动态脚本语言Lua与C++交互方法的研究与实现①

邓正阳　陈和平　苏　鹏 (武汉科技大学 信息科学与工程学院 湖北 武汉 430081)

摘　要： 针对当前利用 C++开发大型复杂程序代价高、周期长的问题，为了充分发挥 Lua 动态脚本语言的特点，重点探讨了如何在 Lua 脚本中模拟消息事件机制和 Lua 访问 C++接口的具体实现方法以及 Lua 调试器的设计方案。

关键词： 动态脚本语言；Lua；消息事件机制；C++接口；调试器

Research and Realization of Lua and C + + Cross-Method

DENG Zheng-Yang, CHEN He-Ping, SU Peng

(College of Information Science and Technology, Wuhan University of Science and Technology, Wuhan 430081, China)

Abstract: To make full use of the characteristics of Lua dynamic scripting language for the current development of large and complex programs, which has the problem of high cost, this article focuses on how to simulate the message mechanism based on event in the Lua script and visiting the C + + interface. The design of Lua debugger is also presented.

Keywords: dynamic scripting languages; Lua; message mechanism based on event; C + + interface; debugger

目前，随着软件行业的发展，如何高效地开发复杂的大型系统已成为业界非常关注的问题。由于 C++是一种静态语言，当系统达到一定复杂度时，会给开发人员带来较大的困难，导致软件开发效率低、成本高。而动态脚本语言 Lua 的出现，大大降低了软件开发代价，并且使得程序设计人员也能够参与到软件开发工作中来。

1　Lua脚本语言概述

动态脚本语言是一种在执行期间才检验数据类型的程序设计语言。通过脚本能够动态改变主程序的逻辑行为，而不需要重新编译整个主程序。

Lua (下载 http://www.lua.org/download.html)是免费的、轻量级的、独立的、可扩展的嵌入式程序语言。Lua 是基于关联数组和可扩展语法结构设计的语言，具有变量无类型、动态定义类型、面向对象结构、编译产生中间代码和内存自动回收等特点 [1,2]。

Lua 没有 "main" 程序的概念：它只能嵌入一个宿主程序中工作，该宿主程序被称作 embedding program 或简称为 host。宿主程序可以通过调用函数执行一小段 Lua 代码，可以读写 Lua 变量，可以注入 C 函数让 Lua 代码调用。这些扩展的 C 函数，可以较大地扩展 Lua 能处理事务的领域，而且共享一个统一的句法格式框架。Lua 还提供了一个保证独立的 Lua 解释器。

Lua 由标准 C 编写而成，代码简洁优美，几乎在所有操作系统和平台上都可以编译，运行。Lua 语法清晰简单，功能强大，可移植性高，能和 C/C++等语言结合紧密，在诸多脚本语言中占有较大的优势。随着在大型网游魔兽世界中的应用，近年来 Lua 已发展成非常热门的语言，深受广大编程人员的欢迎。Lua 不仅作为扩展脚本，也可以作为普通的配置文件，代替 XML,INI 等文件格式，并且更容易理解和维护。后者只是提供静态的参数值的定义，而 Lua 能以函数形式来控制变量等之间的逻辑关系，具有更大的灵活性。

Lua 的主要优点：

(1) Lua 在诸多脚本语言中运行时速度最快而且内存占用最少。

① 收稿时间:2009-08-28;收到修改稿时间:2009-10-05

(2) 集成 Lua 只会增加极少的内存占用率。

(3) 与其它语言的集成性很好。

Lua 的主要缺点：

(1) Lua 缺少完善的文档资料。

(2) Lua 内建的功能很少，并没有对创建大型的复杂应用程序提供足够支持。

若使用者不要求使用 Lua 来单独开发复杂的程序，而想与 C/C++集成，并要求速度快、资源占用率低，Lua 便是不错的选择。

2 Lua与C++交互的实现方法

初始化 Lua：

```
lua_State *L = lua_open(); //创建 Lua 环境
void InitLua()
{
luaL_openlibs(L);     //载入所有 lua 标准库
if (luaL_dofile(L,"test.lua")) //加载 Lua 文件是//否成功
cout << "can't load test.lua" << endl;//打印失败
//信息
}
```

首先，在主程序初始化函数中加载 Lua 库，然后调用 luaL_dofile()来执行 Lua 文件，最后调用 Lua_close(L)释放 Lua(Test.lua 文件的相对路径是可执行文件目录)。

Lua 与 C++的交互分为两部分，第一部分是 Lua 访问 C++的类或函数，第二部分是 C++访问 Lua 函数。如图 1 所示：

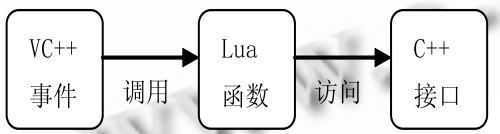

图 1 Lua 与 C++的交互图

Lua 被嵌入到 C++程序中，起到一个桥梁的作用，其功能由 C++提供，它只负责调用这些功能，结合自身的优点来快速重组它们，以此来控制各函数及模块之间的逻辑关系。下面介绍如何利用 Lua 脚本来实现 C++程序的逻辑控制。

2.1 Lua 中模拟 C++消息事件机制

首先，实现 Lua 中的消息机制，实际上是 C++调用 Lua 函数的过程。以 VC++中的重绘函数为例，在 OnPaint()中发送一个事件给 Lua，即当 VC++重绘时，Lua 中也会触发相应的重绘函数。这样就只需要在脚本文件中编写代码，而省去了 C++复杂的语法规则及漫长的编译过程。编程时，先在 Lua 中自定义函数：

```
function OnPaintInLua(dc) //Lua 中的重绘函数

end
```

然后在 VC++中的 OnPaint()函数中调用 OnPaintInLua()。具体步骤如下：

第一，将被调用的函数入栈，在 OnPaint()中调用 lua_getglobal(L,"func")，其中 func 是 Lua 函数名，此时还可调用 lua_isfunction(L,-1)来判断 func 是否为一函数；

第二，依次将所有参数入栈，lua_pushnumber(L, Pa)将数值型参数 Pa 压栈，lua_pushboolean(L, Pa)将布尔型参数 Pa 压栈，lua_pushlstring(L, Pa)将字符串参数 Pa 压栈，并可以同时将多个参数传递至 Lua 函数中；

第三，使用 lua_pcall(L, nCount, 0, 0)调用 Lua 函数，其中 nCount 表示参数的个数；最后，从栈中获取函数执行返回的结果。

此后，主程序重绘时便会触发 Lua 中的 OnPaintInLua()函数，这便实现了 Lua 中的重绘消息机制，类似地，可以实现其它的事件机制(如鼠标、键盘消息等)。

2.2 Lua 调用 C++接口

Lua 调用 C++ 函数，必须遵循相应协议以传递参数和获得返回结果。首先必须注册 C++函数，也就是说，必须把 C++ 函数的地址以适当的方式传递给 Lua 解释器。当 Lua 调用 C++ 函数时，C++函数从交互栈中获取从 Lua 传递过来的参数，调用结束后将返回结果放入栈中，同时还会返回结果的个数。

在 Lua 中注册的函数必须有同样的原型，该原型声明定义是 lua.h 中的 lua_CFunction：typedef int (*lua_CFunction) (lua_State *L);

C++ 函数接受的参数都是 Lua state，返回值表示返回结果的个数。注册 C++函数由下面两个函数完成：

```
lua_pushcfunction(L,cFun);
lua_setglobal(L,"LuaFun");
```

第一个函数把 cFun 函数入栈，第二个函数把 cFun 值赋给全局变量 LuaFun，即 LuaFun 是 cFun 在 Lua

中的别名。在 Lua 中调用 LuaFun 函数时会相应地调用 cFun 函数。其中 cFun 表示已定义的 C++函数。

其次，封装 C++函数 cFun：

```
static int cFun(lua_State *L)
{
lua_gettop(L);          //取得参数个数
lua_Number x = 0;    //定义变量 x
lua_Number y = 0;    //定义变量 y
if (!lua_isnumber(L, 1)) cout << "param must be a number" << endl; //判断第一个变量是否为数字
x = lua_tonumber(L, 1); //获取第一个参数//赋给变量 x
lua_pushnumber(L, x);   //把第一个返回值进栈
if (!lua_isnumber(L, 2)) cout <<
" param must a  number" << endl;//判断第二个//变量是否为数字
y = lua_tonumber(L, 2); //获取第二个参数//赋给变量 y
lua_pushnumber(L, y);   //把第二个返回值进栈
lua_pop(L,2);    //删除堆栈中的内容
return 2;        //返回两个值
}
```

最后，实现在 Lua 中调用 C++函数：

```
function OnPaintInLua(dc)
cFun(Pa1, Pa2) //C++函数
end
```

当主程序重绘时将调用 Lua 中的 OnPaintInLua()函数，进而在该函数中调用已封装好的 C++处理函数 cFun()。从而实现了 C++和 Lua 之间的交互访问。

另外，一种更好的办法是，将 C++函数封装成一个类。首先把封装好的 C++类注册给 Lua，即存入 Lua 的 table 中，接着为表中的每个元素(即类)定义一个元表 Metatable，把类中的成员函数插入到元表中去。此时在 Lua 中创建一个类对象时，调用该对象的 getmetatable()即可得到该类对象中的所有成员函数名，这样即符合 C++中的面向对象思维，也使得 Lua 中的代码更容易维护和理解。如封装 CDC 类的库函数，在此定义一个 C++类 LDC，在类中添加成员函数：

```
static void DrawLine(lua_State *L) //此//函数用来画线
{
……
lua_Number x1 = lua_tonumber(L, 1); //获取第//一个参数
…依次获取另外几个坐标值…;
HDC hDC = (HDC) lua_tonumber(L, 5)    //第//五个参数为设备描述表 DC
CDC* pDC = CDC::FromHandle(hDC);//从//HDC 类型转换为 CDC 类型
if(pDC)
{//从(x1, y1)画直线至(x2, y2)
pDC->MoveTo(x1, y1);
pDC->LineTo(x2, y2);
}
}
```

在 Lua 中：

```
function OnPaintInLua(dc)
ldc = LDC.new(); //ldc 得到 LDC 类对象
ldc:Drawline(x1, y1, x2, y2, dc);//调用//LDC 的成员函数 Drawline 画线
end
```

3　Lua单步断点调试器的设计

Lua 功能的强大无庸置疑，但是编辑 Lua 时也存在较大的问题：语法简单随意、动态数据类型、不如 C/C++严谨，所以在运行时出现问题也难以查找，因为它不会对拼写错误进行输出，在这种情况下，程序照样能够顺利执行，只是不能达到预期的效果。因此，在实际开发中会遇到脚本如何调试的难题。Lua 没有自带的调试器，而只是提供了一套 Debug 调试库。所以 Lua 调试器的设计对 Lua 开发者来说是必不可少的。

Lua 中用 debug.sethook 来设置 Hook，有四种情况可以触发一个 hook 的事件[3]：当 Lua 调用一个函数时，call 事件发生；每次函数返回时，return 事件发生；Lua 开始执行代码的新行时，line 事件发生；运行指定数目的指令后，count 事件发生。

另外，debug 库中主要的自省函数 debug.etinfo(foo)能够返回 foo 函数信息表，它能记录一些有用的错误信息、函数类型、函数行号、及函数本身等。debug 库的 getlocal 函数可以访问任何活动状态的局部变量。该函数有两个参数：将要查询的函数的栈级别和变量的索引。函数有两个返回值：变量名

和变量当前值。

调试器的简要设计：

主要实现功能：

F5---进入调试状态，

F9---设置行断点

F10---跳过函数单步调试

F11---不跳过函数单步调试

并可实时查询当前变量及函数的值及相关信息。

首先，从 CEditView 派生一个视类 CLuaEditew，用来显示 Lua 文本。新建 Lua 文件 Debug.lua，并在该文件中定义：

```
Function BeginDebug()//注册 hook
debug.sethook(OnDebugTrace, "l")
end
```

其次，在 CluaEditView 类 OnKeyDown()函数中判断，若按下 F5 键，便调用 Lua 中的 BeginDebug()函数。每当 line 事件发生时，会触发 OnDebugace()函数进入调试状态。

接下来，在 Lua 中的 OnDebugTrace()函数内判断是否有断点，如有，便循环调用 io.write("(ldb)")和 local RevCmd = io.read()使程序处于中断、待输入状态。

```
if RevCmd == "F5" then //按下 F5 键
程序直接跳到下一个断点；
Else
清除掉断点标记，并调用 debug.sethook()来退出调试；
End
if RevCmd == "F9" then //按下 F9 键
获取 CeditView 中当前行号，并赋值给 行标记变量；
End
if RevCmd == "F10" then //按下 F10 键
程序触发 line 事件，并重新设置相关变量值；
End
if RevCmd == "F11" then //按下 F11 键
if debug.getinfo(n).fun 为新函数 then
进入函数并单步调试，函数返回时触发 return 事件；
Else
与 F10 相同的操作；
end
end
```

最后，在 Lua 中调用 debug.getinfo(foo)和 getlocal(n)封装函数来获取调试过程中的变量及函数的相关信息。在程序初始化时加载 Debug.Lua 文件，在 CluaEditView 中调用其 Lua 函数来获取一些重要调试信息，并在输出窗口中输出(界面效果仿照 VS2008 的调试输出窗口)。效果如图 2：

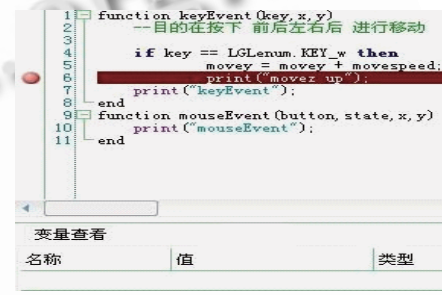

图 2 Lua 调试器效果图

4 结语

由于已有的脚本语言较多，高效率脚本语言的选取是开发的关键。Lua 脚本能够将程序核心和逻辑控制分离开来，在软件开发过程中，只需要提供丰富的 C++接口，便可开发出复杂的应用系统。本文从实际应用出发，在开发游戏引擎应用系统中，将 Lua 集成到 C++中，能够在较大程度上减少程序员的编程负担。

参考文献

1 巴克兰德.游戏人工智能编程案例精粹,罗岱,等译.北京:人民邮电出版社, 2008.

2 Jung K. Lua 程序设计初阶 Beginning Lua Programming. John Wiley & Sons 2007.

3 Ierusalimschy R. Programming in Lua, Second Edition, 周惟迪译.北京:电子工业出版社, 2008.

4 麦克沙夫瑞.游戏编程全接触 Game Coding Complete, 冯兆麟,孔祥一,李华杰译.北京:人民邮电出版社, 2006.