table是Lua中唯一的数据结构，其他语言所提供的数据结构，如：arrays、records、lists、queues、sets等，Lua都是通过table来实现，并且在lua中table很好的实现了这些数据结构。

在传统的C语言或者Pascal语言中我们经常使用arrays和lists（record+pointer）来实现大部分的数据结构，在Lua中不仅可以用table完成同样的功能，而且table的功能更加强大。通过使用table很多算法的实现都简化了，比如你在lua中很少需要自己去实现一个搜索算法，因为table本身就提供了这样的功能。

我们需要花一些时间去学习如何有效的使用table，下面通过一些例子，我们来看看如果通过table来实现一些常用的数据结构。首先，我们从arrays和lists开始，因为两者是其他数据结构的基础，大家也比较熟悉。前面章节，我们已接触了table的一些内容，本章，我们将彻底了解它。

11.1 数组

在lua中通过整数下标访问table中元素，即是数组。并且数组大小不固定，可动态增长。

通常我们初始化数组时，就间接地定义了数组的大小，例如：

a = {} -- new array

for i=1, 1000 do

a = 0

end

数组a的大小为1000，访问1-1000范围外的值，将返回nil。数组下标可以根据需要，从任意值开始，比如：

-- creates an array with indices from -5 to 5

a = {}

for i=-5, 5 do

a = 0

end

然而习惯上，Lua的下标从1开始。Lua的标准库遵循此惯例，因此你的数组下标必须也是从1开始，才可以使用标准库的函数。

我们可以用构造器在创建数组的同时初始化数组：

squares = {1, 4, 9, 16, 25, 36, 49, 64, 81}

这样的语句中，数组的大小可以任意的大。

11.2 矩阵和多维数组

Lua中有两种表示矩阵的方法，一是“数组的数组”。也就是说，table的每个元素是另一个table。例如，可以使用下面代码创建一个n行m列的矩阵：

mt = {} -- create the matrix

for i=1,N do

mt = {} -- create a new row

for j=1,M do

mt[j] = 0

end

end

由于Lua中table是对象，所以每一行我们必须显式地创建一个table，比起c或pascal，这显得冗余，但另一方面也提供了更多的灵活性，例如可修改前面的例子创建一个三角矩阵：

for j=1,M do

改成

for j=1,i do

这样实现的三角矩阵比起整个矩阵，仅使用一半的内存空间。

表示矩阵的另一方法，是将行和列组合起来。如果索引下标都是整数，通过第一个索引乘于一个常量（列）再加上第二个索引，看下面的例子实现创建n行m列的矩阵：

mt = {} -- create the matrix

for i=1,N do

for j=1,M do

mt[i\*M + j] = 0

end

end

如果索引是字符串，可用一个单字符将两个字符串索引连接起来构成一个单一的索引下标，例如一个矩阵m，索引下标为s和t，假定s和t都不包含冒号，代码为：m[s..':'..t]，如果s或者t包含冒号将导致混淆，比如("a:", "b") 和("a", ":b")，当对这种情况有疑问的时候可以使用控制字符来连接两个索引字符串，比如'\0'。

实际应用中常常使用稀疏矩阵，稀疏矩阵指矩阵的大部分元素都为空或者0的矩阵。例如，我们通过图的邻接矩阵来存储图，也就是说：当m,n两个节点有连接时，矩阵的m,n值为对应的x，否则为nil。如果一个图有10000个节点，平均每个节点大约有5条边，为了存储这个图需要一个行列分别为10000的矩阵，总计10000\*10000个元素，实际上大约只有50000个元素非空（每行有五列非空，与每个节点有五条边对应）。很多数据结构的书上讨论采用何种方式才能节省空间，但是在Lua中你不需要这些技术，因为用table实现的数据本身天生的就具有稀疏的特性。如果用我们上面说的第一种多维数组来表示，需要10000个table，每个table大约需要五个元素（table）；如果用第二种表示方法来表示，只需要一张大约50000个元素的表，不管用那种方式，你只需要存储那些非nil的元素。

11.3 链表

Lua中用tables很容易实现链表，每一个节点是一个table，指针是这个表的一个域（field），并且指向另一个节点（table）。例如，要实现一个只有两个域：值和指针的基本链表，代码如下：

根节点：

list = nil

在链表开头插入一个值为v的节点：

list = {next = list, value = v}

要遍历这个链表只需要：

local l = list

while l do

print(l.value)

l = l.next

end

其他类型的链表，像双向链表和循环链表类似的也是很容易实现的。然后在Lua中在很少情况下才需要这些数据结构，因为通常情况下有更简单的方式来替换链表。比如，我们可以用一个非常大的数组来表示栈，其中一个域n指向栈顶。

11.4 队列和双向队列

虽然可以使用Lua的table库提供的insert和remove操作来实现队列，但这种方式实现的队列针对大数据量时效率太低，有效的方式是使用两个索引下标，一个表示第一个元素，另一个表示最后一个元素。

function ListNew ()

return {first = 0, last = -1}

end

为了避免污染全局命名空间，我们重写上面的代码，将其放在一个名为list的table中：

List = {}

function List.new ()

return {first = 0, last = -1}

end

下面，我们可以在常量时间内，完成在队列的两端进行插入和删除操作了。

function List.pushleft (list, value)

local first = list.first - 1

list.first = first

list[first] = value

end

function List.pushright (list, value)

local last = list.last + 1

list.last = last

list[last] = value

end

function List.popleft (list)

local first = list.first

if first > list.last then error("list is empty") end

local value = list[first]

list[first] = nil -- to allow garbage collection

list.first = first + 1

return value

end

function List.popright (list)

local last = list.last

if list.first > last then error("list is empty") end

local value = list[last]

list[last] = nil -- to allow garbage collection

list.last = last - 1

return value

end

对严格意义上的队列来讲，我们只能调用pushright和popleft，这样以来，first和last的索引值都随之增加，幸运的是我们使用的是Lua的table实现的，你可以访问数组的元素，通过使用下标从1到20，也可以16,777,216 到 16,777,236。另外，Lua使用双精度表示数字，假定你每秒钟执行100万次插入操作，在数值溢出以前你的程序可以运行200年。

11.5 集合和包

假定你想列出在一段源代码中出现的所有标示符，某种程度上，你需要过滤掉那些语言本身的保留字。一些C程序员喜欢用一个字符串数组来表示，将所有的保留字放在数组中，对每一个标示符到这个数组中查找看是否为保留字，有时候为了提高查询效率，对数组存储的时候使用二分查找或者hash算法。

Lua中表示这个集合有一个简单有效的方法，将所有集合中的元素作为下标存放在一个table里，下面不需要查找table，只需要测试看对于给定的元素，表的对应下标的元素值是否为nil。比如：

reserved = {

["while"] = true, ["end"] = true,

["function"] = true, ["local"] = true,

}

for w in allwords() do

if reserved[w] then

-- `w' is a reserved word

...

还可以使用辅助函数更加清晰的构造集合：

function Set (list)

local set = {}

for \_, l in ipairs(list) do set[l] = true end

return set

end

reserved = Set{"while", "end", "function", "local", }

11.6 字符串缓冲

假定你要拼接很多个小的字符串为一个大的字符串，比如，从一个文件中逐行读入字符串。你可能写出下面这样的代码：

-- WARNING: bad code ahead!!

local buff = ""

for line in io.lines() do

buff = buff .. line .. "\n"

end

尽管这段代码看上去很正常，但在Lua中他的效率极低，在处理大文件的时候，你会明显看到很慢，例如，需要花大概1分钟读取350KB的文件。（这就是为什么Lua专门提供了io.read(\*all)选项，她读取同样的文件只需要0.02s）

为什么这样呢？Lua使用真正的垃圾收集算法，但他发现程序使用太多的内存他就会遍历他所有的数据结构去释放垃圾数据，一般情况下，这个算法有很好的性能（Lua的快并非偶然的），但是上面那段代码loop使得算法的效率极其低下。

为了理解现象的本质，假定我们身在loop中间，buff已经是一个50KB的字符串，每一行的大小为20bytes，当Lua执行buff..line.."\n"时，她创建了一个新的字符串大小为50,020 bytes，并且从buff中将50KB的字符串拷贝到新串中。也就是说，对于每一行，都要移动50KB的内存，并且越来越多。读取100行的时候（仅仅2KB），Lua已经移动了5MB的内存，使情况变遭的是下面的赋值语句：

buff = buff .. line .. "\n"

老的字符串变成了垃圾数据，两轮循环之后，将有两个老串包含超过100KB的垃圾数据。这个时候Lua会做出正确的决定，进行他的垃圾收集并释放100KB的内存。问题在于每两次循环Lua就要进行一次垃圾收集，读取整个文件需要进行200次垃圾收集。并且它的内存使用是整个文件大小的三倍。

这个问题并不是Lua特有的：其它的采用垃圾收集算法的并且字符串不可变的语言也都存在这个问题。Java是最著名的例子，Java专门提供StringBuffer来改善这种情况。

在继续进行之前，我们应该做个注释的是，在一般情况下，这个问题并不存在。对于小字符串，上面的那个循环没有任何问题。为了读取整个文件我们可以使用io.read(\*all)，可以很快的将这个文件读入内存。但是在某些时候，没有解决问题的简单的办法，所以下面我们将介绍更加高效的算法来解决这个问题。

我们最初的算法通过将循环每一行的字符串连接到老串上来解决问题，新的算法避免如此：它连接两个小串成为一个稍微大的串，然后连接稍微大的串成更大的串。。。算法的核心是：用一个栈，在栈的底部用来保存已经生成的大的字符串，而小的串从栈定入栈。栈的状态变化和经典的汉诺塔问题类似：位于栈下面的串肯定比上面的长，只要一个较长的串入栈后比它下面的串长，就将两个串合并成一个新的更大的串，新生成的串继续与相邻的串比较如果长于底部的将继续进行合并，循环进行到没有串可以合并或者到达栈底。

function newStack ()

return {""} -- starts with an empty string

end

function addString (stack, s)

table.insert(stack, s) -- push 's' into the the stack

for i=table.getn(stack)-1, 1, -1 do

if string.len(stack) > string.len(stack[i+1]) then

break

end

stack = stack .. table.remove(stack)

end

end

要想获取最终的字符串，我们只需要从上向下一次合并所有的字符串即可。table.concat函数可以将一个列表的所有串合并。

使用这个新的数据结构，我们重写我们的代码：

local s = newStack()

for line in io.lines() do

addString(s, line .. "\n")

end

s = toString(s)

最终的程序读取350KB的文件只需要0.5s，当然调用io.read("\*all")仍然是最快的只需要0.02s。

实际上，我们调用io.read("\*all")的时候，io.read就是使用我们上面的数据结构，只不过是用C实现的，在Lua标准库中，有些其他函数也是用C实现的，比如table.concat，使用table.concat我们可以很容易的将一个table的中的字符串连接起来，因为它使用C实现的，所以即使字符串很大它处理起来速度还是很快的。

Concat接受第二个可选的参数，代表插入的字符串之间的分隔符。通过使用这个参数，我们不需要在每一行之后插入一个新行：

local t = {}

for line in io.lines() do

table.insert(t, line)

end

s = table.concat(t, "\n") .. "\n"

io.lines迭代子返回不带换行符的一行，concat在字符串之间插入分隔符，但是最后一字符串之后不会插入分隔符，因此我们需要在最后加上一个分隔符。最后一个连接操作复制了整个字符串，这个时候整个字符串可能是很大的。我们可以使用一点小技巧，插入一个空串：

table.insert(t, "")

s = table.concat(t, "\n")