1.使用local

在代码运行前,Lua会把源码预编译成一种中间码,类似于Java的虚拟机 这种格式然后会通过C的解释器进行解释,整个过程其实就是通过一个while循环,里面有很多的switch...case语句,一个case对应一条指令来解析

Lua 5.0之后,Lua采用了一种类似于寄存器的虚拟机模式 Lua用栈来储存其寄存器 每一个活动的函数,Lua都会其分配一个栈,这个栈用来储存函数里的活动记录 每一个函数的栈都可以储存至多250个寄存器,因为栈的长度是用8个比特表示的 有了这些寄存器,Lua的预编译器能把所有的local变量储存在其中 使得Lua在获取local变量时其效率很高

例: 假设a和b为local变量,a = a + b的预编译会产生一条指令:

```
//a是寄存器0 b是寄存器1
ADD 0 0 1
```

但是若a和b都没有声明为local变量,则预编译会产生如下指令:

```
GETGLOBAL    0 0    ;get a
GETGLOBAL    1 1    ;get b
ADD          0 0 1  ;do add
SETGLOBAL    0 0    ;set a
```

所以在写Lua代码时,应该尽量使用local变量 对比测试:

```
a = os.clock()
for i =1,10000000 do
    local x = main.sin(i)
end
b = os.clock()
print(b - a) //1.113454
把math.sin赋给local变量sin:
a = os.clock()
local sin = math.sin
for i = 1,10000000 do
  local x = sin(i)
end
b = os.clock()
print(b-a) //0.75951
```

直接使用math.sin,耗时1.11秒； 使用local变量sin来保存math.sin,耗时0.76秒 30%的效率提升

2.表table

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,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

```
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

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

上述两段代码类似,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； 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 但是如果创建了非常多的长度很小的表(比如坐标点:point = {x=0,y=0}),这会造成巨大的影响

如果有很多非常多的很小的表需要创建时,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

```
a = os.clock()
for i = 1,2000000 do
    local a = {}
    a[1] = 1; a[2] = 2; a[3] = 3
end
b = os.clock()
print(b-a)  --1.528293
```

如果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

```
a = os.clock()
for i = 1,2000000 do
    local a = {1,1,1}
    a[1] = 1; a[2] = 2; a[3] = 3
end
b = os.clock()
print(b-a)  --0.746453
```

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

3.关于字符串

与其他主流脚本语言不同的是,Lua在实现字符串类型有两方面不同

第一,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

第二,所有的字符串变量,只保存字符串引用,而不保存它的buffer 这使得字符串的赋值变得十分高效,若进行长字符串的复制时,复制整个buffer将会非常消耗性能,这个操作的代价十分昂贵 而在Lua,同样的赋值,只复制引用,十分的高效

但是只保存引用会降低在字符串连接时的速度,需要获取整个字符串的拷贝,然后将新增内容添加到拷贝的末尾,而复制buffer的赋值处理在拼接字符串时,只需要将新增内容插入buffer的末尾即可

在Lua中,并不支持第二种更快的操作 以下代码将花费6.65秒:

```
a = os.clock()
local s = ''
for i = 1,300000 do
    s = s..'a'
end
b = os.clock()
print(b-a)  --6.649481
可以用table来模拟buffer,下面的代码只需花费0.72秒,9倍效率提升
a = os.clock()
local s = ''
local t = {}
for i = 1,300000 do
    t[#t + 1] = 'a'
end
s = table.concat( t, '')
b = os.clock()
print(b-a)  --0.07178
```

在大字符串连接中,应避免.. 使用table来模拟buffer,然后concat得到最终字符串

4.3R原则

3R原则(the rules of 3R)是:

- 减量化(reducing)
- 再利用(reusing)
- 再循环(recycling)

Reducing

有许多办法能够避免创建新对象和节约内存 例如:如果程序中使用了太多的表,需要考虑换一种数据结构来表示

假设有多边形这个类型,用一个表来储存多边形的顶点:

```
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:

```
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

更节省内存的方法

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

但是代码可读性大大降低,虽然性能较之前好了一些

一百万个顶点,内存将只占用32MB,相当于原来的1/5 需要在性能和代码可读性之间做出取舍

在循环中,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
```

应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,需要考虑重用旧对象 考虑下面这段代码:

```lua
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

```lua
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 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,不能一概而论 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 只有通过不断测试,才知道哪种方式更适合