[quote]

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

======

[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，本书的内容已经与现在你实际接触到的游戏与代码内容有所差异，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》的第十二个章节。全书共14个章节和2个附录。这是我翻译的第四个章节，其余章节的翻译内容我会在以后陆续发帖分享。[b]也希望阅读的人不要吝啬你们的点赞&#128077;，只要点一下便能给予我莫大的支持和鼓励。[/b]这是到目前为止所翻译的几个章节的合集，有兴趣的可以自行浏览[tid=24502396][color=orangered][合集]Lua与魔兽世界插件——《Beginning Lua with World of Warcraft Addons》[/color][/tid]

这次第十章还没翻译完，先放出一部分。[del]毕竟之后要开荒了，最近忙着调整WA，想在副本中实现一些有趣的功能。想更改之前的几个章节中的翻译错误。[/del]

[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]

接下来让我们正式开始这次的阅读吧。

[/quote]

[quote]

[align=center][b][color=orangered][size=200%]' 目 录 '[/size][/color][/b][/align]

======

[size=180%][b]&#9632; 第十章 使用库[/b][/size]

[size=130%][b]&#9679; 嵌入库(Embedded Libraries)[/b][/size]

&#9675; [b]库的版本控制(Versioning Libraries)[/b]

&#9675; [b]嵌入库的问题(Problems with Embedded Libraries)[/b]

&#9675; [b]使用LibStub(Using LibStub)[/b]

&#9675; [b]创建我们自己的嵌入库(Creating Our Own Embedded Library)[/b]

· 为我们的库建立一个Skeleton插件(Building a Skeleton Addon for our Library)

· 嵌入LibStub(Embedding LibStub)

· 构建SimpleTimingLib-1.0(Building SimpleTimingLib-1.0)

· 覆盖旧版本的问题(Problems with Overwriting Old Versions)

· 库的TOC文件(A TOC File for Our Library)

· 测试库(Testing the Library)

[color=crimson][size=150%]---本章未完待续---[/size][/color]

[/quote]

======

[align=center][color=orangered][b][size=200%]&#9632; 第十章 使用库[/size][/b]

[size=130%][b](Using Libraries)[/b][/size][/color][/align]

库(library)是一个插件，它提供了一组可以被其他插件所使用的函数。我们在第四章中编写的插件SimpleTimingLib就是一个库。但是在使用库时会涉及到一些问题。假设你编写了一个依赖于该库的插件，并希望发布它。你可以在zip压缩文件中包含该库，也可以让用户自己安装库。

第一个解决方案(内置库)，对于已经拥有此库的用户，当他试图安装你的包时会暴露其缺点。他用你的库版本覆盖了他已经拥有的库版本。但是他如果有一个更新版本的库会发生什么呢？如果他有一个旧版本，但是使用该库的另一个版本会与新版本不兼容，会发生什么呢？只要你的库有不同的版本，这种解决方案几乎肯定会破坏用户界面。用户甚至无法区分库的新旧版本，因为它没有版本号。

另一种解决方案是，告诉用户从其他地方下载所需库的特定版本，如果你的插件被设计为另一个更大的架构或库的插件，那么它会非常有效。假设你已经为一种新的实例编写了boss模块，并且已经使用DBM。之后你可以在某个地方(比如curse.com)发布boss模块，并注明“这是一个DBM的相关插件”。你需要从[url]http://www.deadlybossmods.com[/url]安装DBM来使用这个boss模块。你的用户肯定会理解该模块是基于boss模块创建的架构，因此需要DBM。

但是有很多插件使用了10个或更多的库。你将在本章看到这样的插件。对于用户来说，手动下载插件的所有库是相当费力的。很多人会在任务失败后给你发邮件说“你的插件不工作了，我总是得到一个关于缺少依赖的错误！”。

我们将在本章种看到如何在魔兽世界中使用嵌入式的库来解决这个问题，嵌入式库式嵌入到你的插件主文件夹中，而不是作为单独的插件。

======

[color=blue][b][size=160%]&#9679; 嵌 入 库 [/size]

[size=130%](Embedded Libraries)[/size][/b][/color]

嵌入式库式嵌入在插件中的库，这意味着它将与插件在同一个文件夹中。这样的库通常由单个Lua文件组成，只需将其添加到TOC文件中。之后你可以把你的插件文件打包成一个zip文件，并与它使所有使用的库一起发布。

如果我有两个插件嵌入相同的库会发生什么？我需要使用这个库两次吗？这个问题的答案即是肯定的，也是否定的。你的硬盘上将有两个库，但每个版本只加载一次。但我们的库目前甚至没有版本号，所以两个不同版本的SimpleTimingLib将无法区分。并且要能够确定它是否已经被加载。对于库来说，怎样的数字算是一个好的版本号呢？

[b][size=130%]&#9675; 库的版本控制(Versioning Libraries)[/size][/b]

库的版本号通常由两部分组成：主要版本，主要版本是名称后跟着的第一个普通版本数字，例如LibFoo-1.1。对于这个版本号的更改表示对库的重大更改。这个版本号的第二部分是它的次要版本，它会随着库的每一个次要更改而增加。这个次要版本只是表示为一个数字，例如，9。稍后你将看到如何设置库的这些版本号。

假设我们有一个名为LibFoo的嵌入库。我们安装了以下插件和以下版本的LibFoo。

[quote]

Addon1:LibFoo-1.0(minor:1)

Addon2:LibFoo-1.0(minor:3)

Addon3:LibFoo-1.0(minor:2)

Addon4:LibFoo-1.1(minor:9)

[/quote]

如果他们不是按需加载(load-on-demand)，则插件总是按字母顺序加载。这意味着第一个加载的插件是Addon1，之后加载LibFoo的1.0版本。下一个加载的插件是Addon2，它嵌入了LibFoo的1.0版本。现在情况变得有趣了，因为这里加载了同一个库的一个旧版本，如次要版本号所示的那样。

Addon2的LibFoo-1.0现在检测到它已经有了一个旧版本——你将很快看到我们如何实现这个功能。现在必须用新的库替换这个旧版本。这个新版本必须与旧的库兼容，因为其他插件可能依赖这个旧版本。

下一个插件是Addon3，它尝试加载LibFoo-1.0的另一个次要版本——次要版本为2，它比当前加载的版本3更老。这个LibFoo-1.0版本现在检测到已经有一个更新的版本(3)，它只需要什么也不做。要让程序说明也不做，可以在文件中的所有代码周围使用一个更大的if块，但是有一个更好的解决方案——在函数外部使用return。回想一下，Lua文件在内部作为一个巨大的函数被处理，它会在插件被加载时执行一次。在文件中的函数外部调用return就像在表示文件的函数中调用return一样。这意味着这个返回调用取消了文件的加载。下面的代码展示了这种返回语句：

[code=lua]

if dontLoadMe then

return

end

[/code]

如果dontLoadMe为真，将停止加载文件。我们将在本章的后面看到一个库的例子，以及我们如何处理这种检测。

最后加载的插件是Addon4。它加载了这个库的最新版本。这个版本包含了很多变化，并且不兼容旧版本的库，由于这个原因，它的主要版本是LibFoo-1.1。它不会替换旧的1.0版本的库，如果它被加载，由于可能存在的兼容性问题，它被视为一个完全不同的库。从此时起，你将在内存中加载此库的两个版本。

所有依赖于LibFoo-1.0的插件都将在次要版本3中使用LibFoo-1.0，所有需要LibFoo-1.1的插件都将在次要版本中使用LibFoo-1.1。所有插件和所有库都是向后兼容一个主要版本。

[b][size=130%]&#9675; 嵌入库的问题(Problems with Embedded Libraries)[/size][/b]

但是，如果一个库在主要版本中不是100%向后兼容的，会发生什么呢？或者，如果一个插件只与一个库工作，因为该库的错误吗？库的新版本现在修复了这个bug，旧的插件失效了。这种情况不应该发生，而且在新版中维护完全的向后兼容性实际上是相当困难的。此问题的简单解决方案是，随着可能导致此类问题的每个主要变更，主要版本都要增加。次要版本仅针对不影响库功能的微小更改而增加。

另一个缺点是，嵌入库可能会减慢插件的加载过程，特别是当你安装了很多不同版本的库时。在加载UI时，它可能被替换多次。

由多个插件或程序共享的库以及不同版本的整体问题并没有解决。这不仅适用于《魔兽世界》，每个在许多程序之间使用共享库的系统都有相同的问题。例如，在Windows中，这个问题被称为“DLL Hell”。如果你曾经在Windows中丢失了一个DLL(动态链接库)文件，你就知道我在说什么了。

嵌入库是解决这个问题的一个很好的解决方案，但是它们还远远不够完美。

让我们看看如何实现这些嵌入库。基本上有一个主库(main library)充当库的助手库(helper library)：LibStub。

[b][size=130%]&#9675; 使用LibStub(Using LibStub)[/size][/b]

LibStub是一个管理其他嵌入式库的嵌入库。你可以在wowace.com的项目主页面下载它的最新版：[url]http://www.wowace.com/projects/libstub/[/url]

它提供了三个可由插件和库访问的函数，LibStub:NewLibrary(major, minor)、LibStub:GetLibrary(major, silent)和LibStub:IterateLibraries()。最后一个只是一个辅助函数，可以在一般的for循环中使用它去遍历当前加载的所有库。

方法NewLibrary创建了一个新的库并注册它。这个函数的第一个参数是一个字符串，用作库的唯一标识符。此标识符称为此库的主版本。在前面使用LibFoo的示例中，对于LibFoo-1.0，可以是LibFoo-1.0,对于LibFoo-1.1，可以是LibFoo-1.1。这个函数的第二个参数是一个数字，表示库的次要版本。这个数字应该随着对库的每次变更而增加，以确保总是用新版本替换旧版本。

LibStub检查是否已经存在具有相同版本的库。如果不是这样，它会返回一个新的空表。之后这个空表将被库作为命名空间，这意味着库的所有函数和变量都将放在这个表中。如果已经有这样的库加载了较新的次要版本，则它返回nil。之后会取消库的加载过程，因为这意味着它已经过时，并且已经加载了一个新版本。

第三种情况是，已经有了一个加载相同主要版本的库，但是旧的次要版本比新库的次要版本要低。LibsStub会返回这个旧的库(只是一个表)，然后返回旧库的次要版本。新库现在可以覆盖旧版本库使用的表中的所有函数和其他值。这个过程用你的新版本完全替换了库的旧版本。

最后一种方法是LibStub:GetLibrary(major, silent)，它返回请求的库，后面跟着它的次要版本，如果该库不存在，则抛出一条错误消息。第二个可选参数可以设置为true来抑制(suppress)该错误消息。

LibStub非常简短，因此我们可以在这查看它的源代码。请注意，它也是一个嵌入式库，这意味着它还必须负责替换自身的旧版本，并且能够取消加载过程。回想一下，\_G式保存所有全局变量的表，这意味着LibStub将存储在以下代码中的全局变量LibStub中：

[code=lua]

-- LibStub is a simple versioning stub meant for use in Libraries.

-- http://www.wowace.com/wiki/LibStub for more info

-- Credits: Kaelten, Cladhaire, ckknight, Mikk, Ammo, Nevcairiel, joshborke

local LIBSTUB\_MAJOR, LIBSTUB\_MINOR = “LibStub”, 2

local LibStub = \_G[LIBSTUB\_MAJOR]

if not LibStub or LibStub.minor < LIBSTUB\_MINOR then

LibStub = LibStub or {libs = {}, minors = {} }

\_G[LIBSTUB\_MAJOR] = LibStub

LibStub.minor = LIBSTUB\_MINOR

function LibStub:NewLibrary(major, minor)

assert(type(major) == “string”, “Bad argument #2 to ‘NewLibrary’ (string exected)”)

minor = assert(tonumber(strmatch(minor, “%d+”)), “Minor version must either be a number or contain a number.”)

local oldminor = self.minors[major]

if oldminor and oldminor >=minor then return nil end

self.minorss[major], self.libs[major] = minor, self.libs[major] or {}

end

function LibStub:NewLibrary(major, minor)

assert(type(major) == “string”, “Bad argument #2 to ‘NewLibrary’ (string expected)”)

minor = assert(tonumber(strmatch(minor, “%d+”)), “Minor version must either be a number or contain a number.”)

local oldminor = self.minors[major]

if oldminor and oldminor >= minor then return nil end

self.minor[major], self.libs[major] = minor, self.libs[major] or {}

return self.libs[major], oldminor

end

function LibStub:GetLibrary(major, silent)

if not self.libs[major] and not silent then

error((“Cannot find a library instance of %q.”):format(tostring(major)), 2)

end

return self.libs[major], self.minors[major]

end

function LibStub:IterateLibraries() return pairs(self.libs) end

setmetatable(LibStub, { \_\_call = LibStub.GetLibrary })

end

[/code]

这里LibStub的主要版本就是LibStub，它被用作一个全局变量来存储库。这里的次要版本是2。它检查是否已经加载了LibStub的旧版本，如果这个旧版本存在，并且 LibStub的旧实例的次要版本大于或等于该实例的次要版本，则不执行任何操作(通过使用一个大的if块)。

否则，它用一个包含所有库的表和一个包含所有库的次要版本的表初始化LibStub。如果要覆盖旧的LibStub，它将只使用旧版本LibsStub使用的表(命名空间)。之后这个库定义了它的三个方法。

最后一行中的metatable赋值允许你使用LibStub(“libname”)而不是LibStub:GetLibrary(“libname”)。

[b][size=130%]&#9675; 创建我们自己的嵌入库(Creating Our Own Embedded Library)[/size][/b]

为了使用LibStub，让我们来更新SimpleTimingLib。我们需要设置一个新的插件，作为我们的库的测试，以检查它是否工作。我们需要这个测试插件，因为如果没有被嵌入到插件中，则嵌入式库就不能使用。当前包含库的文件夹SimpleTimingLib，在完成这个嵌入版本后将不再需要。当前我们可以将这个库的工作版本嵌入到目前使用SimpleTimingLib的其他插件中。让我们调用我们的测试插件SimpleTimingLibTest。

[b]· 为我们的库建立一个Skeleton插件(Building a Skeleton Addon for our Library)[/b]

为库创建一个名为SimpleTimingLibTest的文件夹，并在其中放置以下TOC文件：

[quote]

## Interface: 30100

## Title: Test Addon for SimpleTimingLib-1.0

## OptionalDeps: SimpleTimingLib-1.0

libs\LibStub\LibStub.lua

libs\SimpleTimingLib-1.0\SimpleTimingLib-1.0.lua

SimpleTimingLibTest.lua

[/quote]

我们在这里使用字段OptionalDeps。这确保了我们的插件是在插件SimpleTimingLib-1.0之后加载的(如果有这样的插件存在的话)，允许我们以后创建库的独立版本。这种独立版本的有点是在加载过程中只加载一次。缺点是用户必须更新库的独立版本。解决这个问题的方法是为你的插件使用一个更新器，但是目前没有可用的更新器可以从插件中提取嵌入库。旧的WowAce更新程序有这个功能，但是它的继任者，Curse Client，还没有提供这个功能。它计划在未来的版本出现，在你阅读的时候可能已经可以使用了。

接下来的三行处理加载所需的文件。通常使用子文件夹“libs”来存放所有的库。注意，这里列出文件的顺序很重要：首先必须加载的是LibStub，因为库和测试文件都需要它。下一个需要加载的文件是我们的库，因为它是测试文件所需要的，并且依赖于LibStub。最后一个文件是我们的简单脚本，用于测试库是否正常工作，它需要SimpleTimingLib-1.0和LibStub。

一些插件嵌入了很多库，所以这个TOC文件中的列表很快会变得很长且混乱。但是如果使用XML，有一种简单的方法可以将库与属于插件的文件分开。回想一下，XML文件可以通过使用元素&#60;Script&#62;嵌入Lua代码。这意味着你可以创建一个XML文件，其唯一目的是加载其他Lua文件。下面的代码展示了一个XML文件，可以使用它替代前面TOC文件中的两个与库相关的条目。

[code=xml]

<Ui xmlns="http://www.blizzard.com/wow/ui/" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://www.blizzard.com/wow/ui/..\FrameXML\UI.xsd">

<Script file=”libs\LibStub\LibStub.lua”>

<Script file=”libs\SimpleTimingLib-1.0\SimpleTimingLib-1.0.lua”/>

</Ui>

[/code]

这样的XML文件通常称为embeds.xml。之后将它添加到TOC文件中，而不是添加所有的lib/libname/libname.lua条目。

[b]· 嵌入LibStub(Embedding LibStub)[/b]

接下来我们需要的是LibStub。从下载包中获取LibStub文件，并将其放到SimpleTimingLibTest\libs\LibStub文件夹中，以加载到插件中。

[b]· 构建SimpleTimingLib-1.0(Building SimpleTimingLib-1.0)[/b]

我们现在可以在文件libs\SimpleTimingLib-1.0\SimpleTimingLib-1.0.lua中创建库了。在这个文件中，我们需要做的第一件事是通过调用库LibStubd的NewLibrary方法创建一个新库。下面的代码显示了库的开头。它会尝试创建新的库，如果LibStub返回nil，则取消执行。当已加载相同或较新的版本时，就会发生这种情况。

文件的开头是这样的：

[code=lua]

local MAJOR, MINOR = “SimpleTimingLib-1.0”, 1

local SimpleTimingLib = LibStub:NewLibrary(MAJOR, MINOR)

if not simpleTimingLib then

return -- a greater or equal version is already loaded

end

[/code]

本地变量MAJOR和MINOR保存了库的主要和次要版本，它们在整个文件中是可见的。我们现在可以从旧库中复制和粘贴所有代码，除了下面这一行：

[code=lua]

SimpleTimingLib = {}

[/code]

之后，我们的库将只使用LibStub(也就是一个表)，而不是全局变量中的旧表。像下面这样的方法声明就做得很好：

[code=lua]

function SimpleTimingLib:Schedule(time, func, ...)

return schedule(time, func, self, ...)

end

[code]

实际上，整个库使用LibStub就可以很好地工作。出于兼容性的原因，我们甚至可以保留两个全局函数。当新版本的SimpleTimingLib加载到当前版本上时，它们将被覆盖。

[b]· 覆盖旧版本的问题(Problems with Overwriting Old Versions)[/b]

但是，当有一个新的次要版本要覆盖旧版本时，也会出现一个问题。目前，我们将所有任务保存在一个本地表，并且无法从外部访问这个表。这也意味着，如果我们的库必须覆盖它自己的旧版本，它将丢失所有旧的任务。但是，通过该表存储在库中，很容易防止这种情况发生。有问题是下面这一行：

[code=lua]

local tasks = {}

[/code]

将其替换为以下两行以解决此问题：

[code=lua]

SimpleTimingLib.tasks = SimpleTimingLib.tasks or {}

local tasks = SimpleTimingLib.tasks

[/code]

这将在库的字段任务(field tasks)中存储对任务表的引用。如果库已经加载了旧版本的库，则库将使用旧表，如果第一次加载库，则创建一个新表。本地变量任务也存储了对这个表的引用，我在这里使用这个表只是为了我们不需要改变代码中的任何其他东西。

但是还有第二个类似的问题：如果我们在运行时升级库，框体(frame)会发生什么？它也只是存储在一个局部变量中，不能从外部访问。它的OnUpdate脚本处理程序将在升级之后继续被调用。这实际上是完全没有问题的。但是，如果在加载接口时多次覆盖库，则会对性能造成一小部分的影响，因此我们希望回收该框体。

相似的问题，相似的解决方案，替换下面这一行：

[code=lua]

local frame = CreateFrame(“Frame”)

[/code]

替换成下面这两行：

[code=lua]

SimpleTimingLib.frame = SimpleTimingLib.frame or CreateFrame(“Frame”)

local frame = SimpleTimingLib.frame

[/code]

这些只会在框架不存在时创建它。SimpleTimingLib.frame拿着库旧版本使用的框架——如果有这样一个旧版本，这个框架将被覆盖。

目前为止。我们的库现在功能齐全，可以被嵌入。但另一个不错的特性是保留其作为独立库的能力。许多人更喜欢独立库而不是嵌入库，因为大量的嵌入库可能会减慢UI的加载过程。创建一个既可以作为独立库也可以作为嵌入库的库是很容易的。我们只需要添加一个TOC文件。

[b]· 库的TOC文件(A TOC File for Our Library)[/b]

在libs\SimpleTimingLib-1.0中，创建一个名为SimpleTimingLib-1.0.toc的TOC文件。如果你使用库作为一个嵌入库，则这个TOC文件将被游戏忽略，因为它是在一个插件的子文件夹而不是在一个插件的主文件夹。

这个TOC文件的内容如下所示：

[quote]

## Interface: 30100

## Title: SimpleTimingLib-1.0 (Stand-alone)

LibStub\LibStub.lua

SimpleTimingLib-1.0.lua

[/quote]

注意，这个库的独立版本仍然需要嵌入LibStub。这意味着你必须将LibStub文件夹从SimpleTimingLibTest\libs复制粘贴到SimpleTimingLibTest\libs\SimpleTimingLib-1.0。如果你使用这个库作为嵌入库，那么这个版本的LibStub将不会被加载，因为在这种情况下，引用它的TOC文件将不会被加载。

我们的库现在可以作为一个独立的版本使用，只要移动文件夹SimpleTimingLib-1.0，从SimpleTimingLibTest\libs到你的插件文件夹中。多亏了TOC文件，《魔兽世界》把这个文件夹识别为一个单独的插件，并将它与LibStub一起加载。

但是我们怎么才能确定库真的在工作呢？我们想要编写一个小的测试插件来使用这个库。现在让我们来编写测试代码。

[b]· 测试库(Testing the Library)[/b]

放置测试代码的地方是文件SimpleTimingLibTest.lua。下面的代码创建计时库(timing library)的一个新的实例，并创建一个简单的斜杠命令处理程序。这个斜杠命令是/stltest <time> <lua code> (for SimpleTimingLibTest)，它使用一个数字(time)作为参数，后面跟着要在time秒经过之后执行的Lua代码。

在SimpleTimingLibTest文件中，我们使用了Lua函数loadstring(“lua code”)，它将Lua代码作为字符串，并从中创建一个可以执行的函数。使用这个函数时应该非常小心，因为如果将字符串(在最坏的情况下可能来自另一个玩家)作为代码执行，可能会有安全问题。它也非常的慢，因为它必须将字符串编译为Lua代码。由于这些问题，只有在没有其他方法实现给定功能时才应该使用它。

[code=lua]

local timingLib = LibStub(“SimpleTimingLib-1,0”):New()

SLASH\_STL\_TEST1 = “/stltest”

SlashCmdList[“STL\_TEST”] = function(msg)

local time, code = msg:match(“(%d+) (.\*)”)

if time and code then

local func, errorMsg = loadstring(code)

if func then

timingLib:Schedule(tonumber(time), func)

else

error(errorMsg)

end

end

end

[/code]

我们可以通过执行这样的斜杠命令来测试库是否正常工作。

[code=lua]

/stltest 3 print(“test”)

[/code]

这将在三秒后将测试打印到默认聊天框。你也可以测试我们的库作为一个独立的库，或者你可以将库嵌入到另一个小版本的插件中，并确认升级工作正常。你将看到这是没有问题的，库继续正常工作。这种巨大的灵活性是嵌入库和LibStub的主要优点之一。

现在我们可以编写自己的库，但是使用现有的库更有趣。它们允许你只用少量的工作量就可以在你的插件中添加许多令人兴奋的功能。让我们来看一个著名的插件框架，它可以作为LibStub的嵌入式库包含在其中：Ace3

======

[color=crimson][size=150%]---本章未完待续---[/size][/color]