[quote]

[align=center][b][color=orangered][size=200%]' 免 责 声 明 '[/size][/color][/b][/align]

======

&#8195;&#8195;[b][size=150%]该[/size]教材的出版时间是2009年，由于游戏与代码的迭代特性，本书的内容可能已经与现在你实际接触到的游戏或代码内容有所差异，有些内容可能已经过时，甚至是被淘汰“过期的”，请大家自己斟酌参考。若出现由于原文与本帖翻译有所不同而导致的错误或损失，概不负责。[/b]

[/quote]

[quote]

[align=center][b][color=orangered][size=200%]' 扇 贝 碎 碎 念 '[/size][/color][/b][/align]

======

[b][size=150%]这[/size][/b]是国外教材《Beginning Lua with World of Warcraft Addons》的第十三章节。我已经出了我到目前为止所翻译的几个章节的合集，大家可以自行浏览[tid=24502396][color=orangered][合集]Lua与魔兽世界插件——《Beginning Lua with World of Warcraft Addons》[/color][/tid]

这章节可以说是相当难的一个章节了，如果没有较好的Lua基础可能后面的一些内容会读起来稍显吃力。这章节的优化是针对第六章中的一些内容进行的，虽然不是全部，但还是需要了解一些基本知识。特别是在哈希表和协程部分，我翻译起来稍显别扭。大家阅读的时候多关注作者想让你知道的东西，不要太在意具体的句子和字词。

[size=110%][b]如果有人在阅读过程中，如果发现哪里翻译错误，或错别字需要更正，请发评论或发私信告知，谢谢大家。[/b][/size]

祝大家看得开心。

[/quote]

[quote]

[align=center][b][color=orangered][size=200%]' 目&#8195;&#8195;录 '[/size][/color][/b][/align]

======

[size=180%][b]&#9632; 第十三章 窍门、技巧和优化[/b][/size]

[size=130%][b]&#9679; 测量性能[/b][/size]

&#9675; [b]在Lua中测量性能[/b]

· 测量CPU时间

· 测量内存占用

&#9675; [b]分析《魔兽世界》中的插件[/b]

· 《魔兽世界》中的CPU分析

· 《魔兽世界》中的内存分析

&#9675; [b]一些约定[/b]

[size=130%][b]&#9679; 优化的主要规则[/b][/size]

[size=130%][b]&#9679; 字符串[/b][/size]

&#9675; [b]了解字符串[/b]

&#9675; [b]例子：连接插件中的信息[/b]

· 一个简单的事件处理程序

· 《魔兽世界》中的内存分析

· 出现问题后的内存占用

· 解决方案

&#9675; [b]优化string.format[/b]

[size=130%][b]&#9679; 局部变量[/b][/size]

&#9675; [b]全局变量vs局部变量[/b]

&#9675; [b]这意味着什么？[/b]

[size=130%][b]&#9679; 表[/b][/size]

&#9675; [b]了解数组[/b]

· 数组在内存中是什么样子

· 数组大小

&#9675; [b]了解哈希表[/b]

· 哈希表基础

· 创建一个哈希表

· 测试我们的哈希表

&#9675; [b]表优化基础[/b]

[size=130%][b]&#9679; 优化SimpleTimingLib[/b][/size]

&#9675; [b]使用另一种数据结构[/b]

&#9675; [b]构建SimpleTimingLib-1.1[/b]

&#9675; [b]再利用表？[/b]

[size=130%][b]&#9679; 利用Userdata[/b][/size]

&#9675; [b]使用Userdata作为委托[/b]

&#9675; [b]Userdata元方法[/b]

[size=130%][b]&#9679; 协程库[/b][/size]

&#9675; [b]协程基础[/b]

&#9675; [b]协程示例[/b]

[size=130%][b]&#9679; 总结[/b][/size]

[/quote]

======

[align=center][color=orangered][b][size=200%]&#9632; 第十三章 窍门、技巧和优化[/size][/b]

[size=130%][b](Tips, Tricks, and Optimization)[/b][/size][/color][/align]

[b][size=150%]这[/size][/b]是完全专注于Lua语言的第三个章节也是最后一章，并且只使用一些《魔兽世界》的API。我将向你展示一些非常有用的窍门和技巧，你可以在你的插件中使用它们。本章还介绍了协程标准库(coroutine standard library)，它还未被提及是因为它非常复杂，而且很少在插件中使用(但是它仍然非常强大)。

我还将向你展示Lua中的一些内部工作机制，包括字符串和表。如果你想熟练地使用这些元素，那么了解它们的工作原理是非常重要的。每个数据结构都有其优缺点，因此在插件中选择正确的数据结构非常重要。选择正确的数据结构还可以优化代码，使其运行得更快或占用更少的内存。然而，在《魔兽世界》插件中，优化是一个被过度炒作的话题。在阅读插件代码时，你会看到许多完全不必要或错误的优化(这意味着生成的代码比实际上更慢)。

尤其内存占用(memory usage)是一个完全被夸大的话题。许多人声称，内存占用高的插件会降低游戏速度，并导致延迟上升(lag spikes)。他们所说的“内存占用”通常指1M字节或更多，这担心是可笑的，因为你可能有2GB或更多的内存，并且《魔兽世界》本身很少需要超过1G内存。这也为你的操作系统、后台应用程序和插件留下了1G的内存空间。我在第六章讨论lua5.0和lua5.1的垃圾回收器(garbage collectors)时提到了这个话题。《魔兽世界》在燃烧的远征之前使用了Lua5.0，对于游戏这样的应用程序，它是一个非常糟糕的垃圾回收器。这样的垃圾回收器被戏称为“中止魔兽”回收器(“stop the world” collector)。因为它在回收垃圾时停止脚本的执行。但是新的lua5.1垃圾回收器速度非常块，你不应该太担心它。当你在Lua中编写《魔兽世界》插件时，你可能用不到我在这里向你展示的许多优化窍门和技巧。然而，随着你编程技能的成长和编写更大更复杂的插件，你需要对Lua有更深入的了解。

本章节将向你展示何时应该考虑性能、如何测量性能以及可以优化的内容。我还将向你展示许多小技巧，让你可以更加轻松。我们需要讨论的第一件事是衡量插件或任何Lua脚本的性能。

======

[color=blue][b][size=160%]&#9679; 测量性能 [/size]

[size=130%](Measuring Performance)[/size][/b][/color]

Lua只提供了几个用来测量性能的函数，而《魔兽世界》有几个高级的分析函数(profiling functions)。

[b][size=130%]&#9675; 在Lua中测量性能(Measuring Performance in Lua)[/size][/b]

最少有两种资源会被Lua脚本占用：CPU时间和内存。然而缩短CPU周期和节省内存往往是两个相互排斥的目标。许多减少所需CPU时间的优化都是通过使用更多的内存来实现的，例如缓存。你不得不权衡CPU占用和内存占用。但在大多数情况下，你应该选择更少的CPU占用和更多的内存占用，因为内存相对便宜，而且你通常都有足够的内存。

[b]· 测量CPU时间(Measuring CPU Time)[/b]

你可以通过调用os.clock()函数来获取特定Lua脚本使用的CPU时间，但这在《魔兽世界》中是无法使用的。此函数返回当前正在执行的Lua脚本所使用的CPU时间，你可以使用它快速获得在游戏外执行某个脚本所需的时间。下面举个例子：

[code=lua]

for i = 1, 10^8 do end

print(os.clock())

[/code]

运行结果取决于你计算机的速度。

获取执行时间的另一种方法是使用外部程序启动Lua脚本并打印运行它所需的时间。在Linux和Unix中，可以通过使用简单的程序time和调用time lua5.1 myScript.lua来实现这一点。Windows没有这样的程序，但这不是问题，因为os.clock()对我们来说已经足够了。

os.clock()函数的作用是：返回整个脚本的总执行时间。你可以在代码块周围使用以下用法来独立于脚本的其他部分测量其执行时间：

[code=lua]

local t = os.clock()

-- code here

print(os.clock() - t)

[/code]

[quote]

注意：尽管我不会在本章每个示例的末尾都包含print(os.clock())，但你可以把它加到任何你想测量的地方。

[/quote]

[b]· 测量内存占用(Measuring memory usage)[/b]

你不得不使用的另一个重要资源的可用内存。当我们在第六章讨论垃圾回收器时，你已经了解了如何跟踪Lua脚本的内存使用情况。如果option设置为“count”，那么collectgarbage(option)函数将返回Lua当前占用的内存(KB)。

下面的代码通过比较创建表前后的内存使用情况，以字节为单位打印空表的大小。结果的值乘以1024得到以字节为单位的值，这在这里会更易于理解，因为空表相对较小：

[code=lua]

local mem = collectgarbage(“count”)

local x = {}

print((collectgarbage(“count”) - mem) \* 1024) -->32

[/code]

[b][size=130%]&#9675; 分析《魔兽世界》中的插件(Profiling Addons in World of Warcraft)[/size][/b]

《魔兽世界》提供了一些高级功能，可以用来分析你插件的性能。有一些高级功能可用于内存和CPU分析。

[quote]

注意：本章中的大多数例子都可以在不使用《魔兽世界》的情况下通过普通的Lua解释器进行测试。当某个例子由于使用了《魔兽世界》的特定功能而只能在《魔兽世界》中工作时，我会将其标注出来。

[/quote]

[b]· 《魔兽世界》中的CPU分析(CPU Profiling in WoW)[/b]

《魔兽世界》的CPU分析功能非常强大；它们几乎追踪所有东西。它们可以告诉你某个特定函数或给定框体的所有事件处理程序所使用的CPU时间。这些分析特性在默认情况下是禁用的，因为它们会严重影响性能。可以自行通过以下斜杠命令来启用分析：

[quote]

/console set scriptProfile “1”

[/quote]

你必须重新加载UI才能使此更改生效。但不要忘记在完成分析后执行以下命令来禁用此功能：

[quote]

/console set scriptProfile “0”

[/quote]

启用分析后，可以调用以下几个函数来分析代码：

[quote]

ResetCPUUsage()：重置分析统计信息。

UpdateAddOnCPUUsage()：更新由下列所有函数检索的数据。在使用此列表中的函数建所统计信息之前，必须调用此函数。

GetAddOnCPUUsage(addon)：获取addon在最后一次调用UpdateAddOnCPUUsage()之前使用的CPU时间(以毫秒为单位)。参数addon可以将插件的名称表示为字符串或数字。

GetFunctionCPUUsage(func, includeSubs)：获取func使用的CPU时间，以毫秒为单位。参数includeSubs可以设置为false，以忽略func调用的函数所需要的时间；默认值为true，这意味着统计信息中包含所有下层的函数调用。

GetFrameCPUUsage(frame, includeChildren)：返回frame的所有处理程序使用的CPU时间(以毫秒为单位)。参数includeChildren可以设置为false，以忽略框体的子框体的脚本处理程序；默认为true，表示结果中包含子框体的CPU使用情况。

GetEventCPUUsage(event)：获取处理event的所有事件处理程序所使用的CPU时间。

[/quote]

[b]· 魔兽世界》中的内存分析(Memory Profiling in WoW)[/b]

《魔兽世界》中的内存分析函数和CPU分析函数类似。但是你不需要主动去启用它们，因为这些函数不会对性能造成影响。

[quote]

UpdateAddOnMemoryUsage()：更新下面的函数检索的数据。

GetAddOnMemoryUsage(addon)：获取上次调用UpdateAddOnMemoryUsage()时addon(字符串或数字)使用的内存。

[/quote]

[b][size=130%]&#9675; 一些约定(Some Conventions)[/size][/b]

在本章中，你将看到许多简短的示例，它们比较了针对同一问题的不同解决方案的性能。我会经常告诉你我电脑上的脚本运行所需的时间(一台带有2.00GHz的酷睿2双核CPU的笔记本电脑)。之后，你可以很容易地看到哪个脚本更快。但我也会不时地使用百分比值，这可能会引起很多的混乱。

例如，“快100%”是什么意思？假设我们有两个脚本，A和B，前者比后者运行得更快。如果脚本A的运行时间比脚本B少x%，那么脚本A的运行速度比脚本快x%。因此，100%的速度意味着脚本A可以在短时间内运行。

在本章中我将使用的另一个重要约定是大写的O符号。这个符号为我们提供了一种根据输入值的大小来描述算法的资源使用(CPU或内存)情况的方法。我不会给出它的定义，因为它是高度数学化的，但是让我们看看它是如何在没有太多数学理论的情况下工作的。

大写O表示法如下：O(f(n))，其中f(n)替换为所需CPU时间或内存占用增长的函数，n是输入的大小，例如表的大小或字符串的长度。这意味着当你在O(1)函数上使用较大的值时，它不会变慢(需要占用更多内存)。下面的函数就是一个简单的例子：

[code=lua]

function foo(t, key)

return t[key]

end

[/code]

如果使用更大的表t，这个函数不会变慢，因为访问一个键值是O(1)操作(稍后将详细介绍表索引的工作方式)。下面的函数遍历一个表并寻找一个特定的值；在此它是O(n)，因为它随着表t的大小增长(因为它必须遍历整个表)：

[code=lua]

function foo(t, val)

for k, v in pairs(t) do

if v == val then

return k

end

end

end

[/code]

注意，对于某些输入值，O(n)操作不一定比O(1)操作慢。例如，以下毫无意义的函数以O(1)运行，但仍然非常慢。

[code=lua]

function foo(t)

for i =1, 10^9 do

end

return t

end

[/code]

此函数的性能明显较差。但是当你使用一个更大的表t时，它不会变得更糟，这意味着它是O(1)。对于你之前可能使用过的所有输入值，前面的O(n)函数可能更快，但对于真正巨大的表，它比这个O(1)要慢。随着表的增长，它变得越来越慢。这说明大O符号只是告诉我们所需时间增长有多快，而不是实际所需多少时间。

同样有趣的是下面的函数，它接收一个用数字填充并且已经被排序的数组。它检查给定的数字是否在这个数组中，但它比之前的O(n)函数更快，因为它使用所谓的二分查找算法(binary search algorithm)，以O(log n)运行：

[code=lua]

function binSearch(t, val) -- t: sorted (ascending) array

local start, last = 1, #t

while start <= last do

local middle = math.floor(start + (last - start) / 2)

if t[middle] > val then

last =middle - 1

elseif t[middle] < val then

start = start + 1

else -- t[i] == val, found it

return middle

end

end

end

[/code]

这个函数比遍历一个表更加复杂，但速度也更快。你可以使用我前面提到的技巧对其进行测试。binSearch函数首先查看在数组中间的元素，接着继续寻找数组左边部分的值。除非中间的值大于val，否则它会在数组的右侧部分进行寻找，因为我们假设数组已经排序。函数会继续执行此操作，直到搜索的间隔(interval)为空或找到该值为止。

另一个可以帮助你理解大O符号的示例是以下函数。它实现了冒泡排序算法(bubble sort algorithm)，以O(n2)的时间对表进行排序：

[code=lua]

function bubblsSort(t)

for i = 1, #t do

for i2 =1, #t do

if t[i] < t[i2] then

t[i], t[i2] = t[i2], t[i]

end

end

end

end

[/code]

你可能已经猜到O(n log n)位于O(n)和O(n2)之间。这样的函数有一个例子是table.sort(t, compare)。它以O(n log n)对表进行排序，因为它在内部使用快速排序。它实际上不是严格的O(n log n)；在最坏的情况下，它可能以O(n2)运行，但对于大型表来说，这种情况不太可能发生。O(n log n)是快速排序所需的平均时间。还有另一种排序算法可以保证以O(n log n)运行：归并排序(merge sort)。但是对于几乎所有表来说，它仍然比快速排序要慢，因为它有一个步骤比快速排序的开销更大。因此，大多数编程语言都使用快速排序。这里我没有展示快速排序，因为在Lua中你不需要担心对表进行排序；你只需要使用table.sort。

现在你知道了如何测量代码的性能，并且我们已经讨论了必要的约定。但是什么应该被优化，什么不应该呢？

======

[color=blue][b][size=160%]&#9679; 优化的主要规则 [/size]

[size=130%](The Main Rules of Optimization)[/size][/b][/color]

有两个应该始终遵守的基本优化规则：

[quote]

不要这样做(Don't do it)

先别这样做(Don't do it yet)——仅供专家使用(for experts only)

[/quote]

如果不事先考虑就优化代码，结果可能是浪费时间，使代码更难理解，而且可能做错，最终导致性能下降。这意味着你应该经常测量脚本的性能，看它是否需要优化；99%的插件根本不需要优化。

下面的所有优化技巧只适用于对插件性能至关重要的代码。这方面的一个例子是OnUpdate处理程序，它被多个框体使用(就像我们第一个版本的CooldownMonitor中bar对象的OnUpdate处理程序)。另一个例子是COMBAT\_LOG\_EVENT\_UNFILTERED事件，它经常被调用。在某些Boss战中，我测量到每秒高达800个事件的峰值；这意味着这个事件处理程序比OnUpdate处理程序被调用的频率更高。不要浪费时间优化一个偶尔调用的函数。始终测量优化前后的性能，以查看优化是否真的有效。

让我们看看什么东西可以在插件中进行优化。字符串真的是影响性能的一个很好的例子。

======

[color=blue][b][size=160%]&#9679; 字符串 [/size]

[size=130%](Strings)[/size][/b][/color]

字符串似乎没有危害，你想不到例如连接两个字符串的这种简单操作会降低脚本的速度。为了理解之后遇到的问题，你需要了解字符串在内部是如何工作的。

[b][size=130%]&#9675; 了解字符串(Understanding Strings)[/size][/b]

字符串是一种值类型(value type)，这意味着它们是作为值(而不是引用)传递给函数的。之后函数可以修改字符串，而调用者不会注意到它。下面的例子说明了这种过程：

[code=lua]

function test(s)

s = s..“foo”

end

local s = “test”

test(s)

print(s) --> test

[/code]

这会打印test，因为字符串表现得就像它们是作为值传递一样，这意味着test中字符串的修改不会影响原始字符串。但是字符串在内部总是作为引用传递；你只是没有注意到这个实现细节，因为字符串是不可变的。无法更改现有字符串；你只能生成新的字符串。这意味着语句t = t..”foo”创建了一个由t和”foo”串联而成的新字符串，并将其存储在局部变量t中。

每个字符串都被保存在内部的字符串池中(可以将其视为包含所有字符串的巨大哈希表)，而且每个字符串都是唯一的。当你尝试创建一个新的字符串时，会发生以下情况：Lua构建字符串，检查它是否已经在字符串池中，如果它不存在，则插入它。然后在字符串池中获得对该字符串的引用。

到目前为止还没有特别之处；我在第二章提到了这个过程。但这对我们来说意味着什么呢？字符串受到垃圾回收器的影响，这意味着当你删除它的最后一个引用时，未使用的字符串不会立即消失。因此，如果你有一个非常长的字符串(假设5000万个字符长度，或50MB内存使用量)，并附加了一个短字符串(例如，“\n”表示换行)，这将在你的内存中拥有两条这个字符串，直到原始的字符串被回收。

[b][size=130%]&#9675; 例子：连接插件中的信息(Example: Concatenation in Addon Communication)[/size][/b]

让我们从一个例子开始。假设你有一个函数，它接收其他玩家发送的带有字符串的CHAT\_MSG\_ADDON事件。但是你想要接收的消息可能超过聊天消息的255个字符串限制，因此发送方函数会将发送的消息分成多个字符串，而接收方函数将它们连接起来。

[b]· 一个简单的事件处理程序(A Simple Event Handler)[/b]

下面的代码显示了可以在此插件中使用的两个函数：syncHandler和receive。每次插件接收到复合消息的一部分时，都会调用receive函数；前缀“Foo-End”表示消息结束。这个函数连接并用玩家作为键(key)存储所有的局部消息到表received，以防止多个玩家同时向你发送消息时出现问题。之后，当接收到消息的最后一部分，使用完整连接的消息作为参数调用syncHandler。

[code=lua]

local function syncHandler(msg)

-- do stuff here

-- adding print(msg) here is a bad idea as msg will be really long

print(“Received a sync message!”)

end

local received = {}

local function receive(player, prefix, msg)

received[player] = received[player] or “”

received[player] = received[player]..msg

if prefix == “Foo-End” then

return syncHandler(received[player])

end

end

[/code]

你不大可能需要向其它玩家发送大量数据，假设期望接收消息的最大数量是10条，每一条都有245个字符的大小，这给了我们留了10个字符的前缀来识别插件。

[b]· 一个简单的事件处理程序(A Simple Event Handler)[/b]

在函数后面添加以下代码，用测试数据来模拟CHAT\_MEG\_ADDON事件：

[code=lua]

local testMsgs = {}

local player = “TestPlayer”

local msg = string.rep(“x”, 245)

for i = 1, 10 do

table.insert(testMsgs, {player, “Foo”, msg})

end

table.insert(testMsgs, {player, “Foo-End”, msg})

for i, v in ipairs(testMsgs) do

receive(unpack(v))

end

[/code]

这段代码只是创建10条消息并将它们传递给接收方。尝试执行代码，你将会看到它运行得非常快，你可能不认为这是一个问题。但是让我们看下运行代码时会发生什么。

它将第一个消息与第二个消息连接起来，这样你的内存中就有了原始消息(245字节)和连接后的消息(490字节)。然后它再次将这个490字节的消息与245字节的消息连接起来；现在你有三个字符串：原始字符串、连接前两个的字符串和连接前三个的字符串(735字节)。

生成的字符串的总体大小现在已经是1470字节了，而且随着每条消息的增加，情况会越来约糟糕。如果我们连接n条消息，被占用的内存是245(1+2+...+n)=245n(n+1)/2字节。因此，简单地连接10条消息的示例代码已经占用了13MB的内存。一个真正的插件甚至需要更多的内存，因为接收到的同步消息(sync messages)会有所不同。我们的例子为每个测试消息使用相同的内容，因此它不会在每个步骤中生成新的245字节字符串。但与连接相比，这只需要很小的内存。

值得注意的是，垃圾回收器会在连接期间回收未使用的字符串，这样就不会同时分配整个内存。但这期间会减慢脚本的运行速度，特别是因为所需的内存以O(n2)增长。

[b]· 出现问题后的内存占用(Memory Usage If Something Goes Wrong)[/b]

想象一下，如果发生了问题，你收到了许多不带有Foo-End信息的同步消息(synchronization messages)。或者更糟的是，如果你的团队中有一个恶意的玩家一直向你发送虚假消息让你的游戏崩溃，会发生什么？

发送10000条消息可以相对较快地完成。让我们来看看如果有人这么做会发生什么。将语句for i = 1, 10 do改为下面的语句，用来模拟10000条消息：

[code=lua]

for i = 1, 10000 do

[/code]

我们可以用前面的公式计算在连接期间被移动(moved)的内存：245×10,000×(10,000 + 1) / 2 = 12,251,225,000字节，超过12G，现在我的笔记本电脑上运行这个脚本大约需要30秒，因为Lua需要大量内存，而垃圾回收器一直在工作。在执行脚本时，可以在任务管理器中查看lua5.1.exe进程。这可以给你一个Lua如何洗牌内存(shuffle memory)来连接所有这些字符串的印象。

图13-1显示了在我的任务管理器中执行包含10000条消息的脚本时的内存图。你可以清楚地看到垃圾回收器与你的脚本交错运行；它每隔几个执行步骤启动一次，并收集一些被认为是垃圾的对象。

[align=center][img]./mon\_202108/18/-10er0nQ175-bz9lKgT1kSbc-41.jpg[/img]

[size=80%]图13-1 连接10000个字符串时的内存使用情况图[/size][/align]

字符串连接的这个问题不仅局限于Lua；许多语言(最著名的是Java和C#)都有类似的处理字符串的方式，因此也有同样的问题。许多语言的解决方案是StringBuilder对象，它的内部使用更智能的方式连接字符串。Lua没有这样的对象，不过这个问题还是有简单的解决办法的。

[b]· 解决方案(The Solution)[/b]

有一个函数可以用于连接多个字符串而不会二次增加内存使用量：table.concat(tb1, delimiter)。此函数采用一个在其数组部分中只有字符串的表并将它们连接起来(由delimiter分隔，这是可选的)。《魔兽世界》API提供了一个类似的函数，也可以使用：string.join(delimiter, ...)。它适用于多个字符串参数，而不是表，但我们将使用table.concat，因为它对于我们的例子来说更简单。

我们可以修改函数receive来构建一个包含所有接收到的字符串的表；当接收到最后一条消息时，这个表被连接起来：

[code=lua]

local received = {}

local function receive(player, prefix, msg)

received[player] = received[player] or {}

lable.insert(received[player], msg)

if prefix == “Foo-End” then

return syncHandler(table.concat(received[player]))

end

end

[/code]

我们可以用10000甚至100000条消息来测试它；它运行得非常快。表13-1简单地比较了连接(Concatenation)字符串的旧版本与使用table.concat的新receive函数的性能。

[align=center][size=80%]表13-1 Concatenation和table.concat的字符串比较[/size]

[table]

[tr]

[td33]消息的数量[/td]

[td25]以秒为单位的时间间隔(Concatenation)[/td]

[td]以秒为单位的时间间隔(table.concat)[/td]

[/tr]

[tr]

[td]1000[/td]

[td]0.16[/td]

[td]0.01[/td]

[/tr]

[tr]

[td]5000[/td]

[td]5.3[/td]

[td]0.04[/td]

[/tr]

[tr]

[td]10000[/td]

[td]29[/td]

[td]0.06[/td]

[/tr]

[tr]

[td]15000[/td]

[td]80[/td]

[td]0.09[/td]

[/tr]

[tr]

[td]20000[/td]

[td]&#9679;[/td]

[td]0.12[/td]

[/tr]

[tr]

[td]100000[/td]

[td]&#9679;[/td]

[td]0.6[/td]

[/tr]

[/table]

[/align]

“&#9679;”号表示，当我尝试20000条或更多条消息时，字符串连接失败，在我的笔记本电脑(32位操作系统，内存2G)上出现“内存不足”的错误消息。你可以使用更多的内存或64位版本的Lua，但所需的时间仍然增长得非常快。将垃圾回收器设置为更优(more aggressive)会有所帮助，但这并不能解决实际问题。

我们从中学到的是，应该小心字符串连接。大多数情况下，在循环中使用连接操作(concentration operator)是一个特别糟糕的注意；尽可能使用table.concat或string.join。

[b][size=130%]&#9675; 优化string.format(Optimizing string.format)[/size][/b]

这种优化是《魔兽世界》特有的，可以帮助你避免字符串垃圾。假设单位框体中有一个显示单位生命值的FontString对象。现在，你可以注册UNIT\_HEALTH事件并使用如下事件处理程序：

[code=lua]

local healthString = “Health: %d%d”

function myFrame:UNIT\_HEALTH(uId)

if uId == myFrame.unit then

local txt = healthString:format(UnitHealth(uId)), UniHealthMax(uId))

myFrame.fntStrHealth:Set(txt)

end

end

[/code]

现在，每当监视的单位的生命值发生变化时，都会调用该处理程序，这种情况相当频繁。想象你在一个团队框体插件中使用这个处理程序，在一个40人的战场上，例如奥特兰克山谷。如果团队目前处于战斗状态，可能每个框体调用此理程序20到40次。每次调用都会从格式化字符串(the format string)中创建带有20个字符的字符串。

这些字符串发生了什么？它们被传递给SetText方法，Lua不再使用它们，只在框体内部使用。这意味着它们现在是垃圾，迟早会被回收。但它似乎并不多；如果更新30个团队框体，只有大约600字节的垃圾。但每一个经常更新的字体(font)字符串(想想你的战斗日志或计时器上显示的文本)都会生成这些垃圾字符串，这些字符串被传递给一个函数然后被删除。所有这些框体组合在一起可以快速生成大量垃圾，这会让垃圾回收器忙碌并降低游戏速度。

实际上，你已经知道了这个问题的解决方案，就像我在本书前面写CooldownMonitor插件时提到的那样。所有例如FontString这样显示文本的框体都有SetFormattedText方法，它基本都有以下函数：

[code=lua]

function frame:SetFormattedText(str, ...)

return frame:SetText(set:format(...))

end

[/code]

唯一的区别是它在Lua中不调用string.format；它在内部格式化字符串并显示它，而不生成Lua字符串。如果始终如一地使用它，可以节省大量的垃圾。

这个简单的技巧可以安全地用于每次格式化字符串并显示它的时候，这是《魔兽世界》中最简单的优化之一。另一个非常简单的优化是使用局部变量而不是全局变量。

======

[color=blue][b][size=160%]&#9679; 局部变量 [/size]

[size=130%](Local Variables)[/size][/b][/color]

我已经很多次提到，应该更偏向使用局部变量而不是全局变量。但是为什么呢？一个很明显的原因是，如果变量只在特定的代码块中被访问和更改，那么你的代码将更具可读性。此外，全局变量可以在任何地方访问和更改，因此你永远无法确定其他文件或插件也会修改给定的全局变量。

[b][size=130%]&#9675; 全局变量vs局部变量(Global Variables vs. Local Variables)[/size][/b]

此外，你应该更喜欢局部变量而不是全局变量的另一个原因是：访问局部变量的速度大约快30%。我们可以用一个从1到10^8的紧凑循环(tight loop)(用一条指令来衡量这条指令的性能的循环)来测试。下面的代码使用了一个全局变量：

[code=lua]

i = 0

while true do

i = i + 1

if i >= 10^8 then

break

end

end

[/code]

它运行得相当慢；在这个循环中，Lua在我的笔记本上需要13.1秒。回想一下，在Lua中全局变量是如何工作的：所有全局变量都存储在一个巨大的哈希表中，称为全局环境\_G。在本章的后面，你将详细了解表是如何工作的。局部变量是一个“真实(real)”变量，这意味着它的名称只是编译器的标识符，而不是字符串。Lua知道局部变量在内存中的位置，而全局变量需要计算位置。

下面的版本使用了局部变量而不是全局变量，所以它应该更快：

[code=lua]

local i = 0

while true do

i = i + 1

if i >= 10^8 then

break

end

end

[/code]

添加关键字local的简单更改使代码速度提高了约78%(因此只需要2.8秒)。请注意，结果可能会因为略有不同，因为它们取决于许多因素。

另一个有趣的测试是在使用它的函数之外声明所使用的局部变量。这意味着我们要为引用这个局部变量的函数创建一个闭包。这种局部变量在Lua中也称为upvalue。这里有一个例子：

[code=lua]

local i = 0

function test()

while true do

i = i + 1

if i >= 10^8 then

break

end

end

end

test()

[/code]

有人可能会认为局部变量和upvalue之间没有区别。但是有一个巨大的区别；这段代码的运行时间为4.6秒。它仍然比使用全局变量快65%。但是使用一个“真实的(real)”局部变量使比使用一个upvalue要快39%。

[b][size=130%]&#9675; 这意味着什么？(What Does It Mean?)[/size][/b]

访问全局变量仍然非常快。我们在上一节中看到的巨大速度增益在实际中并没有那么大。在之前的例子中，它们之所以这么高是因为这些例子所作的唯一事情就是访问变量。让我们看一个调用函数math.cos和math.sin的例子。

访问这两个函数花费的时间甚至比访问一个简单的全局变量还要长，因为每次调用首先得获取全局变量math，它是一个表。然后Lua需要从这个哈希表中获取字段cos和sin；这样做所需的时间与访问全局变量相当，因为从技术上讲，全局变量也是哈希表中的项。请注意，哈希表的大小并不影响从哈希表中获取条目(entry)所需的时间(我们将在本章的后面看到这是如何工作的)。

[code=lua]

for i = 1, 10^8 do

math.sin(math.cos(5))

end

[/code]

这在我的电脑上运行了31.5秒。让我们将函数存储在局部变量中。这将4次表/全局变量访问减少为2次局部变量访问：

[code=lua]

local sin = math.sin

local cos = math.cos

for i = 1, 10^8 do

sin(cos(5))

end

[/code]

现在代码运行时间为23秒，仅提高了27%。你也可以测试upvalue，但你会发现几乎没有区别。

但这是有区别的，这种优化可以很容易的被应用。在局部变量中保存对经常调用的代码中使用的函数或表的引用是一个很好的办法。你会在很多文件中看到这种优化；它们通常以如下代码开始。这种优化技术通常被称为缓存(caching)，因为局部变量基本上充当全局函数的缓存。

[code=lua]

local pairs, ipairs = pairs, ipairs

local tremove = table.remove

-- and so on with all frequently called functions

[/code]

但有一个重要的问题没有得到解答：为什么访问全局变量这么慢？全局变量被存储在表(\_G)中，所以我们需要了解表是如何工作的，以便了解当我们访问全局变量时发生了什么。

======

[color=blue][b][size=160%]&#9679; 表 [/size]

[size=130%](Tables)[/size][/b][/color]

表分为两部分：数组部分和哈希部分。数组部分相对简单，所以我们先从它开始。

[b][size=130%]&#9675; 了解数组(Understanding Arrays)[/size][/b]

数组是一种非常简单的数据结构，它使用正常整数作为索引；这些索引用于计算条目(entry)在内存中的位置。这意味着我们首先需要看看内存在内部是如何组织的。

[b]· 数组在内存中是什么样子(What Arrays Look Like Internally)[/b]

你的内存被分成许多小单元，每个单元宽8位(1字节)，你只能访问这些单元格(cells)。单元格会被编号，单元格的编号就是它的地址。在32位系统中，内存的大小被限制为2^32字节，或者说4G，因为地址可以使用的最大比特数是32。

幸运的是，在使用Lua编程时，我们不需要担心这个问题，因为Lua已经为我们完成了所有困难的工作。我们不能告诉Lua在给定的地址上独取内存。但是我们可以获得一些Lua对象的内存地址，比如表，因为默认的\_\_toString()方法可以打印它：

[code=lua]

local t = {}

print(t) --> table: 0059AC90

[/code]

这里，0059AC90是我的系统中的基本内存地址；你可能会得到一个不同的内存地址。Lua在内部将关于该表格的各种信息存储在这个内存地址上。其中一项是表的数组部分的内存地址，它可以位于内存中完全不同的部分(同样适用于哈希部分)。从现在开始，我将把这个地址称为数组的基地址(base address)。

有趣的是，当我们现在尝试访问表中的字段(如t[1])时，会发生什么。Lua首先检查索引是否在数组的范围内，因为数组内部有固定大小，即使它们看起来是动态大小的(我们将在本节后面进一步讨论数组的大小)。如果请求的索引不在数组部分的范围内，Lua将检查数组的哈希表部分。

如果它在数组部分，Lua首先从索引中减去1，得到一个从零开始的索引，该索引在内部使用。这个从零开始的索引现在乘以16并添加到数组基地址。计算出的内存地址包含一个表示数组项的对象，该数组的宽度始终为16字节。

下图显示了这样一个数组t在内存中的视图。表头位于tostring(t)返回的内存地址处；这里稍微简化了一下，因为它在内部有32字节宽，并且不仅仅由数组和散列地址组成。在这里讨论存储在表头中的所有内容会走得太远(go too far)，对于理解表是如何工作的并不重要。数组地址保留数组部分中第一个条目t[1]的位置，简短的计算表明这是正确的。

[quote]

arrayAddress+16×(1-1)=arrayAddress+16×0=arrayAddress

[/quote]

t[2]位于arrayAddress+16，t[3]位于arrayAddress+32，以此类推。

[align=center]

[img]./mon\_202108/18/-10er0nQ175-1ulbKcT1kSc5-9k.png[/img][/align]

这解释了为什么访问数组相对较快，并且并且不会随着添加更多的元素而变慢。Lua始终可以计算数组中每个元素的位置。从这种行为中我们可以得出一个明显的结论，即数组的大小必须在内部固定。想象一下，如果它不是固定的，并且你试图在某处插入一个值时，会发生什么。计算后的内存位置不一定是数组的一部分，你可能会覆盖一些重要内容，从而导致脚本崩溃。可惜事实并非如此，因此我们可以看到数组的大小是固定。

[b]· 数组大小(Array Size)[/b]

数组的初始大小取决于使用的表构造函数(table constructor)。简单的用{ }创建一个空数组部分，而{1}创建一个带有一个空位(slot)的数组。你可能已经猜到{1, 2}创建了两个空位，但是你可能没猜到{1, 2, 3}创建了一个包含四个条目(entry)的数组部分，其中一个条目将保存为空。

这是因为一旦数组达到极限(二的幂)，它们的大小总是翻倍。以下代码说明了该行为：

[code=lua]

local t = {} -- 0 entries

t[1] = 1 -- 1 entry

t[2] = 1 -- 2 entries

t[3] = 1 -- 4 entries

t[4] = 1 -- 4 entries

t[5] = 1 -- 8 entries

t[6] = 1 -- 8 entries

-- and so on...

[/code]

与访问或更改值相比，调整数组大小是一项成本相对较高的操作。Lua需要在内存中找到一个新的位置，为所有条目提供足够的连续可用空间，然后将旧数组复制到该位置。当你拥有一个大数组并向其添加新值时，你不希望一直这样做。我们可以通过一个简单的循环来测试不同的表构造函数的性能：

[code=lua]

for i =1, 10^7 do

local t = {1, 2, 3, 4, 5}

end

[/code]

这段代码在我的笔记本上运行8秒，因为表最初是由8个条目创建的(entries)。以下代码创建一个空表并在其后添加条目。

[code=lua]

for i =7, 10^7 do

local t = {}

t[1], t[2], t[3], t[4], t[5] = 1, 2, 3, 4, 5

end

[/code]

这段代码执行同样的操作，需要31秒才能运行，因为它在循环的每次执行中都会调整表的大小三次。

创建这样的一个表的第三种方式是显示使用表的表达式中的整数索引：

[code=lua]

for i = 1, 10^7 do

local t= {[1] = 1, [2] = 2, [3] = 3, [4] = 4, [5] = 5}

end

[/code]

你可能希望它的运行速度和第一个版本一样快，但它需要13秒。这样做的原因是Lua一开始并不将其识别为数组。它将索引误认为哈希表的项，并生成一个包含8个空位的部分哈希表。但它仍然比在构造函数(constructor)之外添加条目快，因为Lua在内部填充数组并调整其大小，而不需要在Lua代码和内部C代码之间切换。

关于数组大小的讨论导致我们在第四章的SimpleTimingLib示例中遇到一个老问题，该示例使用数组存储参数，这些参数后来通过unpack()传递给函数。问题是可能会将nil作为参数传递给函数，这就提出了数组是否可以存储nil值的问题。我给出的答案是肯定的，但只是在某些情况下。

它取决于数组的大小和nil值的数量。当使用整数索引时，Lua试图平衡内存使用和访问速度。仅仅因为要使用条目1和65，就使用一个包含128个条目的数组显然毫无意义。但是，当你想要使用条目1和条目3时，使用4个空位数组是有意义的，并且是可能的。

如果在表构造函数中显示地告诉编译器这样做，Lua就会使用数组部分。下面的例子说明了这一点：

[code=lua]

print(unpack{nil, nil})

print(unpack{nil, nil, nil, 4}) --> nil nil nil 4

print(unpack{nil, nil, nil, 4, 5, 6}) --> nil nil nil 4 5 6

[/code]

注意数组后面不能有nil值；它们总是会被隔断(cut off)。使用可变参数“...”创建数组时，也会这样。

你可能想知道为什么SimpleTimingLib有时会失败，尽管它使用了变量参数(vararg)来创建表，但它的nil值相对较少。原因是修改表(例如使用它的哈希表部分)会导致Lua通过将值移动到哈希表部分来优化数组。

数组部分只有在超过50%的空位被占用时才会被使用。我们可以用一个简单的函数来测试它，该函数向数组的哈希表部分添加一个虚拟字段(dummy field)。我们的SimpleTimingLib使用哈希表来存储要调用的函数和任务到期的时间：

[code=lua]

function foo(...)

local t = {...}

t.x = 5

return unpack(t)

end

-- use 3 of 4 fields works

print(foo(1, nil, 3, 4)) -->1 nil 3 4

-- 2 of 4 fields is not enough

print(foo(1, nil, nil, 4)) --> nil nil 3 4 5 6 7

-- 4 of 8 is not enough

print(foo(nil, nil, 3, 4, 5, 6)) --> (everything in the hash table part)

[/code]

这说明了在开始使用哈希表部分应用的“超过50%”限制。然而，这个限制对我们的计时库来说并不是真正的问题，因为你很少需要向函数传递多个空值。即使必须这样做，大多数函数也接收false，在数组中使用它不会有问题。

[b][size=130%]&#9675; 了解哈希表(Understanding Hash Tables)[/size][/b]

哈希表是相比简单数组更复杂的数据结构。它能够将任意值(如字符串)与其它值关联，因此也称为关联数组。哈希表特别有趣的是，读取或设置哈希表中的值是一个O(1)操作。有人可能会认为关联数组在内部只是对所有条目进行迭代，直到找到请求的条目为止，这是一个O(n)操作。但事实并非如此。Lua可以计算哈希表中给定项的内存地址，在我们构建自己的哈希表时，你将看到这是如何工作的。

在创建示例表之前，我将概述哈希表的工作原理。

[b]· 哈希表基础(Hash Table Basics)[/b]

数组只存储值；哈希表还必须存储键(key)，该键也可以是任意Lua值。哈希表条目也称为键值对(key/value pairs)。你可能已经猜到了这种键值对的内存使用情况：32字节，而数组使用16字节。

哈希表在内部使用固定大小的数组来存储其键/值对。数组开始时没有键/值对的空位，并且按照与表的数组部分相同的规则增长。

该数组中键/值对的位置可以通过使用散列函数，使用键计算出来，散列函数将任意对象转换为称为该对象的散列码(或简称散列)的数字。然后，该散列用作数组中键/值对的索引。简单计算hash % #array(array是用于键/值对的数组)确保该索引在数组的边界内。

注意这个散列不是唯一的；可能有多个不同的对象具有相同的哈希，这会导致哈希表中发生冲突。一个好的散列函数可以确保表中的冲突(collisions)数保持在较低的水平，但不能完全避免它们。这意味着我们需要一种解决这些冲突的方法。

有几种解决冲突的方法；最简单的方法之一是在多个键碰撞的槽(slot)中使用键/值对的链表(link list)。新的冲突键将被添加到列表(list)中，从哈希表中检索键/值对的get操作将遍历列表并检查请求的键是否在其中一个键/值对中。

最坏的情况是，表中的所有键都具有相同的哈希代码，这会降低哈希表的性能。因为，它是一个链表，每当你添加一个新条目或从中读取一个条目时都属于是一个O(n)操作，它都会被遍历。但是，当使用像Lua散列函数(Lua hash function)这样的好散列函数时，这种情况极不可能发生。

理论就是这些；现在让我们创建一个简单的哈希表。

[b]· 创建一个哈希表(Creating a Hash Table)[/b]

我们将使用Lua创建一个哈希表对象，以便更好地理解哈希表。注意，与原本Lua哈希表相比，我们在这里创建的哈希表的性能非常差，特别是因为我们使用带有两个条目的Lua哈希表(在发生冲突时使用三个条目)来表示Lua中的键/值对。这个例子的唯一目的是为了演示哈希表是如何工作的。

我们的哈希表对象将表现得像一个普通的Lua哈希表。我们可以通过使用元方法重载表访问操作(table-access operation)来实现这一点。唯一的区别是性能，正如刚才所指出的那样，性能会非常差；原因之一是它是用Lua而不是该语言提供的哈希表一样用C编写的。另一个原因是哈希表甚至会在内部使用Lua哈希表来表示键/值对。我们的示例也使用了一个非常简单的哈希函数；这将导致很多冲突，因为优秀哈希函数是有点复杂的。如果你了解C语言，并希望详细了解Lua哈希表的工作方式，可以阅读Lua源代码(ltable.c文件)

让我们从创建一个简单的哈希函数开始。我们将使用一个函数简单地对值使用tostring，之后计算字符串中所有字符的字节值的和。这可以通过两个简单的函数来实现：

[code=lua]

local function sum(...)

local result = 0

for i = 1, select(“#”, ...) do

result = result + select(i, ...)

end

return result

end

local function hash(k)

k = tostring(k)

return sum(k:byte(1, #k))

end

[/code]

你可能会尝试使用递归函数求和，但递归通常比循环慢。让我们假设一下，我们真的想在一个插件中使用这个哈希表；这意味着哈希函数是实现的性能关键部分。因此，在这里使用递归求和几个数字是不可能的。

现在，让我们构建实际的哈希表，它通常定义三个操作：一个get操作，用于检索给定键关联的值；一个put操作，用于在表中存储新的键/值对或更新现有的键/值对；以及一个delete操作，以删除现有键。我们不必为此定义方法；可以使用元方法\_\_index进行get操作，使用\_\_newindex进行put和delete操作。

让我们从创建\_\_index方法开始。我们将使用字段大小来存储数组的当前最大大小，以模拟静态大小的数组。kvPairs包含当前存储在散列表中的所有键/值对。该方法计算请求键的位置，然后检查存储在这个bucket(保存键/值对的数组项)中的所有键/值对，除非发生冲突，否则只有一对键/值对。我们必须在计算位置上加上1，因为Lua数组是基于1的；位置0会在数组的哈希表部分，我们不希望那样。

[code=lua]

HashTableMT = {}

HashTableMT.\_\_index = function(self, k)

local pos = hash(k) % self.size + 1 -- calculate the position

if not self.kvPairs[pos] then -- doesn’t exist...

return nil -- ...return nil

else

local pair = self.kvPairs[pos]

while pair do -- traverse the list to find it

if pair.key == k then -- found the key

return pair.value -- return the associated value

end

pair = pair.next -- check next item in list

end

return nil -- requested key didn’t exist in this bucket

end

end

[/code]

这个函数展示了为什么从哈希表中检索值是O(1)操作(如果哈希函数是优秀的)。计算位置，然后操作是一个简单的数组访问，这是一个O(1)操作，因为可以计算实际内存地址。

\_\_newindex方法比\_\_index稍微复杂一些，因为它需要检查冲突并解决它们。在哈希表对象中还需要一个额外的属性：usedSlots，它保存当前已占用的空位数。put操作调用resize函数，如果使用的空位比现有空位多，则resize函数将使哈希表的大小增加一倍。

[code=lua]

HashTableMT.\_\_newindex = function(self, k, v)

local pos = hash(k) % self.size + 1 -- calcultate the position

if not self.kvPairs[pos] then -- entry doesn’t exist yet

if v == nil then

return -- inserting a new nil value is pointless

end

self.kvPairs[pos] = {key = k, value = v} -- just add a new/value pair

self.usedSlots = self.usedSlots + 1 -- increment the number of used slots

else

-- we either have a collision or we are trying to update an existing handler

-- let’s check if the key already exists in this bucket

local pair = self.kvPairs[pos]

while pair do

if pair.key == k then -- found it, let’s update it

pair.value = v -- update...

return -- ...and return, no need to update the size or resize the array

end

pair = pair.next

end

-- the entry does exist but our key is not in the bucket

-- this means we’ve got a collision :(

-- --> insert the new elemtn at the beginning of the linked list

self.kvPairs[pos] = {key = k, value = v, next = self.kvPairs[pos]}

self.usedSlots = self.usedSlots + 1

end

-- check if we reached the limit

if self.usedSlots > self.size then

resize(self) -- double the size

end

end

[/code]

你可能会注意到，将字段设置为nil并不会将其从表中删除。它只是将其值设置为nil，但键/值对仍然存在，表不会收缩。常规Lua哈希表也有相同的行为；真正从它们中删除某些内容的唯一方法是触发调整大小(trigger a resize)。这个调整大小操作必须计算新的位置(重哈希，rehash the table)；这意味着它必须循环遍历数组中的所有条目，所以这是一个删除nil值的好时候。将下列函数置于\_\_newindex元方法之上，因为这里需要它：

[code=lua]

local function resize(self)

self.size = self.size \* 2

self.usedSlots = 0

local oldArray = self.kvPairs

self.kvPairs = {}

-- rehash the table by inserting all non-nil entries into the new re-sized array

for i, v in pairs(oldArray) do

local pair = v

while pair do

if pair.value ~= nil then

self[pair.key] = pair.value

end

pair = pair.next

end

end

end

[/code]

请注意，我们的表实际上永远不会缩小，只会增大。真正的Lua表也可以在重哈希(rehash)被触发时收缩，但是重哈希只能通过添加条目而不是删除条目来触发。

我们的哈希表差不多完成了。在测试它之前，我们需要做的最后一件事是构造函数(constructor)：

[code=lua]

function CreateHashTable()

return setmetatable({size = 1, usedSlots = 0, kvPairs = {}}, HashTableMT)

end

[/code]

[b]· 测试我们的哈希表(Testing Our Hash Table)[/b]

我们可以运行的最简单的测试是使用几个不会冲突的键，检查并确认我们可以读取它们：

[code=lua]

local t = CreateHashTable()

t.a = 1

t.b = 2

t.c = 3

t.d = 4

print(t.a, t.b, t.c, t.d) --> 1 2 3 4

[/code]

a的哈希码是97,b是98，依此类推。这意味着连续字符不会与我们的哈希算法发生冲突。我们知道t看起来就像是一个普通的表，所以让我们通过添加以下代码来测试它是否真的在使用我们的哈希表实现：

[code=lua]

print(rawget(t, “a”), rawget(t, “b”)) --> nil nil

print(rawget(t, “c”), rawget(t, “d”)) --> nil nil

[/code]

rawget函数在不调用\_\_index元方法的情况下执行原始表访问。现在，我们可以尝试从子表kvPairs中读取键/值对。注意，访问t.kvPairs不会调用\_\_index，因为这个元方法只有在rawget返回nil时才会被调用。(对于\_\_newindex也是如此。)

[code=lua]

print(t.kvPairs[1].key, t.kvPairs[1].value) --> d 4

print(t.kvPairs[2].key, t.kvPairs[2].value) --> a 1

print(t.kvPairs[3].key, t.kvPairs[3].value) --> b 2

print(t.kvPairs[4].key, t.kvPairs[4].value) --> c 3

[/code]

第一个条目包含键d：

[quote]

hash(“d”) % 4 + 1 = 100 % 4 + 1 = 1

[/quote]

有人可能会认为“a”是这个哈希表中的第一个条目，因为它是我们添加到哈希表中的第一个条目，但你无法控制哈希表中元素的顺序，也无法对哈希表进行排序。我们已经从普通的Lua哈希表中知道了这一点，迭代(iterator)函数next以似乎是随机的顺序返回键/值对。我们现在可以解释为什么：next只是迭代保存键/值对的数组，并跟踪所有链表，以防发生冲突，所以哈希表条目的顺序甚至可能在表调整大小时改变。

另一个有趣的测试是引发冲突(provoke a collision)，这对于我们糟糕的哈希函数来说非常容易。例如，当哈希调用键上的tostring时，键1和"1"发生冲突：

[code=lua]

local t = CreateHashTable()

t[1] = “number”

t[“1”] = “string”

print(t[1], t[“1”]) --> number string

[/code]

我们还可以通过查看数组kvPairs来查看冲突，其中两个条目都存储在第二个bucket中:

[code=lua]

print(t.kvPairs[2].key, t.kvPairs[2].value) --> 1 number

print(t.kvPairs[2].next.key, t.kvPairs[2].next.value) --> 1 string

[/code]

对哈希表的最后一个测试是检查重哈希(rehash)是否正常工作。让我们从一个拥有两个冲突条目“a”和“c”的哈希表开始：

[code=lua]

local t = CreateHashTable()

t.a = 1

t.c = 2

print(t.kvPairs[2].key, t.kvPairs[2].value) --> a 1

print(t.kvPairs[2].next.key, t.kvPairs[2].next.value) --> c 2

[/code]

当添加第二个条目时，这实际上已经触发了第一次重哈希。这就是为什么“c”在列表的第二个条目中而不是第一个条目中，正如你所想的那样，因为冲突的条目被添加到列表的开头。但是让我们来测试一下，如果我们添加另一个条目，它会将表的大小调整为4。当表格大小调整为4时，我们看到“a”和“c”不再冲突：

[code=lua]

t.b = 3

print(t.kvPairs[2].key, t.kvPairs[2].value) --> a 1

print(t.kvPairs[3].key, t.kvPairs[3].value) --> b 3

print(t.kvPairs[4].key, t.kvPairs[4].value) --> c 2

[/code]

在我们的哈希表中，一切似乎都很好。现在，我们可以利用这些知识来了解在使用表时可以优化哪些内容。

[b][size=130%]&#9675; 表优化基础(Basic Table Optimizations)[/size][/b]

优化表最简单的方法是避免创建不必要的表。一个很好的例子是一个典型的对象构造函数：

[code=lua]

function CreateSomeObject()

return setmetatable({}, {\_\_index = someObjectPrototpye})

end

[/code]

这将为创建的每个对象创建一个新的元表(metatable)。没必要这样做，因为所有对象都有相同的元表。最好在函数外部定义元表，就像我们在整本书中所做的那样：

[code=lua]

local someObjectMT = {\_\_index = someObjectPrototype}

function CreateSomeObject()

return setmetatable({}, someObjectMT)

end

[/code]

这是一个有效的优化。还有更复杂的技巧，但需要理解哈希表的内部细节。其中之一是创建具有适当大小的表。假设你有一个存储对象的表，该对象以4个条目开始，但是你知道稍后将添加第5个条目。让我们编写一个简单的例子：

[code=lua]

for i = 1, 10^7 do

local obj = {a = 1, b = 2, c = 3, d = 4}

obj.e = 5

end

[/code]

问题是，第5个条目会触发表的重哈希，因此代码运行得非常慢，在我的笔记本上需要18秒。我们可以创建一个大小为8的哈希表，方法是在表构造函数中添加一个虚拟表条目并将其设置为nil。

[code=lua]

for i = 1, 10^7 do

local obj = { a = 1, b = 2, c = 3, d = 4, e = nil}

obj.e = 5

end

[/code]

这样就避免了重复，而且速度快了33%，而且它只运行12秒。

让我们通过一个真实的例子来了解更多的优化：我们的库SimpleTimingLib迫切需要优化，因为我们在当前版本中没有考虑性能。