一、前言

Lua 是一门以其性能著称的脚本语言，被广泛应用在很多方面，尤其是游戏。像《魔兽世界》的插件，手机游戏《大掌门》《神曲》《迷失之地》等都是用 Lua 来写的逻辑。

所以大部分时候我们不需要去考虑性能问题。Knuth 有句名言：“过早优化是万恶之源”。其意思就是过早优化是不必要的，会浪费大量时间，而且容易导致代码混乱。

所以一个好的程序员在考虑优化性能前必须问自己两个问题：“我的程序真的需要优化吗？”。如果答案为是，那么再问自己：“优化哪个部分？”。

我们不能靠臆想和凭空猜测来决定优化哪个部分，代码的运行效率必须是可测量的。我们需要借助于分析器来测定性能的瓶颈，然后着手优化。优化后，我们仍然要借助于分析器来测量所做的优化是否真的有效。

我认为最好的方式是在首次编写的时候按照最佳实践去写出高性能的代码，而不是编写了一堆垃圾代码后，再考虑优化。相信工作后大家都会对事后的优化的繁琐都深有体会。

一旦你决定编写高性能的 Lua 代码，下文将会指出在 Lua 中哪些代码是可以优化的，哪些代码会是运行缓慢的，然后怎么去优化它们。

二、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。

自 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。

有了这么多的寄存器，Lua 的预编译器能把所有的 local 变量储存在其中。这就使得 Lua 在获取 local 变量时其效率十分的高。

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

```
1    ;a 是寄存器 0 b 是寄存器 1
2    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：

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

所以你懂的：在写 Lua 代码时，你应该尽量使用 local 变量。

以下是几个对比测试，你可以复制代码到你的编辑器中，进行测试。

```
1    a = os.clock()
2    for i = 1,10000000 do
3        local x = math.sin(i)
4    end
5    b = os.clock()
6    print(b-a) -- 1.113454
```

把 math.sin 赋给 local 变量 sin：

```
1    a = os.clock()
```

```
2    local sin = math.sin
3    for i = 1,10000000 do
4        local x = sin(i)
5    end
6    b = os.clock()
7    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。可以获得 30%的效率提升！

三、关于表(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：

```
1    local a = {}
2    for i=1,3 do
3            a[i] = true
4    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：

```
1    a = {}
2    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：

```
1    a = os.clock()
```

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

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！

```
1    a = os.clock()
2    for i = 1,2000000 do
3            local a = {1,1,1}
4            a[1] = 1; a[2] = 2; a[3] = 3
5    end
6    b = os.clock()
7    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。

四、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。例如在 Perl 中，$x = $y，会将$y 的 buffer 整个的复制到$x 的 buffer 中，当字符串很长时，这个操作的代价将十分昂贵。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。在 Perl 中，$s = $s . 'x'和$s .= 'x' 的效率差距惊人。前者，将会获取整个$s 的拷贝，并将’x’添加到它的末尾；而后者，将直接将’x’插入到$x 的 buffer 末尾。

由于后者不需要进行拷贝，所以其效率和$s 的长度无关，因为十分高效。

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：

```
1    a = os.clock()
2    local s = ''
3    for i = 1,300000 do
4            s = s .. 'a'
5    end
6    b = os.clock()
7    print(b-a)    --6.649481
```

我们可以用 table 来模拟 buffer，下面的代码只需花费 0.72 秒，9 倍多的效率提升：

```
1    a = os.clock()
2    local s = ''
3    local t = {}
4    for i = 1,300000 do
5            t[#t + 1] = 'a'
6    end
7    s = table.concat( t, '')
8    b = os.clock()
9    print(b-a)    --0.07178
```

所以：在大字符串连接中，我们应避免...。应用 table 来模拟 buffer，然后 concat 得到最终字符串。

五、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。

六、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。

举个栗子。 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：

```
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 },
        ...
```

```
6    }
```

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：

```
1    polyline = {
2            x = {1.1, 1.1, 4.6, ...},
3            y = {2.9, 3.7, 5.2, ...}
4    }
```

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。

```
1    for i=1,n do
2            local t = {1,2,3,'hi'}
3            --执行逻辑，但 t 不更改
4            ...
5    end
```

我们应该把在循环中不变的东西放到循环外来创建：

```
1    local t = {1,2,3,'hi'}
2    for i=1,n do
3            --执行逻辑，但 t 不更改
4            ...
5    end
```

七、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：

```
1    local t = {}
2    for i = 1970, 2000 do
3            t[i] = os.time({year = i, month = 6, day = 14})
4    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：

```
1    local t = {}
2    local aux = {year = nil, month = 6, day = 14}
3    for i = 1970, 2000 do
4            aux.year = i;
5            t[i] = os.time(aux)
6    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。

八、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。

九、结语

我们应该在写代码时，按照高标准去写，尽量避免在事后进行优化。

如果真的有性能问题，我们需要用工具量化效率，找到瓶颈，然后针对其优化。当然优化过后需要再次测量，查看是否优化成功。

在优化中，我们会面临很多选择：代码可读性和运行效率，CPU 换内存，内存换 CPU 等等。需要根据实际情况进行不断试验，来找到最终的平衡点。

最后，有两个终极武器：

第一、使用 LuaJIT，LuaJIT 可以使你在不修改代码的情况下获得平均约 5 倍的加速。查看 LuaJIT 在 x86/x64 下的性能提升比。

第二、将瓶颈部分用 C/C++来写。因为 Lua 和 C 的天生近亲关系，使得 Lua 和 C 可以混合编程。但是 C 和 Lua 之间的通讯会抵消掉一部分 C 带来的优势。

注意：这两者并不是兼容的，你用 C 改写的 Lua 代码越多，LuaJIT 所带来的优化幅度就越小。

十、声明

这篇文章是基于 Lua 语言的创造者 Roberto Ierusalimschy 在 Lua Programming Gems 中的 Lua Performance Tips 翻译改写而来。本文没有直译，做了许多删节，可以视为一份笔记。

感谢 Roberto 在 Lua 上的辛勤劳动和付出！