代码编写规范

开发中，大量使用 lua，暂时根据当前状况，总结相对而言较好的规范，在多人协作中可以更好的开发、交流。

介绍

该文档旨在为使用lua编写应用程序建立编码指南。

制订编码规范的目的：

- 统一编码标准，通用。提高开发效率；
- 使代码通俗易懂，易于维护。

切记：善用调试器。

目录

一、 命名惯例

1. 所有 lua 文件命名时使用小写字母
2. 类名、变量名等全小写，尽可能使用有意义的英文，单词若生僻，则用 _ 分割
3. 文件内局部变量加 s_前缀
4. 常量、消息号定义时用大写，单词间 _ 分割
5. 枚举值定义时 加前缀 enum_

二、 文件组织

1. 文件开头加上此文件的功能、职责的简要描述； 每个文件都加 module 限定词； 导入的模块都加 local 限定词；
2. 所有函数都加如下格式的注释。

   例如：

```
--此函数检测是否可以从 A(oldx, oldy)点走到 B 点（newx, newy）
--@param oldx 当前所在点 x
--@param oldy 当前所在点 y
--@param newx 目标点 x
--@param newy 目标点 y
--@return 若可以到达，返回 true；否则返回 false
function obj:checkbar(oldx, oldy, newx, newy)
    ...
end
```

3. 函数与函数间、以及一些定义之间加上空行。
4. 文件内不允许出现全局变量，_G.instance 例外
5. 函数内的临时变量、文件内的局部函数都加上 local 限定词
6. 常量、消息号、枚举值行末都加上分号。
7. 函数的行数过长（大于 100 行）时，尽量拆分为多个子函数；函数中一些晦涩的部分，一定要加上注释。
8. 短小的注释使用 --； 较长的注释使用 --[[ ]]
9. assert 函数开销不小，请慎用。
10. Lua 类设计时，用元表来实现 oop。

    不要直接增加函数成员，因为直接增加函数成员会导致内存增加并且在 jit 下执行效率和用元表方式无差异。

```
--一个典型的类定义如下：
--baseobj.lua
module(..., package.seeall)
--类方法定义
local s_method = {__index = {}}
local function init_method(obj)
 function obj:getname()
     return self.name
 end
 function obj:setname(name)
     self.name = name
 end
end
init_method(s_method.__index)

--创建一个对象
function createobj()
 --类数据定义
 local obj =
 {
```

```
    name = "testname"
}
--设置元表
obj = setmetatable(obj, s_method)
return obj
end
```

一个典型的两层继承的例子如下：

```
--objbase.lua
--基类定义
module(..., package.seeall)
--扩展方法元表
function expandmethod(obj)
 function obj:getname()
     return self.name
 end
 function obj:setname(name)
     self.name = name
 end
end
```

```
--创建一个基类对象
function createobj()
 --类数据定义
 local obj =
 {
     name = "testname"
 }
 return obj
end
--子类定义
--obj.lua
module(..., package.seeall)
local base = require "objbase"


--类方法定义
local s_method = {__index = {}}
local function init_method(obj)
 function obj:setid(id)
     self.id = id
 end
 function obj:getid()
```

```
    return self.id
end

--设置基类方法
base.expandmethod(obj)
end
init_method(s_method.__index)

--创建一个子类，此子类为最终子类
function createchar()
 local obj = base.createobj()

 obj.id = 0

 obj = setmetatable(obj, s_method)
 return obj
end

--test.lua

local obj = require "obj"
```

```
s = obj.createchar()

print(s:getid(), s:getname())

s:setid(100)

s:setname('65')

print(s:getid(), s:getname())
```

请注意在 lua 中，表的应用范围、以及相关特性的实现。

三、 分隔和缩进

1. 使用空行

在下述情况下使用单行的空白行来分隔：

1）在方法之间

2）在方法内部代码的逻辑段落小节之间

3）在注释行之前

注释之前增加一行或者多行空行。

2. 使用空格符

除正常的成分之间以空格符分隔名（如数据类型和变量名之间），在下述情况下也应使用一个空格符来分隔：

1）运算符和运算符之间，如： c = a + b;

2）在参数列表中的逗号后面，如：

```
function ml(int year, int month)

end
```

3) 在 for 语句时，如：

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

end
```

4)在下列情况下不要使用空格。

例如：

函数定义时：

```
function test1(a)
end
```

不要这样：

```
function test1( a )
end
```

函数调用时：

```
test1(3)
```

不要这样：

```
test1( 3 )
```

不要如此的原因在于：

a).容易忘记相关空格，导致风格不统一，这样还不如不加；

b).lua 解析语法时是采用空格等分割来解析的，某些情况下，若不小心加空格会导致非预期的结果。

3. 使用换行符

不建议在一行中写多条语句，一条语句的长度一般超过了 80 个字符时，应该换行

4. 使用小括号

可以使用小括号来强行规定运算顺序

5. 使用缩进

在下述情况下应用缩进

1）类中的成分

2）方法体或语句块中的成分

3）换行时的非起始行

缩减量一般为在上一级成分的基础上跑到下一个制表位

四、 代码建议：

1. 代码中使用的一些函数尽可能在文件开头或者当前局部环境中加 local 前缀重新定义下。
   例如：
   ```
   local assert = assert
   ```
2. 不要使用元表来实现继承
3. 高级特性尽可能不用
4. 写代码时尽可能写的简单，考虑性能时先做好推断，看看能提升多少，增加的复杂度以及造成的代码晦涩有多严重，然后再决定如何做
5. 加载的 xml 数据表，尽可能的做好数据校验，若校验失败，要出发断言，使服务器无法启动；不要等出错时，回过头来检查是数据表问题还是逻辑问题。
6. 出错时，记录好错误日志。
   有的函数开销比较大，而调用的频率很低，那么可以不对他做优化；
   反之，有的函数开销较小，但是调用的频率很高，从如何降低调用频率以及减少函数开销两个角度去思考，然后定下优化方案
7. 提交代码之前，去掉或者注释掉无关的代码； 测试下保证服务器可以正确启动。
8. 尽量减少表中的成员是另一个表的引用。 考虑 lua 的垃圾收集机制、内存泄露等。