多益网络lua编程规范 （版本 0.5）

==========

lua风格规范

排版

空格

程序块间要采用缩进风格编写，缩进空格数为Tab。所有运算符（..除外）两边要有空格。如：a = b，特列：.. 和...相邻时需要空格

空行

相对独立的程序块之间，如函数块之后必须加空行。如下：两个函数之间必须要加空行。

```
local function Foo(p)
    print(p)
end

local function Bar(p)
    print(p)
end
```

代码行

较长的语句要分成多行书写，每行代码不超过80字符，一行代码最好只做一件事情，比如只写一个语句，或只定义一个变量，这样的代码容易阅读，方便注释。 if…then…return…end要分行写，if、for等语句自占一行，执行语句不得紧跟其后。特例：当匿名函数作为参数时可以根据代码上下文酌情考虑是否分行

```
local function Func(f)
end

Func(function() if 1 then print(1) end  end)
```

比较规范

```
NIL     a == nil
INTEGER a == 0
STRING  a == "lua"
BOOL    if a
BOOL    if not a
```

杜绝比较不同类型的对象

注释

注释符号

单行注释：--
多行注释：--[[--]]

注释区域

注释通常用于以下：
（1）版本、版权声明；
（2）函数接口说明；
（3）重要的代码行或段落提示。
注释的位置应与被描述的代码相邻，可以放在代码的上方或右方，不可放在下方。
注释的原则是有助于对程序的阅读理解，注释也不宜太多。注释可以是中文或英文，但最好用英文，防止产生乱码问题。

全局变量注释

代码中尽量勿使用全局变量，如有使用全局变量要有较详细的注释，包括对其功能、取值范围、哪些函数存取它以及存取它时的注意事项等的说明。

TODO注释

为临时代码使用TODO注释，它是一种短期解决方案．不算完美，但够好了．

TODO注释应该在所有开头处包含”TODO”字符串，紧跟着是用括号括起来的你的名字，email地址或其它标识符．然后是一个可选的冒号．接着必须有一行注释，解释要做什么．主要目的是为了有一个统一的TODO格式，这样添加注释的人就可以搜索到(并可以按需提供更多细节)．
写了TODO注释并不保证写的人会亲自解决问题．当你写了一个TODO，请注上你的名字．

```
# TODO(duoyi@henhaoji.com): Use a "*" here for string repetition.
# TODO(duoyi@henhaoji.com) Change this to use relations.
```

命名规范

(1) 变量：驼峰命名法，eg：playerName变量的命名要见名知意，便于阅读和修改。
(2) 常量：大写加下划线 eg：MAX_NAME_LEN
(3) 函数：pascal命名法，首字母大写，eg：function CheckPlayerName()
(4) 文件：所有lua文件命名时使用小写字母，如playerskill.lua
(5) 类名：pascal命名法，首字母大写，PlayerSkill

其他规范

(1) table的数据较多时考虑用如下形式增强可读性：

```
local a = {
    [1] = 1,
    [2] = 2,
    [3] = 3,
}
```

lua语言规范

面向对象

使用自定义的class函数来实现面向对象编程，class支持单继承

变量

(1) 尽量使用local变量而非global变量，原则上不允许出现global变量
(2) 被多次引用的global变量，应提取出来放到local变量中
(3) 被多次引用的其他模块中的变量，应提取出来放到local变量中,以下两种写法是有性能差异的

```
for i in 1, 10000000 do
    math.sin(i)
end

local sin = math.sin()
for i in 1, 10000000 do
    sin(i)
end
```

(4) 从其他模块导入的变量，应提取出来放到local变量中，该local变量放在源文件开头部分，用local变量引用标准库，
用和local和require引用自定义库且引用的名字要和类定义中的名字一致。例

```
local table  = table
local myclass = require "myclass"
locla myfuncs = require "myfuncs"
```

(5) 同一文件中前面的函数需要调用后面函数的，由于是local函数，必须先定义变量

```
local Func

local function Caller()
    Func()
end

function Func()
end
```

模块与接口

模块（文件）的对外接口严格定义为通过return返回，该返回值大多数情况下应该是一个table，也可为某个函数或者变量

```
local module = {}

function module.Foo()
end

function module.Bar()
end

return module
```

metatable

逻辑层禁止使用metatable

coroutine

逻辑层禁止使用coroutine

关于表

表在Lua中使用十分频繁，因为表几乎代替了Lua的所有容器。所以快速了解一下Lua底层是如何实现表，对我们编写Lua代码是有好处的。
Lua的表分为两个部分：数组(array)部分和哈希(hash)部分。数组部分包含所有从1到n的整数键，其他的所有键都储存在哈希部分中。
哈希部分其实就是一个哈希表，哈希表本质是一个数组，它利用哈希算法将键转化为数组下标，若下标有冲突(即同一个下标对应了两个不同的键)，则它会将冲突的下标上创建一个链表，将不同的键串在这个链表上，这种解决冲突的方法叫做：链地址法。
当我们把一个新键值赋给表时，若数组和哈希表已经满了，则会触发一个再哈希(rehash)。再哈希的代价是高昂的。首先会在内存中分配一个新的长度的数组，然后将所有记录再全部哈希一遍，将原来的记录转移到新数组中。新哈希表的长度是最接近于所有元素数目的2的乘方。
当创建一个空表时，数组和哈希部分的长度都将初始化为0，即不会为它们初始化任何数组。让我们来看下执行下面这段代码时在Lua中发生了什么：

```
local a = {}
for i = 1, 3 do
    a[i] = true
end
```

最开始，Lua创建了一个空表a，在第一次迭代中，a[1] = true触发了一次rehash，Lua将数组部分的长度设置为2^0，即1，哈希部分仍为空。在第二次迭代中，a[2] = true再次触发了rehash，将数组部分长度设为2^1，即2。最后一次迭代，又触发了一次rehash，将数组部分长度设为2^2，即4。
下面这段代码：

```
local a = {}
a.x = 1
a.y = 2
a.z = 3
```

与上一段代码类似，只是其触发了三次表中哈希部分的rehash而已。
只有三个元素的表，会执行三次rehash；然而有一百万个元素的表仅仅只会执行20次rehash而已，因为2^20 = 1048576 > 1000000。但是，如果你创建了非常多的长度很小的表（比如坐标点：point = {x=0,y=0}），这可能会造成巨大的影响。

如果你有很多非常多的很小的表需要创建时，你可以将其预先填充以避免rehash。比如：
{true,true,true}，Lua知道这个表有三个元素，所以Lua直接创建了三个元素长度的数组。类似的，{x=1，y=2， z=3}，Lua会在其哈希部分中创建长度为4的数组。
以下代码执行时间为1.53秒：

```
local a = os.clock()
for i = 1, 2000000 do
    local a = {}
    a[1] = 1
    a[2] = 2
    a[3] = 3
end
local b = os.clock()
print(b-a)
```

如果我们在创建表的时候就填充好它的大小，则只需要0.75秒，一倍的效率提升！

```
local a = os.clock()
for i = 1, 2000000 do
    locala = {1, 1, 1}
    a[1] = 1
    a[2] = 2
    a[3] = 3
end
local b=os.clock()
print(b-a)
```

所以，当需要创建非常多的小size的表时，应预先填充好表的大小。

关于字符串

所有的字符串在Lua中都只储存一份拷贝。当新字符串出现时，Lua检查是否有其相同的拷贝，若没有则创建它，否则，指向这个拷贝。这可以使得字符串比较和表索引变得相当的快，因为比较字符串只需要检查引用是否一致即可；但是这也降低了创建字符串时的效率，因为Lua需要去查找比较一遍。在大字符串连接中，我们应避免用..，应用table来模拟buffer，然后concat得到最终字符串。

```
local s = ''
local t = {}
for i = 1, 300000 do
    t[#t+1] = 'a'
end
s=table.concat(t, '')
```

3R原则

3R原则（the rules of 3R）是：减量化（reducing），再利用（reusing）和再循环（recycling）三种原则的简称。
3R原则本是循环经济和环保的原则，但是其同样适用于Lua。

Reducing

有许多办法能够避免创建新对象和节约内存。例如：如果你的程序中使用了太多的表，你可以考虑换一种数据结构来表示。
举个例子， 假设你的程序中有多边形这个类型，你用一个表来储存多边形的顶点：

```
    local polyLine = {
        {x = 1.1, y = 2.9},
        {x = 1.1, y = 3.7},
        {x = 4.6, y = 5.2},
    }
```

以上的数据结构十分自然，便于理解。但是每一个顶点都需要一个哈希部分来储存。如果放置在数组部分中，则会减少内存的占用：

```
    locla polyLine = {
        {1.1, 2.9},
        {1.1, 3.7},
        {4.6, 5.2},
    }
```

一百万个顶点时，内存将会由153.3MB减少到107.6MB，但是代价是代码的可读性降低了。
最变态的方法是：

```
    local polyLine = {
        x = {1.1, 1.1, 4.6,...},
        y = {2.9, 3.7, 5.2,...},
}
```

一百万个顶点，内存将只占用32MB，相当于原来的1/5。你需要在性能和代码可读性之间做出取舍。
在循环中，我们更需要注意实例的创建。

```
    for i=1,n do
        local t = {1, 2, 3, 'hi'}
        --执行逻辑，但t不更改
        ...
    end
```

我们应该把在循环中不变的东西放到循环外来创建：

```
    local t = {1, 2, 3, 'hi'}
```

```
for i = 1, n do
    --执行逻辑，但t不更改
    ...
end
```

Reusing

如果无法避免创建新对象，我们需要考虑重用旧对象。
考虑下面这段代码：

```
local t = {}
for i = 1970, 2000 do
    t[i] = os.time({year=i, month=6, day=14})
end
```

在每次循环迭代中，都会创建一个新表`{year = i, month = 6, day = 14}`，但是只有year是变量。
下面这段代码重用了表：

```
local t = {}
local aux = {year=nil, month=6, day=14}
for i = 1970, 2000 do
    aux.year = i
    t[i] = os.time(aux)
end
```

另一种方式的重用，则是在于缓存之前计算的内容，以避免后续的重复计算。后续遇到相同的情况时，则可以直接查表取出。这种方式实际就是动态规划效率高的原因所在，其本质是用空间换时间。

Recycling

Lua自带垃圾回收器，所以我们一般不需要考虑垃圾回收的问题。
了解Lua的垃圾回收能使得我们编程的自由度更大。
Lua的垃圾回收器是一个增量运行的机制。即回收分成许多小步骤（增量的）来进行。
频繁的垃圾回收可能会降低程序的运行效率。
我们可以通过Lua的`collectgarbage`函数来控制垃圾回收器。
`collectgarbage`函数提供了多项功能：停止垃圾回收，重启垃圾回收，强制执行一次回收循环，强制执行一步垃圾回收，获取Lua占用的内存，以及两个影响垃圾回收频率和步幅的参数。
对于批处理的Lua程序来说，停止垃圾回收`collectgarbage("stop")`会提高效率，因为批处理程序在结束时，内存将全部被释放。
对于垃圾回收器的步幅来说，实际上很难一概而论。更快幅度的垃圾回收会消耗更多CPU，但会释放更多内存，从而也降低了CPU的分页时间。只有小心的试验，我们才知道哪种方式更适合。