lua 的 table 表处理 及注意事项

lua table 分为数组和哈希两个部分。字 key 一般放在数组段中，没有初始化过的 key 值全部设置为 nil 。当数字 key 过于离散的时候，部分较大的数字 key 会被移到 hash 段中去。这个分割线是以数组段的利用率不低于 50% 为准。0 和 负数做 key 时是肯定放在 hash 段中的。

string 和 number 采用 hash，hash 段采用**闭散列方法**，即，所有值存放于 table 中。

lua table 长度问题：

使用 ipairs 方式循环或者递归计算 table 内元素的个数。不要在 lua 的他了中使用 nil 值，如果一个元素删除，直接 remove，不要用 nil 去代替。

判断 lua table 是否为 nil

判断 lua table 是否为 nil 不能用 if a == {} then 【错误的】(这样的结果就是 a == {}永远返回 false，是一个逻辑错误。因为这里比较的是 table a 和一个匿名 table 的内存地址。)；

if table.maxn(a) == 0 then 【错误的】这样做不保险啊，除非 table 的 key 都是数字，而没有 hash 部分。

if #(a) == 0 then 也是不靠谱的，除非你能保证没人这样写这个 table like this：tab = {nil,1,nil;} 用#tab print 出来 的确是 0，能说此 tab 是 nil 的？

可以使用 lua 内置的 next 来判断； if next(a) == 0 then；

字符串的连接操作

警惕临时变量 字符串的连接操作，会产生新的对象。这是由 lua 本身的 string 管理机制导致的。lua 在 VM 内对相同的 string 永远只保留一份唯一 copy ，这样，所有字符串比较就可以简化为地址比较。这也是 lua 的 table 工作很快的原因之一。这种 string 管理的策略，跟 java 等一样，所以跟 java 一样，应该尽量避免在循环内不断的连接字符串，比如 a = a..x 这样。每次运行，都很可能会生成一份新的 copy 。

每次构造一份 table 都会多一份 table 的 copy 。比如在 lua 里，把平面坐标封装成｛x, y｝用于参数传递，就需要考虑这个问题。每次你想构造一个坐标对象传递给一个函数，｛10,20｝这样明确的写出，都会构造一个新的 table 出来。要么，我们想办法考虑 table 的重用；要么，干脆用 x,y 两个参数传递坐标。 同样需要注意的是以 function foo (...) 这种方式定义函数， ... 这种不定参数，每次调用的时候都会被定义出一个 table 存放不定数量的参数。 这些临时构造的对象往往要到 gc 的时候才被回收，过于频繁的 gc 有时候正是效率瓶颈。