======

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

[size=130%](Optimizing SimpleTimingLib)[/size][/b][/color]

当前实现的SimpleTimingLib并不是很好。OnUpdate处理程序执行任务的时间为O(n)，如果你想在调度大量任务的插件中使用这个库，这是无法接受的。

[b][size=130%]&#9675; 使用另一种数据结构(Using Another Data Structure)[/size][/b]

熟悉这个主题的人可能会建议使用所谓的优先队列(priority queue)作为数据结构。你可能已经想到了二叉搜索树例如AVL树，或者堆例如二叉堆，或者斐波那契堆；从理论上讲，这些都是解决我们问题的非常有效的方法。然而，这样的数据结构非常复杂，初学者很难理解。所以我们不会在这里看到它。我确实考虑过解释一个简单的优先队列(一个二进制堆)；它相对较短，只有大约50行代码。然而，解释它肯定会很长很复杂。因此，我决定在这里跳过这一点，因为它会超出本书的范围。如果你对这个话题感兴趣，你可能会想读一本关于算法和数据结构的书。但是，我们仍然可以在这里改进SimpleTimingLib；在大多数用例中，我在这介绍的解决方案与合适的优先队列之间的性能差异是最小的。

我们将看到一个非常简单但有效的解决方案：一个存储所有条目的链表。这似乎与我们当前使用数组的解决方案非常相似。但我们将使用一个简单的技巧来加快代码速度：在插入新值时保持数组的排序。如果它是排序的，OnUpdate处理程序只需要查看这个列表中的第一个条目。因此，它的运行时间为O(1)，而新的插入操作运行时间为O(n)。

我已经测试了DBM(Deadly Boss Mods)中调度程序所使用的所有数据结构。一个合适的优先队列，对于调度(insert)操作和extract min操作(删除最小时间的计时器)都是O(log n)，与链表相比，具有明显的优势。但是我在这个测试中使用了数百万个同步计时器(simultaneous timers)，对于DBM或SimpleTimingLib来说，这都不是一个真实的场景。即使DBM充分利用了它的调度库，但在一场复杂的Boss战斗中，通常只有4到8个计时器在特定时间运行。这样做的好处很小，因为很多时间不是浪费在O(n)插入操作上，而是浪费在构建包含函数参数的表这样的任务上。

但小的优势仍然是优势，因此DBM使用二进制堆作为其调度的优先队列。你可以在DBM-Core文件中阅读带有良好注释的代码。如果你对这种数据结构是如何工作的感到好奇，可以阅读文件DBM-Core.lua(搜索“Scheduler”以找到它)中注释良好的代码。但是要注意：这里不讨论它，因为它超出了初学者指南的范围，所以代码可能看起来非常复杂。

现在有足够的理论；让我们开始在SimpleMingLib中实现链表。

[b][size=130%]&#9675; 构建SimpleTimingLib-1.1(Building SimpleTimingLib-1.1)[/size][/b]

基本上，我们将繁重的工作从OnUpdate处理程序转移到插入条目的函数，这是一个非常聪明的解决方案。OnUpdate调用非常频繁，而insert函数仅由另一个插件时不时地调用。

以下所有代码块都是嵌入式库SimpleTimeingLib-1.0的更新，这意味着你必须替换相应的函数。将该库的主要版本增加到SimpleMingLib-1.1。

我们更改的第一个东西是当前存储所有条目的数组；我们将用链表替换它，因此使用nil而不是空表初始化变量任务。替换此项：

[code=lua]

SimpleTimingLib.tasks = SimpleTimingLib.tasks or {}

[/code]

使用以下代码：

[code=lua]

SimpleTimingLib.tasks = SimpleTimingLib.tasks or nil

[/code]

schedule函数现在创建所有被调度(scheduled)任务的升序排序链表：

[code=lua]

local function schedule(time, func, obj, ...)

local t = {...}

t.func = func

t.time = GetTime() + time

t.obj = obj

if not tasks then

-- list is empty or the new is due before the first one

-- insert the new element at the very beginning

t.next = tasks

tasks = t

else -- list is not empty, find last entry which is < time

local node = tasks

while node.next and node.next.time < time do

node = node.next

end

t.next = node

end

end

[/code]

从代码中可以很容易地看到，这个操作现在是O(n)，因为它可能会迭代整个链表，以找到插入它的正确位置。以下代码显示了新的OnUpdate处理程序：

[code=lua]

local function onUpdate()

local node = tasks

while node do

if node.time <= GetTime() then

tasks = node.next

node.func(unpack(node))

else

break

end

node = node.next

end

end

[/code]

它看起来像一个典型的O(n)操作，因为它似乎在整个数组上循环。当所有被调度任务都需要在当前帧(frame)中结束(due)时，它可能在O(n)模式下运行。但是，我们必须查看整体性能，即使在最坏的情况下，所有任务都需要在此帧中执行，在下一次调用OnUpdate时也会有一个空的链表。在几乎所有的调用中，函数将只查看第一个元素，因为一旦遇到尚未结束的任务，它就会取消循环。

非调度(unschedule)函数也需要调整，但我们无法真正优化它。因为我们必须查看所有任务，以检查它们是否与提供的参数匹配：

[code=lua]

local function unscheduled(func, obj, ...)

local node = tasks

local prev -- previous node required for the removal operation

while node do

if node.obj == obj and (not func or node.func == func) then

local matches = true

for i = 1, select(“#”, ...) do

if select(i, ...) ~= node [i] then

matches = false

break

end

end

if matches then

if not prev then --trying to remove first node

tasks = node.next

else

prev.next = node.next

end

else -- set prev to the current node if it was not removed

prev = node

end

node = node.next

end

end

end

[/code]

SimpleTimeingLib现在看起来稍微复杂一些，但如果被许多插件使用，它会更快。库的用途是提供给多个插件使用，你必须假设一个库被另一个插件广泛使用。你不会想在这里浪费性能。

另一个易于应用的优化是再利用表(recycling tables)。

[b][size=130%]&#9675; 再利用表？(recycling tables？)[/size][/b]

在Lua中，再利用表通常是不值得的。垃圾回收器(The garbage collector)的工作效率非常高，回收表(collecting tables)通常比清空表并填充它们要快。但是《魔兽世界》提供了函数table.wipe(t)，它可以快速删除给定表中的所有条目。

我们可以这样做：创建一个堆栈(stack)，并在擦除后将所有不再需要的表推入这个堆栈。然后，调度函数可以尝试从这个堆栈中弹出一个表，或者在不可能的情况下创建一个新表。然而，有几个问题：

[quote]

&#9679;我们必须使用循环将所有参数插入到再利用表中。我们目前使用简单的{...}，之前已经看到这比循环要快。

&#9679;假设在短时间内需要许多计时器。然后在回收堆栈上就会有很多死的计时器对象，而且它永远不会收缩(shrink back)。可能的解决方案是使用弱表(weak table)来存储所有回收的计时器；之后，垃圾回收器将回收所有未被回收的表。但这也带来了另一个缺点，下面将讨论。

&#9679;使用堆栈作为数据结构是不可能的，因为垃圾回收器会通过在中间回收元素来销毁它。唯一的解决方案是使用哈希表，之后用next从中获取第一个元素。但是请记住，在我们的哈希表实现中，next需要查找表中第一个不是空闲的位置，这可能需要一些时间。这尤其有问题，因为如果启动了很多计时器，哈希表可能会变得相对较大；回想一下，只有向哈希表中添加更多的nil元素，它才能收缩。你稍后将在几乎空的大表上看到更多关于next性能的信息。

&#9679;很难跟踪当前可回收的表的数量，因为垃圾收集器将以不可预测的顺序和时间点删除它们。但是，仍然可以使用finalizer元方法跟踪数字，该方法是在收集对象时调用的函数。但这会产生更大的开销，因为带有终结器(finalizers)的对象需要特殊处理。在下一节讨论userdata值时，我将告诉你关于终结器(finalizers)的更多信息。

[/quote]

尽管有缺点，弱哈希表解决方案似乎仍然很有吸引力，因为你可能没想到next会成为这样一个性能大户(performance hog)。让我们看看下一步会有多糟糕。下面的代码创建了一个包含50000个条目的简单示例哈希表：

[code=lua]

local t = {}

for i = 1, 50000 do

t[-i] = i -- negative entries are always in the hash part

end

[/code]

让我们测试一个从表中删除所有项的简单循环：

[code=lua]

for k, v in pairs(t) do

t[k] = nil

end

[/code]

它的速度和预期的一样快，在我的笔记本电脑上大约0.03秒。现在让我们尝试使用以下代码。它也是删除表中的所有条目，但使用while循环和next(t, nil)检索表的第一个条目，直到表完全为空：

[code=lua]

local k, v = next(t, nil)

while k and v do

t[k] = nil

k, v = next(t, nil)

end

[/code]

这个循环在我的笔记本电脑上需要5秒的CPU时间，因为下一步必须遍历整个表去找一个条目直到最后。这在一个只有几个条目的大表上需要花一些时间，如果一个短的周期内有很多计时器(哈希表增长)，而之后的一个周期只有几个计时器(垃圾回收器删除了大部分计时器)，那么存储再利用表的哈希表将会变成这样的表。

在这种情况下，再利用表是不值得做的。你可以增加CPU使用率以节省几千字节的内存。但是在玩魔兽世界的时候，你的CPU通常会超负荷工作，而几千字节的内存不会有任何区别。

我们现在已经优化了很多表，并且在前面使用了字符串。但是还有两种更有趣的数据类型我们还没有讨论过：userdata值(userdata values)和线程(threads)。我们先看看如何使用userdata。

======

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

[size=130%](Utilizing Userdata)[/size][/b][/color]

我在第二章中告诉过你，你不能从Lua创建userdata值，但这并不意味着我们不能使用它们。现在，我们可以用userdata对象做什么？此对象最常见的用途是表示Lua主机(Lua host)提供的对象，如《魔兽世界》框体(frame)。所有框体都由一个包含userdata对象的表表示。但我们可以对这个对象做的唯一一件事是将它传递给API提供的函数，这些函数知道这个userdata对象实际上代表什么。这意味着我们目前无法对userdata对象执行任何操作，只能将其用作标识符。

但是userdata值可以有元表，它们甚至定义了两个额外的元方法：\_\_len在对userdata对象使用长度运算符#时被调用，\_\_gc是一个终结器(finalizer)，在垃圾回收器收集userdata对象时执行。但仍然存在两个问题：如何创建userdata对象，以及如何对其应用元表？setmetatable对userdata值无效。根据官方文档，无法创建userdata值并对其应用元表。

但是，Lua中有一个没有正式说明的(undocumented)函数，它创建一个新的userdata值(没有附加任何数据)和一个附加到它的元表。此函数的名称为newproxy(mt)，这已经表明了它的用途：将其用作委托(proxy)。

[b][size=130%]&#9675; 使用Userdata作为委托(Using Userdata as a Proxy)[/size][/b]

用一个委托对象包装(wraps)表，可以跟踪、修改或阻止对表的访问。在第6章，你看到过一个委托表，在这里我向你展示一些在你的文件环境中使用委托进行调试的有用技巧。该委托跟踪并传递对全局环境的所有访问；即所有全局变量。

newproxy(mt) 的第一个参数mt可以为true，为新的userdata值创建一个新的空元表，也可以为false，在没有元表的情况下创建它。mt的第三个可能值是另一个已经有元表的userdata对象，该元表将用于新的userdata对象。

下面的例子展示了用于对另一个表跟踪访问的userdata委托：

[code=lua]

local myTable = {}

local p = newproxy(true)

getmetatable(p).\_\_index = function(self, k)

print(“Accessing ”..tostring(k))

return mtTable[k]

end

p.test = “123”

p.foo = 1

p.foo = p.foo + 1

print(myTable.test)

[/code]

你可能想知道使用普通表的优势在哪里。唯一的区别是内存的使用。如果表的唯一目的是通过元表转发访问，则表会分配32字节的内存，这是浪费的。userdata对象不分配任何不必要的内存，但其优势仍然非常小，通常不值得付出努力。

但是，默认UI在许多与安全模板和受限框体相关的代码中以类似的方式使用newproxy。它在那里被用作包装框体(wraps frames)的委托，以保护框体不受污染。文件中的注释表明，它们使用userdata对象而不是表的唯一原因是内存使用。

[b][size=130%]&#9675; Userdata元方法(Userdata Metamethods)[/size][/b]

如前所述，还有两种额外的元方法可用。首先，当对userdata值使用长度运算符#时，将调用\_\_len。我们可以使用它为哈希表对象创建一个包装器(wrapper)，用于跟踪哈希表的大小。我们将使用userdata对象的元表作为哈希表内容的存储。元表中的\_\_size字段大小将存储哈希表的大小。代码如下：

[code=lua]

local function index(self, k)

return getmetatable(self)[k]

end

local function newindex(self, k, v)

local mt = getmetatable(self)

local old = mt[k]

mt[k] = v

if old and v == nil then -- deleting an existing entry

mt.\_\_size = mt.\_\_size -1

elseif not old and v ~=nil then

mt.\_\_size = mt.\_\_size + 1 -- adding a new entry

end

end

local function len(self)

return getmetatable(self).\_\_size

end

function NewHashTable()

local obj = newproxy(true)

getmetatable(obj).\_\_index = index

getmetatable(obj).\_\_newindex = newindex

getmetatable(obj).\_\_len = len

getmetatable(obj).\_\_size = 0

end

[/code]

我们可以通过添加以下代码来测试哈希表对象：

[code=lua]

local t = NewHashTable()

print(#t) --> 0

t.foo = “bar”

print(#t) --> 1

t.x = 1

t.y = 2

t.z = 3

print(#t) --> 4

t.foo = nil

print(#t) --> 3

t[1] = 4 -- also counts entries in the array part

print(#t) --> 4

[/code]

使用这样的表的一个缺点是，访问和设置其中的值会比常规表慢。这是因为每次访问或更改值时都有一个额外的函数调用。

第二个元方法是终结器(finalizer)\_\_gc，当垃圾回收器删除userdata值时调用它。这种元方法存在的原因是userdata值通常存储Lua无法访问的数据，垃圾回收器无法清理常规的userdata值。终结器通常不是Lua函数，而是由底层API提供的清理userdata值的函数。

这个元方法的使用对我们来说有些限制，因为我们只能在这里添加Lua函数。下面的示例只是打印一条消息，因为我想不出这个元方法有什么有用的应用程序：

[code=lua]

local p = newproxy(true)

getmetatable(p).\_\_gc = function(self)

print(“Collected ”..tostring(self))

end

[/code]

不需要通过调用collectgarbage(“collect”) 来触发垃圾回收器，因为当脚本结束时，垃圾回收器还会使用终结器收集所有对象。这样做的原因是userdata对象可能已经打开系统资源(如文件)，而终结器会关闭它们。因此，Lua必须确保调用所有终结器，即使脚本以错误终止。

我们还没有讨论Lua的另一个特性，因为在《魔兽世界》中你很少需要它：协程库(The Coroutine Library)。

======

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

[size=130%](The Coroutine Library)[/size][/b][/color]

这个库是Lua标准库之一，我在书的开头提到过它。该库允许你创建协程对象(coroutine objects)，这些对象基本上是函数，但有一个区别：它们可以让步(yield)，也可以恢复(resumed)。协程中的让步类似于函数中的返回，但协程将在恢复时从让步的位置继续执行。函数不能存储返回的位置并从那里继续。

[b][size=130%]&#9675; 协程基础(Coroutine Basics)[/size][/b]

在给定的时间内只能运行一个协程，因为协程不能实现多线程。因此，不能从外部阻止协程；它必须让步(yield)，然后才能从外部恢复。

让我们概述一下在协程中的可用功能：

[quote]

&#9679;coroutine.create(func)：从给定函数func中创建一个新的协程对象。此函数在第一次协程被恢复(resumed)(即启动时)时执行。

&#9679;coroutine.resume(co, ...)：恢复协程并将参数“...”传递给它。如果协程运行时没有错误，则此函数返回true，后面跟着传递给让步函数(yield function)或返回语句(return statement)的值；否则返回false，后面跟着错误消息。

&#9679;coroutine.yield(...)：让步并且激活被当前活动协程调用的协程。所有被传递到此函数的参数都将作为coroutine.resume调用恢复(resumed)当前活动协程的额外的返回值返回。当协程在下次被恢复时，yield返回传递给resume函数额外的值。

&#9679;coroutine.running()：返回当前正在运行的协程，如果没有任何协程正在运行，则返回nil。

&#9679;coroutine.status(co)：返回协程co的状态，它有以下可能的值：running表示它当前正在运行；suspended意味着它已经让步或尚未启动；normal意味着当前协程已经停止了，因为另一个协程被恢复了；dead表示从中创建协程的函数已返回或发生错误。

&#9679;coroutine.wrap(func)：从func创建一个协程，并返回一个包装函数(wrapper function)，该函数在每次调用包装函数时对创建的协程调用coroutine.resume。此包装函数去除第一个返回值(指示发生的错误)，并在该值为false时生成错误消息。

[/quote]

它看起来很复杂，现在让我们看一个简单的例子。

[b][size=130%]&#9675; 协程示例(Coroutine Basics)[/size][/b]

下面的示例创建了一个协程，该协程封装了一个大体上由无限循环组成的函数。此函数在循环体中生成，并将计数器传递给调用函数：

[code=lua]

local function foo()

for i = 1, math.huge do

coroutine.yield(i)

end

end

local co = coroutine.wrap(foo)

print(co()) --> 1

print(co()) --> 2

print(co()) --> 3

[/code]

这个例子非常简单，但你可以想象得到，协程可能很容易变得非常复杂，特别是当你有多个相互恢复并传递参数的协程时。

然而，在《魔兽世界》的插件中，你不太可能需要如此复杂的协程结构。但是，如果你需要执行更长的任务，并且希望将其分布在多个框体上以避免延迟上升，则可以使用本例中的简单循环。你可以简单地定义一个OnUpdate处理程序，该处理程序检查给定的协程是否仍处于活动状态，并在本例中执行它。然后，协程执行任务的几个步骤并产生让步(yields)。

======

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

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

[b][size=150%]这[/size][/b]一章是本书中最难的一章，因为它非常深入地阐述了细节。如果你不了解每一个细节，不要担心。在你自己编写了第一个插件之后，可能想再次阅读本章。之后，你可以应用我在这里介绍的一些技巧和窍门来改进你的插件。

你了解了我们如何通过许多小技巧来衡量和改进插件的性能，还了解了许多有关Lua内部工作方式的细节。我们详细讨论了字符串和表是如何工作的，并给出了优化它们的示例。

之后，我们通过应用所讨论的优化技术改进了我们的库SimpleTimingLib。插入(insertion)操作现在是一个O(n)操作，OnUpdate处理程序是O(1)；旧版本则相反。这是更好的，因为OnUpdate处理程序在每一帧(frame)都被调用，而插入只是不时被调用。

然后我向你展示了Lua API的两个部分，我们之前没有讨论过。其中一个是函数newproxy()，这是一个完全没有文档记录(undocumented)的特性，非常有用，因为它允许你创建userdata对象。我在这里向你展示的最后一个技巧是如何使用协程标准库，这在《魔兽世界》插件中很少使用。这个库相当有用，功能强大，但它非常难以掌握。