI.Label(new Rect(Screen.width / 2 - 200, Screen.height / 2 - 100,
    400, 300), tips);
49.
50.             if (GUI.Button(new Rect(50, 50, 120, 45), "Test1"))
51.             {
52.                 float time = Time.realtimeSinceStartup;
53.
54.                 for (int i = 0; i < 200000; i++)
55.                 {
56.                     Vector3 v = transform.position;
57.                     transform.position = v + Vector3.one;
58.                 }
59.
60.                 time = Time.realtimeSinceStartup - time;
61.                 tips = "";
62.                 Debugger.Log("c# Transform getset cost time: " + time);
63.                 transform.position = Vector3.zero;
64.
65.                 LuaFunction func = state.GetFunction("Test1");
66.                 func.BeginPCall();
67.                 func.Push(transform);
68.                 func.PCall();
69.                 func.EndPCall();
70.                 func.Dispose();
71.                 func = null;
72.             }
73.             else if (GUI.Button(new Rect(50, 150, 120, 45), "Test2"))
74.             {
75.                 float time = Time.realtimeSinceStartup;
76.
77.                 for (int i = 0; i < 200000; i++)
78.                 {
79.                     transform.Rotate(Vector3.up, 1);
80.                 }
```

```
81.
82.                 time = Time.realtimeSinceStartup - time;
83.                 tips = "";
84.                 Debugger.Log("c# Transform.Rotate cost time: " + time);
85.
86.                 LuaFunction func = state.GetFunction("Test2");
87.                 func.BeginPCall();
88.                 func.Push(transform);
89.                 func.PCall();
90.                 func.EndPCall();
91.                 func.Dispose();
92.                 func = null;
93.             }
94.             else if (GUI.Button(new Rect(50, 250, 120, 45), "Test3"))
95.             {
96.                 float time = Time.realtimeSinceStartup;
97.
98.                 for (int i = 0; i < 2000000; i++)
99.                 {
100.                    new Vector3(i, i, i);
101.                }
102.
103.                time = Time.realtimeSinceStartup - time;
104.                tips = "";
105.                Debugger.Log("c# new Vector3 cost time: " + time);
106.
107.                LuaFunction func = state.GetFunction("Test3");
108.                func.Call();
109.                func.Dispose();
110.                func = null;
111.            }
112.            else if (GUI.Button(new Rect(50, 350, 120, 45), "Test4"))
113.            {
114.                float time = Time.realtimeSinceStartup;
115.
116.                for (int i = 0; i < 20000; i++)
117.                {
118.                    new GameObject();
119.                }
120.
121.                time = Time.realtimeSinceStartup - time;
122.                tips = "";
```

```
123.                 Debugger.Log("c# new GameObject cost time: " + time);
124.
125.                 //光gc了
126.                 LuaFunction func = state.GetFunction("Test4");
127.                 func.Call();
128.                 func.Dispose();
129.                 func = null;
130.             }
131.             else if (GUI.Button(new Rect(50, 450, 120, 45), "Test5"))
132.             {
133.                 int[] array = new int[1024];
134.
135.                 for (int i = 0; i < 1024; i++)
136.                 {
137.                     array[i] = i;
138.                 }
139.
140.                 float time = Time.realtimeSinceStartup;
141.                 int total = 0;
142.
143.                 for (int j = 0; j < 100000; j++)
144.                 {
145.                     for (int i = 0; i < 1024; i++)
146.                     {
147.                         total += array[i];
148.                     }
149.                 }
150.
151.                 time = Time.realtimeSinceStartup - time;
152.                 tips = "";
153.                 Debugger.Log("Array cost time: " + time);
154.
155.                 List<int> list = new List<int>(array);
156.                 time = Time.realtimeSinceStartup;
157.                 total = 0;
158.
159.                 for (int j = 0; j < 100000; j++)
160.                 {
161.                     for (int i = 0; i < 1024; i++)
162.                     {
163.                         total += list[i];
164.                     }
```

```
                }

                time = Time.realtimeSinceStartup - time;
                tips = "";
                Debugger.Log("Array cost time: " + time);

                LuaFunction func = state.GetFunction("TestTable");
                func.Call();
                func.Dispose();
                func = null;
            }
            else if (GUI.Button(new Rect(50, 550, 120, 40), "Test7"))
            {
                float time = Time.realtimeSinceStartup;
                Vector3 v1 = Vector3.zero;

                for (int i = 0; i < 200000; i++)
                {
                    Vector3 v = new Vector3(i,i,i);
                    v = Vector3.Normalize(v);
                    v1 = v + v1;
                }

                time = Time.realtimeSinceStartup - time;
                tips = "";
                Debugger.Log("Vector3 New Normalize cost: " + time);
                LuaFunction func = state.GetFunction("Test7");
                func.Call();
                func.Dispose();
                func = null;
            }
            else if (GUI.Button(new Rect(250, 50, 120, 40), "Test8"))
            {
                float time = Time.realtimeSinceStartup;

                for (int i = 0; i < 200000; i++)
                {
                    Quaternion q1 = Quaternion.Euler(i, i, i);
                    Quaternion q2 = Quaternion.Euler(i * 2, i * 2, i * 2);
                    Quaternion.Slerp(q1, q2, 0.5f);
                }

```

```
                time = Time.realtimeSinceStartup - time;
                tips = "";
                Debugger.Log("Quaternion Euler Slerp cost: " + time);

                LuaFunction func = state.GetFunction("Test8");
                func.Call();
                func.Dispose();
                func = null;
            }
            else if (GUI.Button(new Rect(250, 150, 120, 40), "Test9"))
            {
                tips = "";
                LuaFunction func = state.GetFunction("Test9");
                func.Call();
                func.Dispose();
                func = null;
            }
            else if (GUI.Button(new Rect(250, 250, 120, 40), "Quit"))
            {
                Application.Quit();
            }

            state.CheckTop();
            state.Collect();
        }
    }
```

?

TestLuaStack.cs

```
1.  using UnityEngine;
2.  using System.Collections;
3.  using LuaInterface;
4.  using System;
5.  using System.Runtime.InteropServices;
6.
7.  //检测合理报错
8.  public class TestLuaStack : MonoBehaviour
9.  {
10.     public GameObject go = null;
11.     public GameObject go2 = null;
12.     public static LuaFunction show = null;
13.     public static LuaFunction testRay = null;
14.     public static LuaFunction showStack = null;
15.     public static LuaFunction test4 = null;
16.
17.     private static GameObject testGo = null;
18.     private string tips = "";
19.     public static TestLuaStack Instance = null;
20.
21.     [MonoPInvokeCallbackAttribute(typeof(LuaCSFunction))]
22.     static int Test1(IntPtr L)
23.     {
24.         try
25.         {
26.             show.BeginPCall();
27.             show.PCall();
28.             show.EndPCall();
29.         }
30.         catch (Exception e)
31.         {
32.             return LuaDLL.toluaL_exception(L, e);
33.         }
34.
35.         return 0;
36.     }
37.
38.     [MonoPInvokeCallbackAttribute(typeof(LuaCSFunction))]
39.     static int PushLuaError(IntPtr L)
```

```
40.         {
41.             try
42.             {
43.                 testRay.BeginPCall();
44.                 testRay.Push(Instance);
45.                 testRay.PCall();
46.                 testRay.EndPCall();
47.             }
48.             catch (Exception e)
49.             {
50.                 return LuaDLL.toluaL_exception(L, e);
51.             }
52. 
53.             return 0;
54.         }
55. 
56.         [MonoPInvokeCallbackAttribute(typeof(LuaCSFunction))]
57.         static int Test3(IntPtr L)
58.         {
59.             try
60.             {
61.                 testRay.BeginPCall();
62.                 testRay.PCall();
63.                 testRay.CheckRay();
64.                 testRay.EndPCall();
65.             }
66.             catch (Exception e)
67.             {
68.                 return LuaDLL.toluaL_exception(L, e);
69.             }
70. 
71.             return 0;
72.         }
73. 
74.         [MonoPInvokeCallbackAttribute(typeof(LuaCSFunction))]
75.         static int Test4(IntPtr L)
76.         {
77.             try
78.             {
79.                 show.BeginPCall();
80.                 show.PCall();
81.                 show.EndPCall();
```

```
82.             }
83.             catch (Exception e)
84.             {
85.                 return LuaDLL.toluaL_exception(L, e);
86.             }
87.
88.             return 0;
89.         }
90.
91.         [MonoPInvokeCallbackAttribute(typeof(LuaCSFunction))]
92.         static int Test5(IntPtr L)
93.         {
94.             try
95.             {
96.                 test4.BeginPCall();
97.                 test4.PCall();
98.                 bool ret = test4.CheckBoolean();
99.                 ret = !ret;
100.                test4.EndPCall();
101.            }
102.            catch (Exception e)
103.            {
104.                return LuaDLL.toluaL_exception(L, e);
105.            }
106.
107.            return 0;
108.        }
109.
110.        [MonoPInvokeCallbackAttribute(typeof(LuaCSFunction))]
111.        static int Test6(IntPtr L)
112.        {
113.            try
114.            {
115.                throw new LuaException("this a lua exception");
116.            }
117.            catch (Exception e)
118.            {
119.                return LuaDLL.toluaL_exception(L, e);
120.            }
121.        }
122.
123.        [MonoPInvokeCallbackAttribute(typeof(LuaCSFunction))]
```

```
124.         static int TestOutOfBound(IntPtr L)
125.         {
126.             try
127.             {
128.                 Transform transform = testGo.transform;
129.                 Transform node = transform.GetChild(20);
130.                 ToLua.Push(L, node);
131.                 return 0;
132.             }
133.             catch (Exception e)
134.             {
135.                 return LuaDLL.toluaL_exception(L, e);
136.             }
137.         }
138.
139.         [MonoPInvokeCallbackAttribute(typeof(LuaCSFunction))]
140.         static int TestArgError(IntPtr L)
141.         {
142.             try
143.             {
144.                 LuaDLL.luaL_typerror(L, 1, "number");
145.             }
146.             catch (Exception e)
147.             {
148.                 return LuaDLL.toluaL_exception(L, e);
149.             }
150.
151.             return 0;
152.         }
153.
154.         [MonoPInvokeCallbackAttribute(typeof(LuaCSFunction))]
155.         static int TestTableInCo(IntPtr L)
156.         {
157.             try
158.             {
159.                 LuaTable table = ToLua.CheckLuaTable(L, 1);
160.                 string str = (string)table["name"];
161.                 ToLua.Push(L, str);
162.                 return 1;
163.             }
164.             catch (Exception e)
165.             {
```

```
166.                 return LuaDLL.tolua_exception(L, e);
167.             }
168.         }
169.
170.         [MonoPInvokeCallbackAttribute(typeof(LuaCSFunction))]
171.         static int TestCycle(IntPtr L)
172.         {
173.             try
174.             {
175.                 LuaState state = LuaState.Get(L);
176.                 LuaFunction func = state.GetFunction("TestCycle");
177.                 int c = (int)LuaDLL.luaL_checknumber(L, 1);
178.
179.                 if (c <= 2)
180.                 {
181.                     LuaDLL.lua_pushnumber(L, 1);
182.                 }
183.                 else
184.                 {
185.                     func.BeginPCall();
186.                     func.Push(c - 1);
187.                     func.PCall();
188.                     int n1 = (int)func.CheckNumber();
189.                     func.EndPCall();
190.
191.                     func.BeginPCall();
192.                     func.Push(c - 2);
193.                     func.PCall();
194.                     int n2 = (int)func.CheckNumber();
195.                     func.EndPCall();
196.
197.                     LuaDLL.lua_pushnumber(L, n1 + n2);
198.                 }
199.
200.                 return 1;
201.             }
202.             catch (Exception e)
203.             {
204.                 return LuaDLL.tolua_exception(L, e);
205.             }
206.         }
207.
```

```
208.         [MonoPInvokeCallbackAttribute(typeof(LuaCSFunction))]
209.         static int TestNull(IntPtr L)
210.         {
211.             try
212.             {
213.                 GameObject go = (GameObject)ToLua.CheckUnityObject(L, 1,
    typeof(GameObject));
214.                 ToLua.Push(L, go.name);
215.                 return 1;
216.             }
217.             catch (Exception e)
218.             {
219.                 return LuaDLL.toluaL_exception(L, e);
220.             }
221.         }
222.
223.         [MonoPInvokeCallbackAttribute(typeof(LuaCSFunction))]
224.         static int TestAddComponent(IntPtr L)
225.         {
226.             try
227.             {
228.                 GameObject go = (GameObject)ToLua.CheckUnityObject(L, 1,
    typeof(GameObject));
229.                 go.AddComponent<TestInstantiate2>();
230.                 return 0;
231.             }
232.             catch (Exception e)
233.             {
234.                 return LuaDLL.toluaL_exception(L, e);
235.             }
236.         }
237.
238.         static void TestMul1()
239.         {
240.             throw new Exception("multi stack error");
241.         }
242.
243.         static void TestMul0()
244.         {
245.             TestMul1();
246.         }
247.
```

```
248.         [MonoPInvokeCallbackAttribute(typeof(LuaCSFunction))]
249.         static int TestMulStack(IntPtr L)
250.         {
251.             try
252.             {
253.                 TestMul0();
254.                 return 0;
255.             }
256.             catch (Exception e)
257.             {
258.                 //Debugger.Log("xxxx" + e.StackTrace);
259.                 return LuaDLL.toluaL_exception(L, e);
260.             }
261.         }
262. 
263.         void OnSendMsg()
264.         {
265.             try
266.             {
267.                 LuaFunction func = state.GetFunction("TestStack.Test6");
268.                 func.BeginPCall();
269.                 func.PCall();
270.                 func.EndPCall();
271.             }
272.             catch(Exception e)
273.             {
274.                 state.ThrowLuaException(e);
275.             }
276.         }
277. 
278. 
279.         LuaState state = null;
280.         public TextAsset text = null;
281. 
282.         static Action TestDelegate = delegate { };
283. 
284.         void Awake()
285.         {
286.     #if UNITY_5
287.             Application.logMessageReceived += ShowTips;
288.     #else
289.             Application.RegisterLogCallback(ShowTips);
```

```
290.     #endif
291.             Instance = this;
292.             new LuaResLoader();
293.             testGo = gameObject;
294.             state = new LuaState();
295.             state.Start();
296.             LuaBinder.Bind(state);
297.
298.             state.BeginModule(null);
299.             state.RegFunction("TestArgError", TestArgError);
300.             state.RegFunction("TestTableInCo", TestTableInCo);
301.             state.RegFunction("TestCycle", TestCycle);
302.             state.RegFunction("TestNull", TestNull);
303.             state.RegFunction("TestAddComponent", TestAddComponent);
304.             state.RegFunction("TestOutOfBound", TestOutOfBound);
305.             state.RegFunction("TestMulStack", TestMulStack);
306.             state.BeginStaticLibs("TestStack");
307.             state.RegFunction("Test1", Test1);
308.             state.RegFunction("PushLuaError", PushLuaError);
309.             state.RegFunction("Test3", Test3);
310.             state.RegFunction("Test4", Test4);
311.             state.RegFunction("Test5", Test5);
312.             state.RegFunction("Test6", Test6);
313.             state.EndStaticLibs();
314.             state.EndModule();
315.
316.             //state.DoFile("TestErrorStack.lua");
317.             state.Require("TestErrorStack");
318.             show = state.GetFunction("Show");
319.             testRay = state.GetFunction("TestRay");
320.
321.             showStack = state.GetFunction("ShowStack");
322.             test4 = state.GetFunction("Test4");
323.
324.             TestDelegate += TestD1;
325.             TestDelegate += TestD2;
326.         }
327.
328.         void Update()
329.         {
330.             state.CheckTop();
331.         }
```

```
332.
333.         void OnApplicationQuit()
334.         {
335.     #if UNITY_5
336.             Application.logMessageReceived -= ShowTips;
337.     #else
338.             Application.RegisterLogCallback(null);
339.     #endif
340.             state.Dispose();
341.             state = null;
342.         }
343.
344.         void ShowTips(string msg, string stackTrace, LogType type)
345.         {
346.             tips += msg;
347.             tips += "\r\n";
348.
349.             if (type == LogType.Error || type == LogType.Exception)
350.             {
351.                 tips += stackTrace;
352.             }
353.         }
354.
355.         void TestD1()
356.         {
357.             Debugger.Log("delegate 1");
358.             TestDelegate -= TestD2;
359.         }
360.
361.         void TestD2()
362.         {
363.             Debugger.Log("delegate 2");
364.         }
365.
366.         void OnGUI()
367.         {
368.             GUI.Label(new Rect(Screen.width / 2 - 300, Screen.height / 2 - 150,
    800, 400), tips);
369.
370.             if (GUI.Button(new Rect(10, 10, 120, 40), "Error1"))
371.             {
372.                 tips = "";
```

```
373.                 showStack.BeginPCall();
374.                 showStack.Push(go);
375.                 showStack.PCall();
376.                 showStack.EndPCall();
377.                 showStack.Dispose();
378.                 showStack = null;
379.             }
380.             else if (GUI.Button(new Rect(10, 60, 120, 40), "Instantiate
    Error"))
381.             {
382.                 tips = "";
383.                 LuaFunction func = state.GetFunction("Instantiate");
384.                 func.BeginPCall();
385.                 func.Push(go);
386.                 func.PCall();
387.                 func.EndPCall();
388.                 func.Dispose();
389.             }
390.             else if (GUI.Button(new Rect(10, 110, 120, 40), "Check Error"))
391.             {
392.                 tips = "";
393.                 LuaFunction func = state.GetFunction("TestRay");
394.                 func.BeginPCall();
395.                 func.PCall();
396.                 func.CheckRay();        //返回值出错
397.                 func.EndPCall();
398.                 func.Dispose();
399.             }
400.             else if (GUI.Button(new Rect(10, 160, 120, 40), "Push Error"))
401.             {
402.                 tips = "";
403.                 LuaFunction func = state.GetFunction("TestRay");
404.                 func.BeginPCall();
405.                 func.Push(Instance);
406.                 func.PCall();
407.                 func.EndPCall();
408.                 func.Dispose();
409.             }
410.             else if (GUI.Button(new Rect(10, 210, 120, 40), "LuaPushError"))
411.             {
412.                 //需要改lua文件让其出错
413.                 tips = "";
```

```
414.                 LuaFunction func = state.GetFunction("PushLuaError");
415.                 func.BeginPCall();
416.                 func.PCall();
417.                 func.EndPCall();
418.                 func.Dispose();
419.             }
420.             else if (GUI.Button(new Rect(10, 260, 120, 40), "Check Error"))
421.             {
422.                 tips = "";
423.                 LuaFunction func = state.GetFunction("Test5");
424.                 func.BeginPCall();
425.                 func.PCall();
426.                 func.EndPCall();
427.                 func.Dispose();
428.             }
429.             else if (GUI.Button(new Rect(10, 310, 120, 40), "Test Resume"))
430.             {
431.                 tips = "";
432.                 LuaFunction func = state.GetFunction("Test6");
433.                 func.BeginPCall();
434.                 func.PCall();
435.                 func.EndPCall();
436.                 func.Dispose();
437.             }
438.             else if (GUI.Button(new Rect(10, 360, 120, 40), "out of bound"))
439.             {
440.                 tips = "";
441.                 LuaFunction func = state.GetFunction("TestOutOfBound");
442.                 func.BeginPCall();
443.                 func.PCall();
444.                 func.EndPCall();
445.                 func.Dispose();
446.             }
447.             else if (GUI.Button(new Rect(10, 410, 120, 40), "TestArgError"))
448.             {
449.                 tips = "";
450.                 LuaFunction func = state.GetFunction("Test8");
451.                 func.BeginPCall();
452.                 func.PCall();
453.                 func.EndPCall();
454.                 func.Dispose();
455.             }
```

```
456.             else if (GUI.Button(new Rect(10, 460, 120, 40), "TestFuncDispose"))
457.             {
458.                 tips = "";
459.                 LuaFunction func = state.GetFunction("Test8");
460.                 func.Dispose();
461.                 func.BeginPCall();
462.                 func.PCall();
463.                 func.EndPCall();
464.                 func.Dispose();
465.             }
466.             else if (GUI.Button(new Rect(10, 510, 120, 40), "SendMessage"))
467.             {
468.                 tips = "";
469.                 gameObject.SendMessage("OnSendMsg");
470.             }
471.             else if (GUI.Button(new Rect(10, 560, 120, 40),
    "SendMessageInLua"))
472.             {
473.                 LuaFunction func = state.GetFunction("SendMsgError");
474.                 func.BeginPCall();
475.                 func.Push(gameObject);
476.                 func.PCall();
477.                 func.EndPCall();
478.                 func.Dispose();
479.             }
480.             else if (GUI.Button(new Rect(10, 610, 120, 40), "AddComponent"))
481.             {
482.                 tips = "";
483.                 LuaFunction func = state.GetFunction("TestAddComponent");
484.                 func.BeginPCall();
485.                 func.Push(gameObject);
486.                 func.PCall();
487.                 func.EndPCall();
488.                 func.Dispose();
489.             }
490.             else if (GUI.Button(new Rect(210, 10, 120, 40), "TableGetSet"))
491.             {
492.                 tips = "";
493.                 LuaTable table = state.GetTable("testev");
494.                 int top = state.LuaGetTop();
495.
496.                 try
```

```
497.                 {
498.                     state.Push(table);
499.                     state.LuaGetField(-1, "Add");
500.                     LuaFunction func = state.CheckLuaFunction(-1);
501.
502.                     if (func != null)
503.                     {
504.                         func.Call();
505.                         func.Dispose();
506.                     }
507.
508.                     state.LuaPop(1);
509.                     state.Push(123456);
510.                     state.LuaSetField(-2, "value");
511.                     state.LuaGetField(-1, "value");
512.                     int n = (int)state.LuaCheckNumber(-1);
513.                     Debugger.Log("value is: " + n);
514.
515.                     state.LuaPop(1);
516.
517.                     state.Push("Add");
518.                     state.LuaGetTable(-2);
519.
520.                     func = state.CheckLuaFunction(-1);
521.
522.                     if (func != null)
523.                     {
524.                         func.Call();
525.                         func.Dispose();
526.                     }
527.
528.                     state.LuaPop(1);
529.
530.                     state.Push("look");
531.                     state.Push(456789);
532.                     state.LuaSetTable(-3);
533.
534.                     state.LuaGetField(-1, "look");
535.                     n = (int)state.LuaCheckNumber(-1);
536.                     Debugger.Log("look: " + n);
537.                 }
538.                 catch (Exception e)
```

```
539.                 {
540.                     state.LuaSetTop(top);
541.                     throw e;
542.                 }
543. 
544.                 state.LuaSetTop(top);
545.             }
546.             else if (GUI.Button(new Rect(210, 60, 120, 40), "TestTableInCo"))
547.             {
548.                 tips = "";
549.                 LuaFunction func = state.GetFunction("TestCoTable");
550.                 func.BeginPCall();
551.                 func.PCall();
552.                 func.EndPCall();
553.                 func.Dispose();
554.             }
555.             else if (GUI.Button(new Rect(210, 110, 120, 40), "Instantiate2
    Error"))
556.             {
557.                 tips = "";
558.                 LuaFunction func = state.GetFunction("Instantiate");
559.                 func.BeginPCall();
560.                 func.Push(go2);
561.                 func.PCall();
562.                 func.EndPCall();
563.                 func.Dispose();
564.             }
565.             else if (GUI.Button(new Rect(210, 160, 120, 40), "Instantiate3
    Error"))
566.             {
567.                 tips = "";
568.                 UnityEngine.Object.Instantiate(go2);
569.             }
570.             else if (GUI.Button(new Rect(210, 210, 120, 40), "TestCycle"))
571.             {
572.                 tips = "";
573.                 int n = 20;
574.                 LuaFunction func = state.GetFunction("TestCycle");
575.                 func.BeginPCall();
576.                 func.Push(n);
577.                 func.PCall();
578.                 int c = (int)func.CheckNumber();
```

```
                func.EndPCall();

                Debugger.Log("Fib({0}) is {1}", n, c);
            }
            else if (GUI.Button(new Rect(210, 260, 120, 40), "TestNull"))
            {
                tips = "";
                Action action = ()=>
                {
                    LuaFunction func = state.GetFunction("TestNull");
                    func.BeginPCall();
                    func.PushObject(null);
                    func.PCall();
                    func.EndPCall();
                };

                StartCoroutine(TestCo(action));
            }
            else if (GUI.Button(new Rect(210, 310, 120, 40), "TestMulti"))
            {
                tips = "";
                LuaFunction func = state.GetFunction("TestMulStack");
                func.BeginPCall();
                func.PushObject(null);
                func.PCall();
                func.EndPCall();
            }
        }

        IEnumerator TestCo(Action action)
        {
            yield return new WaitForSeconds(0.1f);
            action();
        }
    }
```

TestInstantiate.cs

```
using UnityEngine;
using System;
```

```
3.      using LuaInterface;
4.
5.      public class TestInstantiate : MonoBehaviour
6.      {
7.          void Awake()
8.          {
9.              LuaState state = LuaState.Get(IntPtr.Zero);
10.
11.             try
12.             {
13.                 LuaFunction func = state.GetFunction("Show");
14.                 func.BeginPCall();
15.                 func.PCall();
16.                 func.EndPCall();
17.                 func.Dispose();
18.                 func = null;
19.             }
20.             catch (Exception e)
21.             {
22.                 state.ThrowLuaException(e);
23.             }
24.         }
25.
26.         void Start()
27.         {
28.             Debugger.Log("start");
29.         }
30.     }
```

TestInstantiate2.cs

```
1.      using UnityEngine;
2.      using System;
3.      using LuaInterface;
4.
5.      public class TestInstantiate2 : MonoBehaviour
6.      {
7.          void Awake()
8.          {
9.              try
```

```
10.             {
11.                 throw new Exception("Instantiate exception 2");
12.             }
13.             catch (Exception e)
14.             {
15.                 LuaState state = LuaState.Get(IntPtr.Zero);
16.                 state.ThrowLuaException(e);
17.             }
18.         }
19.     }
```

?

代码热更新

一、新建场景

在任意物体上添加Main组件。其实Main组件里面只是调用了 `AppFacade.Instance.StartUp()` ，这是框架的起点。框架将会自动完成资源加载、热更新等等事项。

二、删掉事例的调用

现在不需要框架自带的示例了，需要删掉一些代码，使框架只运行我们编写的lua文件。打开 `Assets\LuaFramework\Scripts\Manager\GameManager.cs` ，将 `OnInitalize` 修改成下图这个样子。这是lua的入口，框架会调用 `Main.lua` 的 `Main` 方法。

```
1.      void OnInitialize() {
2.          LuaManager.InitStart();
3.          // LuaManager.DoFile("Logic/Game");         //加载游戏
4.          // LuaManager.DoFile("Logic/Network");      //加载网络
5.          // NetManager.OnInit();                     //初始化网络
6.          // Util.CallMethod("Game", "OnInitOK");     //初始化完成
7.          // ...
8.
9.          initialize = true;
10.     }
```

三、编写lua代码

打开Assets\LuaFramework\Lua\main.lua，编写lua代码。这里只添加一句“LuaFramework.Util.Log("HelloWorld");”（如下所示），它的功能相当于Debug.Log("HelloWorld")。

```
1.      --主入口函数。从这里开始lua逻辑
2.      function Main()
3.
4.          LuaFramework.Util.Log("HelloWorld");
5.
```

```
    end
```

“LuaFramework.Util.Log("HelloWorld")”中的Util是c#里定义的类，供lua中调用。可以打开Assets\LuaFramework\Editor\CustomSettings.cs看到所有可以供lua调用的类，如下图是CustomSettings.cs的部分语句。

```
        //for LuaFramework
        _GT(typeof(RectTransform)),
        _GT(typeof(Text)),

        _GT(typeof(Util)),
        _GT(typeof(AppConst)),
        _GT(typeof(LuaHelper)),
        _GT(typeof(ByteBuffer)),
        _GT(typeof(LuaBehaviour)),

        _GT(typeof(GameManager)),
        _GT(typeof(LuaManager)),
        _GT(typeof(PanelManager)),
        _GT(typeof(SoundManager)),
        _GT(typeof(TimerManager)),
        _GT(typeof(ThreadManager)),
        _GT(typeof(NetworkManager)),
        _GT(typeof(ResourceManager)),
```

再由具体的类可以查找所有的API（参见下面两个图），如下图是Util类的部分语句。

```
        public static void Log(string str) {
            Debug.Log(str);
        }

        public static void LogWarning(string str) {
            Debug.LogWarning(str);
        }

        public static void LogError(string str) {
            Debug.LogError(str);
        }
```

四、运行游戏

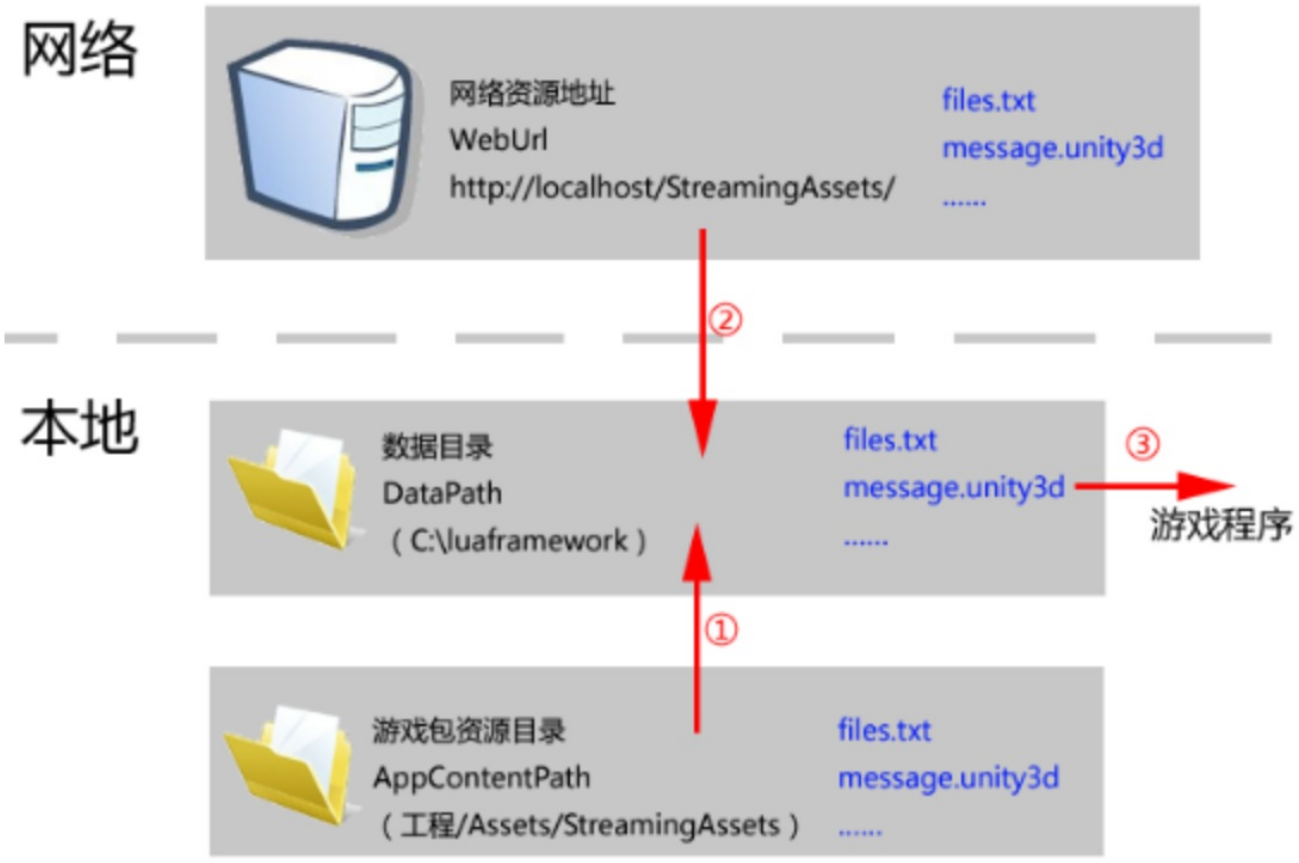
点击菜单栏中LuaFramework→Build Windows Resource，生成资源文件。然后运行游戏，即可在控制台中看到打印出的HelloWorld。

按照默认的设置，每更改一次lua代码，都需要执行Build XXX Resource才能生效。读者可以将Assets\LuaFramework\Scripts\ConstDefine\AppConst.cs中的LuaBundleMode修改为false，这样代码文件便不会以AssetBundle模式读取，会直接生效，以方便调试。

```
1.      //Lua代码AssetBundle模式
2.      public const bool LuaBundleMode = false;
```

五、热更新原理

接下来便要尝试代码热更新，让程序下载服务器上的lua文件，然后运行它。在说明热更新之前，需要先看看Unity3D热更新的一般方法。如下图所示，Unity3D的热更新会涉及3个目录。



热更新的过程

游戏资源目录：里面包含Unity3D工程中StreamingAssets文件夹下的文件。安装游戏之后，这些文件将会被一字不差地复制到目标机器上的特定文件夹里，不同平台的文件夹不同，如下所示（上图以

windows平台为例）

```
1.      Mac OS或Windows：Application.dataPath + "/StreamingAssets";
2.
3.      IOS： Application.dataPath + "/Raw";
4.
5.      Android：jar:file://" + Application.dataPath + "!/assets/";
```

数据目录：由于“游戏资源目录”在Android和IOS上是只读的，不能把网上的下载的资源放到里面，所以需要建立一个“数据目录”，该目录可读可写。第一次开启游戏后，程序将“游戏资源目录”的内容复制到“数据目录中”（步骤1，这个步骤只会执行一次，下次再打开游戏就不复制了）。游戏过程中的资源加载，都是从“数据目录”中获取、解包（步骤3）。不同平台下，“数据目录”的地址也不同，LuaFramework的定义如下：

```
1.      Android或IOS：Application.persistentDataPath + "/LuaFramework"
2.
3.      Mac OS或Windows：c:/LuaFramework/
4.
5.      调试模式下：Application.dataPath + "/StreamingAssets/"
```

> 注：“LuaFramework”和“StreamingAssets”由配置决定，这里取默认值

网络资源地址：存放游戏资源的网址，游戏开启后，程序会从网络资源地址下载一些更新的文件到数据目录。

这些目录包含着不同版本的资源文件，以及用于版本控制的files.txt。Files.txt的内容如下图所示，里面存放着资源文件的名称和md5码。程序会先下载“网络资源地址”上的files.txt，然后与“数据目录”中文件的md5码做比较，更新有变化的文件（步骤2）。

```
1.      StreamingAssets|1738a512698f18f78bfca1591edcf5ae
2.      StreamingAssets.manifest|1994f53ee73e2badeddd65601d587f38
3.      message.unity3d|1ab91c2ccd6a03232c0255c1895c6435
4.      message.unity3d.manifest|9f46935cf1de80e8fab9927c6a25803f
5.      prompt.unity3d|01b68a164fa653bf853bedbda3a5f3d0
6.      prompt.unity3d.manifest|9ebfa552ca1e8a84360944c0dee65bb3
7.      prompt_asset.unity3d|1655812ce2260a4747a7fdb8e39fc007
8.      prompt_asset.unity3d.manifest|ed9d66a485b1d96d504f7002c52ef9a4
9.      shared_asset.unity3d|17ce04b42949f651895d634f852dd972
10.     shared_asset.unity3d.manifest|71721e8dbf69055affcf2a2e0b551f01
11.     tank.unity3d|a5f10487d490d13642d40fce937c195c
12.     tank.unity3d.manifest|202261f1e418ab2bbb8384ed6f54f958
13.     lua/Build.bat|4cfc2f258ecdb9bb43756a2e737c7a1d
```

LuaFramework的热更新代码定义在 `Assets\LuaFramework\Scripts\Manager\GameManager.cs` ，真正用到项目时可能还需少许改动。

六、开始热更新代码

那么开始测试热更新代码的功能吧！热更上述实现的“HelloWorld”。

1）、修改配置

框架的默认配置是从本地加载文件，需要打开AppConst.cs将UpdateMode设置为true（才会执行步骤2），将LuaBundleMode设置为true，将WebUrl设置成服务器地址。

```
1.      public class AppConst {
2.              public const bool DebugMode = true;                         //调试模
    式-用于内部测试
3.              /// <summary>
4.              /// 如果想删掉框架自带的例子，那这个例子模式必须要
5.              /// 关闭，否则会出现一些错误。
6.              /// </summary>
7.              public const bool ExampleMode = true;                       //例子模
    式
8.
9.              /// <summary>
10.             /// 如果开启更新模式，前提必须启动框架自带服务器端。
11.             /// 否则就需要自己将StreamingAssets里面的所有内容
12.             /// 复制到自己的Webserver上面，并修改下面的WebUrl。
13.             /// </summary>
14.             public const bool UpdateMode = true;                        //更新模
    式-默认关闭
15.             public const bool LuaByteMode = false;                       //Lua
    字节码模式-默认关闭
16.             public const bool LuaBundleMode = true;                     //Lua代
    码AssetBundle模式
17.
18.             public const int TimerInterval = 1;
19.             public const int GameFrameRate = 30;                        //游戏帧
    频
20.
21.             public const string AppName = "LuaFramework";               //应用程
    序名称
```

```
22.             public const string LuaTempDir = "Lua/";                    //临时目录
23.             public const string AppPrefix = AppName + "_";              //应用程序前缀
24.             public const string ExtName = ".unity3d";                   //素材扩展名
25.             public const string AssetDir = "StreamingAssets";           //素材目录
26.             public const string WebUrl = "http://localhost/StreamingAssets/";  //测试更新地址
27.
28.             public static string UserId = string.Empty;                 //用户ID
29.             public static int SocketPort = 0;                           //Socket服务器端口
30.             public static string SocketAddress = string.Empty;          //Socket服务器地址
31.
32.             public static string FrameworkRoot {
33.                 get {
34.                     return Application.dataPath + "/" + AppName;
35.                 }
36.             }
37.     }
```

2）、配置“网络资源”

> IIS（Internet Information Services）是一种Web（网页）服务组件，其中包括Web服务器、FTP服务器、NNTP服务器和SMTP服务器，分别用于网页浏览、文件传输、新闻服务和邮件发送等方面，它使得在网络（包括互联网和局域网）上发布信息成了一件很容易的事。控制面板 > 程序和功能 > 启用或关闭Windows功能 > 开启 “Internet Information Services”。打开IIS，win + R > inetMgr

创建ASP.NET项目 MyHotUpdateWebTest

添加index.html

```
1.     // index.html
2.     <!DOCTYPE html>
3.     <html xmlns="http://www.w3.org/1999/xhtml">
4.     <head>
5.     <meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>
6.         <title></title>
```

```
        <meta charset="utf-8" />
    </head>
    <body>
        Hello MyFrameWork!
    </body>
    </html>
```

添加一般处理程序

```
    using System;
    using System.Collections.Generic;
    using System.Linq;
    using System.Web;
    using System.Web.Script.Serialization;

    namespace MyHotUpdateWebTest
    {
        /// <summary>
        /// Login 的摘要说明
        /// </summary>
        public class Login : IHttpHandler
        {

            public void ProcessRequest(HttpContext context)
            {
                context.Response.ContentType = "text/plain";
                //context.Response.Write("Hello World");

                Dictionary<string, object> dic = new Dictionary<string, object>
();
                dic["result"] = "success";

                JavaScriptSerializer jss = new JavaScriptSerializer();
                string result = jss.Serialize(dic);
                context.Response.Write(result);
                context.Response.End();
            }

            public bool IsReusable
            {
                get
                {
```

```
                    return false;
                }
            }
        }
    }
```

配置Web.config

```
    <?xml version="1.0" encoding="utf-8"?>
    <!--
      有关如何配置 ASP.NET 应用程序的详细信息，请访问
      http://go.microsoft.com/fwlink/?LinkId=169433
      -->
    <configuration>
      <system.web>
        <compilation debug="true" targetFramework="4.5" />
        <httpRuntime targetFramework="4.5" />
      </system.web>
      <system.webServer>
        <staticContent>
          <mimeMap fileExtension=".list" mimeType="application/octet-stream" />
          <mimeMap fileExtension=".ab" mimeType="application/octet-stream" />
        </staticContent>
        <directoryBrowse enabled ="true" />
      </system.webServer>
    </configuration>
```

3）、测试热更新

改一下Lua脚本（如将HelloWorld改为Hello Lpy2），点击Build Windows Resource，将“工程目录/StreamingAssets”里面的文件复制到服务器上。再将脚本改成其他内容，然后Build Windows Resource，覆盖掉本地资源。运行游戏，如果程序显示“Hello Lpy2”的代码，证明成功从网上拉取了文件。

> 打包资源需要确认AppConst
>
> ```
>     public const bool ExampleMode = true;                                 //例子模式
> ```

?

资源热更新

热更新涉及资源热更新和代码热更新（其实lua代码也是资源），那接下来看看如何动态加载一个模型，然后热更成其他素材。这一部分涉及资源打包、动态创建资源等内容。

一、创建物体

为了调试的方便，先将框架配置为本地模式，待测试热更新时再改成更新模式。

```
1.          public const bool UpdateMode = false;                    //更新模式-默认关闭
2.          public const bool LuaByteMode = false;                    //Lua字节码模式-默认关闭
3.          public const bool LuaBundleMode = false;                  //Lua代码AssetBundle模式
```

先测试个简单的创建物体，新建一个名为go的物体，然后设置它的坐标为(1,1,1)。这段代码虽然不涉及资源加载，但能展示“把物体添加到场景中”的过程。Main.lua的代码如下：

```
1.      function Main()
2.          local go = UnityEngine.GameObject ('go')
3.          go.transform.position = Vector3.one
4.      end
```

要热更新资源，便需要制作资源。这里制作一个名为tankPrefab的坦克模型预设，然后存到Assets/Tank目录下。接下来对它做打包，然后动态加载。

二、资源打包

LuaFramework在打包方面并没有做太多的工作，我们需要手动打包。打开Assets/LuaFramework/Editor/Packager.cs，按照示例的写法，加上下面这一行：将Assets/Tank目录下的所有预设（.prefab）打包成名为tank的包。

```
1.      /// <summary>
2.      /// 处理框架实例包
```

```
    /// </summary>
    static void HandleExampleBundle() {
        string resPath = AppDataPath + "/" + AppConst.AssetDir + "/";
        if (!Directory.Exists(resPath)) Directory.CreateDirectory(resPath);

        AddBuildMap("prompt" + AppConst.ExtName, "*.prefab",
"Assets/LuaFramework/Examples/Builds/Prompt");
        AddBuildMap("message" + AppConst.ExtName, "*.prefab",
"Assets/LuaFramework/Examples/Builds/Message");

        AddBuildMap("prompt_asset" + AppConst.ExtName, "*.png",
"Assets/LuaFramework/Examples/Textures/Prompt");
        AddBuildMap("shared_asset" + AppConst.ExtName, "*.png",
"Assets/LuaFramework/Examples/Textures/Shared");

        // 坦克的
        AddBuildMap("tank" + AppConst.ExtName, "*.prefab", "Assets/Tank");
    }
```

点击“Build Windows Resource”，即可在StreamingAssets中看到打包好的文件。

> Unity3D资源包里面包含多个资源，就像一个压缩文件一样。在动态加载的时候，便需要有加载包文件、或取包中的资源两步操作（框架已经帮我们做好了这部分工作，直接调用API即可）。

三、动态加载模型

如下图所示，Unity3D资源包里面包含多个资源，就像一个压缩文件一样。在动态加载的时候，便需要有加载包文件、或取包中的资源两步操作（框架已经帮我们做好了这部分工作，直接调用API即可）。

```
    --主入口函数。从这里开始lua逻辑
    function Main()
        LuaHelper = LuaFramework.LuaHelper;
        resMgr = LuaHelper.GetResManager();
        resMgr:LoadPrefab('tank', { 'TankPrefab' }, OnLoadFinish);
    end

    --加载完成后的回调--
    function OnLoadFinish(objs)
        local go = UnityEngine.GameObject.Instantiate(objs[0]);
        LuaFramework.Util.Log("Finish");
    end
```

完成后运行游戏，即可看到动态加载出来的模型。

四、加载资源的过程

只有理解了动态加载，即LoadPrefab的过程，才能算是真正的理解了热更新。LoadPrefab为ResourceManager中定义的方法，在Assets\LuaFramework\Scripts\Manager\ResourceManager.cs中实现。

```
1.      public void LoadPrefab(string abName, string[] assetNames, LuaFunction
    func) {
2.          abName = abName.ToLower();
3.          List<UObject> result = new List<UObject>();
4.          for (int i = 0; i < assetNames.Length; i++) {
5.              UObject go = LoadAsset<UObject>(abName, assetNames[i]);
6.              if (go != null) result.Add(go);
7.          }
8.          if (func != null) func.Call((object)result.ToArray());
9.      }
10.
11.     /// <summary>
12.     /// 载入素材
13.     /// </summary>
14.     public T LoadAsset<T>(string abname, string assetname) where T :
    UnityEngine.Object {
15.         abname = abname.ToLower();
16.         AssetBundle bundle = LoadAssetBundle(abname);
17.         return bundle.LoadAsset<T>(assetname);
18.     }
19.
20.     /// <summary>
21.     /// 载入AssetBundle
22.     /// </summary>
23.     /// <param name="abname"></param>
24.     /// <returns></returns>
25.     public AssetBundle LoadAssetBundle(string abname) {
26.         if (!abname.EndsWith(AppConst.ExtName)) {
27.             abname += AppConst.ExtName;
28.         }
29.         AssetBundle bundle = null;
30.         if (!bundles.ContainsKey(abname)) {
```

```
            byte[] stream = null;
            string uri = Util.DataPath + abname;
            Debug.LogWarning("LoadFile::>> " + uri);
            LoadDependencies(abname);

            stream = File.ReadAllBytes(uri);
            bundle = AssetBundle.LoadFromMemory(stream); //关联数据的素材绑定
            bundles.Add(abname, bundle);
        } else {
            bundles.TryGetValue(abname, out bundle);
        }
        return bundle;
    }

    /// <summary>
    /// 载入依赖
    /// </summary>
    /// <param name="name"></param>
    void LoadDependencies(string name) {
        if (manifest == null) {
            Debug.LogError("Please initialize AssetBundleManifest by calling AssetBundleManager.Initialize()");
            return;
        }
        // Get dependecies from the AssetBundleManifest object..
        string[] dependencies = manifest.GetAllDependencies(name);
        if (dependencies.Length == 0) return;

        for (int i = 0; i < dependencies.Length; i++)
            dependencies[i] = RemapVariantName(dependencies[i]);

        // Record and load all dependencies.
        for (int i = 0; i < dependencies.Length; i++) {
            LoadAssetBundle(dependencies[i]);
        }
    }
```

打包后，Unity3D会产生一个名为AssetBundle.manifest的文件（框架会将该文件放在StreamingAssets中），该文件包含所有包的依赖信息。所以在加载资源前需要先加载这个文件，m_AssetBundleManifest便是指向这个包的变量。相关代码如下：

```
    /// <summary>
```

```
    /// 初始化
    /// </summary>
    public void Initialize() {
        byte[] stream = null;
        string uri = string.Empty;
        bundles = new Dictionary<string, AssetBundle>();
        uri = Util.DataPath + AppConst.AssetDir;
        Debug.Log("uri : " + uri);
        if (!File.Exists(uri)) return;
        stream = File.ReadAllBytes(uri);
        assetbundle = AssetBundle.LoadFromMemory(stream);
        manifest = assetbundle.LoadAsset<AssetBundleManifest>
("AssetBundleManifest");
    }
```

加载这个包后，便可以使用下面的语句获取某个包所依赖的所有包名，然后加载它们。

```
    string[] dependencies = manifest.GetAllDependencies(name);
```

字典类型的bundles保存了所有已经加载资源包。如果某个包已经被加载过，那下次需要用到它时，直接从字典中取出即可，减少重复加载。简化后的代码如下：

```
    /// <summary>
    /// 载入AssetBundle
    /// </summary>
    /// <param name="abname"></param>
    /// <returns></returns>
    public AssetBundle LoadAssetBundle(string abname) {
        if (!abname.EndsWith(AppConst.ExtName)) {
            abname += AppConst.ExtName;
        }
        AssetBundle bundle = null;
        if (!bundles.ContainsKey(abname)) {
            byte[] stream = null;
            string uri = Util.DataPath + abname;
            Debug.LogWarning("LoadFile::>> " + uri);
            LoadDependencies(abname);

            stream = File.ReadAllBytes(uri);
            bundle = AssetBundle.LoadFromMemory(stream); //关联数据的素材绑定
            bundles.Add(abname, bundle);
        } else {
```

```
21.             bundles.TryGetValue(abname, out bundle);
22.         }
23.         return bundle;
24.     }
```

五、资源热更新

“资源热更新”和上一篇的“代码热更新”完全相同，开启更新模式后，将新的资源文件复制到服务器上，框架即可自动下载更新的资源。这里不再复述。

?

编写Lua逻辑

为实现代码热更新，在Unity3D中使用lua，然而为此也需付出不少代价。其一，使代码结构混乱（尽管可以优化），其二降低了运行速度，其三增加学习成本（还要多学一门语言）。为了热更新，所有的逻辑都要用lua编写，那么怎样用lua编写游戏逻辑呢？

一、Lua 的 Update 方法

第一篇“代码热更新”演示了用lua打印HelloWorld的方法，第二篇“资源热更新”演示了加载坦克模型的方法。这一篇要把两者结合起来，用lua实现“用键盘控制坦克移动”的功能。用Lua和用c#编写的Unity3D程序大同小异，只需正确使用API即可。

1）、Update 方法

出于效率的考虑，tolua提供了名为UpdateBeat的对象，在LuaFramework中，只需给UpdateBeat添加回调函数，该函数便会每帧执行，相当于Monobehaviour的Update方法。Lua代码如下所示：

```
1.      function Main()
2.          UpdateBeat:Add(Update, self)
3.      end
4.
5.      function Update()
6.          LuaFramework.Util.Log("每帧执行一次");
7.      end
```

除了UpdateBeat，tolua还提供了LateUpdateBeat和FixedUpdateBeat，对应于Monobehaviour中的LateUpdate和FixedUpdate。

2）、控制坦克

现在编写“用键盘控制坦克移动”的lua代码，加载坦克模型后，使用UpdateBeat注册每帧执行的Update方法，然后在Update方法中调用UnityEngine.Input等API实现功能。代码如下：

```
1.      local go; --加载的坦克模型
2.
3.      --主入口函数。从这里开始lua逻辑
```

```
    function Main()
            LuaHelper = LuaFramework.LuaHelper;
            resMgr = LuaHelper.GetResManager();
            resMgr:LoadPrefab('tank', { 'TankPrefab' }, OnLoadFinish);
    end

    --加载完成后的回调--
    function OnLoadFinish(objs)
            go = UnityEngine.GameObject.Instantiate(objs[0]);
            LuaFramework.Util.Log("LoadFinish");

            UpdateBeat:Add(Update, self)
    end

    --每帧执行
    function Update()
            LuaFramework.Util.Log("每帧执行");

            local Input = UnityEngine.Input;
            local horizontal = Input.GetAxis("Horizontal");
            local verticla = Input.GetAxis("Vertical");

            local x = go.transform.position.x + horizontal
            local z = go.transform.position.z + verticla
            go.transform.position = Vector3.New(x,0,z)
    end
```

运行游戏，即可用键盘的控制坦克移动。

二、自定义API

框架中提供了数十个可供lua调用的c#类，但这些往往不够用，需要自己添加，本节将介绍添加自定义API的方法。

1）、编写C#类

例如，编写TestLuaFun.类，它包含一个静态方法Log，会打印出两行文本。

```
    using UnityEngine;
    using System.Collections;

```

```
4.      public class TestLuaFun
5.      {
6.          public static void Log()
7.          {
8.              Debug.Log("《Unity3D网络游戏实战》是一本好书！");
9.              Debug.Log("《手把手教你用c#制作rpg游戏》也是一本好书！");
10.         }
11.     }
```

2）、修改CustomSetting

打开CustomSetting.cs，在customTypeList中添加一句“_GT(typeof(TestLuaFun))”。

```
1.      //在这里添加你要导出注册到lua的类型列表
2.      public static BindType[] customTypeList =
3.      {
4.          // ...
5.
6.          _GT(typeof(GameManager)),
7.          _GT(typeof(LuaManager)),
8.          _GT(typeof(PanelManager)),
9.          _GT(typeof(SoundManager)),
10.         _GT(typeof(TimerManager)),
11.         _GT(typeof(ThreadManager)),
12.         _GT(typeof(NetworkManager)),
13.         _GT(typeof(ResourceManager)),
14.
15.         // 在这里添加导出注册到lua的类型类标
16.         _GT(typeof(TestLuaFunc)),
17.     }
```

3）、生成wrap文件

点击菜单栏的Lua→Clear wrap files和Lua→Generate All，重新生成wrap文件。由于刚刚在customTypeList添加了类，所以会生成TestLuaFun类的wrap文件TestLuaFunWrap.cs。

打开TestLuaFunWrap.cs，可以看到TestLuaFun注册了Log方法。

```
1.      //this source code was auto-generated by tolua#, do not modify it
2.      using System;
3.      using LuaInterface;
4.
5.      public class TestLuaFuncWrap
```

```
6.      {
7.          public static void Register(LuaState L)
8.          {
9.              L.BeginClass(typeof(TestLuaFunc), typeof(System.Object));
10.             L.RegFunction("Log", Log);
11.             L.RegFunction("New", _CreateTestLuaFunc);
12.             L.RegFunction("__tostring", ToLua.op_ToString);
13.             L.EndClass();
14.         }
15.
16.         [MonoPInvokeCallbackAttribute(typeof(LuaCSFunction))]
17.         static int _CreateTestLuaFunc(IntPtr L)
18.         {
19.             try
20.             {
21.                 int count = LuaDLL.lua_gettop(L);
22.
23.                 if (count == 0)
24.                 {
25.                     TestLuaFunc obj = new TestLuaFunc();
26.                     ToLua.PushObject(L, obj);
27.                     return 1;
28.                 }
29.                 else
30.                 {
31.                     return LuaDLL.luaL_throw(L, "invalid arguments to ctor
    method: TestLuaFunc.New");
32.                 }
33.             }
34.             catch (Exception e)
35.             {
36.                 return LuaDLL.toluaL_exception(L, e);
37.             }
38.         }
39.
40.         [MonoPInvokeCallbackAttribute(typeof(LuaCSFunction))]
41.         static int Log(IntPtr L)
42.         {
43.             try
44.             {
45.                 ToLua.CheckArgsCount(L, 0);
46.                 TestLuaFunc.Log();
```

```
                return 0;
            }
            catch (Exception e)
            {
                return LuaDLL.toluaL_exception(L, e);
            }
        }
    }
```

4）、测试API

修改main.lua，调用TestLuaFun.Log() ，即可看到效果。

```
    --主入口函数。从这里开始lua逻辑
    function Main()
        TestLuaFun.Log()
    end
```

三、原理

tolua实现了LuaInterface，抛开luaFramework，只需创建lua虚拟机，便能在c#中调用lua代码，如下所示。

```
    using UnityEngine;
    using System.Collections;
    using LuaInterface;

    public class test : MonoBehaviour
    {
        void Start ()
        {
            //初始化
            LuaState lua = new LuaState();
            LuaBinder.Bind(lua);
            //lua代码
            string luaStr =
                @"
                    print('hello tolua#, 广告招租')
                    LuaFramework.Util.Log('HelloWorld');
                    TestLuaFun.Log()
```

```
                ";
            //执行lua脚本
            lua.DoString(luaStr);
        }
    }
```

实际上LuaFramework也是用了相似的方法，框架启动后，会创建LuaManager、LuaLooper的实例。LuaManager创建lua虚拟机并调用Main.lua的Main方法，LuaLooper处理了UpdateBeat相关的事情。如下所示：

```
    private LuaState lua;
    private LuaLoader loader;
    private LuaLooper loop = null;

    void Awake() {
        loader = new LuaLoader();
        lua = new LuaState();
        this.OpenLibs();
        lua.LuaSetTop(0);

        LuaBinder.Bind(lua);
        DelegateFactory.Init();
        LuaCoroutine.Register(lua, this);
    }

    public void InitStart() {
        InitLuaPath();
        InitLuaBundle();
        this.lua.Start();    //启动LUAVM
        this.StartMain();
        this.StartLooper();
    }

    void StartMain() {
        lua.DoFile("Main.lua");

        LuaFunction main = lua.GetFunction("Main");
        main.Call();
        main.Dispose();
        main = null;
    }

```

```
33.     void StartLooper() {
34.         loop = gameObject.AddComponent<LuaLooper>();
35.         loop.luaState = lua;
36.     }
```

?

Lua 组件

基于组件的编程模式是Unity3D的核心思想之一，然而使用纯lua编程，基本就破坏了这一模式。那么有没有办法做一些封装，让Lua脚本也能挂载到游戏物体上，作为组件呢？

一、设计思想

在需要添加Lua组件的游戏物体上添加一个LuaComponent组件，LuaComponent引用一个lua表，这个lua表包含lua组件的各种属性以及Awake、Start等函数，由LuaComponent适时调用Lua表所包含的函数。

```
1.  /*
2.   *  created by shenjun
3.   */
4.
5.  using System.Collections;
6.  using System.Collections.Generic;
7.  using UnityEngine;
8.  using LuaInterface;
9.  using LuaFramework;
10.
11. public class LuaComponent : MonoBehaviour {
12.
13.     // Lua 表
14.     public LuaTable table;
15.
16.     // 添加Lua组件
17.     public static LuaTable Add(GameObject go, LuaTable tableClass)
18.     {
19.         LuaFunction fun = tableClass.GetLuaFunction("New");
20.         if (fun == null) return null;
21.
22.         object[] rets = fun.LazyCall(tableClass);
23.         if (rets.Length != 1) return null;
24.
25.         LuaComponent cmp = go.AddComponent<LuaComponent>();
26.         cmp.table = (LuaTable)rets[0];
```

```
            cmp.CallAwake();
            return cmp.table;
        }

        public static LuaTable Get(GameObject go, LuaTable table)
        {
            LuaComponent[] cmps = go.GetComponents<LuaComponent>();
            foreach (LuaComponent cmp in cmps)
            {
                string mat1 = table.ToString();
                string mat2 = cmp.table.GetMetaTable().ToString();
                if (mat1 == mat2) return cmp.table;
            }
            return null;
        }

        void CallAwake()
        {
            LuaFunction fun = table.GetLuaFunction("Awake");

            if (fun != null)
                fun.Call(table, gameObject);
        }

        void Start () {
            LuaFunction fun = table.GetLuaFunction("Start");

            if (fun != null)
                fun.Call(table, gameObject);
        }

        void Update () {
            // 效率问题有待测试和优化
            // 可在lua中调用UpdateBeat替代
            LuaFunction fun = table.GetLuaFunction("Update");

            if (fun != null)
                fun.Call(table, gameObject);
        }
    }
```

下面列举lua组件的文件格式，它包含一个表（如Component），这个表包含property1 、

property2 等属性，包含Awake、Start等方法。表中必须包含用于派生对象的New方法，它会创建一个继承自Component的表o，供LuaComponent调用。

```
Component=    --组件表
{
    property1 = 100,
    property2 = “helloWorld”
}

function Component:Awake()
    print("TankCmp Awake name = "..self.name );
end

function Component:Start()
    print("TankCmp Start name = "..self.name );
End

--更多方法略

function Component:New(obj)
    local o = {}
    setmetatable(o, self)
    self.__index = self
    return o
end
```

二、LuaComponent 组件

LuaComponent主要有Get和Add两个静态方法，其中Get相当于UnityEngine中的GetComponent方法，Add相当于AddComponent方法，只不过这里添加的是lua组件不是c#组件。每个LuaComponent拥有一个LuaTable（lua表）类型的变量table，它既引用上述的Component表。

Add方法使用AddComponent添加LuaComponent，调用参数中lua表的New方法，将其返回的表赋予table。

Get方法使用GetComponents获取游戏对象上的所有LuaComponent（一个游戏对象可能包含多个lua组件，由参数table决定需要获取哪一个），通过元表地址找到对应的LuaComponent，返回lua表。

三、调试LuaComponent

现在编写名为TankCmp的lua组件，测试LuaCompomemt的功能，TankCmp会在Awake、Start和Update打印出属性name。TankCmp.lua的代码如下：

```
TankCmp =
{
    --里面可以放一些属性
    Hp = 100,
    att = 50,
    name = "good tank",
}

function TankCmp:Awake()
    print("TankCmp Awake name = "..self.name );

end

function TankCmp:Start()
    print("TankCmp Start name = "..self.name );
end

function TankCmp:Update()
    print("TankCmp Update name = "..self.name );
end

--创建对象
function TankCmp:New(obj)
    local o = {}
    setmetatable(o, self)
    self.__index = self
    return o
end
```

编写Main.lua，给游戏对象添加lua组件。

```
require "TankCmp"

--主入口函数。从这里开始lua逻辑
function Main()
    --组件1
    local go = UnityEngine.GameObject ('go')
```

```
        local tankCmp1 = LuaComponent.Add(go,TankCmp)
        tankCmp1.name = "Tank1"

        --组件2
        local go2 = UnityEngine.GameObject ('go2')
        LuaComponent.Add(go2,TankCmp)
        local tankCmp2 = LuaComponent.Get(go2,TankCmp)
        tankCmp2.name = "Tank2"
    end
```

运行游戏，即可看到lua组件的运行结果。

四、坦克组件

下面代码演示用lua组件实现“用键盘控制坦克移动”的功能，TankCmp.lua的代码如下：

```
    TankCmp =
    {
        name = "good tank",
    }

    function TankCmp:Update(gameObject)
        print("TankCmp Update name = "..self.name );
        local Input = UnityEngine.Input;
        local horizontal = Input.GetAxis("Horizontal");
        local vertical = Input.GetAxis("Vertical");
        local x = gameObject.transform.position.x + horizontal
        local z = gameObject.transform.position.z + vertical
        gameObject.transform.position = Vector3.New(x,0,z)
    end


    --创建对象
    function TankCmp:New(obj)
        local o = {}
        setmetatable(o, self)
        self.__index = self
        return o
    end
```

Main.lua先加载坦克模型，然后给他添加lua组件，代码如下：

```
    require "TankCmp"

    --主入口函数。从这里开始lua逻辑
    function Main(
        LuaHelper = LuaFramework.LuaHelper;
        resMgr = LuaHelper.GetResManager();
        resMgr:LoadPrefab('tank', { 'TankPrefab' }, OnLoadFinish);
    end

    --加载完成后的回调--
    function OnLoadFinish(objs)
        go = UnityEngine.GameObject.Instantiate(objs[0]);
        LuaComponent.Add(go,TankCmp)
    end
```

运行游戏，即可用键盘的控制坦克移动。

？

热更 UI

界面系统在游戏中占据重要地位。游戏界面是否友好，很大程度上决定了玩家的体验；界面开发是否便利，也影响着游戏的开发进度。Unity3D 的UGUI系统，使用户可以“可视化地”开发界面，那么怎样用Lua去调用UGUI呢？

一、显示 UI 界面

下面演示如何显示一个UI界面。由于UI界面也是一种资源，使用第二篇“资源热更新”的方法即可。这个例子中，制作一个含有按钮的界面，然后组成名为Panel1的UI预设，存放到Tank目录下。

前面（第二篇）已在Packager类HandleExampleBundle方法中添加了一句“AddBuildMap("tank" + AppConst.ExtName, "*.prefab", "Assets/Tank");”（当然也可以添加到其他地方），它会把Tank目录下的所有预设打包成名为tank的资源包。故而点击“Build xxx Resource”后，Panel1也会被打包到tank资源包中。

修改Lua入口函数Main.lua中的Main方法，在加载资源后把panel1放到Canvas下（需要在场景中添加画布），然后调整它的位置和大小。

```
--主入口函数。从这里开始lua逻辑
function Main()
    LuaHelper = LuaFramework.LuaHelper;
    resMgr = LuaHelper.GetResManager();
    resMgr:LoadPrefab('tank', { 'Panel1' }, OnLoadFinish);
end

--加载完成后的回调--
function OnLoadFinish(objs)
    --显示面板
    go = UnityEngine.GameObject.Instantiate(objs[0]);
    local parent = UnityEngine.GameObject.Find("Canvas")
    go.transform:SetParent(parent.transform);
    go.transform.localScale = Vector3.one;
    go.transform.localPosition = Vector3.zero;
end
```

运行游戏，即可看到加载出来的界面。

二、事件响应

c#中可以使用事件监听的方法给UI组件添加事件。例如，添加按钮点击事件的方法如下：

```
1.      Button btn = go.GetComponent ();
2.      btn.onClick.AddListener(()=>{this.OnClick(go);});
```

然而在LuaFramework的API中，没能找到合适的方法，只能根据第三篇中“自定义API”的方法，自己编写一套了。编写UIEvent类，它包含用于添加监听事件的AdonClick和清除监听事件的ClearButtonClick方法，代码如下所示（完成后记得要“修改CustomSetting”和“生成wrap文件”）。

```
1.      using UnityEngine;
2.      using System.Collections;
3.      using LuaInterface;
4.      using UnityEngine.UI;
5.
6.      public class UIEvent
7.      {
8.
9.          //添加监听
10.         public static void AdonClick(GameObject go, LuaFunction luafunc)
11.         {
12.             if (go == null || luafunc == null) return;
13.             Button btn = go.GetComponent<Button> ();
14.             if (btn == null) return;
15.             btn.onClick.AddListener(()=>{luafunc.Call(go);});
16.         }
17.
18.         //清除监听
19.         public static void ClearButtonClick(GameObject go)
20.         {
21.             if (go == null) return;
22.             Button btn = go.GetComponent<Button> ();
23.             if (btn == null) return;
24.             btn.onClick.RemoveAllListeners();
25.         }
26.     }
```

接下来测试下这套API，修改Main.lua，代码如下：

```
    --主入口函数。从这里开始lua逻辑
    function Main()
        略
    end

    --加载完成后的回调--
    function OnLoadFinish(objs)
        --显示面板
        略
        --事件处理
        local btn = go.transform:Find("Button").gameObject
        UIEvent.AdonClick(btn, OnClick)
    end

    function OnClick()
        print("触发按钮事件")
    end
```

运行游戏，点击按钮，OnClick方法即被调用。

三、界面管理器

LuaFramework提供了一套简单的（不完善的）界面管理器，具体代码请参见PanelManager类。PanelManager类的CreatePanel方法完成异步加载资源，在加载完成后，会设置面板的大小和位置，然后调用回调函数。与上面用lua加载界面的方法完全一样。

```
    public void CreatePanel(string name, LuaFunction func = null) {
        string assetName = name + "Panel";
        string abName = name.ToLower() + AppConst.ExtName;
        if (Parent.Find(name) != null) return;

#if ASYNC_MODE
        ResManager.LoadPrefab(abName, assetName, delegate(UnityEngine.Object[] objs) {
            if (objs.Length == 0) return;
            GameObject prefab = objs[0] as GameObject;
            if (prefab == null) return;

            GameObject go = Instantiate(prefab) as GameObject;
            go.name = assetName;
```

```
            go.layer = LayerMask.NameToLayer("Default");
            go.transform.SetParent(Parent);
            go.transform.localScale = Vector3.one;
            go.transform.localPosition = Vector3.zero;
            go.AddComponent<LuaBehaviour>();

            if (func != null) func.Call(go);
                Debug.LogWarning("CreatePanel::>> " + name + " " + prefab);
        });
#else
        GameObject prefab = ResManager.LoadAsset<GameObject>(name, assetName);
        if (prefab == null) return;

        GameObject go = Instantiate(prefab) as GameObject;
        go.name = assetName;
        go.layer = LayerMask.NameToLayer("Default");
        go.transform.SetParent(Parent);
        go.transform.localScale = Vector3.one;
        go.transform.localPosition = Vector3.zero;
        go.AddComponent<LuaBehaviour>();

        if (func != null) func.Call(go);
            Debug.LogWarning("CreatePanel::>> " + name + " " + prefab);
#endif
    }
```

LuaFramework会给每个界面添加名为LuaBehaviour的组件，它拥有用于添加按钮监听的AddClick方法，相关代码如下，与UIEvent的AdonClick方法相似。

```
    using UnityEngine;
    using LuaInterface;
    using System.Collections;
    using System.Collections.Generic;
    using System;
    using UnityEngine.UI;

    namespace LuaFramework {
        public class LuaBehaviour : View {
            private string data = null;
            private Dictionary<string, LuaFunction> buttons = new
Dictionary<string, LuaFunction>();

```

```
13.             protected void Awake() {
14.                 Util.CallMethod(name, "Awake", gameObject);
15.             }
16.
17.             protected void Start() {
18.                 Util.CallMethod(name, "Start");
19.             }
20.
21.             protected void OnClick() {
22.                 Util.CallMethod(name, "OnClick");
23.             }
24.
25.             protected void OnClickEvent(GameObject go) {
26.                 Util.CallMethod(name, "OnClick", go);
27.             }
28.
29.             /// <summary>
30.             /// 添加单击事件
31.             /// </summary>
32.             public void AddClick(GameObject go, LuaFunction luafunc) {
33.                 if (go == null || luafunc == null) return;
34.                 buttons.Add(go.name, luafunc);
35.                 go.GetComponent<Button>().onClick.AddListener(
36.                     delegate() {
37.                         luafunc.Call(go);
38.                     }
39.                 );
40.             }
41.
42.             /// <summary>
43.             /// 删除单击事件
44.             /// </summary>
45.             /// <param name="go"></param>
46.             public void RemoveClick(GameObject go) {
47.                 if (go == null) return;
48.                 LuaFunction luafunc = null;
49.                 if (buttons.TryGetValue(go.name, out luafunc)) {
50.                     luafunc.Dispose();
51.                     luafunc = null;
52.                     buttons.Remove(go.name);
53.                 }
54.             }
```

```
55.
56.             /// <summary>
57.             /// 清除单击事件
58.             /// </summary>
59.             public void ClearClick() {
60.                 foreach (var de in buttons) {
61.                     if (de.Value != null) {
62.                         de.Value.Dispose();
63.                     }
64.                 }
65.                 buttons.Clear();
66.             }
67.
68.             //-------------------------------------------------------------------
69.             protected void OnDestroy() {
70.                 ClearClick();
71.     #if ASYNC_MODE
72.                 string abName = name.ToLower().Replace("panel", "");
73.                 ResManager.UnloadAssetBundle(abName + AppConst.ExtName);
74.     #endif
75.                 Util.ClearMemory();
76.                 Debug.Log("~" + name + " was destroy!");
77.             }
78.         }
79.     }
```

在LuaFramework的PureMVC架构中，如果要添加一个界面，需要编写对应的Controller、View，以及修改3个框架自带的lua文件，比较繁琐。因此在实际项目中有必要重写PanelManager，由它实现界面的加载及事件处理。

?

xLua教程

Lua文件加载

一、执行字符串

最基本是直接用 `LuaEnv.DoString` 执行一个字符串，当然，字符串得符合 `Lua` 语法比如：

```
1.      LuaEnv luaenv = new LuaEnv();
2.      luaenv.DoString("print('hello world')")
3.      luaenv.Dispose();
```

完整代码见 `XLua\Tutorial\LoadLuaScript\ByString` 目录但这种方式并不建议，更建议下面介绍这种方法。

二、加载Lua文件

用lua的 `require` 函数即可比如：

```
1.      luaenv.DoString("require 'byfile'")
```

完整代码见 `XLua\Tutorial\LoadLuaScript\ByFile` 目录

`require` 实际上是调一个个的 `loader` 去加载，有一个成功就不再往下尝试，全失败则报文件找不到。目前xLua除了原生的 `loader` 外，还添加了从 `Resource` 加载的 `loader` ，需要注意的是因为 `Resource` 只支持有限的后缀，放 `Resources` 下的 `lua` 文件得加上 `txt` 后缀（见附带的例子）。

> 建议的加载 `Lua` 脚本方式是：整个程序就一个 `DoString("require 'main'")` ，然后在 `main.lua` 加载其它脚本（类似 `lua` 脚本的命令行执行： `lua main.lua` ）。有童鞋会问：要是我的 `Lua` 文件是下载回来的，或者某个自定义的文件格式里头解压出来，或者需要解密等等，怎么办？问得好， `xLua` 的自定义 `Loader` 可以满足这些需求。

三、自定义Loader

在 `xLua` 加自定义 `loader` 是很简单的，只涉及到一个接口：

```
1.      public delegate byte[] CustomLoader(ref string filepath);
2.      public void LuaEnv.AddLoader(CustomLoader loader)
```

通过 `AddLoader` 可以注册个回调，该回调参数是字符串， `lua` 代码里头调用 `require` 时，参数将会透传给回调，回调中就可以根据这个参数去加载指定文件，如果需要支持调试，需要把 `filepath` 修改为真实路径传出。该回调返回值是一个 `byte` 数组，如果为空表示该 `loader` 找不到，否则则为 `lua` 文件的内容。有了这个就简单了，用IIPS（交互式信息处理系统）的IFS（互联网金融服务）？没问题。写个 `loader` 调用IIPS的接口读文件内容即可。文件已经加密？没问题，自己写 `loader` 读取文件解密后返回即可。。。完整示例见 `XLua\Tutorial\LoadLuaScript\Loader`

C#访问Lua

这里指的是 `C#` 主动发起对 `Lua` 数据结构的访问。本章涉及到的例子都可以在 `XLua\Tutorial\CSharpCallLua` 下找到。

一、获取一个全局基本数据类型

访问 `LuaEnv.Global` 就可以了，上面有个模版 `Get` 方法，可指定返回的类型。

```
1.      luaenv.Global.Get<int>("a")
2.      luaenv.Global.Get<string>("b")
3.      luaenv.Global.Get<bool>("c")
4.
5.      int d;
6.      luaenv.Global.Get("d", out d)
7.      luaenv.Global.SetInPath("d", d+1);
```

二、访问一个全局的table

也是用上面的 `Get` 方法，那类型要指定成啥呢？

1、映射到普通class或struct

定义一个 `class` ，有对应于 `table` 的字段的 `public` 属性，而且有无参数构造函数即可，比如对于 `{f1 = 100, f2 = 100}` 可以定义一个包含 `public int f1;public int f2;` 的 `class` 。

这种方式下 `xLua` 会帮你 `new` 一个实例，并把对应的字段赋值过去。 `table` 的属性可以多于或者少于 `class` 的属性。可以嵌套其它复杂类型。要注意的是，这个过程是值拷贝，如果 `class` 比较复杂代价会比较大。而且修改 `class` 的字段值不会同步到 `table` ，反过来也不会。这个功能可以通过把类型加到 `GCOptimize` 生成降低开销，详细可参见配置介绍文档。那有没有引用方式的映射呢？有，下面这个就是：

2、映射到一个interface

这种方式依赖于生成代码（如果没生成代码会抛 `InvalidCastException` 异常），代码生成器会生成这个 `interface` 的实例，如果 `get` 一个属性，生成代码会 `get` 对应的 `table` 字段，如果 `set` 属性也会设置对应的字段。甚至可以通过 `interface` 的方法访问 `lua` 的函数。

```
    [CSharpCallLua]
    public interface ICalc
    {
        int Add(int a, int b);
        int Mult { get; set; }
    }

    [CSharpCallLua]
    public delegate CalcNew(int mult, params string[] args);
```

```
    LuaEnv luaenv = new LuaEnv();
    luaenv.DoString(@"
        local calc_mt = {
            __index = {
                Add = function(self, a, b)
                    return (a + b) * self.Mult
                end
            }
        }

        Calc = {
            New = function(mult, ...)
                print(...)
                return setmetatable({Mult = mult}, calc_mt)
            end
        }
    ");

    CalcNew calc_new = luaenv.Global.GetInPath<CalcNew>("Calc.New");
    ICalc calc = calc_new(10, "hi", "join");
```

```
21.     Debug.Log(calc.Add(1, 2));    // 30
22.     calc.Mult = 100;
23.     Debug.Log(calc.Add(1, 2));    // 300
```

3、更轻量级的by value方式：映射到 `Dictionary<>` ， `List<>` 不想定义 `class` 或者 `interface` 的话，可以考虑用这个，前提 `table` 下 `key` 和 `value` 的类型都是一致的。

4、另外一种by ref方式：映射到 `LuaTable` 类这种方式好处是不需要生成代码，但也有一些问题，比如慢，比方式2要慢一个数量级，比如没有类型检查。

三、访问一个全局的function

仍然是用 `Get` 方法，不同的是类型映射。

1、映射到delegate

这种是建议的方式，性能好很多，而且类型安全。缺点是要生成代码（如果没生成代码会抛 `InvalidCastException` 异常）。 `delegate` 要怎样声明呢？对于 `function` 的每个参数就声明一个输入类型的参数。多返回值要怎么处理？从左往右映射到 `c#` 的输出参数，输出参数包括返回值， `out` 参数， `ref` 参数。参数、返回值类型支持哪些呢？都支持，各种复杂类型， `out` ， `ref` 修饰的，甚至可以返回另外一个 `delegate` 。 `delegate` 的使用就更简单了，直接像个函数那样用就可以了。

1、基本数据类型

```
1.     [CSharpCallLua]
2.     public delegate int IntParam(int p);
```

```
1.     LuaEnv luaenv = new LuaEnv();
2.     luaenv.DoString(@"
3.         function id(...)
4.             return ...
5.         end
6.     ");
7.     IntParam f1;
8.     luaenv.Global.Get("id", out f1);
9.     Debug.Log(f1(1));  // 1
```

2、结构体

```
1.     [CSharpCallLua]
```

```
2.      public delegate Vector3 Vector3Param(Vector3 p);
```

```
1.      Vector3Param f2;
2.      luaenv.Global.Get("id", out f2);
3.      Vector3 v3 = new Vector3(1, 2, 3);
4.      Debug.Log(f2(v3));    // (1.0, 2.0, 3.0)
```

3、自定义值类型

```
1.      [GCOptimize]
2.      [LuaCallCSharp]
3.      public struct Pedding
4.      {
5.          public byte c;
6.      }
7.
8.      [GCOptimize]
9.      [LuaCallCSharp]
10.     public struct MyStruct
11.     {
12.         public MyStruct(int p1, int p2)
13.         {
14.             a = p1;
15.             b = p2;
16.             c = p2;
17.             e.c = (byte)p1;
18.         }
19.         public int a;
20.         public int b;
21.         public decimal c;
22.         public Pedding e;
23.     }
24.
25.     [CSharpCallLua]
26.     public delegate MyStruct CustomValueTypeParam(MyStruct p);
```

```
1.      CustomValueTypeParam f3;
2.      luaenv.Global.Get("id", out f3);
3.      MyStruct mystruct = new MyStruct(5, 6);
4.      Debug.Log(f3(mystruct).e.c);    // 5
```

4、枚举类型

```
1.      [LuaCallCSharp]
2.      public enum MyEnum
3.      {
4.          E1,
5.          E2
6.      }
7.
8.      [CSharpCallLua]
9.      public MyEnum EnumParam(MyEnum p);
```

```
1.      EnumParam f4;
2.      luaenv.Global.Get("id", out f4);
3.      Debug.Log(f4(MyEnum.E1));    // E1
```

2、映射到LuaFunction

这种方式的优缺点刚好和第一种相反。使用也简单， `LuaFunction` 上有个变参的 `Call` 函数，可以传任意类型，任意个数的参数，返回值是 `object` 的数组，对应于 `lua` 的多返回值。

> #### 四、使用建议1、访问 lua 全局数据，特别是 table 以及 function ，代价比较大，建议尽量少做，比如在初始化时把要调用的 lua function 获取一次（映射到 delegate ）后，保存下来，后续直接调用该 delegate 即可。 table 也类似。2、如果 lua 测的实现的部分都以 delegate 和 interface 的方式提供，使用方可以完全和 xLua 解耦：由一个专门的模块负责 xlua 的初始化以及 delegate 、 interface 的映射，然后把这些 delegate 和 interface 设置到要用到它们的地方。

Lua调用C

本章节涉及到的实例均在 `XLua\Tutorial\LuaCallCSharp` 下

new C#对象

你在 `C#` 这样 `new` 一个对象：

```
1.      var newGameObj = new UnityEngine.GameObject();
```

对应到Lua是这样：

```
local newGameObj = CS.UnityEngine.GameObject()
```

基本类似，除了： `lua` 里头没有 `new` 关键字；所有 `C#` 相关的都放到 `CS` 下，包括构造函数，静态成员属性、方法；

如果有多个构造函数呢？放心， `xlua` 支持重载，比如你要调用 `GameObject` 的带一个 `string` 参数的构造函数，这么写：

```
local newGameObj2 = CS.UnityEngine.GameObject('helloworld')
```

C#创建Lua全局变量

```
luaenv.Global.Set("g_int", 123);
luaenv.Global.Set(123, 456);
int i;
luaenv.Global.Get("g_int", out i);
Debug.Log(i);    // 123
luaenv.Global.Get(123, out i);
Debug.Log(i);    // 456
```

C#传参数到Lua

```
[CSharpCallLua]
public delegate float MyDel(float p);
```

```
public float FloatParamMethod(float p)
{
    return p;
}
```

```
luaenv.DoString(@"
    function lua_access_csharp()
        return monoBehaviour:FloatParamMethod(123);
    end
");

luaenv.Global.Set("monoBehaviour", this);
```

```
    MyDel flua;
    luaenv.Global.Get("lua_access_csharp", out flua);
    Debug.Log(flua(0.5f));    // 0.5
```

访问C#静态属性，方法

读静态属性

```
    CS.UnityEngine.Time.deltaTime
```

写静态属性

```
    CS.UnityEngine.Time.timeScale = 0.5
```

调用静态方法

```
    CS.UnityEngine.GameObject.Find('helloworld')
```

小技巧：如果需要经常访问的类，可以先用局部变量引用后访问，除了减少敲代码的时间，还能提高性能：

```
    local GameObject = CS.UnityEngine.GameObject
    GameObject.Find('helloworld')
```

访问C#成员属性，方法

读成员属性

```
    testobj.DMF
```

写成员属性

```
    testobj.DMF = 1024
```

调用成员方法注意：调用成员方法，第一个参数需要传该对象，建议用冒号语法糖，如下

```
    testobj:DMFunc()
```

父类属性，方法

`xlua` 支持（通过派生类）访问基类的静态属性，静态方法，（通过派生类实例）访问基类的成员属

性，成员方法

参数的输入输出属性（out，ref）

`Lua` 调用测的参数处理规则： `C#` 的普通参数算一个输入形参， `ref` 修饰的算一个输入形参， `out` 不算，然后从左往右对应 `lua` 调用测的实参列表；

Lua调用测的返回值处理规则： `C#` 函数的返回值（如果有的话）算一个返回值， `out` 算一个返回值， `ref` 算一个返回值，然后从左往右对应 `lua` 的多返回值。

重载方法

直接通过不同的参数类型进行重载函数的访问，例如：

```
1.      testobj:TestFunc(100)
2.      testobj:TestFunc('hello')
```

将分别访问整数参数的 `TestFunc` 和字符串参数的 `TestFunc` 。

注意： `xlua` 只一定程度上支持重载函数的调用，因为 `lua` 的类型远远不如 `C#` 丰富，存在一对多的情况，比如 `C#` 的 `int` ， `float` ， `double` 都对应于 `lua` 的 `number` ，上面的例子中 `TestFunc` 如果有这些重载参数，第一行将无法区分开来，只能调用到其中一个（生成代码中排前面的那个）

操作符

支持的操作符有： `+, -, *, /, ==, 一元-, <, <=, %, []`

参数带默认值的方法

和 `C#` 调用有默认值参数的函数一样，如果所给的实参少于形参，则会用默认值补上。

可变参数方法

对于 `C#` 的如下方法：

```
1.      void VariableParamsFunc(int a, params string[] strs)
```

可以在 `lua` 里头这样调用：

```
1.      testobj:VariableParamsFunc(5, 'hello', 'john')
```

使用Extension methods

在 `C#` 里定义了， `lua` 里就能直接使用。

泛化（模版）方法

不直接支持，可以通过 `Extension methods` 功能进行封装后调用。

枚举类型

枚举值就像枚举类型下的静态属性一样。

```
1.      testobj:EnumTestFunc(CS.Tutorial.TestEnum.E1)
```

上面的 `EnumTestFunc` 函数参数是 `Tutorial.TestEnum` 类型的

另外，如果枚举类加入到生成代码的话，枚举类将支持 `__CastFrom` 方法，可以实现从一个整数或者字符串到枚举值的转换，例如：

```
1.      CS.Tutorial.TestEnum.__CastFrom(1)
2.      CS.Tutorial.TestEnum.__CastFrom('E1')
```

delegate使用（调用，+，-）

`C#` 的 `delegate` 调用：和调用普通 `lua` 函数一样 `+` 操作符：对应 `C#` 的 `+` 操作符，把两个调用串成一个调用链，右操作数可以是同类型的 `C# delegate` 或者是 `lua` 函数。 `-` 操作符：和 `+` 相反，把一个 `delegate` 从调用链中移除。

> Ps: `delegate` 属性可以用一个 `luafunction` 来赋值。

event

比如 `testobj` 里头有个事件定义是这样：

```
1.      public event Action TestEvent;
```

增加事件回调

```
1.      testobj:TestEvent('+', lua_event_callback)
```

移除事件回调

```
1.      testobj:TestEvent('-', lua_event_callback)
```

64位整数支持

Lua53版本64位整数（ `long` ， `ulong` ）映射到原生的64位整数，而 `luajit` 版本，相当于lua5.1的标准，本身不支持64位， `xlua` 做了个64位支持的扩展库， `C#` 的 `long` 和 `ulong` 都将映射到 `userdata` ：

```
1.  支持在lua里头进行64位的运算，比较，打印
2.  支持和lua number的运算，比较
```

要注意的是，在64扩展库中，实际上只有 `int64` ， `ulong` 也会先强转成 `long` 再传递到 `lua` ，而对 `ulong` 的一些运算，比较，我们采取和 `java` 一样的支持方式，提供一组API，详情请看API文档。

C#复杂类型和table的自动转换

对于一个有无参构造函数的 `C#` 复杂类型，在 `lua` 侧可以直接用一个 `table` 来代替，该 `table` 对应复杂类型的 `public` 字段有相应字段即可，支持函数参数传递，属性赋值等，例如：

`C#` 下B结构体（ `class` 也支持）定义如下：

```
1.      public struct A
2.      {
3.          public int a;
4.      }
5.
6.      public struct B
7.      {
8.          public A b;
9.          public double c;
10.     }
```

某个类有成员函数如下：

```
1.      void Foo(B b)
```

在 `lua` 可以这么调用

```
1.      obj:Foo({b = {a = 100}, c = 200})
```

获取类型（相当于C#的typeof）

比如要获取 `UnityEngine.ParticleSystem` 类的 `Type` 信息，可以这样

```
1.      typeof(CS.UnityEngine.ParticleSystem)
```

“强”转

`lua` 没类型，所以不会有强类型语言的“强转”，但有个有点像的东西：告诉 `xlua` 要用指定的生成代码去调用一个对象，这在什么情况下能用到呢？有的时候第三方库对外暴露的是一个 `interface` 或者抽象类，实现类是隐藏的，这样我们无法对实现类进行代码生成。该实现类将会被 `xlua` 识别为未生成代码而用反射来访问，如果这个调用是很频繁的话还是很影响性能的，这时我们就可以把这个 `interface` 或者抽象类加到生成代码，然后指定用该生成代码来访问：

```
1.      cast(calc, typeof(CS.Tutorial.Calc))
```

上面就是指定用 `CS.Tutorial.Calc` 的生成代码来访问 `calc` 对象。

练习

1、通过Lua代码实现冒泡排序逻辑，然后通过C#代码调用，实现一个数组的冒泡排序。

?

HelloWorld

执行**Lua**代码

1、引入命名空间 `using XLua;`

2、创建 `LuaEnv` 实例 `luaenv`

3、通过 `luaenv` 实例调用 `DoString` 方法，并传入 `Lua` 代码作为参数

4、 `CS.UnityEngine.Debug.Log('hello world')` 是通过 `Lua` 调用 `C#` 中的 `Debug.Log` 方法。

5、销毁 `luaenv` 对象 `luaenv.Dispose()`

```
1.      using UnityEngine;
2.      using XLua;
3.
4.      public class Helloworld : MonoBehaviour {
5.          // Use this for initialization
6.          void Start () {
7.          LuaEnv luaenv = new LuaEnv();
8.          luaenv.DoString("CS.UnityEngine.Debug.Log('hello world')");
9.          luaenv.Dispose();
10.     }
11.
12.         // Update is called once per frame
13.         void Update () {
14.
15.         }
16.     }
```

?

U3DScripting

LuaBehaviour.cs

```
1.      using UnityEngine;
2.      using System.Collections;
3.      using System.Collections.Generic;
4.      using XLua;
5.      using System;
6.
7.      [System.Serializable]
8.      public class Injection
9.      {
10.         public string name;
11.         public GameObject value;
12.     }
13.
14.     [LuaCallCSharp]
15.     public class LuaBehaviour : MonoBehaviour {
16.         public TextAsset luaScript;
17.         public Injection[] injections;
18.
19.         internal static LuaEnv luaEnv = new LuaEnv(); //all lua behaviour
    shared one luaenv only!
20.         internal static float lastGCTime = 0;
21.         internal const float GCInterval = 1;//1 second
22.
23.         private Action luaStart;
24.         private Action luaUpdate;
25.         private Action luaOnDestroy;
26.
27.         private LuaTable scriptEnv;
28.
29.         void Awake()
30.         {
31.             scriptEnv = luaEnv.NewTable();
32.
33.             LuaTable meta = luaEnv.NewTable();
34.             meta.Set("__index", luaEnv.Global);
35.             scriptEnv.SetMetaTable(meta);
36.             meta.Dispose();
```

```
37.
38.             scriptEnv.Set("self", this);
39.             foreach (var injection in injections)
40.             {
41.                 scriptEnv.Set(injection.name, injection.value);
42.             }
43.
44.             luaEnv.DoString(luaScript.text, "LuaBehaviour", scriptEnv);
45.
46.             Action luaAwake = scriptEnv.Get<Action>("awake");
47.             scriptEnv.Get("start", out luaStart);
48.             scriptEnv.Get("update", out luaUpdate);
49.             scriptEnv.Get("ondestroy", out luaOnDestroy);
50.
51.             if (luaAwake != null)
52.             {
53.                 luaAwake();
54.             }
55.         }
56.
57.         // Use this for initialization
58.         void Start ()
59.         {
60.             if (luaStart != null)
61.             {
62.                 luaStart();
63.             }
64.         }
65.
66.         // Update is called once per frame
67.         void Update ()
68.         {
69.             if (luaUpdate != null)
70.             {
71.                 luaUpdate();
72.             }
73.             if (Time.time - LuaBehaviour.lastGCTime > GCInterval)
74.             {
75.                 luaEnv.Tick();
76.                 LuaBehaviour.lastGCTime = Time.time;
77.             }
78.         }
```

```
79.
80.         void OnDestroy()
81.         {
82.             if (luaOnDestroy != null)
83.             {
84.                 luaOnDestroy();
85.             }
86.             luaOnDestroy = null;
87.             luaUpdate = null;
88.             luaStart = null;
89.             scriptEnv.Dispose();
90.             injections = null;
91.         }
92.     }
```

LuaTestScript.lua.txt

```
1.      local speed = 10
2.
3.      function start()
4.          print("lua start...")
5.      end
6.
7.      function update()
8.          local r = CS.UnityEngine.Vector3.up * CS.UnityEngine.Time.deltaTime *
    speed
9.          self.transform:Rotate(r)
10.     end
11.
12.     function ondestroy()
13.         print("lua destroy")
14.     end
```

?

UIEvent

ButtonInteraction.lua.txt

通过 `LuaBehaviour` 执行此 `Lua` 脚本，并通过Injections数组中添加一组 `input`

```
1. Name:input
2. Value:InputField
```

```
1.     function start()
2.         print("lua start...")
3.
4.         self:GetComponent("Button").onClick:AddListener(function()
5.             print("clicked, you input is '"
   ..input:GetComponent("InputField").text .."'")
6.         end)
7.     end
```

?

LuaObjectOrented

InvokeLua.cs

```
using UnityEngine;
using System.Collections;
using XLua;

public class InvokeLua : MonoBehaviour
{
    [CSharpCallLua]
    public interface ICalc
    {
        int Add(int a, int b);
        int Mult { get; set; }
    }

    [CSharpCallLua]
    public delegate ICalc CalcNew(int mult, params string[] args);

    private string script = @"
                local calc_mt = {
                    __index = {
                        Add = function(self, a, b)
                            return (a + b) * self.Mult
                        end
                    }
                }

                Calc = {
                    New = function (mult, ...)
                        print(...)
                        return setmetatable({Mult = mult}, calc_mt)
                    end
                }
            ";
    // Use this for initialization
    void Start()
    {
        LuaEnv luaenv = new LuaEnv();
        Test(luaenv);//调用了带可变参数的delegate，函数结束都不会释放delegate，即
```

```
    使置空并调用GC
            luaenv.Dispose();
        }

        void Test(LuaEnv luaenv)
        {
            luaenv.DoString(script);
            CalcNew calc_new = luaenv.Global.GetInPath<CalcNew>("Calc.New");
            ICalc calc = calc_new(10, "hi", "john"); //constructor
            Debug.Log("sum(*10) =" + calc.Add(1, 2));
            calc.Mult = 100;
            Debug.Log("sum(*100)=" + calc.Add(1, 2));
        }

        // Update is called once per frame
        void Update()
        {

        }
    }
```

?

NoGc

NoGc.cs

```
1.  using UnityEngine;
2.  using System;
3.  using XLua;
4.
5.  namespace XLuaTest
6.  {
7.      [GCOptimize]
8.      [LuaCallCSharp]
9.      public struct Pedding
10.     {
11.         public byte c;
12.     }
13.
14.     [GCOptimize]
15.     [LuaCallCSharp]
16.     public struct MyStruct
17.     {
18.         public MyStruct(int p1, int p2)
19.         {
20.             a = p1;
21.             b = p2;
22.             c = p2;
23.             e.c = (byte)p1;
24.         }
25.         public int a;
26.         public int b;
27.         public decimal c;
28.         public Pedding e;
29.     }
30.
31.     [LuaCallCSharp]
32.     public enum MyEnum
33.     {
34.         E1,
35.         E2
36.     }
37.
```

```
38.         [CSharpCallLua]
39.         public delegate int IntParam(int p);
40.
41.         [CSharpCallLua]
42.         public delegate Vector3 Vector3Param(Vector3 p);
43.
44.         [CSharpCallLua]
45.         public delegate MyStruct CustomValueTypeParam(MyStruct p);
46.
47.         [CSharpCallLua]
48.         public delegate MyEnum EnumParam(MyEnum p);
49.
50.         [CSharpCallLua]
51.         public delegate decimal DecimalParam(decimal p);
52.
53.         [CSharpCallLua]
54.         public delegate void ArrayAccess(Array arr);
55.
56.         [CSharpCallLua]
57.         public interface IExchanger
58.         {
59.             void exchange(Array arr);
60.         }
61.
62.         [LuaCallCSharp]
63.         public class NoGc : MonoBehaviour
64.         {
65.             LuaEnv luaenv = new LuaEnv();
66.
67.             IntParam f1;
68.             Vector3Param f2;
69.             CustomValueTypeParam f3;
70.             EnumParam f4;
71.             DecimalParam f5;
72.             ArrayAccess farr;
73.
74.             Action flua;
75.
76.             IExchanger ie;
77.
78.             LuaFunction add;
79.
```

```
80.             [NonSerialized]
81.             public double[] a1 = new double[] { 1, 2 };
82.             [NonSerialized]
83.             public Vector3[] a2 = new Vector3[] { new Vector3(1, 2, 3), new
    Vector3(4, 5, 6) };
84.             [NonSerialized]
85.             public MyStruct[] a3 = new MyStruct[] { new MyStruct(1, 2), new
    MyStruct(3, 4) };
86.             [NonSerialized]
87.             public MyEnum[] a4 = new MyEnum[] { MyEnum.E1, MyEnum.E2 };
88.             [NonSerialized]
89.             public decimal[] a5 = new decimal[] { 1.00001M, 2.00002M };
90.
91.             public float FloatParamMethod(float p)
92.             {
93.                 return p;
94.             }
95.
96.             public Vector3 Vector3ParamMethod(Vector3 p)
97.             {
98.                 return p;
99.             }
100.
101.            public MyStruct StructParamMethod(MyStruct p)
102.            {
103.                return p;
104.            }
105.
106.            public MyEnum EnumParamMethod(MyEnum p)
107.            {
108.                return p;
109.            }
110.
111.            public decimal DecimalParamMethod(decimal p)
112.            {
113.                return p;
114.            }
115.
116.            // Use this for initialization
117.            void Start()
118.            {
119.                luaenv.DoString(@"
```

```
                    function id(...)
                        return ...
                    end

                    function add(a, b) return a + b end

                    function array_exchange(arr)
                        arr[0], arr[1] = arr[1], arr[0]
                    end

                    local v3 = CS.UnityEngine.Vector3(7, 8, 9)
                    local vt = CS.XLuaTest.MyStruct(5, 6)

                    function lua_access_csharp()
                        monoBehaviour:FloatParamMethod(123) --primitive
                        monoBehaviour:Vector3ParamMethod(v3) --vector3
                        local rnd = math.random(1, 100)
                        local r = monoBehaviour:Vector3ParamMethod({x = 1, y = 2, z = rnd}) --vector3
                        assert(r.x == 1 and r.y == 2 and r.z == rnd)
                        monoBehaviour:StructParamMethod(vt) --custom struct
                        r = monoBehaviour:StructParamMethod({a = 1, b = rnd, e = {c = rnd}})
                        assert(r.b == rnd and r.e.c == rnd)
                        monoBehaviour:EnumParamMethod(CS.XLuaTest.MyEnum.E2) --enum
                        monoBehaviour:DecimalParamMethod(monoBehaviour.a5[0])
                        monoBehaviour.a1[0], monoBehaviour.a1[1] = monoBehaviour.a1[1], monoBehaviour.a1[0] -- field
                    end

                    exchanger = {
                        exchange = function(self, arr)
                            array_exchange(arr)
                        end
                    }

                    A = { B = { C = 789}}
                    GDATA = 1234;
                ");

                luaenv.Global.Set("monoBehaviour", this);
```

```
158.
159.                 luaenv.Global.Get("id", out f1);
160.                 luaenv.Global.Get("id", out f2);
161.                 luaenv.Global.Get("id", out f3);
162.                 luaenv.Global.Get("id", out f4);
163.                 luaenv.Global.Get("id", out f5);
164.                 luaenv.Global.Get("array_exchange", out farr);
165.                 luaenv.Global.Get("lua_access_csharp", out flua);
166.                 luaenv.Global.Get("exchanger", out ie);
167.                 luaenv.Global.Get("add", out add);
168.
169.                 luaenv.Global.Set("g_int", 123);
170.                 luaenv.Global.Set(123, 456);
171.                 int i;
172.                 luaenv.Global.Get("g_int", out i);
173.                 Debug.Log("g_int:" + i);
174.                 luaenv.Global.Get(123, out i);
175.                 Debug.Log("123:" + i);
176.             }
177.
178.
179.             // Update is called once per frame
180.             void Update()
181.             {
182.                 // c# call lua function with value type but no gc (using
    delegate)
183.                 f1(1); // primitive type
184.                 Vector3 v3 = new Vector3(1, 2, 3); // vector3
185.                 f2(v3);
186.                 MyStruct mystruct = new MyStruct(5, 6); // custom complex value
    type
187.                 f3(mystruct);
188.                 f4(MyEnum.E1); //enum
189.                 decimal d = -32132143143100109.00010001010M;
190.                 decimal dr = f5(d);
191.                 System.Diagnostics.Debug.Assert(d == dr);
192.
193.                 // using LuaFunction.Func<T1, T2, TResult>
194.                 System.Diagnostics.Debug.Assert(add.Func<int, int, int>(34, 56)
    == (34 + 56)); // LuaFunction.Func<T1, T2, TResult>
195.
196.                 // lua access c# value type array no gc
```

```
197.                 farr(a1); //primitive value type array
198.                 farr(a2); //vector3 array
199.                 farr(a3); //custom struct array
200.                 farr(a4); //enum arry
201.                 farr(a5); //decimal arry
202.
203.                 // lua call c# no gc with value type
204.                 flua();
205.
206.                 //c# call lua using interface
207.                 ie.exchange(a2);
208.
209.                 //no gc LuaTable use
210.                 luaenv.Global.Set("g_int", 456);
211.                 int i;
212.                 luaenv.Global.Get("g_int", out i);
213.                 System.Diagnostics.Debug.Assert(i == 456);
214.
215.                 luaenv.Global.Set(123.0001, mystruct);
216.                 MyStruct mystruct2;
217.                 luaenv.Global.Get(123.0001, out mystruct2);
218.                 System.Diagnostics.Debug.Assert(mystruct2.b == mystruct.b);
219.
220.                 decimal dr2 = 0.0000001M;
221.                 luaenv.Global.Set((byte)12, d);
222.                 luaenv.Global.Get((byte)12, out dr2);
223.                 System.Diagnostics.Debug.Assert(d == dr2);
224.
225.                 int gdata = luaenv.Global.Get<int>("GDATA");
226.                 luaenv.Global.SetInPath("GDATA", gdata + 1);
227.                 System.Diagnostics.Debug.Assert(luaenv.Global.Get<int>("GDATA")
    == gdata + 1);
228.
229.                 int abc = luaenv.Global.GetInPath<int>("A.B.C");
230.                 luaenv.Global.SetInPath("A.B.C", abc + 1);
231.                 System.Diagnostics.Debug.Assert(luaenv.Global.GetInPath<int>
    ("A.B.C") == abc + 1);
232.
233.                 luaenv.Tick();
234.             }
235.
236.             void OnDestroy()
```

```
            {
                f1 =  null;
                f2 = null;
                f3 = null;
                f4 = null;
                f5 = null;
                farr = null;
                flua = null;
                ie = null;
                add = null;
                luaenv.Dispose();
            }
        }
    }
```

?

Coroutine

CoroutineTest.cs

```
using UnityEngine;
using XLua;

public class CoroutineTest : MonoBehaviour {
    LuaEnv luaenv = null;
    // Use this for initialization
    void Start()
    {
        luaenv = new LuaEnv();
        luaenv.DoString("require 'coruntine_test'");
    }

    // Update is called once per frame
    void Update()
    {
        if (luaenv != null)
        {
            luaenv.Tick();
        }
    }

    void OnDestroy()
    {
        luaenv.Dispose();
    }
}
```

coroutine_test.lua.txt

```
local util = require 'xlua.util'

local yield_return = (require 'cs_coroutine').yield_return

local co = coroutine.create(function()
    print('coroutine start!')
    local s = os.time()
```

```
        yield_return(CS.UnityEngine.WaitForSeconds(3))
        print('wait interval:', os.time() - s)

        local www = CS.UnityEngine.WWW('http://www.qq.com')
        yield_return(www)
        if not www.error then
            print(www.bytes)
        else
            print('error:', www.error)
        end
    end)

    assert(coroutine.resume(co))
```

cs_coroutine.lua.txt

```
    local util = require 'xlua.util'

    local gameobject = CS.UnityEngine.GameObject('Coroutine_Runner')
    CS.UnityEngine.Object.DontDestroyOnLoad(gameobject)
    local cs_coroutine_runner =
gameobject:AddComponent(typeof(CS.Coroutine_Runner))

    local function async_yield_return(to_yield, cb)
        cs_coroutine_runner:YieldAndCallback(to_yield, cb)
    end

    return {
        yield_return = util.async_to_sync(async_yield_return)
    }
```

Coroutine_Runner.cs

```
    using UnityEngine;
    using XLua;
    using System.Collections.Generic;
    using System.Collections;
    using System;

    [LuaCallCSharp]
    public class Coroutine_Runner : MonoBehaviour
```

```
    {
        public void YieldAndCallback(object to_yield, Action callback)
        {
            StartCoroutine(CoBody(to_yield, callback));
        }

        private IEnumerator CoBody(object to_yield, Action callback)
        {
            if (to_yield is IEnumerator)
                yield return StartCoroutine((IEnumerator)to_yield);
            else
                yield return to_yield;
            callback();
        }
    }

    public static class CoroutineConfig
    {
        [LuaCallCSharp]
        public static List<Type> LuaCallCSharp
        {
            get
            {
                return new List<Type>()
                {
                    typeof(WaitForSeconds),
                    typeof(WWW)
                };
            }
        }
    }
```

?

AsyncTest

AsyncTest.cs

```
using UnityEngine;
using XLua;
using System.Collections.Generic;
using System;

public class AsyncTest : MonoBehaviour
{
    LuaEnv luaenv = null;
    void Start()
    {
        luaenv = new LuaEnv();
        luaenv.DoString("require 'async_test'");
    }

    void Update()
    {
        if (luaenv != null)
        {
            luaenv.Tick();
        }
    }
}
```

async_test.lua.txt

```
local util = require 'xlua.util'
local message_box = require 'message_box'

-------------------------async_recharge-----------------------------
local function async_recharge(num, cb) --模拟的异步充值
    print('requst server...')
    cb(true, num)
end

local recharge = util.async_to_sync(async_recharge)
-------------------------async_recharge end-----------------------------
```

```
    local buy = function()
        message_box.alert("余额提醒","您余额不足，请充值！")
        if message_box.confirm("确认充值10元吗？","确认框" ) then
            local r1, r2 = recharge(10)
            print('recharge result', r1, r2)
            message_box.alert("提示","充值成功！")
        else
            print('cancel')
            message_box.alert("提示","取消充值！")
        end
    end
    --将按钮监听点击事件，绑定buy方法

CS.UnityEngine.GameObject.Find("Button"):GetComponent("Button").onClick:AddLister
```

message_box.lua.txt

```
    local util = require 'xlua.util'

    local sync_alert = util.async_to_sync(CS.MessageBox.ShowAlertBox)
    local sync_confirm = util.async_to_sync(CS.MessageBox.ShowConfirmBox)

    --构造alert和confirm函数
    return {
        alert = function(title, message)
            if not message then
                title, message = message, title
            end
             sync_alert(message,title)
        end;

        confirm = function(title, message)
            local ret = sync_confirm(title,message)
            return ret == true
        end;
     }
```

MessageBox.cs

```
    using UnityEngine;
    using UnityEngine.UI;
```

```
3.      using XLua;
4.      using System.Collections.Generic;
5.      using System;
6.      using UnityEngine.Events;
7.
8.      public class MessageBox : MonoBehaviour{
9.
10.         public static void ShowAlertBox(string message, string title, Action
    onFinished = null)
11.         {
12.             var alertPanel =
    GameObject.Find("Canvas").transform.Find("AlertBox");
13.             if (alertPanel == null)
14.             {
15.                 alertPanel = (Instantiate(Resources.Load("AlertBox")) as
    GameObject).transform;
16.                 alertPanel.gameObject.name = "AlertBox";
17.                 alertPanel.SetParent(GameObject.Find("Canvas").transform);
18.                 alertPanel.localPosition = new Vector3(-6f, -6f, 0f);
19.             }
20.             alertPanel.Find("title").GetComponent<Text>().text = title;
21.             alertPanel.Find("message").GetComponent<Text>().text = message;
22.             alertPanel.gameObject.SetActive(true);
23.             if (onFinished != null)
24.             {
25.                 var button = alertPanel.Find("alertBtn").GetComponent<Button>
    ();
26.                 UnityAction onclick = null;
27.                 onclick = () =>
28.                 {
29.                     onFinished();
30.                     alertPanel.gameObject.SetActive(false);
31.                     button.onClick.RemoveListener(onclick);
32.                 };
33.                 button.onClick.RemoveAllListeners();
34.                 button.onClick.AddListener(onclick);
35.             }
36.         }
37.
38.         public  static void ShowConfirmBox(string message, string title,
    Action<bool> onFinished = null)
39.         {
```

```
            var confirmPanel =
GameObject.Find("Canvas").transform.Find("ConfirmBox");
            if (confirmPanel == null)
            {
                confirmPanel = (Instantiate(Resources.Load("ConfirmBox")) as
GameObject).transform;
                confirmPanel.gameObject.name = "ConfirmBox";
                confirmPanel.SetParent(GameObject.Find("Canvas").transform);
                confirmPanel.localPosition = new Vector3(-8f, -18f, 0f);
            }
            confirmPanel.Find("confirmTitle").GetComponent<Text>().text =
title;
            confirmPanel.Find("conmessage").GetComponent<Text>().text =
message;
            confirmPanel.gameObject.SetActive(true);
            if (onFinished != null)
            {
                var confirmBtn =
confirmPanel.Find("confirmBtn").GetComponent<Button>();
                var cancelBtn =
confirmPanel.Find("cancelBtn").GetComponent<Button>();
                UnityAction onconfirm = null;
                UnityAction oncancel = null;

                Action cleanup = () =>
                {
                    confirmBtn.onClick.RemoveListener(onconfirm);
                    cancelBtn.onClick.RemoveListener(oncancel);
                    confirmPanel.gameObject.SetActive(false);
                };

                onconfirm = () =>
                {
                    onFinished(true);
                    cleanup();
                };

                oncancel = () =>
                {
                    onFinished(false);
                    cleanup();
                };
```

```
76.                 confirmBtn.onClick.RemoveAllListeners();
77.                 cancelBtn.onClick.RemoveAllListeners();
78.                 confirmBtn.onClick.AddListener(onconfirm);
79.                 cancelBtn.onClick.AddListener(oncancel);
80.             }
81.         }
82.     }
83.
84.     public static class MessageBoxConfig
85.     {
86.         [CSharpCallLua]
87.         public static List<Type> CSharpCallLua = new List<Type>()
88.         {
89.             typeof(Action),
90.             typeof(Action<bool>),
91.             typeof(UnityAction),
92.         };
93.     }
```

?

Hotfix

HotfixTest.cs

```
1.      using UnityEngine;
2.      using XLua;
3.
4.      [Hotfix]
5.      public class HotfixTest : MonoBehaviour
6.      {
7.          LuaEnv luaenv = new LuaEnv();
8.
9.          public int tick = 0; //如果是private的，在lua设置
    xlua.private_accessible(CS.HotfixTest)后即可访问
10.
11.         // Use this for initialization
12.         void Start()
13.         {
14.         }
15.
16.         // Update is called once per frame
17.         void Update()
18.         {
19.             if (++tick % 50 == 0)
20.             {
21.                 Debug.Log(">>>>>>>>Update in C#, tick = " + tick);
22.             }
23.         }
24.
25.         void OnGUI()
26.         {
27.             if (GUI.Button(new Rect(10, 10, 300, 80), "Hotfix"))
28.             {
29.                 luaenv.DoString(@"
30.                     xlua.hotfix(CS.HotfixTest, 'Update', function(self)
31.                         self.tick = self.tick + 1
32.                         if (self.tick % 50) == 0 then
33.                             print('<<<<<<<<Update in lua, tick = ' ..
    self.tick)
34.                         end
35.                     end)
```

```
                ");
            }

            string chHint = @"在运行该示例之前，请细致阅读xLua文档，并执行以下步骤：

    1.宏定义：添加 HOTFIX_ENABLE 到 'Edit > Project Settings > Player > Other Settings > Scripting Define Symbols'。
    （注意：各平台需要分别设置）

    2.生成代码：执行 'XLua > Generate Code' 菜单，等待Unity编译完成。

    3.注入：执行 'XLua > Hotfix Inject In Editor' 菜单。注入成功会打印 'hotfix inject finish!' 或者 'had injected!' 。";
            string enHint = @"Read documents carefully before you run this example, then follow the steps below:

    1. Define: Add 'HOTFIX_ENABLE' to 'Edit > Project Settings > Player > Other Settings > Scripting Define Symbols'.
    (Note: Each platform needs to set this respectively)

    2.Generate Code: Execute menu 'XLua > Generate Code', wait for Unity's compilation.


    3.Inject: Execute menu 'XLua > Hotfix Inject In Editor'.There should be 'hotfix inject finish!' or 'had injected!' print in the Console if the Injection is successful.";
            GUIStyle style = GUI.skin.textArea;
            style.normal.textColor = Color.red;
            style.fontSize = 16;
            GUI.TextArea(new Rect(10, 100, 500, 290), chHint, style);
            GUI.TextArea(new Rect(10, 400, 500, 290), enHint, style);
        }
    }
```

HotfixTest2.cs

```
    using UnityEngine;
    using System.Collections.Generic;
    using XLua;

    [CSharpCallLua]
```

```
6.      public delegate int TestOutDelegate(HotfixCalc calc, int a, out double b,
    ref string c);
7.
8.      [Hotfix]
9.      public class HotfixCalc
10.     {
11.         public int Add(int a, int b)
12.         {
13.             return a - b;
14.         }
15.
16.         public Vector3 Add(Vector3 a, Vector3 b)
17.         {
18.             return a - b;
19.         }
20.
21.         public int TestOut(int a, out double b, ref string c)
22.         {
23.             b = a + 2;
24.             c = "wrong version";
25.             return a + 3;
26.         }
27.
28.         public int TestOut(int a, out double b, ref string c, GameObject go)
29.         {
30.             return TestOut(a, out b, ref c);
31.         }
32.
33.         public T Test1<T>()
34.         {
35.             return default(T);
36.         }
37.
38.         public T1 Test2<T1, T2, T3>(T1 a, out T2 b, ref T3 c)
39.         {
40.             b = default(T2);
41.             return a;
42.         }
43.
44.         public static int Test3<T>(T a)
45.         {
46.             return 0;
```

```
47.         }
48.
49.         public static void Test4<T>(T a)
50.         {
51.         }
52.
53.         public void Test5<T>(int a, params T[] arg)
54.         {
55.
56.         }
57.     }
58.
59.     public class NoHotfixCalc
60.     {
61.         public int Add(int a, int b)
62.         {
63.             return a + b;
64.         }
65.     }
66.
67.     [Hotfix]
68.     public class GenericClass<T>
69.     {
70.         T a;
71.
72.         public GenericClass(T a)
73.         {
74.             this.a = a;
75.         }
76.
77.         public void Func1()
78.         {
79.             Debug.Log("a=" + a);
80.         }
81.
82.         public T Func2()
83.         {
84.             return default(T);
85.         }
86.     }
87.
88.     [Hotfix]
```

```
89.     public class InnerTypeTest
90.     {
91.         public void Foo()
92.         {
93.             _InnerStruct ret = Bar();
94.             Debug.Log("{x=" + ret.x + ",y= " + ret.y + "}");
95.         }
96.
97.         struct _InnerStruct
98.         {
99.             public int x;
100.            public int y;
101.        }
102.
103.        _InnerStruct Bar()
104.        {
105.            return new _InnerStruct { x = 1, y = 2 };
106.        }
107.    }
108.
109.    [Hotfix]
110.    public struct StructTest
111.    {
112.        GameObject go;
113.        public StructTest(GameObject go)
114.        {
115.            this.go = go;
116.        }
117.
118.        public GameObject GetGo(int a, object b)
119.        {
120.            return go;
121.        }
122.    }
123.
124.    [Hotfix(HotfixFlag.Stateful)]
125.    public struct GenericStruct<T>
126.    {
127.        T a;
128.
129.        public GenericStruct(T a)
130.        {
```

```
            this.a = a;
        }

        public T GetA(int p)
        {
            return a;
        }
    }

    public class HotfixTest2 : MonoBehaviour {

        // Use this for initialization
        void Start () {
            LuaEnv luaenv = new LuaEnv();
            HotfixCalc calc = new HotfixCalc();
            NoHotfixCalc ordinaryCalc = new NoHotfixCalc();

            int CALL_TIME = 100 * 1000 * 1000 ;
            var start = System.DateTime.Now;
            for (int i = 0; i < CALL_TIME; i++)
            {
                calc.Add(2, 1);
            }
            var d1 = (System.DateTime.Now - start).TotalMilliseconds;
            Debug.Log("Hotfix using:" + d1);

            start = System.DateTime.Now;
            for (int i = 0; i < CALL_TIME; i++)
            {
                ordinaryCalc.Add(2, 1);
            }
            var d2 = (System.DateTime.Now - start).TotalMilliseconds;
            Debug.Log("No Hotfix using:" + d2);

            Debug.Log("drop:" + ((d1 - d2) / d1));

            Debug.Log("Before Fix: 2 + 1 = " + calc.Add(2, 1));
            Debug.Log("Before Fix: Vector3(2, 3, 4) + Vector3(1, 2, 3) = " +
    calc.Add(new Vector3(2, 3, 4), new Vector3(1, 2, 3)));
            luaenv.DoString(@"
                xlua.hotfix(CS.HotfixCalc, 'Add', function(self, a, b)
                    return a + b
```

```
                end)
            ");
            Debug.Log("After Fix: 2 + 1 = " + calc.Add(2, 1));
            Debug.Log("After Fix: Vector3(2, 3, 4) + Vector3(1, 2, 3) = " +
    calc.Add(new Vector3(2, 3, 4), new Vector3(1, 2, 3)));

            double num;
            string str = "hehe";
            int ret = calc.TestOut(100, out num, ref str);
            Debug.Log("ret = " + ret + ", num = " + num + ", str = " + str);

            luaenv.DoString(@"
                xlua.hotfix(CS.HotfixCalc, 'TestOut', function(self, a, c, go)
                        print('TestOut', self, a, c, go)
                        if go then error('test error') end
                        return a + 10, a + 20, 'right version'
                    end)
            ");
            str = "hehe";
            ret = calc.TestOut(100, out num, ref str);
            Debug.Log("ret = " + ret + ", num = " + num + ", str = " + str);

            luaenv.DoString(@"
                xlua.hotfix(CS.HotfixCalc, {
                     Test1 = function(self)
                        print('Test1', self)
                        return 1
                     end;
                     Test2 = function(self, a, b)
                         print('Test1', self, a, b)
                         return a + 10, 1024, b
                     end;
                     Test3 = function(a)
                        print(a)
                        return 10
                     end;
                     Test4 = function(a)
                        print(a)
                     end;
                     Test5 = function(self, a, ...)
                        print('Test4', self, a, ...)
                     end
```

```
                }
            ");

            int r1 = calc.Test1<int>();
            double r2 = calc.Test1<double>();

            Debug.Log("r1:" + r1 + ",r2:" + r2);

            string ss = "heihei";
            int r3 = calc.Test2(r1, out r2, ref ss);
            Debug.Log("r1:" + r1 + ",r2:" + r2 + ",r3:" + r3 + ",ss:" + ss);

            r3 = HotfixCalc.Test3("test3");
            r3 = HotfixCalc.Test3(2);
            r3 = HotfixCalc.Test3(this);
            Debug.Log("r3:" + r3);
            HotfixCalc.Test4(this);
            HotfixCalc.Test4(2);
            calc.Test5(10, "a", "b", "c");
            calc.Test5(10, 1, 3, 5);

            Debug.Log("----------------------before-----------------------");
            TestStateful();
            System.GC.Collect();
            System.GC.WaitForPendingFinalizers();
            luaenv.DoString(@"
                xlua.hotfix(CS.StatefullTest, {
                    ['.ctor'] = function(csobj)
                        return {evt = {}, start = 0}
                    end;
                    set_AProp = function(self, v)
                        print('set_AProp', v)
                        self.AProp = v
                    end;
                    get_AProp = function(self)
                        return self.AProp
                    end;
                    get_Item = function(self, k)
                        print('get_Item', k)
                        return 1024
                    end;
                    set_Item = function(self, k, v)
```

```
                        print('set_Item', k, v)
                    end;
                    add_AEvent = function(self, cb)
                        print('add_AEvent', cb)
                        table.insert(self.evt, cb)
                    end;
                    remove_AEvent = function(self, cb)
                       print('remove_AEvent', cb)
                       for i, v in ipairs(self.evt) do
                           if v == cb then
                               table.remove(self.evt, i)
                               break
                           end
                       end
                    end;
                    Start = function(self)
                        print('Start')
                        for _, cb in ipairs(self.evt) do
                            cb(self.start, 2)
                        end
                        self.start = self.start + 1
                    end;
                    StaticFunc = function(a, b, c)
                       print(a, b, c)
                    end;
                    GenericTest = function(self, a)
                       print(self, a)
                    end;
                    Finalize = function(self)
                       print('Finalize', self)
                    end
                })
            ");
            Debug.Log("----------------------after-----------------------");
            TestStateful();
            luaenv.FullGc();
            System.GC.Collect();
            System.GC.WaitForPendingFinalizers();

            var genericObj = new GenericClass<double>(1.1);
            genericObj.Func1();
            Debug.Log(genericObj.Func2());
```

```
            luaenv.DoString(@"
                xlua.hotfix(CS['GenericClass`1[System.Double]'], {
                    ['.ctor'] = function(obj, a)
                        print('GenericClass<double>', obj, a)
                    end;
                    Func1 = function(obj)
                        print('GenericClass<double>.Func1', obj)
                    end;
                    Func2 = function(obj)
                        print('GenericClass<double>.Func2', obj)
                        return 1314
                    end
                })
            ");
            genericObj = new GenericClass<double>(1.1);
            genericObj.Func1();
            Debug.Log(genericObj.Func2());

            InnerTypeTest itt = new InnerTypeTest();
            itt.Foo();
            luaenv.DoString(@"
                xlua.hotfix(CS.InnerTypeTest, 'Bar', function(obj)
                        print('lua Bar', obj)
                        return {x = 10, y = 20}
                    end)
            ");
            itt.Foo();

            StructTest st = new StructTest(gameObject);
            Debug.Log("go=" + st.GetGo(123, "john"));
            luaenv.DoString(@"
                xlua.hotfix(CS.StructTest, 'GetGo', function(self, a, b)
                        print('GetGo', self, a, b)
                        return nil
                    end)
            ");
            Debug.Log("go=" + st.GetGo(123, "john"));

            GenericStruct<int> gs = new GenericStruct<int>(1);
            Debug.Log("gs.GetA()=" + gs.GetA(123));
            luaenv.DoString(@"
                xlua.hotfix(CS['GenericStruct`1[System.Int32]'], 'GetA',
```

```
    function(self, a)
                        print('GetA',self, a)
                        return 789
                    end)
            ");
            Debug.Log("gs.GetA()=" + gs.GetA(123));

            try
            {
                calc.TestOut(100, out num, ref str, gameObject);
            }
            catch(LuaException e)
            {
                Debug.Log("throw in lua an catch in c# ok, e.Message:" +
    e.Message);
            }
        }

        void TestStateful()
        {
            StatefullTest sft = new StatefullTest();
            sft.AProp = 10;
            Debug.Log("sft.AProp:" + sft.AProp);
            sft["1"] = 1;
            Debug.Log("sft['1']:" + sft["1"]);
            System.Action<int, double> cb = (a, b) =>
            {
                Debug.Log("a:" + a + ",b:" + b);
            };
            sft.AEvent += cb;
            sft.Start();
            sft.Start();
            sft.AEvent -= cb;
            sft.Start();
            StatefullTest.StaticFunc(1, 2);
            StatefullTest.StaticFunc("e", 3, 4);
            sft.GenericTest(1);
            sft.GenericTest("hehe");
        }

        // Update is called once per frame
        void Update () {
```

```
379.
380.         }
381.     }
```

StatefullTest.cs

```
 1.     using UnityEngine;
 2.     using System.Collections;
 3.
 4.     [XLua.Hotfix(XLua.HotfixFlag.Stateful)]
 5.     public class StatefullTest {
 6.         public StatefullTest()
 7.         {
 8.
 9.         }
10.
11.         public StatefullTest(int a, int b)
12.         {
13.             if (a > 0)
14.             {
15.                 return;
16.             }
17.
18.             Debug.Log("a=" + a);
19.             if (b > 0)
20.             {
21.                 return;
22.             }
23.             else
24.             {
25.                 if (a + b > 0)
26.                 {
27.                     return;
28.                 }
29.             }
30.             Debug.Log("b=" + b);
31.         }
32.
33.         public int AProp
34.         {
35.             get;
36.             set;
```

```
        }

        public event System.Action<int, double> AEvent;

        public int this[string field]
        {
            get
            {
                return 1;
            }
            set
            {
            }
        }

        public void Start () {

        }

        void Update () {

        }

        public void GenericTest<T>(T a)
        {

        }

        static public void StaticFunc(int a, int b)
        {
        }
        static public void StaticFunc(string a, int b, int c)
        {
        }

        ~StatefullTest()
        {
            Debug.Log("~StatefullTest");
        }
    }
```

?

GenericMethod

GenericMethodExample.cs

```
1.  using UnityEngine;
2.  using XLua;
3.
4.  public class GenericMethodExample : MonoBehaviour
5.  {
6.      private const string script = @"
7.          local foo1 = CS.Foo1Child()
8.          local foo2 = CS.Foo2Child()
9.
10.         local obj = CS.UnityEngine.GameObject()
11.         foo1:PlainExtension()
12.         foo1:Extension1()
13.         foo1:Extension2(obj) -- overload1
14.         foo1:Extension2(foo2) -- overload2
15.
16.         local foo = CS.Foo()
17.         foo:Test1(foo1)
18.         foo:Test2(foo1,foo2,obj)
19. ";
20.     private LuaEnv env;
21.
22.     private void Start()
23.     {
24.         env = new LuaEnv();
25.         env.DoString(script);
26.     }
27.
28.     private void Update()
29.     {
30.         if (env != null)
31.             env.Tick();
32.     }
33.
34.     private void OnDestroy()
35.     {
36.         env.Dispose();
37.     }
```

```
    }
```

Foo.cs

```
    using System;
    using System.IO;
    using System.Collections.Generic;
    using UnityEngine;
    using XLua;

    [LuaCallCSharp]
    public class Foo1Parent
    {
    }

    [LuaCallCSharp]
    public class Foo2Parent
    {
    }

    [LuaCallCSharp]
    public class Foo1Child : Foo1Parent
    {
    }

    [LuaCallCSharp]
    public class Foo2Child : Foo2Parent
    {
    }

    [LuaCallCSharp]
    public class Foo
    {
        #region Supported methods

        public void Test1<T>(T a) where T : Foo1Parent
        {
            Debug.Log(string.Format("Test1<{0}>", typeof (T)));
        }

        public T1 Test2<T1, T2>(T1 a, T2 b, GameObject c) where T1 : Foo1Parent
    where T2 : Foo2Parent
```

```
38.         {
39.             Debug.Log(string.Format("Test2<{0},{1}>", typeof (T1), typeof
    (T2)), c);
40.             return a;
41.         }
42.
43.         #endregion
44.
45.         #region Unsupported methods
46.
47.         /// <summary>
48.         /// 不支持生成lua的泛型方法（没有泛型约束）
49.         /// </summary>
50.         public void UnsupportedMethod1<T>(T a)
51.         {
52.             Debug.Log("UnsupportedMethod1");
53.         }
54.
55.         /// <summary>
56.         /// 不支持生成lua的泛型方法（缺少带约束的泛型参数）
57.         /// </summary>
58.         public void UnsupportedMethod2<T>() where T : Foo1Parent
59.         {
60.             Debug.Log(string.Format("UnsupportedMethod2<{0}>",typeof(T)));
61.         }
62.
63.         /// <summary>
64.         /// 不支持生成lua的泛型方法（泛型约束必须为class）
65.         /// </summary>
66.         public void UnsupportedMethod3<T>(T a) where T : IDisposable
67.         {
68.             Debug.Log(string.Format("UnsupportedMethod3<{0}>", typeof(T)));
69.         }
70.
71.         #endregion
72.     }
73.
74.     [LuaCallCSharp]
75.     public static class FooExtension
76.     {
77.         public static void PlainExtension(this Foo1Parent a)
78.         {
```

```
79.             Debug.Log("PlainExtension");
80.         }
81.
82.         public static T Extension1<T>(this T a) where T : Foo1Parent
83.         {
84.             Debug.Log(string.Format("Extension1<{0}>", typeof (T)));
85.             return a;
86.         }
87.
88.         public static T Extension2<T>(this T a, GameObject b) where T :
    Foo1Parent
89.         {
90.             Debug.Log(string.Format("Extension2<{0}>", typeof (T)), b);
91.             return a;
92.         }
93.
94.         public static void Extension2<T1, T2>(this T1 a, T2 b) where T1 :
    Foo1Parent where T2 : Foo2Parent
95.         {
96.             Debug.Log(string.Format("Extension2<{0},{1}>", typeof (T1), typeof
    (T2)));
97.         }
98.
99.         public static T UnsupportedExtension<T>(this GameObject obj) where T :
    Component
100.         {
101.             return obj.GetComponent<T>();
102.         }
103.     }
```

?

SignatureLoader

SignatureLoaderTest.cs

```
1.  using UnityEngine;
2.  using System.Collections;
3.  using XLua;
4.  using System.IO;
5.
6.  public class SignatureLoaderTest : MonoBehaviour {
7.      public static string PUBLIC_KEY =
"BgIAAACkAABSU0ExAAQAAAEAAQBVDDC5QJ+0uSCJA+EysIC9JBzIsd6wcXa+FuTGXcsJuwyUkabwIiT2
8.
9.      // Use this for initialization
10.     void Start () {
11.         LuaEnv luaenv = new LuaEnv();
12. #if UNITY_EDITOR
13.         luaenv.AddLoader(new SignatureLoader(PUBLIC_KEY, (ref string
filepath) =>
14.         {
15.             filepath = Application.dataPath +
"/XLua/Examples/10_SignatureLoader/" + filepath.Replace('.', '/') + ".lua";
16.             if (File.Exists(filepath))
17.             {
18.                 return File.ReadAllBytes(filepath);
19.             }
20.             else
21.             {
22.                 return null;
23.             }
24.         }));
25. #else //为了让手机也能测试
26.         luaenv.AddLoader(new SignatureLoader(PUBLIC_KEY, (ref string
filepath) =>
27.         {
28.             filepath = filepath.Replace('.', '/') + ".lua";
29.             TextAsset file = (TextAsset)Resources.Load(filepath);
30.             if (file != null)
31.             {
32.                 return file.bytes;
33.             }
```

```
34.                 else
35.                 {
36.                     return null;
37.                 }
38.             }));
39.     #endif
40.             luaenv.DoString(@"
41.                 require 'signatured1'
42.                 require 'signatured2'
43.             ");
44.             luaenv.Dispose();
45.         }
46.
47.         // Update is called once per frame
48.         void Update () {
49.
50.         }
51.     }
```

RawObject

RawObjectTest.cs

```
1.  using UnityEngine;
2.  using XLua;
3.
4.  namespace XLuaTest
5.  {
6.      public class IntObject : RawObject
7.      {
8.          int mTarget;
9.
10.         public IntObject(int i)
11.         {
12.             mTarget = i;
13.         }
14.
15.         public object Target
16.         {
17.             get
18.             {
19.                 return mTarget;
20.             }
21.         }
22.     }
23.
24.     public class RawObjectTest : MonoBehaviour
25.     {
26.         public static void PrintType(object o)
27.         {
28.             Debug.Log("type:" + o.GetType() + ", value:" + o);
29.         }
30.
31.         // Use this for initialization
32.         void Start()
33.         {
34.             LuaEnv luaenv = new LuaEnv();
35.             //直接传1234到一个object参数，xLua将选择能保留最大精度的long来传递
36.             luaenv.DoString("CS.XLuaTest.RawObjectTest.PrintType(1234)");
37.             //通过一个继承RawObject的类，能实现指明以一个int来传递
```

```
38.
    luaenv.DoString("CS.XLuaTest.RawObjectTest.PrintType(CS.XLuaTest.IntObject(1234))
39.                 luaenv.Dispose();
40.             }
41.
42.             // Update is called once per frame
43.             void Update()
44.             {
45.
46.             }
47.         }
48.     }
```

?

ReImplementInLua

ReImplementInLua.cs

```
1.      using UnityEngine;
2.      using System.Collections;
3.      using XLua;
4.
5.      [GCOptimize(OptimizeFlag.PackAsTable)]
6.      public struct PushAsTableStruct
7.      {
8.          public int x;
9.          public int y;
10.     }
11.
12.     public class ReImplementInLua : MonoBehaviour {
13.
14.         // Use this for initialization
15.         void Start () {
16.             LuaEnv luaenv = new LuaEnv();
17.             //这两个例子都必须生成代码才能正常运行
18.             //例子1：改造Vector3
19.             //沿用Vector3原来的映射方案Vector3 -> userdata，但是把Vector3的方法实现改为lua实现，通过xlua.genaccessor实现不经过C#直接操作内存
20.             //改为不经过C#的好处是性能更高，而且你可以省掉相应的生成代码以达成省text段的效果
21.             //映射仍然沿用的好处是userdata比table更省内存，但操作字段比table性能稍低，当然，你也可以结合例子2的思路，把Vector3也改为映射到table
22.             luaenv.DoString(@"
23.                 function test_vector3(title, v1, v2)
24.                    print(title)
25.                    print(v1.x, v1.y, v1.z)
26.                    print(v1, v2)
27.                    print(v1 + v2)
28.                    v1:Set(v1.x - v2.x, v1.y - v2.y, v1.z - v2.z)
29.                    print(v1)
30.                    print(CS.UnityEngine.Vector3.Normalize(v1))
31.                 end
32.                 test_vector3('----before change metatable----', CS.UnityEngine.Vector3(1, 2, 3), CS.UnityEngine.Vector3(7, 8, 9))
33.
```

```
34.                 local get_x, set_x = xlua.genaccessor(0, 8)
35.                 local get_y, set_y = xlua.genaccessor(4, 8)
36.                 local get_z, set_z = xlua.genaccessor(8, 8)
37.
38.                 local fields_getters = {
39.                     x = get_x, y = get_y, z = get_z
40.                 }
41.                 local fields_setters = {
42.                     x = set_x, y = set_y, z = set_z
43.                 }
44.
45.                 local ins_methods = {
46.                     Set = function(o, x, y, z)
47.                         set_x(o, x)
48.                         set_y(o, y)
49.                         set_z(o, z)
50.                     end
51.                 }
52.
53.                 local mt = {
54.                     __index = function(o, k)
55.                         --print('__index', k)
56.                         if ins_methods[k] then return ins_methods[k] end
57.                         return fields_getters[k] and fields_getters[k](o)
58.                     end,
59.
60.                     __newindex = function(o, k, v)
61.                         return fields_setters[k] and fields_setters[k](o, v) or
    error('no such field ' .. k)
62.                     end,
63.
64.                     __tostring = function(o)
65.                         return string.format('vector3 { %f, %f, %f}', o.x, o.y,
    o.z)
66.                     end,
67.
68.                     __add = function(a, b)
69.                         return CS.UnityEngine.Vector3(a.x + b.x, a.y + b.y, a.z
    + b.z)
70.                     end
71.                 }
72.
```

```
                xlua.setmetatable(CS.UnityEngine.Vector3, mt)
                test_vector3('----after change metatable----',
    CS.UnityEngine.Vector3(1, 2, 3), CS.UnityEngine.Vector3(7, 8, 9))
            ");

    Debug.Log("++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++");

            //例子2：struct映射到table改造
            //PushAsTableStruct传送到lua侧将会是table，例子里头还为这个table添加了一个
    成员方法SwapXY，静态方法Print，打印格式化，以及构造函数
            luaenv.DoString(@"
                local mt = {
                    __index = {
                        SwapXY = function(o) --成员函数
                            o.x, o.y = o.y, o.x
                        end
                    },

                    __tostring = function(o) --打印格式化函数
                        return string.format('struct { %d, %d}', o.x, o.y)
                    end,
                }

                xlua.setmetatable(CS.PushAsTableStruct, mt)

                local PushAsTableStruct = {
                    Print = function(o) --静态函数
                        print(o.x, o.y)
                    end
                }

                setmetatable(PushAsTableStruct, {
                    __call = function(_, x, y) --构造函数
                        return setmetatable({x = x, y = y}, mt)
                    end
                })

                xlua.setclass(CS, 'PushAsTableStruct', PushAsTableStruct)
            ");

            PushAsTableStruct test;
```

```
112.             test.x = 100;
113.             test.y = 200;
114.             luaenv.Global.Set("from_cs", test);
115.
116.             luaenv.DoString(@"
117.                 print('--------------from csharp----------------------')
118.                 assert(type(from_cs) == 'table')
119.                 print(from_cs)
120.                 CS.PushAsTableStruct.Print(from_cs)
121.                 from_cs:SwapXY()
122.                 print(from_cs)
123.
124.                 print('--------------from lua----------------------')
125.                 local from_lua = CS.PushAsTableStruct(4, 5)
126.                 assert(type(from_lua) == 'table')
127.                 print(from_lua)
128.                 CS.PushAsTableStruct.Print(from_lua)
129.                 from_lua:SwapXY()
130.                 print(from_lua)
131.             ");
132.
133.             luaenv.Dispose();
134.         }
135.
136.         // Update is called once per frame
137.         void Update () {
138.
139.         }
140.     }
```

?

xLua的配置

`xLua` 所有的配置都支持三种方式：打标签；静态列表；动态列表。

打标签

`xLua` 用白名单来指明生成哪些代码，而白名单通过 `attribute` 来配置，比如你想从 `lua` 调用 `c#` 的某个类，希望生成适配代码，你可以为这个类型打一个 `LuaCallCSharp` 标签：

```
1.      [LuaCallCSharp]
2.      public class A
3.      {
4.      }
```

该方式方便，但在 `il2cpp` 下会增加不少的代码量，不建议使用。

静态列表

有时我们无法直接给一个类型打标签，比如系统 `api` ，没源码的库，或者实例化的泛化类型，这时你可以在一个静态类里声明一个静态字段，该字段的类型
除 `BlackList` 和 `AdditionalProperties` 之外只要实现了 `IEnumerable<Type>` 就可以了（这两个例外后面具体会说），然后为这字段加上标签：

```
1.      [LuaCallCSharp]
2.      public static List<Type> mymodule_lua_call_cs_list = new List<Type>()
3.      {
4.          typeof(GameObject),
5.          typeof(Dictionary<string, int>),
6.      };
```

这个字段需要放到一个静态类里头，建议放到 `Editor` 目录。

动态列表

声明一个静态属性，打上相应的标签即可。

```
1.      [Hotfix]
2.      public static List<Type> by_property
3.      {
4.          get
5.          {
```

```
            return (from type in Assembly.GetExecutingAssembly().GetTypes()
                    where type.Namespace == "XXXX"
                    select type).ToList();
        }
    }
```

`Getter` 是代码，你可以实现很多效果，比如按名字空间配置，按程序集配置等等。

这个属性需要放到一个静态类里头，建议放到 `Editor` 目录。

XLua.LuaCallCSharp

一个 `C#` 类型加了这个配置， `xLua` 会生成这个类型的适配代码（包括构造该类型实例，访问其成员属性、方法，静态属性、方法），否则将会尝试用性能较低的反射方式来访问。

一个类型的扩展方法（ `Extension Methods` ）加了这配置，也会生成适配代码并追加到被扩展类型的成员方法上。

`xLua` 只会生成加了该配置的类型，不会自动生成其父类的适配代码，当访问子类对象的父类方法，如果该父类加了 `LuaCallCSharp` 配置，则执行父类的适配代码，否则会尝试用反射来访问。

反射访问除了性能不佳之外，在 `il2cpp` 下还有可能因为代码剪裁而导致无法访问，后者可以通过下面介绍的 `ReflectionUse` 标签来避免。

XLua.ReflectionUse

一个 `C#` 类型类型加了这个配置， `xLua` 会生成 `link.xml` 阻止 `il2cpp` 的代码剪裁。

对于扩展方法，必须加上 `LuaCallCSharp` 或者 `ReflectionUse` 才可以被访问到。

建议所有要在 `Lua` 访问的类型，要么加 `LuaCallCSharp` ，要么加上 `ReflectionUse` ，这才能够保证在各平台都能正常运行。

XLua.CSharpCallLua

如果希望把一个 `lua` 函数适配到一个 `C# delegate` （一类是 `C#` 侧各种回调： `UI` 事件， `delegate` 参数，比如 `List<T>:ForEach` ；另外一类场景是通过 `LuaTable` 的 `Get` 函数指明一个 `lua` 函数绑定到一个 `delegate` ）。或者把一个 `lua table` 适配到一个 `C# interface` ，该 `delegate` 或者 `interface` 需要加上该配置。

XLua.GCOptimize

一个 `C#` 纯值类型（注：指的是一个只包含值类型的 `struct` ，可以嵌套其它只包含值类型

的 `struct` ）或者 `C#` 枚举值加上了这个配置。 `xLua` 会为该类型生成 `gc` 优化代码，效果是该值类型在 `lua` 和 `c#` 间传递不产生（ `C#` ） `gc alloc` ，该类型的数组访问也不产生 `gc` 。各种无 `GC` 的场景，可以参考 `05_NoGc` 例子。

除枚举之外，包含无参构造函数的复杂类型，都会生成 `lua table` 到该类型，以及改类型的一维数组的转换代码，这将会优化这个转换的性能，包括更少的 `gc alloc` 。

XLua.AdditionalProperties

这个是 `GCOptimize` 的扩展配置，有的时候，一些 `struct` 喜欢把 `field` 做成是私有的，通过 `property` 来访问 `field` ，这时就需要用到该配置（默认情况下 `GCOptimize` 只对 `public` 的 `field` 打解包）。

标签方式比较简单，配置方式复杂一点，要求是 `Dictionary<Type, List<string>>` 类型， `Dictionary` 的 `Key` 是要生效的类型， `Value` 是属性名列表。可以参考 `XLua` 对几个 `UnityEngine` 下值类型的配置， `SysGCOptimize` 类。

XLua.BlackList

如果你不要生成一个类型的一些成员的适配代码，你可以通过这个配置来实现。

标签方式比较简单，对应的成员上加就可以了。

由于考虑到有可能需要把重载函数的其中一个重载列入黑名单，配置方式比较复杂，类型是 `List<List<string>>` ，对于每个成员，在第一层 `List` 有一个条目，第二层 `List` 是个 `string` 的列表，第一个 `string` 是类型的全路径名，第二个 `string` 是成员名，如果成员是一个方法，还需要从第三个 `string` 开始，把其参数的类型全路径全列出来。例如下面是对 `GameObject` 的一个属性以及 `FileInfo` 的一个方法列入黑名单：

```
1.      [BlackList]
2.      public static List<List<string>> BlackList = new List<List<string>>()  {
3.          new List<string>(){"UnityEngine.GameObject", "networkView"},
4.          new List<string>(){"System.IO.FileInfo", "GetAccessControl",
    "System.Security.AccessControl.AccessControlSections"},
5.      };
```

下面是生成期配置，必须放到Editor目录下

CSObjectWrapEditor.GenPath

配置生成代码的放置路径，类型是 `string` 。默认放在“ `Assets/XLua/Gen/` ”下。

CSObjectWrapEditor.GenCodeMenu

该配置用于生成引擎的二次开发，一个无参数函数加了这个标签，在执行“`XLua/Generate Code`”菜单时会触发这个函数的调用。

ExampleGenConfig.cs

```
using System.Collections.Generic;
using System;
using UnityEngine;
using XLua;

//配置的详细介绍请看Doc下《XLua的配置.doc》
public static class ExampleGenConfig
{
    //lua中要使用到C#库的配置，比如C#标准库，或者Unity API，第三方库等。
    [LuaCallCSharp]
    public static List<Type> LuaCallCSharp = new List<Type>() {
                typeof(System.Object),
                typeof(UnityEngine.Object),
                typeof(Vector2),
                typeof(Vector3),
                typeof(Vector4),
                typeof(Quaternion),
                typeof(Color),
                typeof(Ray),
                typeof(Bounds),
                typeof(Ray2D),
                typeof(Time),
                typeof(GameObject),
                typeof(Component),
                typeof(Behaviour),
                typeof(Transform),
                typeof(Resources),
                typeof(TextAsset),
                typeof(Keyframe),
                typeof(AnimationCurve),
                typeof(AnimationClip),
                typeof(MonoBehaviour),
                typeof(ParticleSystem),
                typeof(SkinnedMeshRenderer),
                typeof(Renderer),
```

```
                    typeof(WWW),
                    typeof(System.Collections.Generic.List<int>),
                    typeof(Action<string>),
                    typeof(UnityEngine.Debug)
                };

        //C#静态调用Lua的配置（包括事件的原型），仅可以配delegate，interface
        [CSharpCallLua]
        public static List<Type> CSharpCallLua = new List<Type>() {
                    typeof(Action),
                    typeof(Func<double, double, double>),
                    typeof(Action<string>),
                    typeof(Action<double>),
                    typeof(UnityEngine.Events.UnityAction),
                    typeof(System.Collections.IEnumerator)
                };

        //黑名单
        [BlackList]
        public static List<List<string>> BlackList = new List<List<string>>()
{
                    new List<string>(){"UnityEngine.WWW", "movie"},
        #if UNITY_WEBGL
                    new List<string>(){"UnityEngine.WWW", "threadPriority"},
        #endif
                    new List<string>(){"UnityEngine.Texture2D",
"alphaIsTransparency"},
                    new List<string>(){"UnityEngine.Security",
"GetChainOfTrustValue"},
                    new List<string>(){"UnityEngine.CanvasRenderer",
"onRequestRebuild"},
                    new List<string>(){"UnityEngine.Light", "areaSize"},
                    new List<string>()
{"UnityEngine.AnimatorOverrideController", "PerformOverrideClipListCleanup"},
        #if !UNITY_WEBPLAYER
                    new List<string>(){"UnityEngine.Application",
"ExternalEval"},
        #endif
                    new List<string>(){"UnityEngine.GameObject",
"networkView"}, //4.6.2 not support
                    new List<string>(){"UnityEngine.Component", "networkView"},
//4.6.2 not support
```

```
70.                     new List<string>(){"System.IO.FileInfo",
    "GetAccessControl", "System.Security.AccessControl.AccessControlSections"},
71.                     new List<string>(){"System.IO.FileInfo",
    "SetAccessControl", "System.Security.AccessControl.FileSecurity"},
72.                     new List<string>(){"System.IO.DirectoryInfo",
    "GetAccessControl", "System.Security.AccessControl.AccessControlSections"},
73.                     new List<string>(){"System.IO.DirectoryInfo",
    "SetAccessControl", "System.Security.AccessControl.DirectorySecurity"},
74.                     new List<string>(){"System.IO.DirectoryInfo",
    "CreateSubdirectory", "System.String",
    "System.Security.AccessControl.DirectorySecurity"},
75.                     new List<string>(){"System.IO.DirectoryInfo", "Create",
    "System.Security.AccessControl.DirectorySecurity"},
76.                     new List<string>(){"UnityEngine.MonoBehaviour",
    "runInEditMode"},
77.                 };
78.     }
```

?

CS调用Lua

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
using XLua;
using System;

public class CSCallLua : MonoBehaviour {
    LuaEnv luaenv = null;
    string script = @"
        a = 1
        b = 'hello world'
        c = true

        d = {
           f1 = 12, f2 = 34,
           1, 2, 3,
           add = function(self, a, b)
              print('d.add called')
              return a + b
           end
        }

        function e()
            print('i am e')
        end

        function f(a, b)
            print('a', a, 'b', b)
            return 1, {f1 = 1024}
        end

        function ret_e()
            print('ret_e called')
            return e
        end
    ";

    public class DClass
```

```
        {
            public int f1;
            public int f2;
        }

        [CSharpCallLua]
        public interface ItfD
        {
            int f1 { get; set; }
            int f2 { get; set; }
            int add(int a, int b);
        }

        [CSharpCallLua]
        public delegate int FDelegate(int a, string b, out DClass c);

        [CSharpCallLua]
        public delegate Action GetE();

        // Use this for initialization
        void Start()
        {
            luaenv = new LuaEnv();
            luaenv.DoString(script);

            Debug.Log("_G.a = " + luaenv.Global.Get<int>("a"));
            Debug.Log("_G.b = " + luaenv.Global.Get<string>("b"));
            Debug.Log("_G.c = " + luaenv.Global.Get<bool>("c"));


            DClass d = luaenv.Global.Get<DClass>("d");//映射到有对应字段的class, by value
            Debug.Log("_G.d = {f1=" + d.f1 + ", f2=" + d.f2 + "}");

            Dictionary<string, double> d1 = luaenv.Global.Get<Dictionary<string, double>>("d");//映射到Dictionary<string, double>, by value
            Debug.Log("_G.d = {f1=" + d1["f1"] + ", f2=" + d1["f2"] + "}, d.Count=" + d1.Count);

            List<double> d2 = luaenv.Global.Get<List<double>>("d"); //映射到List<double>, by value
```

```
            Debug.Log("_G.d.len = " + d2.Count);

            ItfD d3 = luaenv.Global.Get<ItfD>("d"); //映射到interface实例，by ref，这个要求interface加到生成列表，否则会返回null，建议用法
            d3.f2 = 1000;
            Debug.Log("_G.d = {f1=" + d3.f1 + ", f2=" + d3.f2 + "}");
            Debug.Log("_G.d:add(1, 2)=" + d3.add(1, 2));

            LuaTable d4 = luaenv.Global.Get<LuaTable>("d");//映射到LuaTable，by ref
            Debug.Log("_G.d = {f1=" + d4.Get<int>("f1") + ", f2=" + d4.Get<int>("f2") + "}");


            Action e = luaenv.Global.Get<Action>("e");//映射到一个delgate，要求delegate加到生成列表，否则返回null，建议用法
            e();

            FDelegate f = luaenv.Global.Get<FDelegate>("f");
            DClass d_ret;
            int f_ret = f(100, "John", out d_ret);//lua的多返回值映射：从左往右映射到c#的输出参数，输出参数包括返回值，out参数，ref参数
            Debug.Log("ret.d = {f1=" + d_ret.f1 + ", f2=" + d_ret.f2 + "}, ret=" + f_ret);

            GetE ret_e = luaenv.Global.Get<GetE>("ret_e");//delegate可以返回更复杂的类型，甚至是另外一个delegate
            e = ret_e();
            e();

            LuaFunction d_e = luaenv.Global.Get<LuaFunction>("e");
            d_e.Call();

        }

        // Update is called once per frame
        void Update()
        {
            if (luaenv != null)
            {
                luaenv.Tick();
            }
```

```
111.         }
112.
113.         void OnDestroy()
114.         {
115.             luaenv.Dispose();
116.         }
117.     }
```

?

一、调用Lua基本类型

```
1.  /*
2.   *  created by shenjun
3.   */
4.
5.  using System.Collections;
6.  using System.Collections.Generic;
7.  using UnityEngine;
8.
9.  using XLua;
10.
11. namespace shenjun
12. {
13.     public class CSCallLuaBasicType : MonoBehaviour {
14.
15.         LuaEnv luaEnv = new LuaEnv();
16.
17.         void Start () {
18.
19.             luaEnv.DoString("require 'BasicLua'");
20.
21.             int a = luaEnv.Global.Get<int>("a");
22.             //int a;
23.             //luaEnv.Global.Get("a", out a);
24.
25.             float b = luaEnv.Global.Get<float>("b");
26.
27.             string c = luaEnv.Global.Get<string>("c");
28.
29.             bool d = luaEnv.Global.Get<bool>("d");
30.
31.             string n = luaEnv.Global.GetInPath<string>("e.f.name");
32.
33.             Debug.Log(string.Format("a :{0}, b :{1}, c :{2}, d :{3}, name :
    {4}", a, b, c, d, n));
34.         }
35.
36.         void Update () {
37.             if(luaEnv != null)
```

```
                {
                    luaEnv.Tick();
                }
            }

            void OnDestroy()
            {
                luaEnv.Dispose();
            }
        }
    }
```

BasicLua.lua.txt

```
    a = 1
    b = 1.5
    c = 'hello world'
    d = true

    e = {
        ["f"] = { ["name"] = "shenjun" },
        "unity"
    }

    --f = { 2, 3 }
```

?

调用Lua Table类型

TableLua.lua.txt

```
student = {
    name = "xm", age = 18, Sex = "man",
    80, 90, 95,
    getSex = function(self)
        return "人妖"
    end,

    totalScore = function(self, a, b)
        return a + b;
    end
}
```

一、映射到class和struct

```
/*
 *  created by shenjun
 */

using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using XLua;

namespace shenjun
{
    public class TableToClass : MonoBehaviour {

        LuaEnv luaEnv = new LuaEnv();

        void Start () {

            luaEnv.DoString("require 'TableLua'");

            // 对应public字段，table的属性可以多于或少于class的字段，没有对应的使用该类型的默认值
            Student xm = luaEnv.Global.Get<Student>("student");
```

```
                Debug.Log(xm);

            }

            void Update () {
                if(luaEnv != null)
                {
                    luaEnv.Tick();
                }
            }

            private void OnDestroy()
            {
                luaEnv.Dispose();
            }

            class Student
            {
                public string name;
                public int age;
                public string Sex { get; set; } // 无法对应table中的键

                public override string ToString()
                {
                    return string.Format("name : {0}, age : {1}, sex : {2}", name, age, Sex);
                }

                //public string get_Sex()
                //{
                //    return "";
                //}
            }
        }
    }
```

```
    /*
     *  created by shenjun
     */

    using System.Collections;
    using System.Collections.Generic;
```

```
7.      using UnityEngine;
8.
9.      using XLua;
10.
11.     namespace shenjun
12.     {
13.         public class TableToStruct : MonoBehaviour
14.         {
15.
16.             LuaEnv luaEnv = new LuaEnv();
17.
18.             void Start()
19.             {
20.                 luaEnv.DoString("require 'TableLua'");
21.
22.                 // 对应public字段，table的属性可以多于或少于class的字段，没有对应的使用该类型的默认值
23.                 Student xm = luaEnv.Global.Get<Student>("student");
24.                 Debug.Log(xm);
25.             }
26.
27.             void Update()
28.             {
29.                 if (luaEnv != null)
30.                 {
31.                     luaEnv.Tick();
32.                 }
33.             }
34.
35.             private void OnDestroy()
36.             {
37.                 luaEnv.Dispose();
38.             }
39.
40.             /*
41.              * xLua复杂值类型（struct）的默认传递方式是引用传递，这种方式要求先对值类型boxing，传递给lua，
42.              * lua使用后释放该引用。由于值类型每次boxing将产生一个新对象，当lua侧使用完毕释放该对象的引用时，
43.              * 则产生一次gc。为此，xLua实现了一套struct的gc优化方案，您只要通过简单的配置，
44.              * 则可以实现满足条件的struct传递到lua侧无gc。
```

```
45.              *
46.              * struct需要满足什么条件？
47.              * 1、struct允许嵌套其它struct，但它以及它嵌套的struct只能包含这几种基本类
    型：
48.              * byte、sbyte、short、ushort、int、uint、long、ulong、float、double；
49.              * 例如UnityEngine定义的大多数值类型：Vector系列，Quaternion，Color。。。
    均满足条件，
50.              * 或者用户自定义的一些struct
51.              * 2、该struct配置了GCOptimize属性（对于常用的UnityEngine的几个struct，
52.              * Vector系列，Quaternion，Color。。。均已经配置了该属性），这个属性可以通
    过配置文件或者C# Attribute实现；
53.              * 3、使用到该struct的地方，需要添加到生成代码列表；
54.              */
55.             [GCOptimize]
56.             [LuaCallCSharp]
57.             struct Student
58.             {
59.                 public string name;
60.                 public int age;
61.                 public string Sex { get; set; }  // 无法对应table中的键
62.
63.                 public override string ToString()
64.                 {
65.                     return string.Format("name : {0}, age : {1}, sex : {2}",
    name, age, Sex);
66.                 }
67.             }
68.
69.             //[LuaCallCSharp]
70.             //public static class StructConfig
71.             //{
72.             //    public static List<System.Type> LuaCallCSharp
73.             //    {
74.             //        get
75.             //        {
76.             //            return new List<System.Type>()
77.             //            {
78.             //                typeof(Student)
79.             //            };
80.             //        }
81.             //    }
82.             //}
```

```
        }
    }
```

二、映射到Interface

```
    /*
     *  created by shenjun
     */

    using System.Collections;
    using System.Collections.Generic;
    using UnityEngine;
    using XLua;

    namespace shenjun
    {
        public class TableToInterface : MonoBehaviour {

            LuaEnv luaEnv = new LuaEnv();

            void Start () {

                luaEnv.DoString("require 'TableLua'");

                // 引用类型映射 代码生成器会生成一个实例
                IStudent iTable = luaEnv.Global.Get<IStudent>("student");

                // 修改table中某键的值
                luaEnv.Global.SetInPath<int>("student.age", 20);

                string _name = iTable.name;
                int age = iTable.age;
                string sex = iTable.Sex;

                Debug.Log(string.Format("name :{0}, age :{1}, sex :{2}", _name, age, sex));

                // 引用类型映射
                iTable.name = "xz";
                Debug.Log(luaEnv.Global.GetInPath<string>("student.name"));

                int result = iTable.totalScore(100, 200);
```

```
                Debug.Log(result);
            }

            void Update () {
                if(luaEnv != null)
                {
                    luaEnv.Tick();
                }
            }

            private void OnDestroy()
            {
                luaEnv.Dispose();
            }

            [CSharpCallLua]
            interface IStudent
            {
                string name { get; set; }
                int age { get; set; }
                string Sex { get; set; }

                //string get_Sex();

                int totalScore(int a, int b);
            }
        }
    }
```

三、映射到Dictionary和List

```
    /*
     *  created by shenjun
     */

    using System.Collections;
    using System.Collections.Generic;
    using UnityEngine;
    using XLua;

    namespace shenjun
    {
```

```
12.         public class TableToDicOrList : MonoBehaviour {
13.
14.             LuaEnv luaEnv = new LuaEnv();
15.
16.             void Start () {
17.
18.                 luaEnv.DoString("require 'TableLua'");
19.
20.                 // 值拷贝 只能获取匹配的
21.
22.                 Dictionary<string, string> dic = new Dictionary<string, string>
    ();
23.                 dic = luaEnv.Global.Get<Dictionary<string, string>>("student");
24.                 foreach (var item in dic.Keys)
25.                 {
26.                     Debug.Log("key :" + item + ", value :" + dic[item]);
27.                 }
28.
29.                 List<int> list = new List<int>();
30.                 list = luaEnv.Global.Get<List<int>>("student");
31.                 foreach (var item in list)
32.                 {
33.                     Debug.Log(item);
34.                 }
35.             }
36.
37.             void Update () {
38.
39.             }
40.
41.             private void OnDestroy()
42.             {
43.                 luaEnv.Dispose();
44.             }
45.         }
46.     }
```

四、映射到LuaTable类型

```
1.     /*
2.      *  created by shenjun
3.      */
```

```
4.
5.      using System.Collections;
6.      using System.Collections.Generic;
7.      using System.Linq;
8.      using UnityEngine;
9.      using XLua;
10.
11.     namespace shenjun
12.     {
13.         public class TableToLuaTable : MonoBehaviour {
14.
15.             LuaEnv luaEnv = new LuaEnv();
16.
17.             void Start () {
18.                 luaEnv.DoString("require 'TableLua'");
19.
20.                 LuaTable table = luaEnv.Global.Get<LuaTable>("student");
21.                 string _name = table.Get<string>("name");
22.                 int age = table.Get<int>("age");
23.                 string sex = table.Get<string>("Sex");
24.
25.                 Debug.Log(string.Format("name :{0}, age :{1}, sex :{2}", _name,
    age, sex));
26.
27.                 Debug.Log("Length:" + table.Length);  // 3
28.
29.                 List<int> scores = luaEnv.Global.Get<List<int>>("student");
30.                 Debug.Log("Total Scores :" + scores.Aggregate((a, b) => a +
    b));
31.
32.                 var list = table.GetKeys();
33.                 foreach (var item in list)
34.                 {
35.                     Debug.Log(item);
36.                 }
37.
38.                 int csharpScore = table.Get<int, int>(1);
39.                 int unityScore = table.Get<int, int>(2);
40.                 int shaderScore = table.Get<int, int>(3);
41.                 Debug.Log("csharpScore : " + csharpScore + ", unityScore : " +
    unityScore + ", shaderScore : " + shaderScore);
42.
```

```
                LuaFunction func = table.Get<LuaFunction>("totalScore");
                object[] results = func.Call(table, 100, 200);
                Debug.Log(results[0]);
            }

            void Update () {

            }

            private void OnDestroy()
            {
                luaEnv.Dispose();
            }
        }
    }
```

```
    /*
     *  created by shenjun
     */

    using System.Collections;
    using System.Collections.Generic;
    using System.Linq;
    using UnityEngine;
    using XLua;

    namespace shenjun
    {
        public class TableToLuaTable2 : MonoBehaviour {

            [CSharpCallLua]
            public delegate int AddDel(LuaTable self, int a, int b);
            [CSharpCallLua]
            public delegate string GetSexDel();

            public TextAsset luaText;

            LuaEnv luaEnv = new LuaEnv();
            LuaTable luaTableGlobal;

            LuaTable luaTableLocal;

```

```
27.             GetSexDel luaGetSex;
28.             AddDel luaAdd;
29.
30.
31.             void Start () {
32.
33.                 #region 全局table元表
34.                 luaTableGlobal = luaEnv.NewTable();
35.
36.                 // 设置检索元表 为全局
37.                 LuaTable meta = luaEnv.NewTable();
38.                 meta.Set("__index", luaEnv.Global);
39.                 luaTableGlobal.SetMetaTable(meta);
40.                 meta.Dispose();
41.
42.                 luaEnv.DoString(luaText.text, "LuaBehaviour", luaTableGlobal);
43.
44.                 string _name = luaTableGlobal.GetInPath<string>
    ("student.name");
45.                 int age = luaTableGlobal.GetInPath<int>("student.age");
46.                 string sex = luaTableGlobal.GetInPath<string>("student.Sex");
47.
48.                 Debug.Log(string.Format("1 name :{0}, age :{1}, sex :{2}",
    _name, age, sex));
49.
50.                 luaGetSex = luaTableGlobal.GetInPath<GetSexDel>
    ("student.getSex");
51.                 if (luaGetSex != null)
52.                     Debug.Log("1 : " + luaGetSex()) ;
53.
54.                 luaAdd = luaTableGlobal.GetInPath<AddDel>
    ("student.totalScore");
55.                 if (luaAdd != null)
56.                     Debug.Log("1 : " + luaAdd(luaEnv.NewTable() ,100, 200));
57.
58.                 #endregion
59.
60.                 #region 局部table元表
61.                 luaTableLocal = luaEnv.NewTable();
62.
63.                 luaEnv.DoString(luaText.text);
64.                 LuaTable metaContent = luaEnv.Global.Get<LuaTable>("student");
```

```
65.
66.                 meta = luaEnv.NewTable();
67.                 meta.Set("__index", metaContent);
68.                 luaTableLocal.SetMetaTable(meta);
69.                 meta.Dispose();
70.
71.                 _name = luaTableLocal.Get<string>("name");
72.                 age = luaTableLocal.Get<int>("age");
73.                 sex = luaTableLocal.Get<string>("Sex");
74.
75.                 Debug.Log(string.Format("2 name :{0}, age :{1}, sex :{2}",
    _name, age, sex));
76.
77.                 luaGetSex = luaTableLocal.Get<GetSexDel>("getSex");
78.                 if(luaGetSex != null)
79.                 {
80.                     Debug.Log("2 : " + luaGetSex());
81.                 }
82.
83.                 luaAdd = luaTableLocal.Get<AddDel>("totalScore");
84.                 if (luaAdd != null)
85.                     Debug.Log("2 : " + luaAdd(luaEnv.NewTable(), 500, 600));
86.
87.                 #endregion
88.
89.
90.             }
91.
92.             void Update () {
93.
94.             }
95.
96.             private void OnDestroy()
97.             {
98.                 luaGetSex = null;
99.                 luaAdd = null;
100.                luaEnv.Dispose();
101.            }
102.        }
103.    }
```

?

调用Function

FunctionLua.lua.txt

```
function NoParam()
    print('LuaFunction NoParam')
end

function HaveParam(a, b)
    print('LuaFunction HaveParam : ', 'a: ', a, 'b: ', b)
end

function HaveMultiParam(...)
    print(...)
end

function ReturnString()
    print('LuaFunction ReturnString')
    return "i am the return string!"
end

function ReturnFunction()
    print('LuaFunction ReturnFunction')
    return NoParam
end

function ReturnMultiValue()
    print("LuaFunction ReturnMultiValue")
    return true, "ReturnMultiValue", 1, "num"
end
```

一、映射到Delegate

```
/*
 *  created by shenjun
 */

using System.Collections;
using System.Collections.Generic;
using UnityEngine;
```

```
8.      using XLua;
9.      using System;
10.
11.     namespace shenjun
12.     {
13.         public class FunctionToDelegate : MonoBehaviour {
14.
15.             LuaEnv luaEnv = new LuaEnv();
16.
17.             void Start () {
18.
19.                 luaEnv.DoString("require 'FunctionLua'");
20.
21.                 NoParam();
22.                 HaveParam();
23.                 HaveMultiParam();
24.                 HaveReturnString();
25.                 HaveReturnFunction();
26.                 HaveMultiReturn();
27.             }
28.
29.             void Update () {
30.
31.             }
32.
33.             private void OnDestroy()
34.             {
35.                 luaEnv.Dispose();
36.             }
37.
38.             #region 无参数
39.             void NoParam()
40.             {
41.                 // 无返回值
42.                 Action del1 = luaEnv.Global.Get<Action>("NoParam");
43.                 del1();
44.             }
45.             #endregion
46.
47.             #region 有两个参数
48.             [CSharpCallLua]
49.             public delegate void HaveParamDel(int a, int b);
```

```
50.             void HaveParam()
51.             {
52.                 HaveParamDel del = luaEnv.Global.Get<HaveParamDel>
    ("HaveParam");
53.                 del(10, 20);
54.             }
55.             #endregion
56.
57.             #region 有任意个参数
58.             [CSharpCallLua]
59.             public delegate void HaveMultiParamDel(params int[] nums);
60.             void HaveMultiParam()
61.             {
62.                 HaveMultiParamDel del = luaEnv.Global.Get<HaveMultiParamDel>
    ("HaveMultiParam");
63.                 del(10, 20, 30, 40);
64.             }
65.             #endregion
66.
67.             #region 有字符串返回值
68.             [CSharpCallLua]
69.             public delegate string HaveReturnStringDel();
70.             void HaveReturnString()
71.             {
72.                 HaveReturnStringDel del =
    luaEnv.Global.Get<HaveReturnStringDel>("ReturnString");
73.                 string r = del();
74.                 Debug.Log("GetResturn :" + r);
75.             }
76.             #endregion
77.
78.             #region 有方法返回值
79.             [CSharpCallLua]
80.             public delegate Action HaveReturnFunctionDel();
81.             void HaveReturnFunction()
82.             {
83.                 HaveReturnFunctionDel del =
    luaEnv.Global.Get<HaveReturnFunctionDel>("ReturnFunction");
84.                 Action result = del();
85.                 result();
86.             }
87.             #endregion
```

```
88.
89.             #region 多返回值
90.             // 多返回值的，返回值从左到右的依次映射，返回值，out参数或ref参数 。。。
91.             [CSharpCallLua]
92.             public delegate bool HaveMultiReturnDel(ref string a, out int b);
93.
94.             void HaveMultiReturn()
95.             {
96.                 HaveMultiReturnDel del = luaEnv.Global.Get<HaveMultiReturnDel>
    ("ReturnMultiValue");
97.                 string a = "";
98.                 int b;
99.                 bool result = del(ref a, out b);
100.                Debug.Log("result :" + result + ", a :" + a + ", b :" + b);
101.            }
102.            #endregion
103.        }
104.    }
```

二、映射到LuaFunction

```
1.    /*
2.     *  created by shenjun
3.     */
4.
5.    using System.Collections;
6.    using System.Collections.Generic;
7.    using UnityEngine;
8.    using XLua;
9.
10.   namespace shenjun
11.   {
12.       public class FunctionToLuaFunction : MonoBehaviour {
13.
14.           LuaEnv luaEnv = new LuaEnv();
15.
16.           void Start () {
17.
18.               luaEnv.DoString("require 'FunctionLua'");
19.
20.               NoParam();
21.               HaveParam();
```

```
22.                 HaveReturnString();
23.                 HaveReturnFunction();
24.                 HaveMultiReturn();
25.             }
26.
27.             void Update () {
28.
29.             }
30.
31.             private void OnDestroy()
32.             {
33.                 luaEnv.Dispose();
34.             }
35.
36.             #region 无参数
37.             void NoParam()
38.             {
39.                 LuaFunction del = luaEnv.Global.Get<LuaFunction>("NoParam");
40.                 del.Call();
41.             }
42.             #endregion
43.
44.             #region 有参数
45.             void HaveParam()
46.             {
47.                 LuaFunction del = luaEnv.Global.Get<LuaFunction>("HaveParam");
48.                 del.Call(10, 20);
49.             }
50.             #endregion
51.
52.             #region 有返回值是字符串
53.             void HaveReturnString()
54.             {
55.                 LuaFunction del = luaEnv.Global.Get<LuaFunction>
    ("ReturnString");
56.                 object[] r = del.Call();
57.                 Debug.Log(r[0]);
58.             }
59.             #endregion
60.
61.             #region 有返回值是方法
62.             void HaveReturnFunction()
```

```
63.             {
64.                 LuaFunction del = luaEnv.Global.Get<LuaFunction>
    ("ReturnFunction");
65.                 object[] result = del.Call();
66.                 ((LuaFunction)result[0]).Call();
67.             }
68.             #endregion
69.
70.             #region 多返回值
71.             void HaveMultiReturn()
72.             {
73.                 LuaFunction del = luaEnv.Global.Get<LuaFunction>
    ("ReturnMultiValue");
74.                 object[] result = del.Call();
75.                 Debug.Log("result :" + (bool)result[0] + ", " +
    (string)result[1] + ", " + int.Parse(result[2]+"") + ", " + (string)result[3]);
76.             }
77.             #endregion
78.         }
79.     }
```

?

Lua调用CS

```
using UnityEngine;
using System.Collections;
using System;
using XLua;
using System.Collections.Generic;

namespace Tutorial
{
    [LuaCallCSharp]
    public class BaseClass
    {
        public static void BSFunc()
        {
            Debug.Log("Driven Static Func, BSF = "+ BSF);
        }

        public static int BSF = 1;

        public void BMFunc()
        {
            Debug.Log("Driven Member Func, BMF = " + BMF);
        }

        public int BMF { get; set; }
    }

    public struct Param1
    {
        public int x;
        public string y;
    }

    [LuaCallCSharp]
    public enum TestEnum
    {
        E1,
        E2
    }
```

```
39.
40.         [LuaCallCSharp]
41.         public class DrivenClass : BaseClass
42.         {
43.             [LuaCallCSharp]
44.             public enum TestEnumInner
45.             {
46.                 E3,
47.                 E4
48.             }
49.
50.             public void DMFunc()
51.             {
52.                 Debug.Log("Driven Member Func, DMF = " + DMF);
53.             }
54.
55.             public int DMF { get; set; }
56.
57.             public double ComplexFunc(Param1 p1, ref int p2, out string p3,
    Action luafunc, out Action csfunc)
58.             {
59.                 Debug.Log("P1 = {x=" + p1.x + ",y=" + p1.y + "},p2 = "+ p2);
60.                 luafunc();
61.                 p2 = p2 * p1.x;
62.                 p3 = "hello " + p1.y;
63.                 csfunc = () =>
64.                 {
65.                     Debug.Log("csharp callback invoked!");
66.                 };
67.                 return 1.23;
68.             }
69.
70.             public void TestFunc(int i)
71.             {
72.                 Debug.Log("TestFunc(int i)");
73.             }
74.
75.             public void TestFunc(string i)
76.             {
77.                 Debug.Log("TestFunc(string i)");
78.             }
79.
```

```
80.             public static DrivenClass operator +(DrivenClass a, DrivenClass b)
81.             {
82.                 DrivenClass ret = new DrivenClass();
83.                 ret.DMF = a.DMF + b.DMF;
84.                 return ret;
85.             }
86.
87.             public void DefaultValueFunc(int a = 100, string b = "cccc", string
    c = null)
88.             {
89.                 UnityEngine.Debug.Log("DefaultValueFunc: a=" + a + ",b=" + b +
    ",c=" + c);
90.             }
91.
92.             public void VariableParamsFunc(int a, params string[] strs)
93.             {
94.                 UnityEngine.Debug.Log("VariableParamsFunc: a =" + a);
95.                 foreach (var str in strs)
96.                 {
97.                     UnityEngine.Debug.Log("str:" + str);
98.                 }
99.             }
100.
101.             public TestEnum EnumTestFunc(TestEnum e)
102.             {
103.                 Debug.Log("EnumTestFunc: e=" + e);
104.                 return TestEnum.E2;
105.             }
106.
107.             public Action<string> TestDelegate = (param) =>
108.             {
109.                 Debug.Log("TestDelegate in c#:" + param);
110.             };
111.
112.             public event Action TestEvent;
113.
114.             public void CallEvent()
115.             {
116.                 TestEvent();
117.             }
118.
119.             public ulong TestLong(long n)
```

```
120.             {
121.                 return (ulong)(n + 1);
122.             }
123.
124.             class InnerCalc : Calc
125.             {
126.                 public int add(int a, int b)
127.                 {
128.                     return a + b;
129.                 }
130.
131.                 public int id = 100;
132.             }
133.
134.             public Calc GetCalc()
135.             {
136.                 return new InnerCalc();
137.             }
138.
139.             public void GenericMethod<T>()
140.             {
141.                 Debug.Log("GenericMethod<" + typeof(T) + ">");
142.             }
143.         }
144.
145.         [LuaCallCSharp]
146.         public interface Calc
147.         {
148.             int add(int a, int b);
149.         }
150.
151.         [LuaCallCSharp]
152.         public static class DrivenClassExtensions
153.         {
154.             public static int GetSomeData(this DrivenClass obj)
155.             {
156.                 Debug.Log("GetSomeData ret = " + obj.DMF);
157.                 return obj.DMF;
158.             }
159.
160.             public static int GetSomeBaseData(this BaseClass obj)
161.             {
```

```
                Debug.Log("GetSomeBaseData ret = " + obj.BMF);
                return obj.BMF;
            }

            public static void GenericMethodOfString(this DrivenClass obj)
            {
                obj.GenericMethod<string>();
            }
        }
    }

    public class LuaCallCs : MonoBehaviour {
        LuaEnv luaenv = null;
        string script = @"
            function demo()
                --new C#对象
                local newGameObj = CS.UnityEngine.GameObject()
                local newGameObj2 = CS.UnityEngine.GameObject('helloworld')
                print(newGameObj, newGameObj2)

                --访问静态属性，方法
                local GameObject = CS.UnityEngine.GameObject
                print('UnityEngine.Time.deltaTime:',
    CS.UnityEngine.Time.deltaTime) --读静态属性
                CS.UnityEngine.Time.timeScale = 0.5 --写静态属性
                print('helloworld', GameObject.Find('helloworld')) --静态方法调用

                --访问成员属性，方法
                local DrivenClass = CS.Tutorial.DrivenClass
                local testobj = DrivenClass()
                testobj.DMF = 1024--设置成员属性
                print(testobj.DMF)--读取成员属性
                testobj:DMFunc()--成员方法

                --基类属性，方法
                print(DrivenClass.BSF)--读基类静态属性
                DrivenClass.BSF = 2048--写基类静态属性
                DrivenClass.BSFunc();--基类静态方法
                print(testobj.BMF)--读基类成员属性
                testobj.BMF = 4096--写基类成员属性
                testobj:BMFunc()--基类方法调用

```

```
203.                 --复杂方法调用
204.                 local ret, p2, p3, csfunc = testobj:ComplexFunc({x=3, y =
    'john'}, 100, function()
205.                     print('i am lua callback')
206.                 end)
207.                 print('ComplexFunc ret:', ret, p2, p3, csfunc)
208.                 csfunc()
209.
210.                 --重载方法调用
211.                 testobj:TestFunc(100)
212.                 testobj:TestFunc('hello')
213.
214.                 --操作符
215.                 local testobj2 = DrivenClass()
216.                 testobj2.DMF = 2048
217.                 print('(testobj + testobj2).DMF = ', (testobj + testobj2).DMF)
218.
219.                 --默认值
220.                 testobj:DefaultValueFunc(1)
221.                 testobj:DefaultValueFunc(3, 'hello', 'john')
222.
223.                 --可变参数
224.                 testobj:VariableParamsFunc(5, 'hello', 'john')
225.
226.                 --Extension methods
227.                 print(testobj:GetSomeData())
228.                 print(testobj:GetSomeBaseData()) --访问基类的Extension methods
229.                 testobj:GenericMethodOfString()  --通过Extension methods实现访问泛
    化方法
230.
231.                 --枚举类型
232.                 local e = testobj:EnumTestFunc(CS.Tutorial.TestEnum.E1)
233.                 print(e, e == CS.Tutorial.TestEnum.E2)
234.                 print(CS.Tutorial.TestEnum.__CastFrom(1),
    CS.Tutorial.TestEnum.__CastFrom('E1'))
235.                 print(CS.Tutorial.DrivenClass.TestEnumInner.E3)
236.                 assert(CS.Tutorial.BaseClass.TestEnumInner == nil)
237.
238.                 --Delegate
239.                 testobj.TestDelegate('hello') --直接调用
240.                 local function lua_delegate(str)
241.                     print('TestDelegate in lua:', str)
```

```
                end
                testobj.TestDelegate = lua_delegate + testobj.TestDelegate --
    combine，这里演示的是C#delegate作为右值，左值也支持
                testobj.TestDelegate('hello')
                testobj.TestDelegate = testobj.TestDelegate - lua_delegate --
    remove
                testobj.TestDelegate('hello')

                --事件
                local function lua_event_callback1()
    print('lua_event_callback1') end
                local function lua_event_callback2()
    print('lua_event_callback2') end
                testobj:TestEvent('+', lua_event_callback1)
                testobj:CallEvent()
                testobj:TestEvent('+', lua_event_callback2)
                testobj:CallEvent()
                testobj:TestEvent('-', lua_event_callback1)
                testobj:CallEvent()
                testobj:TestEvent('-', lua_event_callback2)

                --64位支持
                local l = testobj:TestLong(11)
                print(type(l), l, l + 100, 10000 + l)

                --typeof
                newGameObj:AddComponent(typeof(CS.UnityEngine.ParticleSystem))

                --cast
                local calc = testobj:GetCalc()
                calc:add(1, 2)
                assert(calc.id == 100)
                cast(calc, typeof(CS.Tutorial.Calc))
                calc:add(1, 2)
                assert(calc.id == nil)
            end

            demo()

            --协程下使用
            local co = coroutine.create(function()
                print('-----------------------------------------------------')
```

```
                demo()
            end)
            assert(coroutine.resume(co))
        ";

        // Use this for initialization
        void Start () {
            luaenv = new LuaEnv();
            luaenv.DoString(script);
        }

        // Update is called once per frame
        void Update () {
            if (luaenv != null)
            {
                luaenv.Tick();
            }
        }

        void OnDestroy()
        {
            luaenv.Dispose();
        }
    }
```

创建对象

NewObjFunc.lua.txt

```
function New()
    local obj = CS.UnityEngine.GameObject()
    return obj
end

function New(name)
    local obj = CS.UnityEngine.GameObject(name)
    return obj
end

function NewToParent(parent)
    local obj = CS.UnityEngine.GameObject()
    obj.transform:SetParent(parent)
    return obj
end
```

NewObj.cs

```
/*
 *  created by shenjun
 */

using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using XLua;

namespace shenjun
{
    public class NewObj : MonoBehaviour {

        LuaEnv luaEnv = new LuaEnv();

        void Start () {

            luaEnv.DoString("require 'NewObjFunc'");
```

```
19.
20.                 GameObject obj1 = CreateObj();
21.                 GameObject obj2 = CreateObj("NewObj");
22.                 GameObject obj3 = CreateObj(obj2.transform);
23.             }
24.
25.             void Update () {
26.                 if(luaEnv != null)
27.                 {
28.                     luaEnv.Tick();
29.                 }
30.             }
31.
32.             private void OnDestroy()
33.             {
34.                 luaEnv.Dispose();
35.             }
36.
37.             [XLua.CSharpCallLua]
38.             public delegate GameObject CreateObjDel();
39.             GameObject CreateObj()
40.             {
41.                 CreateObjDel del = luaEnv.Global.Get<CreateObjDel>("New");
42.                 return del();
43.             }
44.
45.             [XLua.CSharpCallLua]
46.             public delegate GameObject CreateObjByNameDel(string name);
47.             GameObject CreateObj(string name)
48.             {
49.                 CreateObjByNameDel del = luaEnv.Global.Get<CreateObjByNameDel>
    ("New");
50.                 return del(name);
51.             }
52.
53.             [CSharpCallLua]
54.             public delegate GameObject CreateObjToParentDel(Transform parent);
55.             GameObject CreateObj(Transform parent)
56.             {
57.                 CreateObjToParentDel del =
    luaEnv.Global.Get<CreateObjToParentDel>("NewToParent");
58.                 return del(parent);
```

```
59.             }
60.         }
61.     }
```

?

调用静态成员

CallStatic.lua.txt

```
    --访问静态属性
    local deltaTime = CS.UnityEngine.Time.deltaTime - CS.UnityEngine.Time.deltaTime%0.001
    print("Time deltaTime :", deltaTime)

    --对静态属性进行赋值
    CS.UnityEngine.Time.timeScale = 0.5
    print("Time timeScale :", CS.UnityEngine.Time.timeScale)

    --调用静态方法
    local GameObject = CS.UnityEngine.GameObject
    local obj = GameObject.Find('GameObject')
    if nil ~= obj then
        print("Destroy GameObject")
        CS.UnityEngine.Object.Destroy(obj)
    end
```

CallStatic.cs

```
    /*
     *  created by shenjun
     */

    using System.Collections;
    using System.Collections.Generic;
    using UnityEngine;
    using XLua;

    namespace shenjun
    {
        public class CallStatic : MonoBehaviour {

            void Start () {
                LuaEnv luaEnv = new LuaEnv();
                luaEnv.DoString("require 'CallStatic'");
                luaEnv.Dispose();
```

```
18.             }
19.         }
20.     }
```

?

调用实例成员

CallMember.lua.txt

```lua
local GameObject = CS.UnityEngine.GameObject
local obj = GameObject.Find('GameObject')

--访问成员属性
local transform = obj.transform
print(transform.position)

--对成员属性进行赋值
transform.position = CS.UnityEngine.Vector3(1,1,1)

--调用成员方法
local camera = CS.UnityEngine.Camera.main
camera.transform:LookAt(transform)
```

CallMember.cs

```
/*
 *  created by shenjun
 */

using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using XLua;

namespace shenjun
{
    public class CallMember : MonoBehaviour {

        LuaEnv luaEnv = new LuaEnv();

        void Start () {
            luaEnv.DoString("require 'CallMember'");

        }

```

```
21.             void Update () {
22.                 if(luaEnv != null)
23.                 {
24.                     luaEnv.Tick();
25.                 }
26.             }
27.
28.             private void OnDestroy()
29.             {
30.                 luaEnv.Dispose();
31.             }
32.         }
33.     }
```

?

调用父类成员

CallCSParent.lua.txt

```
    local camera = CS.UnityEngine.Camera.main
    if camera ~= nil then
        print("Find Camera")
    end

    local callParent = camera.transform:GetComponent(typeof(CS.shenjun.CallParent))
    if nil ~= callParent then
        print("GetComponent CallParent")
    else
        print("Not Find Component CallParent")
    end


    --调用父类的静态字段
    print(CS.shenjun.CallParent.index)

    --调用父类的非静态字段
    print(callParent.tag)

    --调用父类的静态方法
    CS.shenjun.CallParent.SetIndex()

    --调用父类的非静态方法
    print(callParent:CompareTag("Player"))
```

CallParent.cs

```
    /*
     *  created by shenjun
     */

    using System.Collections;
    using System.Collections.Generic;
    using UnityEngine;
    using XLua;
```

```
 9.
10.     namespace shenjun
11.     {
12.         public class CallParent : Parent {
13.
14.             void Start () {
15.
16.                 LuaEnv luaEnv = new LuaEnv();
17.                 luaEnv.DoString("require 'CallCSParent'");
18.                 luaEnv.Dispose();
19.             }
20.
21.             void Update () {
22.
23.             }
24.         }
25.
26.         public class Parent : MonoBehaviour
27.         {
28.             public static int index = 0;
29.
30.             public static void SetIndex()
31.             {
32.                 index++;
33.                 Debug.Log("index : " + index);
34.             }
35.         }
36.     }
```

?

Ref、Out参数

RefOutParam.lua.txt

```
local RefOutParam = CS.UnityEngine.Object.FindObjectOfType(typeof(CS.shenjun.RefOutParam))

-- 1、out参数不需要传入实参
-- 2、返回值顺序：如果函数有返回值那么第一个是函数返回值
-- 其他的都是ref，out参数的传出的，按顺序传出
-- 3、如果函数没有返回值，那么按顺序返回ref，out参数
local a, b, c = RefOutParam:RefOutParamFunc("shenjun", 10, 30)
print("Lua Return :" ,a, b, c)
```

RefOutParam.cs

```
/*
 *  created by shenjun
 */

using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using XLua;

namespace shenjun
{
    public class RefOutParam : MonoBehaviour {


        void Start () {
            LuaEnv luaEnv = new LuaEnv();
            luaEnv.DoString("require 'RefOutParam'");
            luaEnv.Dispose();
        }

        void Update () {

        }

```

```
25.             public void RefOutParamFunc(string s, ref int a, out int b, ref int
    c)
26.             {
27.                 b = 1;
28.                 Debug.Log("s :" + s + ", a : " + a + ", b : " + b + ", c : " +
    c);
29.             }
30.         }
31.     }
```

?

调用重载方法

OverloadMethod.lua.txt

```
1.      local overloadMethod =
    CS.UnityEngine.Object.FindObjectOfType(typeof(CS.shenjun.OverLoadMethod))
2.
3.      --调用无参的方法
4.      overloadMethod:Func()
5.      --调用参数为int类型的方法
6.      overloadMethod:Func(10)
7.      --调用参数为string类型的方法
8.      overloadMethod:Func("shenjun")
```

OverloadMethod.cs

```
1.      /*
2.       *  created by shenjun
3.       */
4.
5.      using System.Collections;
6.      using System.Collections.Generic;
7.      using UnityEngine;
8.      using XLua;
9.
10.     namespace shenjun
11.     {
12.         public class OverLoadMethod : MonoBehaviour {
13.
14.             void Start () {
15.                 LuaEnv luaEnv = new LuaEnv();
16.                 luaEnv.DoString("require 'OverloadMethod'");
17.                 luaEnv.Dispose();
18.             }
19.
20.             void Update () {
21.
22.             }
23.
24.             public void Func()
```

```
25.             {
26.                 Debug.Log("Func NoParam");
27.             }
28.
29.             public void Func(int num)
30.             {
31.                 Debug.Log("Func int param :" + num);
32.             }
33.
34.             public void Func(string name)
35.             {
36.                 Debug.Log("Func string param :" + name);
37.             }
38.         }
39.     }
```

?

使用运算符重载

LuaCallCSOperator.lua.txt

```
    local operator = CS.UnityEngine.Object.FindObjectOfType(typeof(CS.shenjun.LuaCallCSOperator))
    local Quaternion = CS.UnityEngine.Quaternion
    local Vector3 = CS.UnityEngine.Vector3
    operator.transform.rotation = operator.transform.rotation * Quaternion.AngleAxis(45, Vector3.up)

    operator.transform.position = operator.transform.position + Vector3.up * 10
```

LuaCallCSOperator.cs

```
    /*
     *  created by shenjun
     */

    using System.Collections;
    using System.Collections.Generic;
    using UnityEngine;
    using XLua;

    namespace shenjun
    {
        public class LuaCallCSOperator : MonoBehaviour {

            LuaEnv luaEnv = new LuaEnv();

            void Start () {
                luaEnv.DoString("require 'LuaCallCSOperator'");
            }

            void Update () {

                if(luaEnv != null)
                {
                    luaEnv.Tick();
                }
```

```
            }

            private void OnDestroy()
            {
                luaEnv.Dispose();
            }
        }
    }
```

?

调用有默认参数方法

DefaultValueParam.lua.txt

```
    local default =
CS.UnityEngine.Object.FindObjectOfType(typeof(CS.shenjun.DefaultValueParam))

    --调用方法，使用默认参数
    default:DefaultValueMethod()
    --调用方法，不使用默认参数
    default:DefaultValueMethod(100)
```

DefaultValueParam.cs

```
    /*
     *  created by shenjun
     */

    using System.Collections;
    using System.Collections.Generic;
    using UnityEngine;
    using XLua;

    namespace shenjun
    {
        public class DefaultValueParam : MonoBehaviour {

            void Start () {

                LuaEnv luaEnv = new LuaEnv();
                luaEnv.DoString("require 'DefaultValueParam'");
                luaEnv.Dispose();
            }

            void Update () {

            }

            public void DefaultValueMethod(int num = 0)
            {
```

```
27.                 Debug.Log("num :" + num);
28.             }
29.         }
30.     }
```

?

Params参数

Params.lua.txt

```
1.    local params =
  CS.UnityEngine.Object.FindObjectOfType(typeof(CS.shenjun.Params))
2.    local result = params:Split(params.msg, ' ', '#')
3.
4.    local t = {}
5.    for i = 1, result.Length do
6.       t[#t+1] = result[i-1]
7.    end
8.
9.    for i,v in ipairs(t) do
10.       print(v)
11.   end
```

Params.cs

```
1.     /*
2.      *  created by shenjun
3.      */
4.
5.     using System.Collections;
6.     using System.Collections.Generic;
7.     using UnityEngine;
8.     using XLua;
9.
10.    namespace shenjun
11.    {
12.        public class Params : MonoBehaviour
13.        {
14.
15.            public string msg = "a b C#d";
16.
17.            void Start()
18.            {
19.                LuaEnv luaEnv = new LuaEnv();
20.                luaEnv.DoString("require 'Params'");
21.                luaEnv.Dispose();
```

```
22.             }
23.
24.             void Update()
25.             {
26.
27.             }
28.
29.             public string[] Split(string s, params string[] chs)
30.             {
31.                 return s.Split(chs,
    System.StringSplitOptions.RemoveEmptyEntries);
32.             }
33.         }
34.     }
```

?

调用扩展方法

ExtensionMethod.lua.txt

```
1.  local obj =
    CS.UnityEngine.Object.FindObjectOfType(typeof(CS.shenjun.ExtensionMethod))
2.  obj:ShowInfo()
```

ExtensionMethod.cs

```
1.  /*
2.   *  created by shenjun
3.   */
4.
5.  using System.Collections;
6.  using System.Collections.Generic;
7.  using UnityEngine;
8.  using XLua;
9.
10. namespace shenjun
11. {
12.     public class ExtensionMethod : MonoBehaviour {
13.
14.         void Start () {
15.             LuaEnv luaEnv = new LuaEnv();
16.             luaEnv.DoString("require 'ExtensionMethod'");
17.             luaEnv.Dispose();
18.         }
19.
20.         void Update () {
21.
22.         }
23.     }
24.
25.     [LuaCallCSharp]
26.     public static class ExtensionClass
27.     {
28.         public static void ShowInfo(this ExtensionMethod self)
29.         {
30.             Debug.Log(self.gameObject.name);
```

```
            }
        }
    }
```

?

调用泛型方法

Generic.lua.txt

```
1.  local obj =
    CS.UnityEngine.Object.FindObjectOfType(typeof(CS.shenjun.Generic))
2.  local cam = obj:GetCamera()
3.  local cam1 = obj:GetComponent();
4.
5.  if cam1 == cam then
6.      print("the same camera")
7.  end
```

Generic.cs

```
1.  /*
2.   *  created by shenjun
3.   */
4.
5.  using System.Collections;
6.  using System.Collections.Generic;
7.  using UnityEngine;
8.  using XLua;
9.
10. namespace shenjun
11. {
12.     public sealed class Generic : MonoBehaviour {
13.
14.
15.         void Start () {
16.             LuaEnv luaEnv = new LuaEnv();
17.             luaEnv.DoString("require 'Generic'");
18.             luaEnv.Dispose();
19.         }
20.
21.         void Update () {
22.
23.         }
24.
25.         public Camera GetCamera()
```

```
            {
                return this.GetComponent<Camera>();
            }
        }

        [LuaCallCSharp]
        public static class GenericExtension
        {
            public static Camera GetComponent(this Generic self)
            {
                return self.GetComponent<Camera>();
            }
        }
    }
```

?

调用枚举类型

CallCSEnum.lua.txt

```
    local callEnum = CS.UnityEngine.Object.FindObjectOfType(typeof(CS.shenjun.CallEnum))
    callEnum:EnemyAttack(callEnum.myType)

    --直接访问枚举
    local boss = CS.shenjun.EnemyType.Boss
    print("===========", boss)  -- Boss : 1
    callEnum:EnemyAttack(2)

    --把字符串转换成枚举
    local npc = CS.shenjun.EnemyType.__CastFrom('NPC')
    print("===========", npc)
    callEnum:EnemyAttack(npc)

    --把数字转换成枚举
    local mon = CS.shenjun.EnemyType.__CastFrom(0)
    print("===========", mon)
    callEnum:EnemyAttack(mon)
```

CallCSEnum.cs

```
    /*
     *  created by shenjun
     */

    using System.Collections;
    using System.Collections.Generic;
    using UnityEngine;
    using XLua;

    namespace shenjun
    {
        public class CallEnum : MonoBehaviour {

            public EnemyType myType = EnemyType.Boss;

```

```
16.             void Start () {
17.                 LuaEnv luaEnv = new LuaEnv();
18.                 luaEnv.DoString("require 'CallCSEnum'");
19.                 luaEnv.Dispose();
20.             }
21.
22.             void Update () {
23.
24.             }
25.
26.             public void EnemyAttack(EnemyType type)
27.             {
28.                 Debug.Log("EnemyType : " + type);
29.             }
30.         }
31.
32.         [LuaCallCSharp]
33.         public enum EnemyType
34.         {
35.             Monster,
36.             Boss,
37.             NPC,
38.         }
39.     }
```

?

调用委托

CallCSDelegate.lua.txt

```
    local callDel =
CS.UnityEngine.Object.FindObjectOfType(typeof(CS.shenjun.CallDelegate))

    -- 对委托进行初始化赋值
    callDel.del = callDel.del2

    -- 对委托添加成员方法 错误
    -- callDel.del = callDel.Test

    -- 在使用前定义
    function Test()
        print("Lua Func")
    end

    -- 右操作数可以是同类型的C# delegate/静态方法或者是lua函数。
    callDel.del = callDel.del + Test

    -- 删除一个委托
    --callDel.del = callDel.del - callDel.del2

    -- 判断委托是否为空
    if nil ~= callDel.del then
        -- 调用委托
        callDel:del()
    end

    --把委托变为空
    callDel.del = nil
```

CallCSDelegate.cs

```
    /*
     *  created by shenjun
     */

    using System.Collections;
```

```
6.      using System.Collections.Generic;
7.      using UnityEngine;
8.      using XLua;
9.
10.     namespace shenjun
11.     {
12.         public class CallDelegate : MonoBehaviour {
13.
14.             public System.Action del = null;
15.
16.             public System.Action del2 = null;
17.
18.             LuaEnv luaEnv = new LuaEnv();
19.             void Start () {
20.                 del2 = Test;
21.                 luaEnv.DoString("require 'CallCSDelegate'");
22.             }
23.
24.             void Update () {
25.
26.             }
27.
28.             private void OnDestroy()
29.             {
30.                 del = null;
31.                 del2 = null;
32.                 luaEnv.Dispose();
33.             }
34.
35.             public void Test()
36.             {
37.                 Debug.Log("CS Func");
38.             }
39.         }
40.     }
```

?

调用事件

CallCSEvent.lua.txt

```
1.      local callEvent =
    CS.UnityEngine.Object.FindObjectOfType(typeof(CS.shenjun.CallEvent))
2.
3.      -- 注册静态方法
4.      callEvent:onClick('+', CS.shenjun.CallEvent.StaticTest)
5.      callEvent:onValueChanged('+', CS.shenjun.CallEvent.StaticTest)
6.
7.      -- 取消注册的静态方法
8.      callEvent:onClick('-', CS.shenjun.CallEvent.StaticTest)
9.      callEvent:onValueChanged('-', CS.shenjun.CallEvent.StaticTest)
10.
11.     -- 注册一个实例方法 错误
12.     -- callEvent:onClick('+', callEvent.Test)
13.
14.     function test1()
15.         print('lua no param')
16.     end
17.
18.     function test2(n)
19.         print('lua have param :', n)
20.     end
21.
22.     -- 注册lua方法
23.     callEvent:onClick('+', test1)
24.     callEvent:onValueChanged('+', test2)
25.
26.     -- 调用事件执行
27.     callEvent:Invoke(10)
```

CallCSEvent.cs

```
1.      /*
2.       *  created by shenjun
3.       */
4.
5.      using System.Collections;
```

```
using System.Collections.Generic;
using UnityEngine;
using XLua;

namespace shenjun
{
    public class CallEvent : MonoBehaviour {

        public event System.Action onClick;

        [CSharpCallLua]
        public delegate void Del(int n);
        public event Del onValueChanged;

        void Start () {

            LuaEnv luaEnv = new LuaEnv();
            luaEnv.DoString("require 'CallCSEvent'");

            onClick = null;
            onValueChanged = null;
            luaEnv.Dispose();
        }

        void Update () {

        }

        public void Test()
        {
            Debug.Log("CS Func");
        }

        public static void StaticTest(int value = 0)
        {
            Debug.Log("CS Static Func :" + value);
        }

        public void Invoke(int value = 0)
        {
            if(onClick != null)
            {
```

```
48.                     onClick.Invoke();
49.                 }
50.
51.                 if(onValueChanged != null)
52.                 {
53.                     onValueChanged(value);
54.                 }
55.             }
56.         }
57.     }
```

?

复杂转换

ComplexConvert.lua.txt

```lua
local studentInfo = {
    name = "zs",
    age = 18,
    scores = {
        90, 80, 100
    }
}

local students = {}
students[1] = studentInfo;
students[1].name = "xm"

students[2] = { name = "ls", age = 20 }
students[2].scores = { 90, 50, 100 }


-- 创建Unity1711对象
local unity1711 = CS.shenjun.Unity1711()
unity1711:Add(studentInfo)
unity1711:Add(students[1])
unity1711:Add(students[2])

unity1711:ShowAll()

-- 获取所有成绩都及格的学生数据
print("show pass ======================")
local pass = unity1711:GetPass()
for i=1, pass.Length do
    CS.UnityEngine.Debug.Log(pass[i-1])
end

-- 获取所有某成绩不及格的学生数据
print("show fail ======================")
local fail = unity1711:GetFail()

for i=1, fail.Length do
    CS.UnityEngine.Debug.Log(fail[i-1])
```

```
38.     end
```

ComplexConvert.cs

```
1.      /*
2.       *  created by shenjun
3.       */
4.
5.      using System.Collections;
6.      using System.Collections.Generic;
7.      using System.Linq;
8.      using UnityEngine;
9.      using XLua;
10.
11.     namespace shenjun
12.     {
13.         public class ComplexConvert : MonoBehaviour
14.         {
15.
16.             void Start()
17.             {
18.
19.                 LuaEnv luaEnv = new LuaEnv();
20.                 luaEnv.DoString("require 'ComplexConvert'");
21.                 luaEnv.Dispose();
22.             }
23.
24.             void Update()
25.             {
26.
27.             }
28.         }
29.
30.         public class Unity1711
31.         {
32.             public List<StudentInfo> studentsInfo = new List<StudentInfo>();
33.
34.             public void Add(StudentInfo s)
35.             {
36.                 studentsInfo.Add(s);
37.             }
38.
```

```
39.             public void ShowAll()
40.             {
41.                 foreach (var item in studentsInfo)
42.                 {
43.                     Debug.Log(item);
44.                 }
45.             }
46.
47.             // 获取所有成绩都及格的学生数据
48.             public StudentInfo[] GetPass()
49.             {
50.                 var result = studentsInfo.Where(
51.                     n =>
52.                     {
53.                         return n.scores.All(m => m > 60);
54.                     }
55.                 );
56.                 return result.ToArray();
57.             }
58.
59.             // 获取所有某成绩不及格的学生数据
60.             public StudentInfo[] GetFail()
61.             {
62.                 var result = studentsInfo.Where(
63.                     n =>
64.                     {
65.                         return n.scores.Any(m => m < 60);
66.                     }
67.                 );
68.                 return result.ToArray();
69.             }
70.         }
71.
72.         [XLua.LuaCallCSharp]
73.         public struct StudentInfo
74.         {
75.             public string name;
76.             public int age;
77.             public int[] scores;
78.
79.             public override string ToString()
80.             {
```

```
81.                 //var score = from n in scores select n + "";
82.                 var score = scores.Select(n => n + "");
83.                 string scoreStr = string.Join("#", score.ToArray());
84.                 return string.Format("[name:{0}, age:{1}, score:{2}]", name,
    age, scoreStr);
85.             }
86.         }
87.     }
```

?

调用协同程序

LuaCoroutine.lua.txt

```
    local util = require "xlua.util" -- 加载 Resources/xlua/util.lua.txt

    local coroutineInstance = CS.UnityEngine.GameObject.FindObjectOfType(typeof(CS.shenjun.Coroutine))

    local function async_yield_return(to_yield, cb)
        coroutineInstance:InvokeSpawn(to_yield, cb)
    end

    -- 将调用C#协同的方法 转换为可以在lua协同中调用的方法 yield_return需要在lua协同中执行
    local yield_return = util.async_to_sync(async_yield_return)

    local function coroutine_callback()
        print("lua coroutine over! callback done!")
    end

    local function co_function()
        print("lua coroutine start")
        local wt = CS.UnityEngine.WaitForSeconds(3)
        yield_return(wt, coroutine_callback)
    end

    -- local co = coroutine.create(co_function)

    function Start()
        -- coroutine.resume(co)
        util.coroutine_call(co_function)()
    end
```

```
37.
38.
39.
40.
41.
42.     --[===[  lua coroutine 语法
43.
44.     --[[ 协同案例1 （基本语法）
45.
46.     fun1 = function()
47.         print("hello")
48.     end
49.
50.     -- 创建一个协同程序 co
51.     co = coroutine.create(fun1)
52.
53.     -- 协同程序的4种状态 suspended、running、dead、normal
54.     -- 创建完协同程序以后，处于suspended状态
55.     print("status : ", coroutine.status(co))
56.
57.     -- 启动协同程序（第一次唤醒）
58.     coroutine.resume(co)
59.     -- 协同程序运行完以后，处于dead状态，再次执行没效果
60.     print("status : ", coroutine.status(co))
61.
62.     -- 再次唤醒已经dead的协同程序，无反应，说明不会再次执行了
63.     coroutine.resume(co)
64.     print("status : ", coroutine.status(co))
65.
66.     -- 协同程序处理dead状态，再次唤醒，返回false及一条错误信息
67.     print(coroutine.resume(co))
68.
69.     ]]
70.
71.
72.     --[[ 协同案例2 （yield方法）
73.
74.     -- 协同的强大之处 coroutine.yield()
75.     -- 方法中有yield语句
76.     -- 作用：让一个运行中的程序挂起，而之后可以恢复它的运行
77.
78.     co = coroutine.create(function()
```

```
            print("start coroutine function")
            for i=1, 3 do
                print("co :", i)
                coroutine.yield()
            end
        end)

    -- 处于suspended状态
    print(coroutine.status(co))

    -- 第一次唤醒，运行到yield方法时返回，挂起发生在yield调用中
    coroutine.resume(co)  -- 打印1
    print("1", coroutine.status(co))

    -- 第二次唤醒，从上一次的yield的调用中返回，继续执行，碰到yield再挂起
    coroutine.resume(co)  -- 打印2
    print("2", coroutine.status(co))

    -- 第三次唤醒
    coroutine.resume(co)  -- 打印3
    print("3", coroutine.status(co))

    print("3-4", coroutine.status(co))  -- suspended状态

    -- 第四次唤醒
    coroutine.resume(co)  -- 什么都不打印
    print("4", coroutine.status(co))

    --]]




    -- 当一个协同程序A唤醒一个协同程序B时，协同程序A就处于一个特殊状态，
    -- 既不是挂起状态（无法继续A的执行），也不是运行状态（是B在运行）。
    -- 所以将这时的状态称为“正常”状态。

    -- Lua的协同程序还具有一项有用的机制，就是可以通过一对resume-yield来交换数据。
```

```
121.      --[[ 协同案例3 （带参数协同）
122.
123.      -- 创建一个带参数的协同程序
124.      co = coroutine.create(function(a, b)
125.              for i=1,3 do
126.                  print("co", a, b, i)
127.                  coroutine.yield()
128.              end
129.
130.          end)
131.
132.      -- 第一次唤醒 参数1, 2传给function
133.      coroutine.resume(co, 10, 20)   -- 缺少参数使用nil, 多余的参数舍弃
134.
135.      -- 第二次唤醒
136.      coroutine.resume(co, 100, 200) -- 传入100, 200无效, 有其他用途
137.
138.      --]]
139.
140.
141.      --[[ 协同案例4 （resume的返回值）
142.
143.      co = coroutine.create(function(a, b)
144.              coroutine.yield(a + b, a - b)
145.              return 6, 7
146.          end)
147.
148.      -- 第一次唤醒
149.      -- 在resume的返回值中, 第一个返回值为true表示没有错误
150.      -- 第2个返回值以及之后的, 都对应yield传入的参数
151.      print(coroutine.resume(co, 10, 3))  -- true  13  7
152.      -- 当一个协同程序结束时, 它的所有返回值都作为resume的返回值。
153.      print(coroutine.resume(co))         -- true  6   7
154.      --]]
155.
156.
157.      --[[ 协同案例5 yield的返回值
158.
159.      co = coroutine.create(function()
160.              print("co", coroutine.yield("yield"))
161.          end)
162.
```

```
    -- 第一次唤醒，传入的参数“a”是传给方法的，由于方法没有参数，所以舍弃“a”
    print("1", coroutine.resume(co, "a"))   -- "1"  true  yield

    -- 第二次唤醒，传入的参数1，2，作为yield的额外返回值使用（yield返回的额外值就是对应resume传入的参数）
    print("2", coroutine.resume(co, 1, 2))  -- "co"  1  2  -- "2"  true

    print("3", coroutine.resume(co, 100, 200)) -- "3"  false  cannot resume dead coroutine
    --]]


    --[[ *协同案例6* 协同程序的嵌套使用（管道与过滤器）

    -- 唤醒传入的迭代器
    function receive(prod)
        local status, value = coroutine.resume(prod);
        return value;
    end

    funcition send(x)
        coroutine.yield(x)
    end

    -- 返回一个挂起的协同程序 协同程序中需要有yield
    function producer()
        return coroutine.create(function()
            while true do
                local x = io.read
                send(x)                 -- 产生新值
            end
        end)
    end

    function filter(prod)
        return coroutine.create(function()
            for line = 1, math.huge do
                local x = receive(prod) -- 接收producer产生的值
                x = string.format("%5d %s", line, x)
                send(x)                 -- 把新值发送给消费者
            end
        end)
```

```
    end

    function consumer(prod)
        while true do
            local x = receive(prod)     -- 获取新值
            io.write(x, "\n")           -- 消费新值
        end
    end

    -- 运行代码
    consumer(filter(producer))

    --]]

    ]===]
```

LuaCoroutine.cs

```
    /*
     *  created by shenjun
     */

    using System.Collections;
    using System.Collections.Generic;
    using UnityEngine;
    using XLua;

    namespace shenjun
    {
        [LuaCallCSharp]
        public class Coroutine : MonoBehaviour {

            LuaEnv luaEnv = new LuaEnv();

            void Start () {
                luaEnv.DoString("require 'LuaCoroutine'");
                System.Action luaStart = luaEnv.Global.Get<System.Action>
("Start");

                if (null != luaStart)
                    luaStart();
            }
```

```
24.
25.             void Update () {
26.
27.             }
28.
29.             private void OnDestroy()
30.             {
31.                 luaEnv.Dispose();
32.             }
33.
34.             public void InvokeSpawn(object yield_return, System.Action
    callback)
35.             {
36.                 StartCoroutine(SpawnObj(yield_return, callback));
37.             }
38.
39.             IEnumerator SpawnObj(object yield_return, System.Action callback)
40.             {
41.                 for (int i = 0; i < 10; i++)
42.                 {
43.                     GameObject go =
    GameObject.CreatePrimitive(PrimitiveType.Sphere);
44.                     go.transform.position = Random.insideUnitSphere * 5;
45.                     if(yield_return is IEnumerator)
46.                     {
47.                         yield return StartCoroutine(yield_return as
    IEnumerator);
48.                     }
49.                     else
50.                     {
51.                         yield return yield_return;
52.                     }
53.                 }
54.                 callback();
55.             }
56.         }
57.
58.         [LuaCallCSharp]
59.         public static class CoroutineConfig
60.         {
61.             public static List<System.Type> LuaCallCSharp
62.             {
```

```
63.                 get{
64.                     return new List<System.Type>()
65.                     {
66.                         typeof(WaitForSeconds)
67.                     };
68.                 }
69.             }
70.         }
71.     }
```

?

载入Lua脚本

一、通过文件

```
using UnityEngine;
using System.Collections;
using XLua;

public class ByFile : MonoBehaviour {
    LuaEnv luaenv = null;
    void Start()
    {
        luaenv = new LuaEnv();
        // 使用默认loader加载Resources下的"byfile.lua.txt"文件
        luaenv.DoString("require 'byfile'");
    }

    void Update()
    {
        if (luaenv != null)
        {
            luaenv.Tick();
        }
    }

    void OnDestroy()
    {
        luaenv.Dispose();
    }
}
```

将 `byfile.lua.txt` 存放在 `Resources` 文件夹下

```
print('hello world')
```

二、通过字符串

```
/*
 *  created by shenjun
 */
```

```
4.
5.      using System.Collections;
6.      using System.Collections.Generic;
7.      using UnityEngine;
8.
9.      /*
10.      * 1、引入命名空间
11.      */
12.     using XLua;
13.
14.     namespace shenjun
15.     {
16.         public class ByString : MonoBehaviour {
17.
18.             LuaEnv luaEnv = new LuaEnv();
19.
20.             [Multiline(10)]
21.             public string luaStr1 =
22.                 @"function Show(...)
23.                     local t = {...}
24.                     for i=1,#t do
25.                         print(t[i])
26.                     end
27.                   end
28.                   Show(5,3,4,1,2)";
29.
30.             [Multiline(10)]
31.             public string luaStr2 =
32.                 @"function GetResult()
33.                       return 1, '2'
34.                   end
35.                   return GetResult()";
36.
37.             void Start () {
38.
39.                 // public object[] DoString(string chunk, string chunkName =
    "chunk", LuaTable env = null)
40.                 // 参数1: Lua代码
41.                 // 参数2: 发生error时debug显示信息时使用，指明某某代码块的某行错误
42.                 // 参数3: 为这个代码块
43.                 // 返回值：Lua代码的返回值
44.
```

```
45.                 // 执行Lua代码
46.                 luaEnv.DoString("print('Hello World!')");
47.                 luaEnv.DoString(luaStr1);
48.                 object[] results = luaEnv.DoString(luaStr2);
49.                 if(results != null)
50.                 {
51.                     foreach (var item in results)
52.                     {
53.                         Debug.Log(item);
54.                     }
55.                 }
56.             }
57.
58.             void Update () {
59.
60.                 if(luaEnv != null)
61.                 {
62.                     // 清除Lua未手动调用的LuaBase（比如LuaTable, LuaFunction），及其他一些事情。
63.                     // 需要定期调用。
64.                     luaEnv.Tick();
65.                 }
66.
67.             }
68.
69.             private void OnDestroy()
70.             {
71.                 luaEnv.Dispose();
72.             }
73.         }
74.     }
```

三、通过加载器

Example01 :

```
1.     using UnityEngine;
2.     using System.Collections;
3.     using XLua;
4.
5.     public class CustomLoader : MonoBehaviour {
6.         LuaEnv luaenv = null;
```

```
        void Start()
        {
            luaenv = new LuaEnv();
            luaenv.AddLoader((ref string filename) => {
                 if (filename == "InMemory")
                 {
                     string script = "return {ccc = 9999}";
                     return System.Text.Encoding.UTF8.GetBytes(script);
                 }
                 return null;
             });
            luaenv.DoString("print('InMemory.ccc=', require('InMemory').ccc)");
        }

        void Update()
        {
            if (luaenv != null)
            {
                luaenv.Tick();
            }
        }

        void OnDestroy()
        {
            luaenv.Dispose();
        }
    }
```

Example02 :

```
    /*
     *  created by shenjun
     */

    using System.Collections;
    using System.Collections.Generic;
    using UnityEngine;
    using System.IO;

    using XLua;

    namespace shenjun
```

```
13.     {
14.         public class ByLoader : MonoBehaviour {
15.
16.             LuaEnv luaEnv = new LuaEnv();
17.
18.             void Start () {
19.
20.                 // 注册自定义loader
21.                 luaEnv.AddLoader((ref string filePath)=>{
22.                     string path = Application.dataPath +
    "/LessonXLua_shenjun/01_LoadLuaScript/03_ByLoader/" + filePath + ".lua.txt";
23.                     if(File.Exists(path))
24.                     {
25.                         string text = "";
26.                         using (StreamReader sr = new StreamReader(path))
27.                         {
28.                             try
29.                             {
30.                                 text = sr.ReadToEnd();
31.                             }
32.                             catch(System.Exception ex)
33.                             {
34.                                 Debug.LogError("ReadFile Error :" + path + ";
    ErrorMsg :" + ex.Message);
35.                                 return null;
36.                             }
37.                         }
38.                         return System.Text.Encoding.UTF8.GetBytes(text);
39.
40.                         //byte[] buffer;
41.                         //using(FileStream fsReader = new FileStream(path,
    FileMode.Open, FileAccess.Read))
42.                         //{
43.                         //    buffer = new byte[fsReader.Length];
44.                         //    try
45.                         //    {
46.                         //        fsReader.Read(buffer, 0,
    (int)fsReader.Length);
47.                         //        return buffer;
48.                         //    }
49.                         //    catch
50.                         //    {
```

```
                        //        return null;
                        //    }
                        //}
                    }
                    else
                    {
                        return null;
                    }
                });

                // 如果注册了自定义的loader，会先查找自定义loader，如果自定义loader返回null，则会使用默认loader，如果默认loader也没找到，则报错。
                luaEnv.DoString("require 'byloader'");
            }

            void Update () {

                if(luaEnv != null)
                {
                    luaEnv.Tick();
                }

            }

            private void OnDestroy()
            {
                luaEnv.Dispose();
            }
        }
    }
```

byloader.lua.txt

```
    print('byCustomLoader: hello world')
```

?

UI事件

Editor目录下MyGenericTypesStaticList.cs

```
/*
 *  created by shenjun
 */

using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using System;
using System.Linq;
using System.Reflection;
using XLua;

[LuaCallCSharp]
public static class MyGenericTypeStaticList
{

    [CSharpCallLua]
    public static List<Type> mymodule_lua_call_cs_list = new List<Type>()
        {
            typeof(UnityEngine.EventSystems.IPointerDownHandler),
        };

    [Hotfix]
    public static List<Type> by_property
    {
        get
        {
            return (from type in Assembly.GetExecutingAssembly().GetTypes()
                    where type.Namespace == "UnityEngine.EventSystems"
                    select type).ToList();
        }
    }

}
```

Resources目录下LuaUIEventLocal.lua.txt

```
    LuaPanel = {}
    local this = LuaPanel

    function LuaPanel.Awake(self, csObj)
        print("LuaPanel Awake")
        this.self = csObj
        this.Button = this.self.transform:GetComponentInChildren(typeof(CS.UnityEngine.UI.Button));
        this.Button.onClick:AddListener(this.OnBtnClick)

        this.Text = this.self.transform:Find("Text"):GetComponent("Text");
        this.Text.text = "LuaPanel"
    end

    function LuaPanel.OnDestroy()
        print("LuaPanel OnDestroy")
        this.Button.onClick:RemoveListener(this.OnBtnClick)
        this.Button.onClick:Invoke()
    end

    function LuaPanel.OnBtnClick()
        print("LuaPanel OnBtnClick")
        this.Text.text = "Button Clicked"
    end

    function LuaPanel.OnPanelClick(eventData)
        print("LuaPanel OnPanelClick")
        this.Text.text = "LuaPanel"
    end

    function LuaPanel.OnPointerDown(eventData)
        this.OnPanelClick(eventData)
    end
```

Resources目录下UIEventGlobal.lua.txt

```
    this = {}

    function Awake(luaEvent)
        this.self = luaEvent
        this.Button = this.self.transform:GetComponentInChildren(typeof(CS.UnityEngine.UI.Button))
```

```
        this.Button.onClick:AddListener(OnBtnClick)
        this.Text = this.self.transform:Find("Text"):GetComponent("Text")
        print("Lua Awake!")
    end

    function Start()
        print("Lua Start!")
    end

    function Update()
        print("Lua Update!")
    end

    function OnDestroy()
        this.Button.onClick:RemoveListener(OnBtnClick)
        this.Button.onClick:Invoke()
        print("Lua OnDestroy!")
    end

    function OnBtnClick()
        this.Text.text = "Lua OnBtnClick"
        print("Lua OnBtnClick!")
    end

    function OnPanelClick(eventData)
        this.Text.text = "Lua OnPanelClick"
        print("Lua OnPanelClick!")
    end

    function OnPointerDown(eventData)
        OnPanelClick(eventData)
    end
```

LuaUIEventGlobal.cs

```
    /*
     *  created by shenjun
     */

    using System.Collections;
    using System.Collections.Generic;
    using UnityEngine;
```

```
8.      using UnityEngine.EventSystems;
9.      using System;
10.     using XLua;
11.
12.     namespace shenjun
13.     {
14.         [CSharpCallLua]
15.         public delegate void AwakeSet(LuaUIEventGlobal self);
16.
17.         [CSharpCallLua]
18.         public delegate void OnPanelClick(PointerEventData eventData);
19.
20.         public class LuaUIEventGlobal : MonoBehaviour, IPointerDownHandler {
21.
22.             LuaEnv luaEnv = new LuaEnv();
23.
24.             AwakeSet luaAwake;
25.             Action luaStart;
26.             Action luaUpdate;
27.             Action luaOnDestroy;
28.
29.
30.             OnPanelClick onPanelClickDel;
31.
32.             private void Awake()
33.             {
34.                 luaEnv.DoString("require 'UIEventGlobal'");
35.
36.                 luaAwake = luaEnv.Global.Get<AwakeSet>("Awake");
37.                 luaAwake(this);
38.
39.                 luaStart = luaEnv.Global.Get<Action>("Start");
40.                 luaUpdate = luaEnv.Global.Get<Action>("Update");
41.                 luaOnDestroy = luaEnv.Global.Get<Action>("OnDestroy");
42.
43.                 onPanelClickDel = luaEnv.Global.Get<OnPanelClick>
    ("OnPanelClick");
44.             }
45.
46.             void Start () {
47.                 luaStart();
48.             }
```

```
49.
50.             void Update () {
51.                 luaUpdate();
52.             }
53.
54.             private void OnDestroy()
55.             {
56.                 luaAwake = null;
57.                 luaStart = null;
58.                 luaUpdate = null;
59.                 luaOnDestroy();
60.                 luaOnDestroy = null;
61.                 onPanelClickDel = null;
62.
63.                 luaEnv.Dispose();
64.             }
65.
66.             public void OnPointerDown(PointerEventData eventData)
67.             {
68.                 if (onPanelClickDel != null)
69.                 onPanelClickDel(eventData);
70.
71.             }
72.         }
73.     }
```

LuaUIEventLocal.cs

```
1.      /*
2.       *  created by shenjun
3.       */
4.
5.      using System.Collections;
6.      using System.Collections.Generic;
7.      using UnityEngine;
8.      using UnityEngine.EventSystems;
9.      using XLua;
10.
11.     namespace shenjun
12.     {
13.         //[CSharpCallLua]
14.         //public delegate void OnPanelClickDel(PointerEventData eventData);
```

```
15.
16.         [LuaCallCSharp]
17.         public class LuaUIEventLocal : MonoBehaviour, IPointerDownHandler {
18.
19.             LuaEnv luaEnv = new LuaEnv();
20.             ILuaPanel panel;
21.
22.             IPointerDownHandler panelPointDown;
23.
24.             public void OnPointerDown(PointerEventData eventData)
25.             {
26.                 //if (panel.OnPanelClick != null)
27.                 //panel.OnPanelClick(eventData);
28.
29.                 if (panelPointDown != null)
30.                     panelPointDown.OnPointerDown(eventData);
31.             }
32.
33.             void Awake()
34.             {
35.                 luaEnv.DoString("require 'LuaUIEventLocal'");
36.
37.                 panel = luaEnv.Global.Get<ILuaPanel>("LuaPanel");
38.                 panelPointDown = luaEnv.Global.Get<IPointerDownHandler>
    ("LuaPanel");
39.                 //panel.self = this; // lua中也会有self的键值
40.
41.                 panel.Awake(this);
42.             }
43.
44.             void Start () {
45.
46.             }
47.
48.             void Update () {
49.
50.                 if(luaEnv != null)
51.                 {
52.                     luaEnv.Tick();
53.                 }
54.             }
55.
```

```
            void OnDestroy()
            {
                panel.OnDestroy();
                //if(panel.OnPanelClick != null)
                //{
                //    panel.OnPanelClick = null;
                //}

                if (panelPointDown != null)
                    panelPointDown = null;
                luaEnv.Dispose();
            }
        }

        [CSharpCallLua]
        interface ILuaPanel
        {
            //LuaUIEventLocal self { get; set; }
            //OnPanelClickDel OnPanelClick { get; set; }
            void Awake(LuaUIEventLocal self);
            void OnDestroy();
        }

    }
```

?

热更新

Editor目录下CreateAssetBundle.cs

```
1.  /*
2.   *  created by shenjun
3.   */
4.
5.  using System.Collections;
6.  using System.Collections.Generic;
7.  using UnityEngine;
8.  using UnityEditor;
9.  using UnityEngine.Networking;
10.
11. namespace shenjun
12. {
13.     public class CreateAssetBundle : EditorWindow
14.     {
15.
16.         public static string createAssetPath{ get { return
    Application.dataPath +
    "/LessonXLua_shenjun/04_Examples/02_HotUpdate/AssetBundle/Build/"; }}
17.
18.
19.         static CreateAssetBundle window;
20.
21.         int platformSelect = 0;
22.         string[] platformName = { "Windows", "Mac OS", "IOS", "Android",
    "All" };
23.
24.         string assetbundlePath = "";
25.         string Path
26.         {
27.             set {
28.                 assetbundlePath = value;
29.             }
30.             get
31.             {
32.                 if(string.IsNullOrEmpty(assetbundlePath))
33.                 {
34.                     assetbundlePath = createAssetPath;
```

```
35.                     }
36.                     return assetbundlePath;
37.                 }
38.             }
39.
40.
41.             [MenuItem("AssetBundle/CreateAssetBundleTools")]
42.             static void ShowWindow()
43.             {
44.                 if (window == null)
45.                 {
46.                     window = EditorWindow.GetWindow<CreateAssetBundle>(true,
    "AssetBundle Tools");
47.                     window.minSize = new Vector2(800, 600);
48.                 }
49.                 window.Show();
50.             }
51.
52.             void OnGUI()
53.             {
54.                 platformSelect = GUILayout.SelectionGrid(platformSelect,
    platformName, platformName.Length);
55.
56.                 GUILayout.BeginHorizontal();
57.
58.                 GUILayout.Label("打包路径：");
59.                 GUILayout.Label(Path);
60.
61.                 if(GUILayout.Button("选择路径", GUILayout.MaxWidth(100)))
62.                 {
63.                     Path = EditorUtility.OpenFolderPanel("选择文件夹",
    Application.dataPath, Path);
64.                 }
65.
66.                 GUILayout.EndHorizontal();
67.                 if (GUILayout.Button("打包"))
68.                 {
69.                     CreateBundle(platformSelect);
70.                     EditorUtility.DisplayDialog("资源打包信息提示", "打包完成", "确
    定");
71.                 }
72.             }
```

```
73.
74.             void CreateBundle(int plat)
75.             {
76.                 string combinePath = "";
77.                 BuildTarget target = BuildTarget.NoTarget;
78.
79.                 switch(plat)
80.                 {
81.                     case 0:
82.                         combinePath = Path + "Windows";
83.                         target = BuildTarget.StandaloneWindows;
84.                         break;
85.                     case 1:
86.                         combinePath = Path + "Mac OS";
87.                         target = BuildTarget.StandaloneOSX;
88.                         break;
89.                     case 2:
90.                         combinePath = Path + "IOS";
91.                         target = BuildTarget.iOS;
92.                         break;
93.                     case 3:
94.                         combinePath = Path + "Android";
95.                         target = BuildTarget.Android;
96.                         break;
97.                     case 4:
98.                         for (int i = 0; i < 4; i++)
99.                         {
100.                            CreateBundle(i);
101.                        }
102.                        return;
103.                }
104.
105.                if (!System.IO.Directory.Exists(combinePath))
106.                {
107.                    System.IO.Directory.CreateDirectory(combinePath);
108.                }
109.                CreateBundle(combinePath, target);
110.            }
111.
112.            void CreateBundle(string path ,BuildTarget target)
113.            {
114.                BuildPipeline.BuildAssetBundles(path,
```

```
    BuildAssetBundleOptions.ForceRebuildAssetBundle, target);
115.                 AssetDatabase.Refresh();
116.             }
117.
118.             [MenuItem("AssetBundle/资源清理(删除本地及远程数据)")]
119.             public static void ClearDirectory()
120.             {
121.                 string[] clearPath = { LoadHotUpdate.localAssetPath,
    LoadHotUpdate.remoteAssetPath };
122.                 for (int i = 0; i < clearPath.Length; i++)
123.                 {
124.                     string[] fileNames =
    System.IO.Directory.GetFiles(clearPath[i]);
125.                     for (int j = 0; j < fileNames.Length; j++)
126.                     {
127.                         System.IO.File.Delete(fileNames[j]);
128.                     }
129.                 }
130.                 AssetDatabase.Refresh();
131.             }
132.         }
133.     }
```

HotUpdateLua.cs

```
 1.     /*
 2.      *  created by shenjun
 3.      */
 4.
 5.     using System.Collections;
 6.     using System.Collections.Generic;
 7.     using UnityEngine;
 8.     using System;
 9.     using UnityEngine.EventSystems;
10.     using XLua;
11.
12.     namespace shenjun
13.     {
14.         public class HotUpdateLua : MonoBehaviour, IPointerDownHandler {
15.
16.             [CSharpCallLua]
17.             public delegate void AwakeSet(HotUpdateLua self);
```

```
18.
19.             [CSharpCallLua]
20.             public delegate void OnPanelClick(PointerEventData eventData);
21.
22.             LuaEnv luaEnv = new LuaEnv();
23.
24.             AwakeSet awake;
25.             Action start;
26.             Action update;
27.             Action destroy;
28.
29.             OnPanelClick panelClick;
30.
31.             void Awake()
32.             {
33.                 AssetBundle bundle =
    AssetBundle.LoadFromFile(LoadHotUpdate.localAssetPath + "luahotupdate_lua.ab");
34.                 TextAsset luaAsset = bundle.LoadAsset<TextAsset>
    ("LuaHotUpdate.lua.txt");
35.
36.                 luaEnv.DoString(luaAsset.text);
37.
38.                 awake = luaEnv.Global.Get<AwakeSet>("Awake");
39.                 start = luaEnv.Global.Get<Action>("Start");
40.                 update = luaEnv.Global.Get<Action>("Update");
41.                 destroy = luaEnv.Global.Get<Action>("OnDestroy");
42.
43.                 panelClick = luaEnv.Global.Get<OnPanelClick>("OnPanelClick");
44.
45.                 if (awake != null) awake(this);
46.             }
47.
48.             void Start () {
49.
50.                 if (start != null) start();
51.             }
52.
53.             void Update () {
54.                 if (update != null) update();
55.             }
56.
57.             void OnDestroy()
```

```
            {
                awake = null;
                start = null;
                update = null;

                if (destroy != null)
                {
                    destroy();
                    destroy = null;
                }

                panelClick = null;
                luaEnv.Dispose();
            }


            public void OnPointerDown(PointerEventData eventData)
            {
                if (panelClick != null) panelClick(eventData);
            }
        }
    }
```

LoadHotUpdate.cs

```
    /*
     *  created by shenjun
     */

    using System.Collections;
    using System.Collections.Generic;
    using UnityEngine;
    using System;
    using System.IO;
    using UnityEngine.Networking;
    using UnityEngine.UI;

    namespace shenjun
    {
        public class LoadHotUpdate : MonoBehaviour
        {
            static Dictionary<string, AssetBundle> bundleCaches = new
```

```
    Dictionary<string, AssetBundle>();
18.             static AssetBundleManifest manifest = null;
19.
20.
21.             public static string localAssetPath
22.             {
23.                 get
24.                 {
25.                     // FIXME : 移动平台的路径不一样
26.                     // 移动平台的话 请使用路径（具有可读写权限）：
    Application.persistentDataPath
27.                     return Application.dataPath +
    "/LessonXLua_shenjun/04_Examples/02_HotUpdate/AssetBundle/Local/";
28.     //#if UNITY_STANDALONE_WIN || UNITY_STANDALONE_OSX
29.     //#elif UNITY_IOS
30.     //#elif UNITY_ANDROID
31.     //#endif
32.                 }
33.             }
34.
35.             public static string remoteAssetPath
36.             {
37.                 get
38.                 {
39.                     // FIXME : 移动平台的路径不一样 同 localAssetPath
40.                     return Application.dataPath +
    "/LessonXLua_shenjun/04_Examples/02_HotUpdate/AssetBundle/Remote/";
41.                 }
42.             }
43.
44.             void Start()
45.             {
46.                 // 从服务器 下载配置文件列表  配置文件格式：txt json xml cvs
47.                 // 和本地配置文件对比  版本号、资源校验码等 不同的则更新
48.                 // 更新
49.                 // 运行程序
50.
51.                 StartCoroutine(LoadRemoteProfile(LoadAsset));
52.             }
53.
54.             void Update()
55.             {
```

```
56.             }
57.
58.             private void OnDestroy()
59.             {
60.                 bundleCaches.Clear();
61.                 AssetBundle.UnloadAllAssetBundles(true);
62.             }
63.
64.             IEnumerator LoadRemoteProfile(Action<string> onFinished)
65.             {
66.                 //Caching.ClearCache();
67.                 //while (!Caching.ready)
68.                 //yield return null;
69.
70.                 string url =
71.     #if UNITY_STANDALONE_WIN
72.                 "file:///" +
73.     #elif UNITY_STANDALONE_OSX
74.                 "file://" +
75.     #endif
76.                     remoteAssetPath + "luahotupdate_profile.ab";
77.
78.                 AssetBundle bundle = null;
79.
80.                 // 1、WWW.LoadFromCacheOrDownload 加载
81.                 //using(WWW www = WWW.LoadFromCacheOrDownload(url, 0))
82.                 //{
83.                 //    yield return www;
84.                 //    if (!string.IsNullOrEmpty(www.error))
85.                 //    {
86.                 //        Debug.Log(www.error);
87.                 //        yield break;
88.                 //    }
89.                 //    bundle = www.assetBundle;
90.                 //}
91.
92.                 // 2、UnityWebRequest.GetAssetBundle 加载
93.                 UnityWebRequest webRequest =
    UnityWebRequest.GetAssetBundle(url);
94.                 yield return webRequest.SendWebRequest();
95.                 if (!string.IsNullOrEmpty(webRequest.error))
96.                 {
```

```
97.                     Debug.Log(webRequest.error);
98.                     yield break;
99.                 }
100.                bundle = DownloadHandlerAssetBundle.GetContent(webRequest);
101.                // 3、AssetBundle.LoadFromFile 加载
102.                //bundle = AssetBundle.LoadFromFile(remoteAssetPath +
    "luahotupdate_profile.ab");
103.
104.                // 4、AssetBundle.LoadFromFileAsync 加载
105.                //AssetBundleCreateRequest request =
    AssetBundle.LoadFromMemoryAsync(File.ReadAllBytes(remoteAssetPath +
    "luahotupdate_profile.ab"));
106.                //yield return request;
107.                //bundle = request.assetBundle;
108.
109.                TextAsset profile = bundle.LoadAsset<TextAsset>("file");
110.                onFinished(profile.text);
111.            }
112.
113.            void LoadAsset(string remoteVersion)
114.            {
115.                AssetBundle.UnloadAllAssetBundles(true);
116.
117.                // 加载本地配置文件
118.                string profilePath = localAssetPath +
    "luahotupdate_profile.ab";
119.                if (!File.Exists(profilePath))
120.                {
121.                    // 加载服务器文件列表
122.                    LoadRemoteAsset();
123.                }
124.                else
125.                {
126.                    // 加载本地列表并比较
127.                    AssetBundle bundle = AssetBundle.LoadFromFile(profilePath);
128.                    TextAsset profile = bundle.LoadAsset<TextAsset>("file");
129.
130.                    float localVersion, removeVersion;
131.                    if (float.TryParse(profile.text, out localVersion) &&
    float.TryParse(remoteVersion, out removeVersion))
132.                    {
133.                        if (removeVersion > localVersion)
```

```
                        {
                            // 加载服务器文件列表
                            LoadRemoteAsset();
                        }
                    }
                }

                LoadPanel();
            }

            // 加载服务器文件列表
            void LoadRemoteAsset()
            {
                if (Directory.Exists(localAssetPath))
                {
                    string[] oldFiles = Directory.GetFiles(localAssetPath);
                    for (int i = 0; i < oldFiles.Length; i++)
                    {
                        File.Delete(oldFiles[i]);
                    }

                    Directory.Delete(localAssetPath);
                    Directory.CreateDirectory(localAssetPath);
                }
                string[] files = Directory.GetFiles(remoteAssetPath);
                for (int i = 0; i < files.Length; i++)
                {
                    string fileName =
    files[i].Substring(files[i].LastIndexOf("/", StringComparison.Ordinal) + 1);
                    File.Copy(remoteAssetPath + fileName, localAssetPath +
    fileName);
                }
            }

            // 加载UI
            void LoadPanel()
            {
                LoadAssetBundleInstance("luahotupdate_ui.ab", "Panel",
    transform);
            }

            public static GameObject LoadAssetBundleInstance(string bundleName,
```

```
    string objName, Transform parent = null)
173.             {
174.                 AssetBundle bundle = LoadAssetBundle(bundleName);
175.                 GameObject obj = bundle.LoadAsset<GameObject>(objName);
176.                 return Instantiate(obj, parent);
177.             }
178.
179.             // 加载AssetBundle包以及依赖文件
180.             public static AssetBundle LoadAssetBundle(string bundleName)
181.             {
182.                 if (manifest == null)
183.                 {
184.                     string platStr = "";
185.     #if UNITY_STANDALONE_WIN
186.                     platStr = "Windows";
187.     #elif UNITY_STANDALONE_OSX
188.                     platStr = "Mac OS";
189.     #elif UNITY_IOS
190.                     platStr = "IOS";
191.     #elif UNITY_ANDROID
192.                     platStr = "Android";
193.     #endif
194.
195.                     AssetBundle platBundle =
    AssetBundle.LoadFromFile(localAssetPath + platStr);
196.                     manifest = platBundle.LoadAsset<AssetBundleManifest>
    ("AssetBundleManifest");
197.                 }
198.
199.                 if (bundleCaches.ContainsKey(bundleName))
200.                 {
201.                     return bundleCaches[bundleName];
202.                 }
203.                 else
204.                 {
205.                     AssetBundle bundle =
    AssetBundle.LoadFromFile(localAssetPath + bundleName);
206.                     bundleCaches[bundleName] = bundle;
207.
208.                     // 注意：GetAllDependencies会返回直接和间接关联的AssetBundle
209.                     string[] depends = manifest.GetAllDependencies(bundleName);
210.                     for (int i = 0; i < depends.Length; i++)
```

```
                    {
                        if (!bundleCaches.ContainsKey(depends[i]))
                        {
                            AssetBundle depAB =
    AssetBundle.LoadFromFile(localAssetPath + depends[i]);
                            bundleCaches[depends[i]] = depAB;
                        }
                    }
                    return bundle;
                }
            }
        }
    }
```

?