======

[color=blue][b][size=160%]&#9679; 建 立 一 个 冷 却 监 视 器[/size]

[size=130%](Building a Cooldown Monitor)[/size][/b][/color]

我们将建立一个插件，显示你的团队成员的法术冷却时间，如英勇/嗜血或重生。我们将这个插件称为冷却监视器(CooldownMonitor)。为它创建一个文件夹，并创建一个包含有以下条目的.toc文件。

[code]

## Interface:30100

## Title: Cooldown Monitor

CooldownMonitor.xml

CooldownMonitor.lua

CooldownBars.lua

[/code]

你现在还应该创建XML和Lua文件，这样当我们以后使用这些文件时，你就不用重新启动你的游戏了。让我们从构建主要Lua文件CooldownMonitor.lua开始。

[b][size=130%]&#9675; 检测法术(Detecting Spells)[/size][/b]

插件需要检测某些法术并对它们做出反应。最简单的解决方案是在处理程序中使用一串长长的if-then-elseif-end代码块来检查该事件是否是我们的法术之一。一个更好的解决方案是使用一个表(table)来存储我们想要使用的所有法术信息。我们需要存储的只是这个法术的事件、法术ID和冷却时间。下表存储了一些有趣的法术及其事件信息。

[code=lua]

local spell = {

SPELL\_CAST\_SUCCESS = {

[29166] = 360, -- Druid: Innervate (6 min cooldown)

[32182] = 600, -- Shaman: Heroism (Alliance)

[2825] = 600, -- Shaman: Bloodlust (Horde)

[22700] = 600, -- Field Repair Bot 74A (10 min duration)

[44389] = 600, -- Field Repair Bot 110G (10 min duration)

},

SPELL\_RESURRECT = {

[20748] = 1200, -- Druid: Rebirth (20 min cooldown)

}

SPELL\_CREATE = { -- SPELL\_CREATE is used for portals, and they vanish after 1 min

[53142] = 60, -- Portal: Dalaran (Alliance/Horde)

[33691] = 60, -- Portal: Shattrath (Alliance)

[35717] = 60, -- Portal: Shattrath (Horde)

[11416] = 60, -- Portal: Ironforge

[10059] = 60, -- Portal: Stormwind

[49360] = 60, -- Portal: Theramore

[11419] = 60, -- Portal: Darnassus

[32266] = 60, -- Portal: Exodar

[11417] = 60, -- Portal: Orgrimmar

[11418] = 60, -- Portal: Undercity

[11420] = 60, -- Portal: Thunder Bluff

[32667] = 60, -- Portal: Silvermoon

[49361] = 60, -- Portal: Stonard

},

}

[/code]

你可以很容易地添加新的法术，只需要在Wowhead或你的战斗记录中查找法术ID，然后把它添加到正确的事件中。当你运行斜杠指令/combatlog时，你可以使用创建的战斗日志文件来确定一个正确的法术事件。为了达到测试的目的，添加一个简单的法术是个不错的主意。我添加了下表作为测试法术的子表，48071是快速治疗(11级)的法术ID。

[code=lua]

SPELL\_HEAL = {

[48071] = 60 Flash Heal (Test)

}

[/code]

事件处理程序现在需要读取该表。下面的代码使用可变参数(vararg)上的select来检索感兴趣的参数。你还可以将我们需要的所有10个参数添加到头函数中，但这种处理不是很清楚。我们只需要子事件，施法者的名字、施法者的标志位(flags)、法术ID和法术名称。然后我们可以使用事件和法术ID来检查表中是否包含法术/事件组合。

施法者的标志位(flags)可以被用来检查玩家是否在我们团队中，如果没有设置从属(affiliation)：局外人(outsider)标志是否被设置。还可以检查是否是我的(mine)、阵营(party)、或副本标志(raid flags)被设置。但是这四个标志位总是相互排斥的。每个单位总是恰好只有其中之一。这意味着，如果没有设置外部标志位，则必须设置其他标志位中的一个。

我们还将使用一个被称为CooldownMonitor的表，放置所有需要从外部访问的所有函数。我们需要做的另一件事是创建局部变量onSpellCast，它被我们的事件处理程序访问。稍后我们将在其中存储一个函数。

[code=lua]

CooldownMonitor = {}

local onSpellCast

local function onEvent(self, event, ...)

if event == “COMBANT\_LOG\_EVENT\_UNFILTERED” then

local event = select(2, ...) get the subevent

local sourceName, sourceFlags = select(4, ...) caster’s nameand flags

local spellID, spellName = select(9, ...) spell ID and name

--check if we need the event and spell ID

-- and check if the outsider bit is not set, meaning the unit is in our group

if spells[event] and spells[event][spellId]

and bit.band(sourceFlags, COMBATLOG\_OBJECT\_AFFILIATION\_OUTSIDER) == 0 then

local cooldown = spells[event][spellId]

onSpellCast(cooldown, sourceName, spellId, spellName)

end

end

end

local frame = CreateFrame(“Frame”)

frame:RegisterEvent(“COMBAT\_LOG\_EVENT\_UNFILTERED”)

frame:SetScript(“OnEvent”, onEvent)

[/code]

每当有人施放了我们前面定义的法术，就会调用onSpellCast函数。onSpellCast函数调用startTimer(我们稍后会写这个函数)来显示一个施法条，显示玩家的名字和法术的冷却时间。onSpellCast的另一个功能是在聊天框中生成和显示一个简短的通知。例如这样一条信息。

[quote]

[player] just cast [spell] (Cooldown: x Minutes)

[/quote]

其中的[player]和[spell]应该是可以点击的链接。我们在构建DKP插件的时候使用了聊天链接。但我们还没有创建任何链接。在这样的消息中使用一些颜色来突出显示链接会更好。我们需要转义序列来创建聊天框中的链接和彩色文本。

[b][size=130%]&#9675; 在聊天框中使用转义序列(Using Escape Sequences in Chat Frame)[/size][/b]

转义序列(具有特殊含义的序列，如颜色代码)可以在游戏中的所有聊天框和字体字符串中使用。转义序列是管道符号(pipe symbol)：“|”。我们可以用一个简单的例子来测试这一点，该例子应该打印输出一些红色文本(更多颜色将在下节中介绍)：

[code=lua]

print(“|cFFFF0000red text!”)

[/code]

只有当您将其写入一个插件的Lua文件时，这才能工作。尝试通过/script或TinyPad这样的游戏编辑器来执行它，只会打印没有颜色的原始代码。这样做的原因是，游戏会自动将用户输入的“|”替换成“||”。这两个管道符号“||”是用于表示没有特殊含义的普通管道符号，就像Lua字符串中的反斜杠一样。

但是有一个Lua转义代码允许我们向游戏中输入的Lua字符串注入一个普通管道：“\124”。尝试在游戏中输入以下内容：

[code=lua]

print(“\124Cffff0000red text!”)

[/code]

这显示了一条红色消息“red text！”在聊天框中。但为什么是红色的，这些十六进制数是什么意思呢？

[b]· 彩色文本(Colored Texts)[/b]

颜色代码的格式是“|cAARRGGBB”。它是由四个值组成：alpha(透明度A)、red(红R)、green(绿G)和blue(蓝B)。所有这些值都是用两个十六进制数写入的，这意味着可能最低值是00，最高值是FF(十进制的255)。alpha的值实际上是被游戏忽略了，你可以在这里使用任何有效的十六进制值。但你应该在这里使用FF(完全可见)，以防某个补丁添加了对它的支持。然后根据红、绿、蓝三原色的值来构建颜色。(如果你不熟悉RGB颜色模型，维基百科有一个很好的解释。[url]http://en.wikipedia.org/wiki/RGB\_color\_model[/url])

举个例子，|cFF00FF00生成绿色文本、|cFF0000FF生成蓝色文本、|cFF111111生成灰色文本、|cFF000000是黑色、|cFFFFFFFF是白色。颜色代码不区分大小写，所以你也可以使用小写字母。

一旦定义，颜色将一直使用到下一个颜色定义或直到遇到游戏的颜色重置代码“|r”。颜色重置后的文本将是该文本元素的默认颜色。通过使用print添加的一行的默认颜色总是白色。但是也可以通过使用聊天框对象的AddMessage方法，向聊天框中添加另一种默认颜色的行。

当我们之前写ChatlinkTooltips插件的时候，你看到了聊天框对象。它们存储在全局变量ChatFrame1到ChatFrameN中。但哪个聊天框是默认的呢？着很可能是ChatFrame1，但你不能确定。因此UI定义了另一个全局变量，它始终代表默认的聊天框：DEFAULT\_CHAT\_FRAME。这意味着我们需要使用以下参数，来调用此框架的方法AddMseeage，以打印使用默认颜色的消息：

[code=lua]

DEFAULT\_CHAT\_FRAME:AddMessage(message, r, g, b)

[/code]

参数r、g、b代表颜色红、绿、蓝的值。注意，这里的最大值是1，最小值是0。所以，r=1，g=1，b=1得到了默认的白色。

如果你想给整个聊天信息上色，使用AddMessage应用默认颜色总是比使用|c转义序列更好。就像我们在第六章中写的迷你插件，这个默认的颜色也会被添加到聊天框的时间戳的钩子使用。使用print总是会得到一个白色的时间戳。

[quote]

注意：不可能在聊天框中嵌入颜色代码。被传递给SendChatMessage的字符串中使用颜色代码会生成一条错误信息。

[/quote]

许多插件为AddMessage方法使用了一个包装函数(wrapper function)来添加插件的名称作为前缀，并设置颜色。让我们为冷却监视器编写这样一个函数。对于像我们这样的小插件，将这个函数保存在一个局部变量种就足够了。较大的插件应该将其打印处理程序(print handler)存储在表中(作为插件的命名空间)，因为许多文件都可以在表中访问该函数。我们将简单地在文件CooldownMonitor.lua的作用域中创建一个名为print的本地变量，这样我们就可以像在以前的插件中所做的那样使用print了。此函数应防止在文件的开头，以确保本地变量在文件中的所有其他函数中都可见。

[code=lua]

local chatPrefix = “|cffff7d0a<|r|cffffd200CooldownMonitor|r|cffff7d0a|r”

local function print(...)

DEFAULT\_CHAT\_FRAME:AddMessage(chatPrefix..string.join(“ ”, tostringall(...)), 0.41, 0.8, 0.94)

end

[/code]

该函数利用了tostringall和string.join，因此，它的表现就像原始的打印处理程序。但它给我们的消息添加了应该漂亮的彩色前缀。

[b]· 超链接(Hyperlink)[/b]

使用以下格式的转义序列创建链接：

[code=lua]

|Hlink|htext|h

[/code]

“|H”标记链接的开始，下面的“link”是该链接的数据。数据字符串的末尾是“|H”,后面是显示的可点击的“text”，后面跟着“|H”。在前面创建DKP插件时，我们看到了一些链接类型；链接基本上只是一个字符串，它被传递给处理超链接事件(如OnHyperLinkClicked)的事件处理程序。让我们来看一个完整的条目链接。我们已经看到了商品的链接，它由描述商品的长串数字组成。例如，一个制作符文铜棒(Runed Titanium Rod)的背后代码是这样的：

[code=lua]

|cff0070dd|Hitem:44452:0:0:0:0:0:0:0:1352691344:80|h[Runed Titanium Rod]|h|r

[/code]

我们可以清楚地看到开头链接的颜色、链接ID和显示的文本，最后，后面是链接(|h)和颜色的结束代码(|r)。

[quote]

注意：不可能将一个未知的项目链接到服务器。这些项目可能是从你的战斗群组(battlegroup)中还没被杀死的boss。服务器还会检查聊天信息中链接的名称和颜色。如果名称或颜色不正确，该链接将从聊天信息中删除。

[/quote]

链接中第一个冒号之前的子字符串用于指示链接的类型。这个例子的类型是一个物品链接，但是我们需要一个玩家和一个法术链接。一个玩家的链接看起来是这样的：

[code=lua]

|Hplayer:name|htext|h

[/code]

并且一个法术链接是这样的：

[code=lua]

|Hspell:spellId|htext|h

[/code]

我们现在可以创建一个函数onSpellcast。添加该函数到文件CooldownMonitor.lua的末尾：

[code=lua]

local castInfo = “|Hplayer:%1$s|h[%1$s]|h cast |cFF71D5FF|Hspll:%d|h[%s]|h|r(cooldown: %d minutes)”

function onSpellCast(timer, player, spellId, spellName)

print(castInfo:format(player, spellId, spellName, timer / 60))

CooldownMonitor.StartTimer(timer, player, spellName, texture)

end

[/code]

这里使用的格式字符串可能看起来比实际要困难得多。第一个是玩家链接，它需要两次玩家的名字：作为链接和作为显示文本。我们可以通过使用“%1$s”启动替换指令来选择传递给string.format的一个参数，其中n是所需参数的数目(s也可以是任何其他格式指令)。这允许我们在这里两次使用第一个参数。第二个链接是一个拼写连接，其默认颜色取值默认UI。

另一个不错的功能是在信息中加入一个法术的小图标。我们也可以用这些转义代码来创建。

[b]· 纹理(Texture)[/b]

表示纹理的转义序列是这样的：

[code=lua]

|Tfile:height:width:xoffset:yoffset|t

[/code]

除文件和高度以外的所有参数都是可选的。默认宽度与高度相同，默认偏移量为0。如果高度(height)设置为0，则使用字体高度。你应该始终将高度设置为0。因为错误的值会扭曲字体。如果高度为0且宽度被设置，它将被理解为相对于字体高度的一个值。

例如，代码“|Tfoo.tga:0|t”显示的纹理文件foo.tga缩放到一个正方形文本宽度相同的高度，“|Tfootga:0:2|t”显示相同的纹理，高度设置为字体高度，宽度设置为字体高度的两倍，“|Tfootga:10:20|t”显示大小为10x20的foo.tga。

[quote]

注意：出于安全原因，不可能将纹理嵌入到聊天信息中。

[/quote]

现在我们有法术ID，但我们需要它的纹理。有一个可用的API函数可以返回几乎所有我们需要的关于法术的信息。这个函数名称是GetSpellInfo，它的参数可以是一个法术的ID、一个法术名称或法术链接。

[quote]

name, rank, iconTexture, cost, isFunnel,

powerType, castTime, minRange, maxRange

- GetSpellInfo(spell)

[/quote]

我们只需要第三个返回值，它包含与这个法术相关的纹理的文件名。将旧版本的onSpellCast功能替换为这个新的改进的功能，它还会在您的聊天中显示一个图标。

[code=lua]

local castInfo = “|Hplayer:%1$s|h[%1$s]|h cast |T%s:0|t|cFF1D5FF|Hspell:%d|h[%s]|h|r (Cooldown: %d minutes)”

function onSpellCast(timer, player, spellId, spellName)

local texture - select(3, GetSpellInfo(spellId))

print(castInfo:format(player, texture, spellId, spellName, timer / 60))

CooldownMonitor.StartTimer(timer, player, spellName, texture)

end

[/code]

该代码在拼写链接前面添加了一个|T转义序列，并将纹理作为附加参数添加到string.format。这将在信息消息中添加一个小图标。纹理的路径现在也被传递到startTimer，因为纹理在时间上也是有用的。

还有一个小问题：如果我们的法术冷却时间只有1分钟会发生什么？文字仍然会说：“Cooldown： 1 minutes”。这个游戏为我们提供了一种处理这种情况的简单方法。

[b]· 语法转义序列(Grammatical Escape Sequence)[/b]

有三种转义序列可用来处理格式化文本时可能出现的语法问题。第一种格式如下。

[code=lua]

digit |1singular;plural1;plural2;

[/code]

如果前面的数字是1，它将显示singular，否则为plural1。plural2是可选的，如果提供的数字是2，他将被用来代替plural1。第二种形式的复数在俄语等语言中很有用。

这个转义序列只考虑它左边的数字。所以11 |1singular;plural;将导致11 singular。很少需要这个转义序列，下面这个更有用，因为它考虑了整个数字。

[code=lua]

number |4singular:plural1:plural2;

[/code]

它的工作原理与|1类似，但是使用整体来确定是使用singular还是plural1/2。plural2文本也是可选的。这是我们在格式字符串castInfo中需要的。让我们把那个改字符串改为下面的字符串：

[code=lua]

local castInfo = “|Hplayer:%1$s|h[%1$s]|h cast |T%s:0|t|cFF71D5FF|Hspell:%d|h[%s]|h|r (Cooldown: %d |4minute:minutes;)”

[/code]

[quote]

注意：|1使用分号分隔不同的形式，而|4需要冒号作为分隔符，但在末尾使用分号。

[/quote]

第三个转义序列是|2。格式就是|2sometext。如果sometext以元音开头，则显示为d’sometext，否则显示为de sometext。

[quote]

注意：这些语法转义序列不能在聊天信息中使用。

[/quote]

现在我们可以将漂亮的消息打印到聊天框中，functiononSpellCast也完成了。我们的下一个任务是构建CooldownMonitor.StartTimer函数，它将创建一个可视化计时器并将其显示在屏幕上。我们将在一个单独的文件中创建这个函数及其所有辅助函数(你可能已经猜到了它的名字：CooldownBars.lua)，以便在下一章中用一个库轻松替换它。对于这种冷却计数器来说，一个好的框架类型是状态栏。它代表一个进度条，可以被你在屏幕上看到的施法条、生命条和法力条使用。状态栏与滑块(sliders)非常相识，因为它们有最小值、最大值和当前设置的值。因为框架类型的名称，使用状态栏显示的计时器通常被称为“状态栏计时器”。

[b][size=130%]&#9675; 建立状态栏计时器(Building Status Bar Timer)[/size][/b]

让我们首先为这样的计时器构建一个XML模板。创建模板的第一个问题总是，是否已经有一个由暴雪定义的模板？答案是肯定的，文件Interface\FrameXML\CastingbarFrame.xml定义了模板CastingBarFrameTemplate，它适合于我们的目的。

[b]· 建立一个模板(Building a Templat)[/b]

在使用这个模板之前，我们需要对它做一些修改。默认的施法栏不显示计时器，所以我们必须添加计时器，最好是在框架右边作为一个小字体字符串。另一个问题是在这个施法条框体架中居中。我们想要的锚点在它的左边，因为我们需要额外的空间在我们的冷却计时器的右边半个框体。第三个问题是模板定义了一些特定于施法栏的脚本处理程序。我们将不得不改写它们。

我们将从这个已存在的模板派生我们的模板，因为它仍然比重写整个模板更容易。下面的代码显示了我们的新模板。在查看以下代码时，请在文件Interface\FrameXML\CastingbarFram.xml中打开模板CastingBarFrameTemplate。着使得理解完整的模板看起来更容易。

[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”>

<StatusBar name=”CooldownBarTemplate” virtual=”true” minValue=”0” maxValue=”1” inherits=”CastingBarFrameTemplate”>

<Size>

<AbsDimension x=”195” y=”13”/>

</Size>

<Layers>

<Layer level=”ARTWORK”>

<!—add a fontstring that shows the timer -->

<FontString name=”$parentTimer” justify=”RIGHT” inherits=”GameFontHighlight”>

<Anchors>

<Anchor point=”RIGHT”>

<Offset>

<AbsDimension x=”-6” y=”3”/>

</Offset>

</Anchor>

</Anchors>

</FontString>

</Layer>

</Layers>

<Scripts>

<OnShow/> <!--we do not want the old OnShow handler -->

<OnEvent/> <!-- the old OnEvent handler is also ont required -->

<OnLoad> <!-- modify the text and icon -->

\_G[self:GetName()..”Text”]:ClearAllpoints()

\_G[self:GetName()..”Text”]:SetPoint(“LEFT”,6,3)

\_G[self:GetName()..”Text”]:SetWidth(155)

\_G[self:GetName()..”Text”]:SetJustifyH(“LEFT”)

\_G[self:GetName()..”Text”]:ClearAllPoints()

\_G[self:GetName()..”Text”]:SetPoint(“RIGHT”, self, “LEFT”, -5, 2)

\_G[self:GetName()..”Text”]:SetWidth(20)

\_G[self:GetName()..”Text”]:SerHeight(20)

</OnLoad>

<OnUpdate> <!-- the function’s head is function(self, elapsed) -->

self.obj:Update(elapsed)

</OnUpdate>

</Scripts>

</StatusBar>

</Ui>

[/code]

我们的模板CooldownBarTemplate现在有以下几个子节点(children)：

[quote]

Text：固定在框架中心的字体字符串。这是由原始模板定义的，我们修改它的唯一方法是使用Onload处理程序。

Timer：剩下的时间，这个字体字符串是由模板添加的。

Border：框架周围的边框，由原始模板定义。

Icon:定时器左边的一个16x16纹理，它稍后会显示法术的图标。这也是由原始模板定义的。这个图标的大小在处理程序中更改为20x20，它的位置稍微调整了一下。

Spark：一个需要移动到状态栏当前位置的发光效果纹理。这也是由原始模板创建的。

Flash：可以用来在工具条上添加flash效果的纹理。该子组件是由原来模板创建的。

[/quote]

我们不会在XML文件中使用这个模板，因为我们不会用XML创建框体。没有办法预测插件需要多少定时器，可能同时存在10个CD或只有1个。我们将使用CreateFrame动态地从这个模板创建计时器。注意，它将生成应该错误消息，因为我们还没有实现OnUpdate方法。忽略此错误，或通过将其标记为注释来禁用OnUpdate方法。

[code=lua]

local f = CreateFrame(“StatusBar”, “TestTimer”, UIParent, “CooldownBarTemplate”)

\_G[f:GetName()..”Text”]:SetText(“Text Timer”)

\_G[f:GetName()..”Timer”]:SetText(“1:00”)

\_G[f:GetName()..”Flash”]:Hide()

f:SetPoint(“CENTER”)

[/code]

第一行创建一个StatuBar类型的新框体，它继承自模板。它的名字是TestTimer，它的父框体是UIParent。第二行访问框架的子Text，并调用此字体字符串上的SetText来设置测试文本。第三行对计时器执行相同的操作。第四行隐藏默认显示的纹理Flash。最后一行设置了一个点，这样我们的框架在屏幕中央就可见了。你可以试着和各种各样的孩子一起玩，就像图标一样。另一个有趣的测试是尝试将一个很长的字符串设置为文本，这样检查它是否被正确截断。

但定时器还不起作用。它不会倒计时，而且火花(spark)在中间而不是末尾。为此，我们将需要OnUpdate方法。它将通过调用其方法SetValue来更新状态栏的位置，将火花的位置设置为与该栏和计时器的文本相匹配。它也会在时间为0时隐藏它。

该功能存储在对象的方法更新中，对象的方法更新存储在状态栏的键self(the key self)下。也可以使用状态栏框架作为对象，但它已经有了自己的元表。我们必须将调用转发给这个原始的元表，因此在表示帧的表中存储对象会更容易一些。

[b]· 处理定时器(Handing the Timers)[/b]

表示一组状态栏计时器的良好数据结构是双向链表。它允许我们通过访问表中的字段来获取上一个计时器和下一个定时器。这允许我们在帧集中移除计时器。这个双向链表中的每一个条目(entry)都是一个计时器对象。

下面所有的Lua函数都应该放在文件CooldownBar.lua中。我们首先需要创建一些稍后使用的局部变量。

[code=lua]

-- the first and last timer objects in the double-linked list

local firstTimer, lastTimer

-- prototype for the timer object

local timer = {}

[/code]

现在我们需要构建实际的计时器对象。我们的构造函数是startTimer函数。它创建一个状态栏对象并显示它。实际对象也存储在这个框架(frame)中。这个对象持有一个框架的引用，因此我们可以很容易地获得属于这个对象的框架。这个对象被添加到表示一组计时器的双向链表中。

但是如果计时器过期了，框体被隐藏了，会发生什么呢？没有办法删除现有的框体(垃圾回收器不能删除框体)，因此重用它是一个好主意。我们需要将当前所有未使用的框体存储在某个地方，如果存在旧框体，构造函数将回收旧框体。堆栈是存储未使用框体的良好数据结构。我们将过期计时器的对象推入堆栈。这里的弹出操作与我们在前面示例中看到的堆栈略有不同。它永远不会返回nil，因为它只是创建一个新的框体，并在堆栈为空时返回它。我们将调用函数popOrCreateFrame来表示这种行为。

每个框体都需要一个名字，因为我们需要这个名字来获得框体的子框体。我们只是在这里使用了一个计数器，它计算每次弹出操作创建新框体的次数。

下面的代码片段展示了这样一个堆栈。它在这里被实现为一个链表，这意味着我们只将当前在堆栈顶部的对象存储在一个变量中(在本例中为frameStack)。然后这个对象有next字段，它指向堆栈上的下一个对象。

[code=lua]

local popOrCreateFrame, pushFrame

do

local id = 1

local frameStack -- the object on the top of the stack

-- pops a frame form the stack or creates a new frame

function popOrCreateFrame()

local frame

if frameStack then -- old frame exists

frame = frameStack -- re-use it...

-- ...and remove it from the stack by changing the object on the top

frameStack = frameStack.next

frame:Show() -- make sure that it’s shown as it might be hidden

else -- stack is empty...

-- ...so we have to create a new frame

frame = CreateFrame(“StatuBar”, “CooldownMonitor\_Bar”..id, CooldownMonitor\_Anchor, “CooldownBarTemplate”)

id = id + 1 -- increase the ID

end

return frame

end

-- pushes a frame on the stack

function pushFrame(frame)

-- delete the reference to the object to allow

-- the garbage collector to collect it

frame.obj = nil

-- the next object on the stack is the one that is currently on the top

frame.next = frameStack

-- the new object on the top is the new one

frameStack = frame

end

end

[/code]

我们在pop函数中引用一个当前未定义的框体。这个框体的名字是CoolownMonitor\_Anchor。它将是第一个bar的锚点，也是所有bar的父节点。第二个bar将被固定在第一个bar上，以此类推。这个锚点的位置是通过使用框架的SetUserPlaced方法保存的。将以下框架放置在CooldownMonitor.xml文件的<UI>标记中。把这些代码放在模板之前还是之后都没有关系。不要把它放在模板中，因为它是一个全新的框架，与我们的状态栏模板没有任何关系。

[code=xml]

<Frame name=”CooldownMonitor\_Anchor” movable=”true” parent=”UIParent”>

<Anchors>

<Anchor point=”CENTER”>

<Offset>

<AbsDimension x=”300” y=”0”/>

</Offset>

</Anchor>

</Anchors>

<Size> <!-- required in order to make the frame visible -->

<AbsDimension x=”1” y=”1”/> <!-- 0x0 would not work here!-->

</Size>

<Scripts>

<OnLoad>

self:SetUserPlaced(1)

</OnLoad>

</Scripts>

</Frame>

[/code]

[quote]

注意：没有有效大小的框体不仅是不可见的，它们也不能被其他框体作为锚点使用。0x0不是有效的大小。也可以使用多个跨越框架的锚点。

[/quote]

如果我们能够在Lua中创建框体并将其存储在一个局部变量中，那么它将变得更短。但是SetUserPlaced方法要求我们的框体有一个名称，因为该名称用于表示框体。因此，无论如何，我们需要一个全局变量。在使用Lua创建框架时，我们还需要注意默认的位置。在调用SerPoint创建框体后，我们不能覆盖保存的位置。这意味着在设置默认位置之前，我们必须检查是否已经有一个保存的位置。由于这些问题，在这里使用XML创建框架更容易。

现在我们有了框体模板和锚点，我们的下一个任务是构建框体对象和构造函数。

[b]· 状态栏计时器的构造(The Status Bar Timer Constructor)[/b]

这个构造函数创建一个带有相应框体的计时器对象，并将这个对象添加到双向链表中。它还是设置计时器上显示的文本、法术图标的纹理和计时器的颜色。我们将使用为计时器施放法术的玩家的职业颜色。我们需要一个助手函数，它接受一个玩家的名称并返回这个玩家的类。这个功能需要迭代整个团队(party)或副本(raid)来找到玩家。

下面的代码应该放在文件CooldownMonitor.lua的末尾：

[code=lua]

-- gets a class of a player in your party or raid

function CooldownMonitor.GetClassByName(name)

-- check if we are looking for ourselves

if UnitName(“player”) == name then

-- the first return value of UnitClass is localized, the second one is not

return select(2, UnitClass(“player”))

end

-- iterate over the party (if we are in a party)

for i = 1, GetNumPartyMembers() do

if UnitName(“party”..i) == name then

return select(2, UnitCalss(“party”..i))

end

end

-- still no match, iterate over the whole raid (if we are in a raid)

for i = 1, GetNumRaidMembers() do

-- no need to work with the GUID here as the player’s name is unique

if UnitName(“raid”..i) == name then

return select(2, UnitClass(“raid”..i))

end

end

-- that player isn’t part of our party/raid

return “unknown”

end

[/code]

现在我们可以创建CooldownMonitor.StartTimer函数，它类似于状态栏计时器对象的构造函数。它完成所有的困难功能，并创建一个新的状态栏计时器并启动它。把它放在CooldownBars.lua的末尾。

[code=lua]

local mt = {\_\_index = timer} -- metastable

-- the constructor

function CooldownMonitor.StartTimer(timer, player, spell, texture)

local frame = popOrCreateFrame() -- create or recycle a frame

local class = CooldownMonitor.GetClassByName(player)

-- set the color the status bar by using color informations from the table

-- RAID\_CLASS\_COLORS that contains the default colors of all classes

if RAID\_CLASS\_COLORS[class] then

local color = RAID\_CLASS\_COLORS[class]

frame:SetStatuBarColor(color.r, color.g, color.b)

else -- this should actually never happen

frame:SetStatusBarColor(1, 0.7, 0) -- default color from the template

end

-- set the text

\_G[frame:GetName()..”Text”]:SetFormattedText(“%s: %s”, player, spell)

-- and the icon

local ok = \_G[frame:GetName()..”Text”]:SetTexture(texture)

if ok then

\_G[frame:GetName()..”Icon”]:Show()

else -- hide the texture if it couldn’t be loaded for some reason

\_G[frame:GetName()..”Icon”]:Hide()

end

-- add a short flash effect by fading out the flash texture

UIFrameFadeOut(\_G[frame:GetName()..”Flash”], 0.5, 1, 0)

local obj = setmetatable({ -- this is the actual object

frame = frame, -- the frame is stored in the object...

totalTime = timer,

timer = timer -- this is the remaining time, it will be decremented later}, mt)

frame.obj = obj -- ...and the object in the frame

-- add the object to the end of the list

if firstTimer == nil then -- our list is empty

firstTimer ==obj

lastTimer = obj

else -- our list is not empty, so append it after the last entry

-- the element in front of our object is the old last element

obj.prev = lastTimer

-- the element after the old last element is our object

lastTimer.next = obj

-- the new last element is our object

lastTimer = object

end

obj:SetPosition()

obj:Update(0)

return obj -- return the object

end

[/code]

该函数从popOrCreateFrame获取框体，并通过设置其文本、图标和颜色来初始化它。然后它在纹理flash上使用UI函数UIFrameFadeOut，这导致一个flash效果。然后构造函数创建实际对象，并将其添加到双链表中。

然后调用对象的SetPosition方法，该方法将设置框体的位置。接下来我们将实现这个方法。它还调用参数为0的Update方法。这将设置状态栏的初始值和字体字符串计时器的文本。

[b]· 定位(Positioning)[/b]

让我们创建SetPosition方法(将其放在文件的末尾)。该方法只需要根据链表中框体的位置来设置点。如果它是列表中的第一个元素，它需要锚定到我们之前创建的锚点。否则，它使用前一框体作为锚点。

[code=lua]

function timer:SetPosition()

self.frame:ClearAllPoints()

if self == firstTimer then -- it’s the first timer

self.frame:SetPoint(“CENTER”, CooldownMonitor\_Anchor, “CENTER”)

else -- it’s not the first timer, anchor it to the previous one

self.frame:SetPoint(“TOP”, self.prev.frame, “BOTTOM”, 0, -11)

end

end

[/code]

这个方法说明了为什么在这里选择双向链表是明智的。我们可以只写self.prev来获取前一个计时器对象，而self.prev.frame来获取它的框体。

我们的下一个方法是Update方法，它更新框体。

[b]· 更新计时器(Updating Timers)[/b]

Update方法将对象的属性计时器减少自上次调用该对象以来经过的时间。如果剩余时间小于或等于0，该方法将取消计时器。如果不是这种情况，它将更新状态栏的值、计时器的文本和火花(spark)的位置。这里我们需要一个小的助手函数，将计时器格式化为我们可读的格式，然后显示出来。这个函数应该放在文件的末尾。

[code=lua]

local function stringFromTimer(t)

if t < 60 then -- less then 60 seconds --> don’t show minutes

return string.format(“%.1f”, t)

else -- 60 seconds or more remaining --> display minutes

return string.format(“%d:%0.2d”, t/60, t%60)

end

end

function timer:Update(elapsed)

self.timer = self.timer - elapsed

if self.timer <= 0 then -- time’s up

self:Cancel() -- cancel the timer

else

-- currentBarPos holds a value between 0 and 1

local currentBarPos = self.timer / self.totalTime

-- the min value of a status bar timer is 0 and the max value 1

self.frame:SetValue(currentBarPos)

-- update the text

\_G[self.frame:GetName()..”Timer”]:SetText(stringFromTimer(self.timer))

-- set the position of the spark

\_G[self.frame:GetName()..”Spark”]:SetPoint(“CENTER”, self.frame, “LEFT”, self.frame:GetWidth()\*currentBarPos, 2)

end

end

[/code]

[quote]

注意：OnUpdate脚本处理程序仅在显示框架时调用。所以我们可以假设计时器(timer)在运行。

[/quote]

我们的下一个任务是创建Cancel方法来取消计时器。这个Cancel方法必须负责从双向链表中移除对象并回收框体。

[b]· 取消计时器(Canceling Timers)[/b]

Cancel方法基本上只是从列表中删除一个计时器，隐藏框体，并将框体推到堆栈上，堆栈存储当前未使用的框体。这个函数应该放在Lua文件的末尾：

[code=lua]

function timer:Cancel()

-- remove it from the list

if self == firstTimer then

firstTimer = self.next

else

node.prev.next = node.next

end

if self == lastTimer then

lastTimer = self.prev

else

self.next.prev = self.prev

end

-- update the position of the next timer if there is a next timer

if self.next then

self.next:SetPosition()

end

self.frame:Hide() -- hide the frame...

pushFrame(self.frame) -- ...and recycle it

end

[/code]

插件现在可以正如我们所预期的那样正常工作。但仍有一个问题:我们目前无法将栏(bars)移动到另一个位置。

[b]· 移动计时器(Moving the Timers)[/b]

我们需要一个斜杠命令处理程序，显示一个可以移动的虚拟计时器。斜杠命令“移动”或“解锁”将启动移动过程。我们的斜杠命令处理程序将解锁框体45秒，并显示计时器显示剩余时间。我们还需要我们的计时库SimpleTimngLib在45秒后锁定框体，因此请确保它被安装，并将其作为依赖项添加。我们将使用锚的解锁属性来表示它已解锁。

下面的代码显示了斜杠命令，它应该放在Lua文件的末尾。

[code=lua]

local timingLib = simpleTimingLib:New()

local function lock()

CooldownMonitor\_Anchor.unlocked = false

end

SLASH\_COOLDOWNMONITOR1 = “/cooldownmonitor”

SLASH\_COOLDOWNMONITOR2 = “/cm”

SlashCmdList[“COOLDOWNMONITOR”] = function(msg)

local cmd = msg:trim():lower()

if cmd == “unlock” or cmd == “move” then

startTimer(45, “Status”, “unlocked”)

CooldownMonitor\_Anchor.unlocked = true

timingLib:Schedule(45, lock)

end

end

[/code]

注意：我们的插件现在依赖于SimpleTimingLib，所以您应该将他作为应该依赖项添加到TOC文件中。我们将在下一章中看到如何摆脱这种依赖，并将库嵌入到我们的插件中，这一章将详细解释库。

但是计时器不会因为我们在这里设置了一个属性而变得可移动。我们的模板中还需要一些脚本处理程序。我们需要在状态栏上使用RegisterForDrag方法来接受拖动鼠标事件。我们可以通过XML文件中的状态栏模板CooldownBarTemplate的Onload脚本处理程序中添加以下代码来包含它：

[code=xml]

self:RegisterForDrag(“LeftButton”)

[/code]

然后，我们需要将OnDragStart脚本处理程序添加到状态栏模板CooldownBarTemplate的<Scripts>脚本中。

[code=xml]

<OnDragStart>

if self:GetParent().unlocked then

self:GetParent().StartMoving()

self.GetParent().moving = self

end

</OnDragStart>

[/code]

处理程序检查是否设置了锚点的解锁(unlocked)属性(父属性)，如果是这种情况，则将父属性附加到鼠标上。我们还将启动移动过程的框体保存在锚点中。

我们需要告诉这个锚点停止移动，如果我们停止拖动或者当我们移动它的手框体被隐藏了。隐藏框体不会触发OnDragStop脚本，所以当我们移动框体的时候，如果我们点击的计时器过期了，框体会继续移动。

OnHide处理程序仅当用户单击发起移动的框体时才停止锚点。这可以防止当我们移动另一个计时器时，定位停止。

[code=xml]

<OnDragStop>

self:GetParent():StopMovingOrSizing()

</OnDragStop>

<OnHide>

if self:GetParent().moving == self then

self:GetParent():StopMovingOrSizing()

end

</OnHide>

[/code]

我们现在有了一个功能齐全的冷却时间监视器。但它有一个问题，我们没有考虑到一件重要的事情，而且我们的插件目前在某些情况下不能正常工作。你注意到那个bug了吗？

[b]· 修复上一个错误(Fixing the Last Bug)[/b]

我们目前依赖于框架的OnUpdate脚本处理程序计数计时器。但是谁能保证我们的状态栏计时器一直显示呢？如果它们因为任何原因被隐藏，它们将中止。例如，因为用户按下Alt+Z隐藏UIParent，它们可能被隐藏。或者用户可能在计时器运行时打开世界地图。你可以通过输入/cm move并打开世界地图来测试这个错误。计时器将在世界地图打开期间冻结。但SimpleTimingLib不会。所以在你关闭世界地图后，你仍然会看到“status:unlocked”计时器，但45秒可能已经结束，画面再次锁定。

这里有用的是框架对象的IsVisible方法。如果框体没有显示，或者它的父框体没有显示，它返回nil。这意味着如果我们的锚显示但不可见，我们可以检测世界地图是打开的还是UI隐藏的。我们创建了一个不是UIparent的子框体，并使用它的OnUpdate方法来检查我们是否丢失了Onupdate事件。把它放在文件CooldownBars的末尾。

[code=lua]

local updater=CreateFrame(“Frame”)

updater:SetScript(“OnUpdate”, function(self, elpased))

if CooldownMonitor\_Anchor:IsShow() and

not CooldownMonitor\_Anchor:IsVisible() then

local timer = firstTimer

while timer do

timer:Update(elapsed)

timer = timer.next

end

end

end

[/code]

注意：我们只实现了双向链表的几个函数。我们没有实现迭代器，因为它不值得为一次循环而努力。因此，我们在这里只使用了while循环。该循环从所有框体调用Update方法。

======

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

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

[b][size=150%]本[/size][/b]章讨论了如何使用战斗日志。你现在可以创建响应战斗事件的插件。我们还讨论了位域，以及它们如何在《魔兽世界》的战斗日志事件中使用。许多流行的插件，如伤害统计(Damage Meters)或滚动战斗文字(Scrolling Combat Text)，只处理战斗日志事件。你现在可以使用这些事件来写你自己的战斗日志插件，这里有很多令人兴奋的东西。

我们在本章中所编写的示例插件是一个功能齐全的冷却监视器，带有相当不错的冷却时间显示。下一章是关于库(libraries)和如何使用它们。作为一个演示，我们将用库替换状态栏计时器的实现。这些库将给我们的插件带来一个全新的外观(有很多漂亮的东西)，并添加一些其他令人兴奋的功能，如配置菜单。