[Lua的线程和状态](http://www.jellythink.com/archives/616)

2014-09-08 分类：[Lua](http://www.jellythink.com/archives/category/language/lua) 阅读(2534) 评论(5)

那不是真的多线程

Lua不支持真正的多线程，这句话我在《[Lua中的协同程序](http://www.jellythink.com/archives/508)》这篇文章中就已经说了。根据我的编程经验，在开发过程中，如果可以避免使用线程，那就坚决不用线程，如果实在没有更好的办法，那就只能退而用之。为什么？首先，多个线程之间的通信比较麻烦，同时，线程之间共享内存，对于共享资源的访问，使用都是一个不好控制的问题；其次，线程之间来回切换，也会导致一些不可预估的问题，对性能也是一种损耗。Lua不支持真正的多线程，而是一种协作式的多线程，彼此之间协作完成，并不是抢占完成任务，由于这种协作式的线程，因此可以避免由不可预知的线程切换所带来的问题；另一方面，Lua的多个状态之间不共享内存，这样便为Lua中的并发操作提供了良好的基础。

多个线程

从C API的角度来看，将线程想象成一个栈可能更形象些。从实现的观点来看，一个线程的确就是一个栈。每个栈都保留着一个线程中所有未完成的函数调用信息，这些信息包括调用的函数、每个调用的参数和局部变量。也就是说，一个栈拥有一个线程得以继续运行的所有信息。因此，多个线程就意味着多个独立的栈。

当调用Lua C API中的大多数函数时，这些函数都作用于某个特定的栈。当我们调用lua\_pushnumber时，就会将数字压入一个栈中，那么Lua是如何知道该使用哪个栈的呢？答案就在类型lua\_State中。这些C API的第一个参数不仅表示了一个Lua状态，还表示了一个记录在该状态中的线程。

只要创建一个Lua状态，Lua就会自动在这个状态中创建一个新线程，这个线程称为“主线程”。主线程永远不会被回收。当调用lua\_close关闭状态时，它会随着状态一起释放。调用lua\_newthread便可以在一个状态中创建其他的线程。

lua\_State \*lua\_newthread(lua\_State \*L);

这个函数返回一个lua\_State指针，表示新建的线程。它会将新线程作为一个类型为“thread”的值压入栈中。如果我们执行了：

L1 = lua\_newthread(L);

现在，我们拥有了两个线程L和L1，它们内部都引用了相同的Lua状态。每个线程都有其自己的栈。新线程L1以一个空栈开始运行，老线程L的栈顶就是这个新线程。

除了主线程以外，其它线程和其它Lua对象一样都是垃圾回收的对象。当新建一个线程时，线程会压入栈，这样能确保新线程不会成为垃圾，而有的时候，你在处理栈中数据时，不经意间就把线程弹出栈了，而当你再次使用该线程时，可能导致找不到对应的线程而程序崩溃。为了避免这种情况的发生，可以保持一个对线程的引用，比如在注册表中保存一个对线程的引用。

当拥有了一个线程以后，我们就可以像主线程那样来使用它，以前博文中提到的对栈的操作，对这个新的线程都适用。然而，使用多线程的目的不是为了实现这些简单的功能，而是为了实现协同程序。

为了挂起某些协同程序的执行，并在稍后恢复执行，我们可以使用lua\_resume函数来实现。

int lua\_resume(lua\_State \*L, int narg);

lua\_resume可以启动一个协同程序，它的用法就像lua\_call一样。将待调用的函数压入栈中，并压入其参数，最后在调用lua\_resume时传入参数的数量narg。这个行为与lua\_pcall类似，但有3点不同。

1. lua\_resume没有参数用于指出期望的结果数量，它总是返回被调用函数的所有结果；
2. 它没有用于指定错误处理函数的参数，发生错误时不会展开栈，这就可以在发生错误后检查栈中的情况；
3. 如果正在运行的函数交出（yield）了控制权，lua\_resume就会返回一个特殊的代码LUA\_YIELD，并将线程置于一个可以被再次恢复执行的状态。

当lua\_resume返回LUA\_YIELD时，线程的栈中只能看到交出控制权时所传递的那些值。调用lua\_gettop则会返回这些值的数量。若要将这些值移到另一个线程，可以使用lua\_xmove。

为了恢复一个挂起线程的执行，可以再次调用lua\_resume。在这种调用中，Lua假设栈中所有的值都是由yield调用返回的，当然了，你也可以任意修改栈中的值。作为一个特例，如果在一个lua\_resume返回后与再次调用lua\_resume之间没有改变过线程栈中的内容，那么yield恰好返回它交出的值。如果能很好的理解这个特例是什么意思，那就说明你已经非常理解Lua中的协同程序了，如果你还是不知道我说的这个特例是什么意思，请再去读一遍《[Lua中的协同程序](http://www.jellythink.com/archives/508)》，如果你还不懂，那你就在下放留言吧（提醒：这个特例主要利用的是resume-yield之间的传参规则）。

现在，我就通过一个简单的程序来做个试验，以便更好的理解Lua的线程。使用C代码来调用Lua脚本，Lua函数作为一个协同程序来启动，这个Lua函数可以调用其它Lua函数，任意的一个Lua函数都可以交出控制权，从而使lua\_resume调用返回。对于使用C调用Lua不熟悉的伙计，请再去仔细的读读《[Lua与C](http://www.jellythink.com/archives/554)》和《[C“控制”Lua](http://www.jellythink.com/archives/560)》这两篇文章吧。先贴上重要的代码吧。下面是Lua代码：

function Func1(param1)

Func2(param1 + 10)

print("Func1 ended.")

return 30

end

function Func2(value)

coroutine.yield(10, value)

print("Func2 ended.")

end

下面是C++代码：

lua\_State \*L1 = lua\_newthread(L);

if (!L1)

{

return 0;

}

lua\_getglobal(L1, "Func1");

lua\_pushinteger(L1, 10);

// 运行这个协同程序

// 这里返回LUA\_YIELD

bRet = lua\_resume(L1, 1);

cout << "bRet:" << bRet << endl;

// 打印L1栈中元素的个数

cout << "Element Num:" << lua\_gettop(L1) << endl;

// 打印yield返回的两个值

cout << "Value 1:" << lua\_tointeger(L1, -2) << endl;

cout << "Value 2:" << lua\_tointeger(L1, -1) << endl;

// 再次启动协同程序

// 这里返回0

bRet = lua\_resume(L1, 0);

cout << "bRet:" << bRet << endl;

cout << "Element Num:" << lua\_gettop(L1) << endl;

cout << "Value 1:" << lua\_tointeger(L1, -1) << endl;

上面的程序，你可以先运行一下；你能想到运行结果么？单击这里下载完整工程[LuaThreadDemo.zip](http://www.400gb.com/file/72855535)。

上面的例子是C语言调用Lua代码，Lua可以自己挂起自己；如果Lua去调用C代码呢？C函数不能自己挂起它自己，一个C函数只有在返回时，才会交出控制权。因此C函数实际上是不会停止自身执行的，不过它的调用者可以是一个Lua函数，那么这个C函数调用lua\_yield，就可以挂起Lua调用者：

int lua\_yield(lua\_State \*L, int nresults);

你没有听错，C代码调用lua\_yield不能挂起自己，但是它却可以将它的Lua调用者挂起。其中nresults是准备返回给相应resume的栈顶值的个数，当协同程序再次恢复执行时，Lua调用者会收到传递给resume的值。lua\_yield在使用时，只能作为一个返回的表达式，而不能独自使用。比如：

return lua\_yield(L, 0);

对于多线程编程，本身就是麻烦的问题，而这里枯燥的文字总结，也会没有效果，下面来一个简短的例子。先贴Lua代码，这段代码需要结合C代码一起看，否则就是云里雾里的。

require "lua\_yieldDemo"

local function1 = function ()

local value

repeat

value = Module.Func1()

until value

return value

end

local thread1 = coroutine.create(function1)

-- 现在运行到了Module.Func1()

-- 100这个值将会被赋值给value

coroutine.resume(thread1)

--print(coroutine.status(thread1))

-- 设置C函数环境

Module.Func2(10)

print(coroutine.resume(thread1))

C代码如下：

// 判断环境表中JellyThink是否被设置了

static int IsSet(lua\_State \*L)

{

lua\_getfield(L, LUA\_ENVIRONINDEX, "JellyThink");

if (lua\_isnil(L, -1))

{

printf("Not set\n");

return 0;

}

return 1;

}

static int Func1(lua\_State \*L)

{

// 没有被设置就挂起

if (!IsSet(L))

{

printf("Begin yield\n");

return lua\_yield(L, 0);

}

// 被设置了，就取值，返回被设置的值

printf("Resumed again\n");

lua\_getfield(L, LUA\_ENVIRONINDEX, "JellyThink");

return 1;

}

// 设置JellThink的值

static int Func2(lua\_State \*L)

{

luaL\_checkinteger(L, 1);

// 设置到环境表中

lua\_pushvalue(L, 1);

lua\_setfield(L, LUA\_ENVIRONINDEX, "JellyThink");

return 0;

}

当我在Lua中调用coroutine.resume时，我都只传递了一个参数，其它参数都没有；这里需要注意，如果我传值了，就相当于给value赋值了。当我恢复thread1运行时，它是从Module.Func1()返回处继续执行，也就是对value赋值，而这里赋予value的值实际上是传给resume的值。上面的代码中，我没有传值，如果传了，就无法验证我设置的10了。单击这里下载完整工程[lua\_yieldDemo.zip](http://www.400gb.com/file/72855534)。Any question? No? OK, Next.

Lua状态

每次调用luaL\_newstate（或者lua\_newstate）都会创建一个新的Lua状态。不同的Lua状态是各自完全独立的，它们之间不共享任何数据。这个概念是不是很熟悉，是不是特别像Windows中的进程的概念。也就是说，在一个Lua状态中发生的错误也不会影响其它的的Lua状态，windows的进程也是这样的。并且，Lua状态之间不能直接沟通，必须写一些辅助代码来完成这点。

由于所有交换的数据必须经由C代码中转，所以只能在Lua状态间交换那些可以在C语言中表示的类型，例如字符串和数字。由于Lua状态我目前没有使用过，也就没有足够的信心和资格去总结这个东西，还是怕会误导大家，如果以后在实际项目中使用了Lua状态，我还会回过头来总结Lua状态的。相信我，我还会回来的。

总结

哦，这篇文章拖的时间够长的啊。由于最近项目紧，赶着上线，很忙啊，加班啊。又赶上中秋节，也没有太多的时间来写。这篇就这样的，对于Lua状态的总结，还是不够深刻，或者说，基本就没有。哦，算了，后续在总结吧，也不能，也不可能一口吃成一个胖子的。中秋快乐，各位。

2014年9月8日 于深圳。