Luabind使用手册*

翻译：missdeer†

2010 年 11 月 27 日

*http://www.rasterbar.com/products/luabind/docs.html，for Luabind v0.9.1

†blog:http://www.missdeer.com/，Email:missdeer@gmail.com

目录

表格

摘要

作者	Daniel Wallin, Arvid Norberg
版权	版权所有Daniel Wallin, Arvid Norberg 2003.
日期	2006-01-11
版本	1.26.2.5
翻译	missdeer 于2009-11-7 v0.1

呃，只是为了强迫自己仔细从头读一遍Luabind使用手册!

— missdeer

1 介绍

Luabind是个帮助你创建C++和Lua间绑定的库。它可把用C++写成的函数和类导出到Lua。它也可以支持在Lua中定义类，并让类从其他Lua类或C++类派生。Lua类可以覆写C++基类中的虚函数。它是为Lua 5.0编写的，不支持Lua 4.

它使用模板元编程实现。这意味着你在编译你的工程前不需要进行预处理（编译器已经帮你完成了）。这也意味着你（通常）不需要知道每个你注册的函数的准确的签名，因为库会根据编译期的函数类型（包括签名）生成代码。本方案唯一的缺点是文件进行注册时编译时间会变长，所以建议你把所有的注册工作都放在同一个cpp文件中。

Luabind以MIT license[1]发布.

我们很乐意听到有关工程使用Luabind，如果你的项目使用了Lua，请告诉我们。

获取帮助和反馈的主要途径是luabind mailing list[2]。还有一个IRC频道#luabind在irc.freenode.net 上。

2 特性

Luabind 支持:

- 自由函数重载
- Lua中使用C++类
- 成员函数重载
- 操作符
- 属性
- 枚举
- C++中使用Lua函数
- C++中使用Lua类
- Lua类（单继承体系）
- 从Lua类或C++类派生
- 重载C++类中的虚函数
- 已注册类型间的隐匿转换
- 最佳签名匹配
- 返回值策略和参数策略

[1] http://www.opensource.org/licenses/mit-license.php
[2] https://lists.sourceforge.net/lists/listinfo/luabind-user

3 可移植性

经过测试Luabind可工作于以下编译器:

- Visual Studio 7.1
- Visual Studio 7.0
- Visual Studio 6.0 (sp 5)
- Intel C++ 6.0 (Windows)
- GCC 2.95.3 (cygwin)
- GCC 3.0.4 (Debian/Linux)
- GCC 3.1 (SunOS 5.8)
- GCC 3.2 (cygwin)
- GCC 3.3.1 (cygwin)
- GCC 3.3 (Apple, MacOS X)
- GCC 4.0 (Apple, MacOS X)

已确认不能工作的:

- GCC 2.95.2 (SunOS 5.8)

Metrowerks 8.3 (Windows) 可以编译，但在const测试中失败。就是说const成员函数会被当成non-const成员函数。

如何你在其他没有列在这里的编译器中进行了测试，请告诉我们结果。

4 构建Luabind

4.1 事先准备

Luabind依赖Boost 1.34 中的很多库。同时它还依赖 Boost Jam 和 Boost Build V2来构建库并运行测试。Boost 提供了多种平台的预编译的 bjam[3] 二进制文件。如果没有你使用的平台的预编译二进制文件，你需要自己编译一个[4]。

[3] http://sourceforge.net/project/showfiles.php?group_id=7586&package_id=72941
[4] http://www.boost.org/doc/libs/1_36_0/doc/html/jam/building.html

4.2 Windows

首先需要设置LUA_PATH 环境变量，用于指定 Lua 头文件所在目录和编译好的库所在目录。为了能运行测试套，建议直接从 Lua Binaries [5] 处获取 Windows x86 DLL 和包含头文件的包。

然后，需要环境变量BOOST_ROOT 用于指定 Boost 安装目录。

4.3 Linux和其他*nix

如果你的系统中已经安装了Lua，那么构建系统能很顺利地自动找到它。如果你把Lua安装在非标准位置，你需要设置 Lua_PATH 以指定安装位置。

BOOST_PATH用于指定 Boost 安装目录。如果不指定，构建系统会尝试使用标准包含路径中的 Boost 头文件。

4.3.1 Mac OS X

如果你同时安装了10.4和10.5版本的SDK，Boost Build 会默认使用 10.4。Lua 会与 10.5 SDK 链接（至少从 MacPorts 安装的 Lua 如此）。如果 Luabind 由于链接错误而构建失败，你需要显式地指定用 10.5 SDK 构建:

```
$ bjam macosx-version=10.5
```

4.4 构建和测试

构建库的默认的变种，是一个 shared debug 库，可以简单地在 luabind 根目录下运行 bjam 得到:

```
$ bjam
...patience...
...found 714 targets...
...updating 23 targets...
```

当在Linux下使用GCC构建，结果如下:

```
bin/gcc-4.2.3/debug/libluabind.so
```

在Windows下则生成一个dll和匹配的导入库。

要运行单元测试，则使用 bjam 生成目标 test :

```
$ bjam test
```

这将以4种变种构建并运行单元测试: debug 、release 、debug-static-lib 、release-static-lib。一次干净的测试运行结果应该以这样的内容结束:

[5] http://luabinaries.luaforge.net/

```
... updated xxx targets...
```

失败的运行将以如下的结束结束:

```
...failed updating xxx target...
...skipped xxx targets...
```

如果你不使用 Boost Build 构建你的程序，又想使用 shared 库，就需要定义 LUABIND_DYNAMIC_LINK 以导入符号。

5 基本用法

使用 Luabind，你必须包含 lua.h 和 luabind 主头文件:

```
extern "C"
{
    #include "lua.h"
}

#include <luabind/luabind.hpp>
```

这个包含可以支持类和函数的注册。如果你只想要支持函数或只支持类，你可以分别包含 luabind/function.hpp 和 luabind/class.hpp :

```
#include <luabind/function.hpp>
#include <luabind/class.hpp>
```

你首先要做的是调用 luabind::open(lua_State*) ，这会注册那些用于从 Lua 中创建类的函数，并初始化一些 luabind 使用的全局 state 结构。如果你不调用这个函数，随后库里面就会报断言失败。它没有对应的关闭函数，因为一旦一个类被注册到 Lua 中，确实没什么好的办法把它移除。还有部分原因是那些类的仍然存在的实例会依赖于这些类。所有的东西都会在 state 关闭时被清除。

Luabind的头文件从来不直接包含lua.h , 但可以通过 <luabind/lua_include.hpp>包含。如果你因为某些原因需要包含其他 Lua 头文件，可以修改这个文件。

5.1 Hello world

```
#include <iostream>
#include <luabind/luabind.hpp>

void greet()
{
    std::cout << "hello world!\n";
```

```
}

extern "C" int init(lua_State* L)
{
    using namespace luabind;

    open(L);

    module(L)
    [
        def("greet", &greet)
    ];

    return 0;
}
```

```
Lua 5.0  Copyright (C) 1994-2003 Tecgraf, PUC-Rio
> loadlib('hello_world.dll', 'init')()
> greet()
Hello world!
>
```

6 作用域

在Lua中注册的每个东西都是注册在名字空间（Lua 表）或全局作用域（称为 module ）中。所有注册必须在它的作用域中。定义一个 module 要用到 luabind::module 类，如下所示:

```
module(L)
[
    // 声明
];
```

这会将所有声明的函数或类都注册到 Lua 的全局名字空间。如果你想要为你的 module 有个自己的名字空间（像标准库那样）你可以给构造函数一个名字，像这样:

```
module(L, "my_library")
[
    // 声明
];
```

这样所有的声明会被放到 my_library 表中。

如果你要内嵌名字空间，可以使用 luabind :: namespace_ 类。它严格按照 luabind :: module 一样工作，除了它的构造函数不用 lua_State*。它的用法可以看下面这个例子:

```
module(L, "my_library")
[
    // 声明

    namespace_("detail")
    [
        // 库的私有声明
    ]
];
```

你可能已经发现，下面这2个声明是等价的:

```
module(L)
[
    namespace_("my_library")
    [
        // 声明
    ]

];


module(L, "my_library")
[
    // 声明
];
```

每个声明必须用逗号分隔，像这样:

```
module(L)
[
    def("f", &f),
    def("g", &g),
    class_<A>("A")
        .def(constructor<int, int>),
    def("h", &h)
];
```

更多关于声明的信息请看7 绑定函数到 Lua 和8 绑定类到 Lua 章节.

小提示，如果你发现程序性能糟糕，那是因为将函数放在表中会增加查找时间。

7 绑定函数到Lua

使用luabind::def()绑定函数到Lua。它的原型如下:

```
template<class F, class policies>
void def(const char* name, F f, const Policies&);
```

- name 是Lua中使用的函数的名字。
- F 是要注册的函数的指针。
- Policies 参数用于描述函数如何处理参数和返回值，这是个可选参数。更多信息请看14 策略章节。

如果要注册函数float std::sin(float)，示例如下:

```
module(L)
[
    def("sin", &std::sin)
];
```

7.1 重载函数

如果你有多于一个函数有相同的名字，并想将它们都注册到 Lua 中，需要显式地给出函数签名。这可以让 C++ 知道你指的是哪个函数。例如，如果有两个函数 int f(const char*) 和 void f(int) :

```
module(L)
[
    def("f", (int(*)(const char*)) &f),
    def("f", (void(*)(int)) &f)
];
```

7.2 签名匹配

Luabind会生成代码，用于检测 Lua 栈上的值是否匹配函数签名。它能处理派生类间的隐匿转换，会优先选取需要最少隐匿转换的那个。在一次函数调用中，如果函数是被重载过，并且并没有某个重载函数特别地比其他重载函数要匹配，就会搞不清楚。它会生成一个运行时错误，说那个函数调用是模棱两可的。一个简单的例子是注册一个函数接受一个 int 参数，另一个函数接受一个 float 参数。因为 Lua 不能区分浮点数和整数，那么两个都会匹配。

因为所有重载都经过测试，它总是能找到最佳的匹配（不是首次匹配）。这意味着它能处理这样的情况：函数签名上所有的区别在于一个成员函数是 const 的，另一个成员函数不是 const 的。

例如，下面的函数和类被注册：

```
struct A
{
    void f();
    void f() const;
};

const A* create_a();

struct B: A {};
struct C: B {};

void g(A*);
void g(B*);
```

可以执行下面的Lua代码：

```
a1 = create_a()
a1:f() -- 版本被调用const

a2 = A()
a2:f() -- 非版本被调用const

a = A()
b = B()
c = C()

g(a) -- 调用g(A*)
g(b) -- 调用g(B*)
g(c) -- 调用g(B*)
```

所有权转移：为了正确处理所有权转移，create_a() 需要一个 adopt 返回值策略。更多信息请看14 策略章节。

7.3 调用Lua函数

调用Lua函数，可以使用 call_function() 或 object。

```
template<class Ret>
Ret call_function(lua_State* L, const char* name, ...)
template<class Ret>
Ret call_function(object const& obj, ...)
```

call_function有两个重载版本，一个可以通过给定函数名字来调用，另一个需要 object，它是一个可以作为函数来调用的 Lua 值。

使用函数名字来调用的重载版本只能调用全局 Lua 函数。其中...代表发送给 Lua 函数的参数个数可变。函数调用失败会抛出 luabind::error 。

返回值类型不是真正的 Ret（那个模板参数），而是个用于真正完成函数调用的代理对象。这使得可以通过操作符[]给此次调用一个策略。将策略放在括号中，像这样:

```
1  int ret = call_function<int>(
2    L
3  , "a_lua_function"
4  , new complex_class()
5 )[ adopt(_1) ];
```

如果想要传递引用参数，要用 Boost.Ref 包裹起来。像这样:

```
1  int ret = call_function(L, "fun", boost::ref(val));
```

如果要使用自定义的错误处理器，请查看16.1 pcall errorfunc 章节的 set_pcall_callback。

7.4 使用Lua线程

启动一个 Lua 线程，要调用 lua_resume() ，就是说不能使用前面提到的那个 call_function() 来启动线程。要用这个:

```
1 template<class Ret>
2 Ret resume_function(lua_State* L, const char* name, ...)
3 template<class Ret>
4 Ret resume_function(object const& obj, ...)
```

和

```
1 template<class Ret>
2 Ret resume(lua_State* L, ...)
```

第一次启动线程时，必须给出一个函数来执行。比如，使用 resume_function ，当 Lua 函数yield了，它会返回第一个传递给lua_yield()的值。当想要继续执行时，只要在原来的 lua_State 上调用 resume() ，因为它已经执行过函数，不用再传了。传给 resume() 的参数会在 Lua 侧通过 yield() 返回。

要想 yield C++ 函数（不能在 Lua 侧和 C++ 侧来回传递数据），可以使用 yield 策略。

使用 object 参数的 resume_function 重载版本很重要，因为 object 随线程的 lua_State* 一起创建。像这样:

```
1 lua_State* thread = lua_newthread(L);
2 object fun = get_global(thread)["my_thread_fun"];
```

```
resume_function(fun);
```

8 绑定类到Lua

使用class_类来注册类。这个类的名字是为了贴近C++的关键字，使得它看起来更直观。它有一个重载的成员函数 def() 用于注册类的成员函数，操作符，构造函数，枚举和属性。它会返回自己的 this 指针，使得可以注册直接注册更多的成员。

我们先开始一个简单的例子。考虑以下 C++ 类:

```
class testclass
{
public:
    testclass(const std::string& s): m_string(s) {}
    void print_string() { std::cout << m_string << "\n"; }

private:
    std::string m_string;
};
```

要在一个Lua环境中注册它，编写以下代码 (假设使用 luabind 的名字空间):

```
module(L)
[
    class_<testclass>("testclass")
        .def(constructor<const std::string&>())
        .def("print_string", &testclass::print_string)
];
```

这将以 testclass 这个名字注册类，构造函数接受一个 string 作为参数，一个成员函数名为 print_string。

```
Lua 5.0  Copyright (C) 1994-2003 Tecgraf, PUC-Rio
> a = testclass('a string')
> a:print_string()
a string
```

还可以将自由函数注册成成员函数。要求是函数将该类的一个指针，const指针，引用或const引用作为第一个参数。其余参数就是 Lua 中可见的参数，而对象指针作为第一个参数。如果有以下 C++ 代码:

```
struct A
{
    int a;
```

```
};

int plus(A* o, int v) { return o->a + v; }
```

可以这样将 plus() 函数注册为 A 的成员函数:

```
class_<A>("A")
    .def("plus", &plus)
```

plus()现在就可以作为 A 的成员函数来调用，它接受一个 int 参数。如果对象指针参数是 const 的，函数会表现得它是个 const 成员函数的样子（可以被 const 对象调用）。

8.1 重载成员函数

当绑定多于一个的重载成员函数时，或只是绑定一个重载成员函数时，要区分清楚传递给 def 的成员函数指针。可以用普通的 C 风格的转换来转成正确的重载版本。你要知道如何用 C++ 表达成员函数类型，这里有个简短的教程（更多信息，请参考你喜欢的 C++ 书籍）:

下面是成员函数指定的句法:

```
return-value (class-name::*)(arg1-type, arg2-type, ...)
```

这是一个例子:

```
struct A
{
    void f(int);
    void f(int, int);
};

class_<A>()
    .def("f", (void(A::*)(int))&A::f)
```

这里选择绑定第一个重载成员函数f。第二个重载没有被绑定。

8.2 属性

注册类的全局数据成员很容易。考虑以下类:

```
struct A
{
    int a;
};
```

这个类这样注册:

```
module(L)
[
    class_<A>("A")
        .def_readwrite("a", &A::a)
];
```

这赋予了对成员变量 A::a 的可读可写访问权限。同样可以注册一个只读访问的属性:

```
module(L)
[
    class_<A>("A")
        .def_readonly("a", &A::a)
];
```

当绑定非原始类型的成员时，自动生成的 getter 会返回对它的引用。这使得可以使用连接的.操作符。例如，当有一个结构包含另一个结构，像这样:

```
struct A { int m; };
struct B { A a; };
```

当绑定B到Lua，下面的表达式就可以工作:

```
b = B()
b.a.m = 1
assert(b.a.m == 1)
```

这要求首次优先查找（对 a ）来返回对 A 的引用，而不是一份复本。这种情况下，Luabind 会自动使用 dependency 策略使得返回值依赖于它保存的对象国。所以，如果返回的引用生命期长于所有其他对这个对象（这里是 b ）的引用，它会一直保存这个对象，以免悬挂指针。

也可以注册 getter 和 setter 函数，让他们看起来像一个公共访问属性的数据成员。看以下代码:

```
class A
{
public:
    void set_a(int x) { a = x; }
    int get_a() const { return a; }

private:
    int a;
};
```

它可以被注册成好像它有个公共访问属性的数据成员一样:

```
class_<A>("A")
    .property("a", &A::get_a, &A::set_a)
```

这样 get_a() 和 set_a() 函数会被调用，而不是直接写到数据成员中。如果想要让它变成只读，可以省略掉最后一个参数。请注意，get 函数必须是 const 的，不然编译不通过。这看起来是个常见错误源。

8.3 枚举

如果类包含了枚举常量，可以注册枚举使它们可以在 Lua 中使用。注意，它们不是类型安全的，所有枚举在 Lua 中都是个整数，所有使用枚举的函数，都是接受成整数。像这样注册:

```
module(L)
[
    class_<A>("A")
        .enum_("constants")
        [
            value("my_enum", 4),
            value("my_2nd_enum", 7),
            value("another_enum", 6)
        ]
];
```

在 Lua 中它们就像其他数据成员一样被访问，它们是只读的，并且只能通过类本身访问而不是类的实例。

```
Lua 5.0  Copyright (C) 1994-2003 Tecgraf, PUC-Rio
> print(A.my_enum)
4
> print(A.another_enum)
6
```

8.4 操作符

绑定操作符要包含<luabind/operator.hpp>。

在类上注册操作的机制非常简单。使用一个全局的名字 luabind::self 指代类自身，在 def() 调用中写入操作符表达式。这个类:

```
struct vec
{
    vec operator+(int s);
};
```

像这样注册:

```
module(L)
[
    class_<vec>("vec")
        .def(self + int())
];
```

这可以工作，不管是否在这个类中定义了+操作符或是自由函数。

如果操作符是 const 的（或者，被定义成自由函数，并接受一个指向类自己的 const 的引用），要用 const_self 代替 self 。像这样:

```
module(L)
[
    class_<vec>("vec")
        .def(const_self + int())
];
```

Lua中可用的操作符支持如下:

```
+    -    *    /    ==    <    <=
```

就是说，没有到位操作符。相等操作符(==)有点麻烦；如果引用相等时，它不会被调用到。这意味着 == 操作符要多作一些工作。

Lua不支持像 != , > 或 >= 等操作符。那就是为什你只能注册上面列出的操作符。当要用到这些操作符时，Lua 会定义成这些有效的操作符中的一个。

在上面的例子中，其他操作数类型被初始化成 int() 。如果操作数类型是个复杂类型，不能简单地初始化，可以将类型包装到一个称为 other<> 的类中。例如:

注册这个类，我们不想初始化 string 只想注册这个操作符。

```
struct vec
{
    vec operator+(std::string);
};
```

相反我们用 other<> 这样包装:

```
module(L)
[
    class_<vec>("vec")
        .def(self + other<std::string>())
];
```

注册一个应用（函数调用）操作符:

```
module(L)
[
    class_<vec>("vec")
        .def( self(int()) )
];
```

有一个特殊的操作符。在 Lua 中它被称为 __tostring，它不是个真正的操作符。它用于在 Lua 中将对象以标准方式转换成字符串。如果要注册这个函数，需要使用 lua 标准函数 tostring() 将对象转换成字符串。

要在 C++ 中实现这个操作符，要为 std::ostream 函数提供一个 operator<< 。像这个例子:

```
class number {};
std::ostream& operator<<(std::ostream&, number&);

...

module(L)
[
    class_<number>("number")
        .def(tostring(self))
];
```

8.5 内嵌scope和静态函数

可以向一个类添加内嵌 scope 。当需要包装一个内嵌类，或静态函数时，这就有用了:

```
class_<foo>("foo")
    .def(constructor<>()
    .scope
    [
        class_<inner>("nested"),
        def("f", &f)
    ];
```

这个例子中，f将表现得像类 foo 的静态成员函数一样，而类 nested 会表现得像类 foo 的内嵌类。

用这种方法也可以向类添加名字空间。

8.6 派生类

如果要注册一个从其他它派生的类，需要在 class_ 实例化时指定模板参数 bases<>。下面的继承体系：

```
struct A {};
struct B : A {};
```

可以这样注册

```
module(L)
[
    class_<A>("A"),
    class_<B, A>("B")
];
```

如果是多重继承，可以指定多个基类。如果 B 还从类 C 继承，应该这样注册：

```
module(L)
[
    class_<B, bases<A, C> >("B")
];
```

注意不能在单继承时省略 bases<> 。

注意 8.1 *如果不指定类是从其他类继承来的，Luabind 将不能在类型之间进行隐式转换。*

8.7 智能指针

注册类时，可以告诉 luabind 所有该类的实例会被某种形式的智能指针（比如 boost::shared_ptr ）持有。通过向 class_ 类传递一个额外的模板参数来实现这一点，像这样：

```
module(L)
[
    class_<A, boost::shared_ptr<A> >("A")
];
```

还可以为智能指针提供两个函数。一个用于返回智能指针类型的 const 版本（这里是 boost::shared_ptr<const A> ）。另一个函数则从智能指针中提取出原始的指针。需要第一个函数是因为 luabind 将值从 Lua 传递给 C++ 时必须能进行非 const 的->转换。需要第二个函数是因为当 Lua 调用持有类型的成员函数时，这个指针必须是原始类型指针，还因为要使从 Lua 到 C++ 的智能指针转换成原始指针。看起来像这样：

```
namespace luabind {
```

```
2
3       template<class T>
4       T* get_pointer(boost::shared_ptr<T>& p)
5       {
6           return p.get();
7       }
8
9       template<class A>
10      boost::shared_ptr<const A>*
11      get_const_holder(boost::shared_ptr<A>*)
12      {
13          return 0;
14      }
15  }
```

第二个函数只在编译期间将 boost::shared_ptr<A> 映射到它的 const 版本 boost :: shared_ptr<const A> 时才会用到。它从来不会被调用，所以不用管它的返回值（只有返回值类型）。

可以进行转换有（B 是 A 的基类）（见从Lua转换到C++和从C++转换到Lua）：

表 1: 从Lua转换到C++

源	宿
holder_type<A>（持有类型 A）	A*
holder_type<A>（持有类型 A）	B*
holder_type<A>（持有类型 A）	A const*
holder_type<A>（持有类型 A）	B const*
holder_type<A>（持有类型 A）	holder_type<A>
holder_type<A>（持有类型 A）	holder_type<A const>
holder_type<A const>（持有类型 A）	A const*
holder_type<A const>（持有类型 A）	B const*
holder_type<A const>（持有类型 A）	holder_type<A const>

表 2: 从C++转换到Lua

源	宿
holder_type<A>（持有类型 A）	holder_type<A>
holder_type<A const>（持有类型 A）	holder_type<A const>
holder_type<A>const&（持有类型 A）	holder_type<A>
holder_type<A const>const&（持有类型 A）	holder_type<A>

当使用持有类型时，它可以用来检测指针是否有效（比如是否为 NULL ）。例如使用 std :: auto_ptr ，当它作为参数传递给函数时，原来持有者就会无效。为了这个目的，luabind 中所有对象实例都有一个成员：__ok。

```
struct X {};
void f(std::auto_ptr<X>);

module(L)
[
    class_<X, std::auto_ptr<X> >("X")
        .def(constructor<>()),

    def("f", &f)
];
```

```
Lua 5.0  Copyright (C) 1994-2003 Tecgraf, PUC-Rio
> a = X()
> f(a)
> print a.__ok
false
```

当注册一组有继承关系的类时，如果所有实例都被智能指针持有，则所有类都需要有基类的持有类型。像这样:

```
module(L)
[
    class_<base, boost::shared_ptr<base> >("base")
        .def(constructor<>()),
    class_<derived, base, boost::shared_ptr<base> >("base")
        .def(constructor<>())
];
```

Luabind 内部会对原始指针进行必要的转换，它是第一个从持有类型中进行提取的。

8.8 拆分类注册

某些情况下要求将类拆分，并在不同的编译单元中注册。部分原因是当修改绑定的某部分时可以节省构建时间，另一些情况下是编译器的限制使得强迫需要拆分。很简单，看以下代码:

```
void register_part1(class_<X>& x)
{
```

```
3     x.def(/*...*/);
4 }
5
6 void register_part2(class_<X>& x)
7 {
8     x.def(/*...*/);
9 }
10
11 void register_(lua_State* L)
12 {
13     class_<X> x("x");
14
15     register_part1(x);
16     register_part2(x);
17
18     module(L) [ x ];
19 }
```

这个类X分两步注册。两个函数 register_part1 和 register_part2 会放在不同的编译单元中。

要区分模块注册和类注册，请看拆分注册章节。

9 为用户定义类型添加转换器

可以通过特化luabind::default_converter<> 来让 luabind 像处理内建类型一样处理用户定义类型:

```
1  struct int_wrapper
2  {
3      int_wrapper(int value)
4        : value(value)
5      {}
6
7      int value;
8  };
9
10 namespace luabind
11 {
12     template <>
13     struct default_converter<X>
14       : native_converter_base<X>
15     {
16         static int compute_score(lua_State* L, int index)
```

```
17          {
18              return lua_type(L, index) == LUA_TNUMBER ? 0 : -1;
19          }
20
21          X from(lua_State* L, int index)
22          {
23              return X(lua_tonumber(L, index));
24          }
25
26          void to(lua_State* L, X const& x)
27          {
28              lua_pushnumber(L, x.value);
29          }
30      };
31
32      template <>
33      struct default_converter<X const&>
34        : default_converter<X>
35      {};
36  }
```

注意default_converter<> 通过被绑定函数的实参和返回值类型实例化。在上面的例子中，我们添加了 X const& 的特化，它会转发到 X 类型转换器上。这让我们可以导出能接受 X const& 的函数。

native_converter_base<> 要作为特化转换器的基类。它简化了转换器接口，并因为底层接口的原因，实现了向后兼容。

10 使用显式签名绑定函数对象

使用luabind::tag_function<> 可以让我们导出 luabind 不能自动推演签名的函数对象。这可用于区分被绑定函数的签名，甚至用于绑定具有状态的函数对象。

```
1  template <class Signature, class F>
2  implementation-defined tag_function(F f);
```

其中，Signature描述了F的签名的函数类型。可以这样使用：

```
1  int f(int x);
2
3  // alter the signature so that the return value is ignored
4  def("f", tag_function<void(int)>(f));
5
6  struct plus
```

```
{
    plus(int x)
      : x(x)
    {}

    int operator()(int y) const
    {
        return x + y;
    }
};

// bind a stateful function object
def("plus3", tag_function<int(int)>(plus(3)));
```

11 对象

由于函数要能接受 Lua 值（变量类型），我们要将它们包裹起来。这个包裹器称为 luabind :: object 。如果注册的函数接受一个 object ，它就可以使用接受任何 Lua 值。包含<luabind/object.hpp>来使用：

总览

```
class object
{
public:
    template<class T>
    object(lua_State*, T const& value);
    object(from_stack const&);
    object(object const&);
    object();

    ~object();

    lua_State* interpreter() const;
    void push() const;
    bool is_valid() const;
    operator safe_bool_type () const;

    template<class Key>
    implementation-defined operator[](Key const&);

    template<class T>
    object& operator=(T const&);
```

```
22      object& operator=(object const&);
23
24      bool operator==(object const&) const;
25      bool operator<(object const&) const;
26      bool operator<=(object const&) const;
27      bool operator>(object const&) const;
28      bool operator>=(object const&) const;
29      bool operator!=(object const&) const;
30
31      template <class T>
32      implementation-defined operator[](T const& key) const
33
34      void swap(object&);
35
36      implementation-defined operator()();
37
38      template<class A0>
39      implementation-defined operator()(A0 const& a0);
40
41      template<class A0, class A1>
42      implementation-defined operator()(A0 const& a0, A1 const& a1);
43
44      /* ... */
45  };
```

使用Lua object时，可以使用赋值操作符(=)向它赋个新值。这时需要 default_policy 将 C++ 值转换成 Lua 值。如果 luabind :: object 是个表，可以通过操作符[]或迭代器访问它的成员。从操作符[]返回的值是个代理对象，用于向表中读取和向表中写入（使用操作符=）。

注意，这不可能知道一个 Lua 值是否可索引 (lua_gettable不会失败，它要不成功，要不崩溃)。也就是说如果试图索引某个不能被索引的值，得自己负责这个风险。Lua 会调用它的 panic() 函数。见16.3 lua panic 章节。

也有自由函数可用于索引表，见11.2 相关函数章节。

接受 from_stack 对象的构造函数用于想要初始化一个从 lua 栈上获取值的对象的情况。from_stack 类型有以下构造函数:

```
1  from_stack(lua_State* L, int index);
```

其中index是普通的 lua 栈的索引，负值表示从栈顶开始索引。可以这样使用:

```
1  object o(from_stack(L, -1));
```

这将创建一个对象o，并将值从 lua 栈顶复制过来。

函数interpreter()返回保存本对象的 Lua state。如果想要直接使用 Lua 函数操作对象，可以调用 push() 将其压入 Lua 栈。

操作符==会对操作数调用 lua_equal()，并返回它的结果。

函数is_valid()告诉你是否对象已经被初始化。当使用缺省构造函数创建的，那对象就是无效的。要使对象有效，可以给它赋个值。如果想要让一个对象无效，可以简单地给它赋个无效对象。

操作符operator safe_bool_type()与 is_valid() 等价。就是说下面的片段是等价的:

```
object o;
// ...
if (o)
{
    // ...
}

...

object o;
// ...
if (o.is_valid())
{
    // ...
}
```

因为是个函数，应用程序操作符会调用该值。可以给它任意数量的参数（目前 default_policy 用于转换）。返回的对象表示返回值（目前只支持一个返回值）。如果函数调用失败，该操作符会抛出 luabind :: error。如果想要给函数调用指定策略，可以使用 index 操作符（operator[]）于该函数调用，并将策略写在[和] 内。像这样:

```
my_function_object(
    2
  , 8
  , new my_complex_structure(6)
) [ adopt(_3) ];
```

这告诉 luabind 进行 Lua 所有权转移和传递给 lua 函数的指针职责转移。

重要的是所有 object 的实例在 Lua state 关闭时必须已经都销毁。object 会持有指向 lua state 的指针，并在它销毁时释放 Lua object 。

函数使用表的示例如下:

```
void my_function(object const& table)
{
    if (type(table) == LUA_TTABLE)
```

```
4       {
5           table["time"] = std::clock();
6           table["name"] = std::rand() < 500 ? "unusual" : "usual";
7
8           std::cout << object_cast<std::string>(table[5]) << "\n";
9       }
10  }
```

如果函数使用 luabind::object 作为参数，任何 Lua 值都能匹配该参数。那是为什么必须在索引某个值前确保是它个表。

```
1   std::ostream& operator<<(std::ostream&, object const&);
```

有个stream操作符可以打印 object，或使用 boost::lexical_cast 将其转换成字符串。这会使用 lua 的字符串转换函数。所以如果转换一个定义了 tostring 操作符的 C++ 对象，就会为那种类型使用 stream 操作符。

11.1 迭代器

有两类迭代器。普通迭代器，在获取值时，会使用对象的 metamethod （如果有的话）。这类迭代器简单地被 luabind :: iterator 调用。另一类迭代器被 luabind :: raw_iterator 调用，并将 metamethod 透传，返回表中的真正内容。它们有相同的接口，实现了 ForwardIterator [6] 概念。提取了标准迭代器的一部分成员，它们有下列成员和构造函数:

```
1   class iterator
2   {
3       iterator();
4       iterator(object const&);
5
6       object key() const;
7
8       standard iterator members
9   };
```

接受 luabind::object 的构造函数是一个模板，可以与 object 一起使用。将一个 object 作为参数传递给迭代器，会构造一个指向 object 中第一个元素的迭代器。

缺省构造函数会初始化迭代器为“一遍结束”迭代器。它用于测试是否到了序列的结束端。

迭代器的值类型是支持与 luabind::object 相同操作的代理类型的实现。就是说，大多数时候，可以将它作为一个普通对象看待。不同之处在于任何对该代理进行赋值都会导致把值插入到表中迭代器的位置。

[6] http://www.sgi.com/tech/stl/ForwardIterator.html

key()成员返回迭代器在索引关联的 Lua 表时使用的 key 。

使用迭代器的示例:

```
for (iterator i(globals(L)["a"]), end; i != end; ++i)
{
  *i = 1;
}
```

名为 end 的迭代器使用缺省构造函数创建，因此指定序列的结束处。本例简单地迭代了全局表 a 的所有表项，并所有表项值设为1。

11.2 相关函数

有一组与 object 和表相关的函数。

```
int type(object const&);
```

该函数会返回给定 object 的 Lua 类型索引号，如 LUA_TNIL ， LUA_TNUMBER 等。

```
template<class T, class K>
void settable(object const& o, K const& key, T const& value);
template<class K>
object gettable(object const& o, K const& key);
template<class T, class K>
void rawset(object const& o, K const& key, T const& value);
template<class K>
object rawget(object const& o, K const& key);
```

这些函数用于在表中索引。settable 和 gettable 将分别被转换成对 lua_settable 和 lua_gettable 的调用。意思是说，可以只用对象的索引操作符。

rawset 和 rawget 将分别被转换成对 lua_rawset 和 lua_rawget 的调用。所以它们会透传所有metamethod，并返回表项的真实值。

```
template<class T>
T object_cast<T>(object const&);
template<class T, class Policies>
T object_cast<T>(object const&, Policies);

template<class T>
boost::optional<T> object_cast_nothrow<T>(object const&);
template<class T, class Policies>
boost::optional<T> object_cast_nothrow<T>(object const&, Policies);
```

object_cast 函数把一个 Lua object 转换成 C++ 值。可以提供一个策略来处理从 Lua 到 C++ 的转换。如果转换不能进行，会抛出 cast_failed 异常。如果定义了

LUABIND_NO_ERROR_CHECKING （见17 构建选项）就不会进行检测，如果转换无效，应用程序就很可能崩溃。不抛出异常的版本会返回一个未初始化的 boost :: optional<T> 对象，以指明转换不能进行。

上面所有函数的签名都是接受 object 参数的模板，但意味着你只能传 object 过去，这是为什么它没有文档记载的原因。

```
object globals(lua_State*);
object registry(lua_State*);
```

这些函数分别返回全局环境和注册表。

```
object newtable(lua_State*);
```

这个函数创建一个新的表，将作为 object 返回。

12 Lua中定义类

除了可以绑定 C++ 函数和类到 Lua，luabind 还提供了一套 Lua 中的OO（面向对象）系统。

```
class 'lua_testclass'

function lua_testclass:__init(name)
    self.name = name
end

function lua_testclass:print()
    print(self.name)
end

a = lua_testclass('example')
a:print()
```

可以在 lua 类间进行继承:

```
class 'derived' (lua_testclass)

function derived:__init() super('derived name')
end

function derived:print()
    print('Derived:print() -> ')
    lua_testclass.print(self)
end
```

这里初始化基类时在构造函数中使用了 super 关键字。用户要在构造函数中先调用 super。

从示例中可以看到，可以调用基类的成员函数。可以找到基类中所有的成员函数，但必须将 this 指针（self）作为第一个参数。

12.1 Lua中派生类

可以从 C++ 类派生 Lua 类，并用 Lua 函数覆写虚函数。要做到这点，需要为 C++ 基类创建一个包裹器，这是个会在实例化 Lua 类时持有 Lua 对象的类。

```
1  class base
2  {
3  public:
4      base(const char* s)
5      { std::cout << s << "\n"; }
6
7      virtual void f(int a)
8      { std::cout << "f(" << a << ")\n"; }
9  };
10
11 struct base_wrapper : base, luabind::wrap_base
12 {
13     base_wrapper(const char* s)
14         : base(s)
15     {}
16
17     virtual void f(int a)
18     {
19         call<void>("f", a);
20     }
21
22     static void default_f(base* ptr, int a)
23     {
24         return ptr->base::f(a);
25     }
26 };
27
28 ...
29
30 module(L)
31 [
32     class_<base, base_wrapper>("base")
33         .def(constructor<const char*>())
```

```
34        .def("f", &base::f, &base_wrapper::default_f)
35 ];
```

重要 12.1 *由于MSVC6.5不支持成员函数的显式模板参数，就不要调用成员函数 call() 了，而应该使用自由函数 call_member() 并将 this 指针作为第一个参数传给它。*

注意，如果既有基类，双有基类的包裹器，必须把它们的类型作为模板参数传递给 class_（像上面的示例一样）。它们的顺序不重要。还必须注册包裹器的静态版本和虚版本的函数，为了使 luabind 能同时使用函数的动态和静态分派，这很重要。

重要 12.2 *非常重要的一点是，静态（缺省）函数的签名跟虚函数相同。它们一个是自由函数，另一个是成员函数，这没关系，但从 Lua 看到的参数必须匹配。如果静态函数把 base_wrapper* 作为第一个参数，将不能正常工作，因为虚函数把 base* 作为它的第一个参数（它的 this 指针）。目前 luabind 中没有检测来确保签名匹配。*

如果没有包裹器类，将不能把 Lua 类回传给 C++。这是因为虚函数的入口仍然指向 C++ 基类，而不是在 Lua 中定义的函数。这就是为什么需要一个函数来调用基类的真正的函数（如果 Lua 函数没有重定义它），另一个虚函数来分派函数调用到 luabind 中，允许它选择是否 Lua 函数被调用，还是原始的函数被调用。如果不想从 C++ 类派生，或者类没有任何虚成员函数，注册时就不需要类包裹器了。

要从类派生，并不是一定需要一个类包裹的，但如果它有虚函数，就会有隐含的错误。

类包裹器必须从 luabind::wrap_base 类派生，它包含一个 Lua 引用，会持有一个对象的 Lua 实例，该对象用来分派虚函数调用到 Lua 中去。这些工作由重载成员函数完成：

```
1 template<class Ret>
2 Ret call(char const* name, ...)
```

在 call_function 中也是类似的用法，带着不用 lua_State 指针的异常，而名字是 Lua 类中的成员函数。

警告 12.1 *目前 call_member 的实现不能区分 const 成员函数和非 const 成员函数。如果有重载的虚函数只是它们的签名中 const 属性不同，call_member 调用重载就会搞错。不过这种情形很少见。*

12.1.1 对象标识

当传递一个已注册类的指针或引用的包装到 Lua 时，luabind 会查询它的动态类型。如果动态类型从 wrap_base 继承过来，对象标识就被保存下了。

```
struct A { .. };
struct A_wrap : A, wrap_base { .. };

A* f(A* ptr) { return ptr; }

module(L)
[
    class_<A, A_wrap>("A"),
    def("f", &f)
];
```

```
> class 'B' (A)
> x = B()
> assert(x == f(x)) -- 当对象通过C传递过来，对象标识被保存++
```

此功能需要RTTI全能（即 LUABIND_NO_RTTI 没有定义）。

12.2 操作符重载

可以在类中重载大部分 Lua 操作符。只要简单地声明一个成员函数，使用与操作符相同的名字（Lua 中 metamethods 的名字）。可以重载的操作符有：

- __add
- __sub
- __mul
- __div
- __pow
- __lt
- __le
- __eq
- __call
- __unm
- __tostring

__tostring不是个真正的操作符，但它是被标准库中的 tostring() 函数调用的 metamethod。对于二进制操作符，有一个奇怪的行为。不能保证获取到的 self 指针指向类的实例。这是因为 Lua 不区分两种情况：获取操作符是左值还是右值。考虑以下示例：

```
class 'my_class'

  function my_class:__init(v)
      self.val = v
  end

  function my_class:__sub(v)
      return my_class(self.val - v.val)
  end

  function my_class:__tostring()
      return self.val
  end
```

只在 my_class 实例间进行减法操作，这就可以正常工作。但如果想要能从类去减普通数字，就需要手工检测两个操作数的类型了，包括 self 对象。

```
function my_class:__sub(v)
    if (type(self) == 'number') then
        return my_class(self - v.val)

    elseif (type(v) == 'number') then
        return my_class(self.val - v)

    else
        -- 假设两个操作数都是的实例 my_class
        return my_class(self.val - v.val)

    end
end
```

为什么示例中使用 __sub 的原因是减法操作是不可交换的（操作数顺序相关）。这也是为什么 luabind 不能交换两个操作数的顺序，以使 self 引用总是指向实际的类实例。

如果有两个不同的 Lua 类有一个重载的操作符，右侧类型的操作符会被调用。如果另一个操作数是有着相同操作符重载的 C++ 类，它会优先于 Lua 类的操作符。如果 C++ 类中的重载不匹配，才会调用 Lua 类中的操作符。

12.3 结束器

如果一个对象需要在它被回收时进行一些操作，我们提供了一个__finalize函数，可以让 lua 类覆写。__finalize函数会在继承体系链中所有的类中被调用，从最底层的派生类开始。

```
...

function lua_testclass:__finalize()
    -- 当对象被回收时被调用
end
```

12.4 切片

如果lua中使用的C++类没有包裹器（见12.1 Lua 中派生类）并在 Lua 中派生了新类，它们可能会被切片。意思是，如果一个对象以指向它基类的指针传递给 C++，lua 部分则只会有 C++ 中的基类部分。这意味着可以调用这个 C++ 对象的虚函数，它们不会被分派到 lua 类中。也意味着如果 adopt 该对象，lua 部分会被垃圾回收。

```
+--------------------+
| C++ 对象           |    <- 当被adopt时，这部分的所有权
|                    |       被转移到C++中
+--------------------+
| lua 类实例         |    <- 当实例被adopt时，这部分会被垃圾回收，
| 和 lua 成员        |       因为它不能被C++持有
+--------------------+
```

这个问题可以通过下面的例子来阐明：

```
struct A {};

A* filter_a(A* a) { return a; }
void adopt_a(A* a) { delete a; }

using namespace luabind;

module(L)
[
    class_<A>("A"),
    def("filter_a", &filter_a),
    def("adopt_a", &adopt_a, adopt(_1))
]
```

在lua中：

```
a = A()
b = filter_a(a)
adopt_a(b)
```

这个例子中，lua不知道 b 实际上跟 a 是同一个对象，因此会在 C++ 侧获取对象所有权。然后 b 的指针被 adopt，会报个运行时错误上来，因为对象的所有权不在 lua 却被adopt到 C++ 去。

13 异常

当调用 lua 中注册函数抛出异常时，该异常会被 luabind 捕获，转换成一个错误字符串，并调用 lua_error()。如果异常是个 std :: exception 或是被压入 Lua 栈的 const char* 字符串，作为错误字符串，就使用 std :: exception :: what() 返回的字符串或 const char* 字符串本身。如果是未知异常，就只是压入一个字符串说函数抛出了异常。

从用户定义的函数中抛出的异常必须被 luabind 捕获。如果不被捕获，就会抛过 lua，一般 lua 被作为 C 代码编译，就不能支持异常隐含的栈展开。

任何调用了 Lua 代码的函数都可能会抛出 luabind :: error 。该异常的意思是发生了 Lua 运行时错误。错误消息保存在栈顶。在异常中不包含错误字符串本身，是因为可能那时候的堆分配会失败。如果异常类在被抛出时又自己抛出一个异常，应用程序就会退出。

Error类总览如下:

```
class error : public std::exception
{
public:
    error(lua_State*);
    lua_State* state() const throw();
    virtual const char* what() const throw();
};
```

State函数返回指向 Lua state 的指针。如果捕获异常时 lua state 已经销毁了，这个指针就无效了。如果 Lua state 还是有效的，那么就可以通过它获取 Lua 栈顶的错误消息了。

Lua state指针可能指向无效的 state 示例如下:

```
struct lua_state
{
    lua_state(lua_State* L): m_L(L) {}
    ~lua_state() { lua_close(m_L); }
    operator lua_State*() { return m_L; }
    lua_State* m_L;
```

```
};

int main()
{
    try
    {
        lua_state L = lua_open();
        /* ... */
    }
    catch(luabind::error& e)
    {
        lua_State* L = e.state();
        // L will now point to the destructed
        // Lua state and be invalid
        /* ... */
    }
}
```

另一个luabind可能抛出的异常是luabind :: cast_failed, 这个异常从call_function<>或call_member<>中抛出。这个异常的意思是，Lua函数的返回值不能被转换成C++值。如果转换不能进行，它也会被object_cast<>抛出。

luabind::cast_failed总览如下:

```
class cast_failed : public std::exception
{
public:
    cast_failed(lua_State*);
    lua_State* state() const throw();
    LUABIND_TYPE_INFO info() const throw();
    virtual const char* what() const throw();
};
```

同样，state成员函数返回错误发生时指向Lua state的指针。见上面的例子，该指针可能无效。

info成员函数返回用户定义的LUABIND_TYPE_INFO, 默认是const std :: type_info*。该类型信息描述了试图将Lua值转换后的类型。

如果定义了LUABIND_NO_EXCEPTIONS则这些异常都不会抛出，需要设置两个回调函数供调用。这两个函数只有在LUABIND_NO_EXCEPTIONS定义时才能用。

```
luabind::set_error_callback(void(*)(lua_State*))
```

设置的这个函数会在Lua代码中发生运行时错误时被调用。可以在Lua栈顶取得错误消息。该函数不需要返回值，如果它硬是返回东西了，luabind会调用std ::

terminate()。

```
1 luabind::set_cast_failed_callback(void(*)(lua_State*, LUABIND_TYPE_INFO))
```

设置的这个函数作为抛出 cast_failed 的替代物。该函数不需要返回值，如果它硬是返回东西了，luabind 会调用 std :: terminate()。

14 策略

有时能控制 luabind 如何传递参数和返回值非常重要，为此我们提供了策略。所有策略使用一个索引来让它们与函数签名中的一个参数关联。这些指数是 result 和 _N（其中 N >=1）。当处理成员函数时，_1 指代 this 指针。

目前实现的策略有:

- adopt
- dependency
- out_value
- pure_out_value
- return_reference_to
- copy
- discard_result
- return_stl_iterator
- raw
- yield

14.1 adopt

14.1.1 动机

在语言边界转移所有权。

14.1.2 定义于

```
1 #include <luabind/adopt_policy.hpp>
```

14.1.3 总览

```
adopt(index)
```

14.1.4 参数

表 3: adopt策略参数

参数	作用
index	要转移所有权的索引号，_N 或 result

14.1.5 示例

```
X* create()
{
    return new X;
}

...

module(L)
[
    def("create", &create, adopt(result))
];
```

14.2 dependency

14.2.1 动机

用于在值之间创建生命周期依赖。例如返回类的内部引用。

14.2.2 定义于

```
#include <luabind/dependency_policy.hpp>
```

14.2.3 总览

```
dependency(nurse_index, patient_index)
```

表 4: dependency策略参数

参数	作用
nurse_index	能让patient存活的索引号
patient_index	一直存活的索引号

14.2.4 参数

14.2.5 示例

```
struct X
{
    B member;
    B& get() { return member; }
};

module(L)
[
    class_<X>("X")
        .def("get", &X::get, dependency(result, _1))
];
```

14.3 out_value

14.3.1 动机

该策略使得可以包装函数，以便让函数可以接受非 const 引用或指针作为参数，以此来返回值。由于 lua 不可能传递原始类型的引用，本策略会在调用完后添加另一个返回值。如果函数已经有返回值了，本策略的一个实现会添加另一个返回值（请阅读 lua 手册中关于返回多个值的章节）。

14.3.2 定义于

```
#include <luabind/out_value_policy.hpp>
```

14.3.3 总览

```
out_value(index, policies = none)
```

14.3.4 参数

14.3.5 示例

表 5: out_value策略参数

参数	作用
index	作为输出参数的索引号。
policies	内部使用的策略，用于向/从 Lua 转换。_1 表示转换到C++, _2 表示从C++ 转换.

```
1 void f1(float& val) { val = val + 10.f; }
2 void f2(float* val) { *val = *val + 10.f; }
3
4 module(L)
5 [
6     def("f", &f, out_value(_1))
7 ];
```

```
Lua 5.0  Copyright (C) 1994-2003 Tecgraf, PUC-Rio
> print(f1(10))
20
> print(f2(10))
20
```

14.4 pure_out_value

14.4.1 动机

本策略像out_value一样工作，但它会传递一个缺省构造的对象而不是从 Lua 转换参数。就是说，参数会被从 lua 签名中移除。

14.4.2 定义于

```
1 #include <luabind/out_value_policy.hpp>
```

14.4.3 总览

```
1 pure_out_value(index, policies = none)
```

14.4.4 参数

14.4.5 示例

注意，没有传参数给f1和f2。

表 6: pure_out_value策略参数

参数	作用
index	作为输出参数的索引号。
policies	内部使用的策略，用于转换输出参数到 Lua。_1 是用于内部的索引号。

```
1 void f1(float& val) { val = 10.f; }
2 void f2(float* val) { *val = 10.f; }
3
4 module(L)
5 [
6     def("f", &f, pure_out_value(_1))
7 ];
```

```
Lua 5.0  Copyright (C) 1994-2003 Tecgraf, PUC-Rio
> print(f1())
10
> print(f2())
10
```

14.5 return_reference_to

14.5.1 动机

在C++中返回参数引用或 this 指针以便串链调用是很常见的。

```
1  struct A
2  {
3      float val;
4
5      A& set(float v)
6      {
7          val = v;
8          return *this;
9      }
10 };
```

当Luabind生成代码时，会为返回值创建一个新的对象，指向本身的对象。这不会有问题，但会有点低效。使用本策略可以告诉 luabind 返回值已经在 lua 的栈上。

14.5.2 定义于

```
#include <luabind/return_reference_to_policy.hpp>
```

14.5.3 总览

```
return_reference_to(index)
```

14.5.4 参数

表 7: return_reference_to策略参数

参数	作用
index	用于返回引用的参数索引号，可以是除了 result 外的任意参数。

14.5.5 示例

```
struct A
{
    float val;

    A& set(float v)
    {
        val = v;
        return *this;
    }
};

module(L)
[
    class_<A>("A")
        .def(constructor<>())
        .def("set", &A::set, return_reference_to(_1))
];
```

警告 14.1 *此策略忽略了所有类型信息，只有当参数类型与返回值类型完美匹配时才可用（像示例中那样）。*

14.6 copy

14.6.1 动机

创建参数的复本。当参数以值传递时，这是默认行为。注意，只有从 C++ 传递到 Lua 时才用本策略。本策略要求参数类型有可用的拷贝构造函数。

14.6.2 定义于

```
#include <luabind/copy_policy.hpp>
```

14.6.3 总览

```
copy(index)
```

14.6.4 参数

表 8: copy策略参数

参数	作用
index	被复制的参数的索引号。封装 C++ 函数时使用 result 。当传递参数给 Lua 时使用_N。

14.6.5 示例

```
X* get()
{
    static X instance;
    return &instance;
}

...

module(L)
[
    def("create", &create, copy(result))
];
```

14.7 discard_result

14.7.1 动机

这是个很简单的策略，即丢弃 C++ 函数的返回值，而不是转换为 Lua 值。

14.7.2 定义于

```
#include <luabind/discard_result_policy.hpp>
```

14.7.3 总览

```
discard_result
```

14.7.4 示例

```
struct X
{
    X& set(T n)
    {
        ...
        return *this;
    }
};

...

module(L)
[
    class_<X>("X")
        .def("set", &simple::set, discard_result)
];
```

14.8 return_stl_iterator

14.8.1 动机

本策略将 STL 容器转换成生成器函数，以便在 lua 中迭代容器。只要定义了 begin() 和 end() 成员函数（返回迭代器）的容器，它都可以工作。

14.8.2 定义于

```
#include <luabind/iterator_policy.hpp>
```

14.8.3 总览

```
return_stl_iterator
```

14.8.4 示例

```
struct X
{
    std::vector<std::string> names;
};

...

module(L)
[
    class_<A>("A")
        .def_readwrite("names", &X::names, return_stl_iterator)
];

...
```

```
> a = A()
> for name in a.names do
>  print(name)
> end
```

14.9 raw

14.9.1 动机

这个转换器策略会传递未修改的 lua_State*。例如要绑定一个返回 luabind :: object 的函数时，这就有用了。参数将从函数签名中被移除，以减少一个函数元数。

14.9.2 定义于

```
#include <luabind/raw_policy.hpp>
```

14.9.3 总览

```
raw(index)
```

14.9.4 参数

表 9: raw策略参数

参数	作用
index	lua_State* 参数的索引号。

14.9.5 示例

```
void greet(lua_State* L)
{
    lua_pushstring(L, "hello");
}

...

module(L)
[
    def("greet", &greet, raw(_1))
];
```

```
> print(greet())
hello
```

14.10 yield

14.10.1 动机

让C++函数在返回时yield。

14.10.2 定义于

```
#include <luabind/yield_policy.hpp>
```

14.10.3 总览

```
yield
```

14.10.4 示例

```
void do_thing_that_takes_time()
{
    ...
}

...

module(L)
[
    def("do_thing_that_takes_time", &do_thing_that_takes_time, yield)
];
```

15 拆分注册

可能需要将模块注册拆分到几个翻译单元，而不让每次注册都一定要指明模块。

a.cpp:

```
luabind::scope register_a()
{
    return
        class_<a>("a")
            .def("f", &a::f)
            ;
}
```

b.cpp:

```
luabind::scope register_b()
{
    return
        class_<b>("b")
            .def("g", &b::g)
            ;
}
```

module_ab.cpp:

```
luabind::scope register_a();
luabind::scope register_b();

void register_module(lua_State* L)
{
```

```
    module("b", L)
    [
        register_a(),
        register_b()
    ];
}
```

16 错误处理

16.1 pcall errorfunc

在Lua文档[7]中说到，可以传个错误处理函数给 lua_pcall()。Luabind 在调用成员函数和自由函数时内部使用了 lua_pcall()。可以设置一个供 luabind 全局使用的错误处理函数:

```
typedef int(*pcall_callback_fun)(lua_State*);
void set_pcall_callback(pcall_callback_fun fn);
```

这主要用于在保护式调用失败时添加更多错误信息。更多关于如何使用 pcall_callback 函数的信息，请阅读 Lua 文档 pcall [8]章节中的 errfunc 信息。

关于如何从 Lua 获取调试信息的更多内容，请阅读 Lua 文档调试[9]章节。

从 pcall_callback 返回的信息放在 lua 栈顶以供访问。例如，如果想要以 luabind object 的形式访问它，可以这样做:

```
catch(error& e)
{
    object error_msg(from_stack(e.state(), -1));
    std::cout << error_msg << std::endl;
}
```

16.2 文件和行号

如果要向 luabind 生成的错误信息中添加文件名和行号，可以定义自己的 pcall errorfunc。可以修改这个回调函数以便更好地适应需求，基本的功能可以这样实现:

```
int add_file_and_line(lua_State* L)
{
   lua_Debug d;
   lua_getstack(L, 1, &d);
   lua_getinfo(L, "Sln", &d);
```

[7] http://www.lua.org/manual/5.1/manual.html
[8] http://www.lua.org/manual/5.1/manual.html#3.6
[9] http://www.lua.org/manual/5.1/manual.html#3.8

```
6     std::string err = lua_tostring(L, -1);
7     lua_pop(L, 1);
8     std::stringstream msg;
9     msg << d.short_src << ":" << d.currentline;
10
11    if (d.name != 0)
12    {
13       msg << "(" << d.namewhat << "␣" << d.name << ")";
14    }
15    msg << "␣" << err;
16    lua_pushstring(L, msg.str().c_str());
17    return 1;
18 }
```

更多关于可以添加哪些信息到错误信息中的内容，请阅读 Lua 文档调试[10]章节。

注意，通过 set_pcall_callback() 设置的回调函数只在 luabind 执行 lua 代码时使用。当调用 lua_call 的时候，需要自己提供错误处理函数。

16.3 lua panic

如果 lua 遇到了由 C++ 侧 bug 引起的致命错误时，会调用其内部的 panic 函数。例如，当调用 lua_gettable 于一个值上，而该值并不是个表，就会发生这种错误。如果在 lua 中发生同样的错误，它当然会报个错误消息然后失败。

缺省的 panic 函数会执行 exit() 退出程序。如果想要自己处理这个情况，不退出程序，需要使用 lua_atpanic 定义自己的 panic 函数。最好的从 panic 函数继续执行的办法是确保 lua 用 C++ 编译，并在 panic 函数中抛出异常。抛出异常，而不是使用 setjmp 和 longjmp，可以保证栈正确展开。

当 panic 函数被调用时，lua 的状态是无效的，唯一允许的操作是关闭它。

更多信息，请阅读 Lua 文档 3.6 [11]章节。

16.4 结构化异常(MSVC)

由于lua一般作为C库编译，因此任何从 lua 调用的回调函数在任何情况下都不能抛出异常。因为这一点，luabind 必须能捕获到所有异常，并将它们转换到合适的 lua 错误（通过调用 lua_error() ）。就是说，这里有一个 catch(...) 。

在Visual Studio中，catch(...) 将不止捕获 C++ 异常，它也捕获结构化异常，如段错误。这个意思是，如果被 luabind 调用的函数弄出了个无效的内存地址，又没注意到，lua 就会返回个错误消息“unknown exception”。

要补救，可以创建一个自己的异常翻译器:

[10] http://www.lua.org/manual/5.1/manual.html#3.8
[11] http://www.lua.org/manual/5.1/manual.html#3.6

```
1 void straight_to_debugger(unsigned int, _EXCEPTION_POINTERS*)
2 { throw; }
3
4 #ifdef _MSC_VER
5    ::_set_se_translator(straight_to_debugger);
6 #endif
```

这里会使用结构化异常，像段错误，实际上被调试器捕获到了。

16.5 错误消息

由 luabind 生成的错误消息有更深入的解释。

the attribute 'class-name.attribute-name' is read only

在名为*class-name*的类中没有名为*attribute-name*的数据成员，或没有那样名字的已注册的 setter 函数。见8.2 属性章节。

the attribute 'class-name.attribute-name' is of type: (class-name) and does not match (class-name)

当尝试向一个属性赋予一个不能转换成该属性类型的值时，报这个错误。

class-name() threw an exception, class-name:function-name() threw an exception

类的构造函数或成员函数抛出未知异常。已知异常有 const char* ， std::exception。见13 异常章节。

no overload of 'class-name:function-name' matched the arguments (parameter-types) no match for function call 'function-name' with the parameters (parameter-types) no constructor of class-name matched the arguments (parameter-types) no operator operator-name matched the arguments (parameter-types)

没有指定名字的函数/操作符使用你给出的参数。可能是函数名字拼写错误，或给了不正确的参数。该错误后有一个可能的候选函数列表以帮助你发现哪个参数使用了错误的类型。如果候选列表是空的，表示没有那个名字的函数。见7.2 签名匹配章节。

call of overloaded 'class-name:function-name*(*parameter-types)' is ambiguous ambiguous match for function call 'function-name' with the parameters (parameter-types) call of overloaded constructor 'class-name*(*parameter-types)' is ambiguous call of overloaded operator operator-name (parameter-types) is ambiguous

这个意思是调用的函数/操作符至少有一个重载版本与第一个重载版本一样匹配参数。

cannot derive from C++ class 'class-name'. It does not have a wrapped type.

17 构建选项

构建 luabind 时有一组配置选项可用。非常重要的是，工程中使用的配置选项要与构建库时的配置选项严格一致。例外的是 LUABIND_MAX_ARITY 和 LUABIND_MAX_BASES 是基于模板的选项，只影响库的使用（可以与库编译时的设置不同）。

如果不进行其他设置，就使用在luabind/config.hpp中的默认设置。

如果要修改库的设置，可以修改配置文件。它被包含在所有的 makefile 中。在那里还可以修改 Lua 和 Boost 的路径。

LUABIND_MAX_ARITY

控制注册到 Lua 的函数的元数的个数。不能注册参数个数多于这个宏定义数目的函数。默认值是5，所以如果函数元数大于这个值，就需要重新定义。这会严重增加编译时间。

LUABIND_MAX_BASES

控制在 luabind 中一个类可以派生的类的数量（类的数量在 base<> 中指定）。LUABIND_MAX_BASES 默认为4.这会严重增加编译时间。

LUABIND_NO_ERROR_CHECKING

如果定义了这个宏，即假设所有 lua 代码只使用合法的调用。如果有非法函数调用（比如给出了不匹配函数签名的参数），luabind 是不会检测到的，应用程序就可能会崩溃。可以在 release 构建版本（假设最终用户不会写 Lua 代码）中禁用错误检测。注意，如果函数被重载，仍然会进行函数参数匹配，否则就不知道调用哪个。当错误检测禁用时，函数仍然可以抛出异常。

如果函数抛出异常，会被 luabind 捕获，并通过 lua_error() 进行后续处理。

LUABIND_NO_EXCEPTIONS

该定义禁用所有 luabind 中的 try ， catch 和 throw。这在很多情况下，如进行无效的转换、调用 Lua 函数失败、返回值不能根据指定的策略转换等，会禁用运行期错误。Luabind 要求由 luabind 直接或间接调用的函数不要抛出异常（从 Lua 抛出异常是未定义行为）。

任何想要从 luabind 通过异常来获取错误报告的地方，都会把异常通过使用 set_error_callback() 和 set_cast_failed_callback()设置一个回调函数来替代。

LUA_API

如果想动态链接 lua ，可以将这个宏定义为编译器平台的导入关键字。在 Windows 的 Visual Studio 上如果动态链接 lua ，则这个宏应该定义为__declspec(dllimport) 。

LUABIND_EXPORT, LUABIND_IMPORT

如果想动态链接 luabind（在 Visual Studio 中），可以定义 LUABIND_EXPORT 为 __declspec(dllexport)，LUABIND_IMPORT 定义为 __declspec(dllimport)或者在 GCC 4 中定义为 __attribute__ ((visibility("default")))。注意，同时需要把 lua 也设置成动态链接，才能正常工作。

LUABIND_NO_RTTI

如果不想让 luabind 使用 dynamic_cast<> 就定义这个宏。它会禁用对象标识。

LUABIND_TYPE_INFO, LUABIND_TYPE_INFO_EQUAL(i1,i2), LUABIND_TYPEID(t), LUABIND_INVALID_TYPE_INFO

如果不想使用由 C++ 提供的 RTTI，可以提供自己的 type-info 结构体，并定义 LUABIND_TYPE_INFO 宏。自定义的 type-info 结构体必须是可复制的，必须是可与其他 type-info 结构体进行比较的。通过宏定义 LUABIND_TYPE_INFO_EQUAL() 来提供自己的比较函数。它将比较两个 type-info 结构体，如果代表的是相同的类型，就返回 true，否则返回 false。还需要提供一个函数用来生成 type-info 结构体，这通过 LUABIND_TYPEID() 完成。它将返回自定义的 type-info 结构体，并以一个类型作为参数，这是编译期参数。LUABIND_INVALID_TYPE_INFO 宏必须定义成一个无效类型。其他类型就不应该能生成这个类型信息。必须为所有要带有 type-info 的类，生成特化的 trait 类，才能正常使用。像这样:

```
class A;
class B;
class C;

template<class T> struct typeinfo_trait;

template<> struct typeinfo_trait<A> { enum { type_id = 0 }; };
template<> struct typeinfo_trait<B> { enum { type_id = 1 }; };
template<> struct typeinfo_trait<C> { enum { type_id = 2 }; };
```

如果像这样（使用整数来定义类型）建立了自己的RTTI系统，就可以让 luabind 通过下列定义来使用:

```
#define LUABIND_TYPE_INFO const std::type_info*
#define LUABIND_TYPEID(t) &typeid(t)
#define LUABIND_TYPE_INFO_EQUAL(i1, i2) *i1 == *i2
#define LUABIND_INVALID_TYPE_INFO &typeid(detail::null_type)
```

目前，通过 LUABIND_TYPE_INFO 定义的类型必须定义小于比较符。

NDEBUG

该定义会禁用断言，在 release 构建版本中必须要定义。

18 实现提示

类和对象在 Lua 中以 user data 实现。要保证 user data 确实是内部结果，我们在它们的 metatable 中打了标签。在 metatable 中包含了名字为 __luabind_classrep 的布尔成员，就被认为是由 luabind 导出的类。在 metatable 中包含了名字为 __luabind_class 的布尔成员，就被认为是 luabind 类的实例。

这意思是说，如果你在自己的 user data 的 metatable 中打上了相同名字的标签，那么很容易愚弄 luabind 从而使应用程序崩溃。

在 Lua 注册表中，Luabind 保存了一条名为 __luabind_classes 的表项。不要删除或改写它。

在全局表中，有个字为 super 的变量，用于每次 lua 类的构造函数中调用。这是为了让构造函数可以方便地调用到它的基类的构造函数。所以，如果有一个全局变量名为 super，它可能会被改写。这可能不是个最佳的解决方案，而且可能以后会去年这个限制。

Luabind在它注册的函数中使用两个 upvalue。第一个是个 user data，包含了个所有该函数的重载函数组成的列表，另一个是个值为0x1337的 light user data，这个值用于标识由 luabind 注册的函数。实际上是不可能会有这样值的指针作为第二个 upvalue 的。意思是，如果试图用 luabind 函数替换一个已存在的函数，luabind 会看一下第二个 upvalue 不是那个魔鬼数字，并替换掉。如果可以认出该函数是个 luabind 函数，它不会替换，但会添加一个它的重载版本。

在 luabind 的名字空间内，有另一个名为 detail 的名字空间。这个名字空间包含着非共有的类，并且不希望被用户直接使用。

19 FAQ

关于__cdecl和__stdcall?

如果你遇到问题说函数不能从void (__stdcall *)(int,int)转换到 void (__cdecl*)(int,int)。你可以修改工程设置，使编译器生成使用 __cdecl 调用转换的函数。这是个在开发者环境中的问题。

函数使用可变数量的参数?

你不能注册签名中使用了省略号的函数。因为省略号不是类型安全的，这无论如何应该被避免。

VC中内部结构溢出?

如果在 Visual Studio 中遇到了 fatal error C1204: compiler limit : internal structure overflow。你应该试一下将编译单元拆分成几个更小的。见15 拆分注册和8.8 拆分类注册。

VC中预编译头文件错误?

Visual Studio不喜欢在它的预编译头文件中有匿名的名字空间。如果你遇到了这个问题，可以禁用掉使用了 luabind 的编译单元的预编译头文件。

error C1076: compiler limit - VC中达到了内部堆限制

在Visual Studio中可能会遇到这个错误。要修正这个问题，必须在命令行选项中增加内部堆。我们可以通过 /Zm300 来编译测试套，但你可能需要更大的堆。

error C1055: compiler limit : VC中超出键范围

这个问题看起来像是一个程序中用了太多的 assert()，或者更确切点，是 __LINE__ 宏。似乎通过把 /ZI（用于编辑和继续的程序数据库）改为 /Zi（程序数据库）可以修正这个问题。

什么使得我的可执行程序变得巨大？

如果你在 debug 模式编译，可能会有很多调用信息和符号（luabind 由很多函数组成）。还有，如果在 debug 模式编译，是不会有优化的，luabind 依赖于编译器将函数在线展开。如果你在 release 模式编译，试一下在可执行程序上执行 strip 移除导出符号，这可以减小尺寸。

我们的测试发现，cygwin 上的 gcc 生成的可执行程序比其他平台的 gcc 和其他编译器生成的要大。

可以用 luabind 注册类模板吗？

可以，但只能注册类的显式实例化。因为没有对应于 C++ 模板的 Lua 概念。例如，你可以这样注册 std :: vector<> 的显示实例化:

```
module(L)
[
    class_<std::vector<int> >("vector")
        .def(constructor<int>)
        .def("push_back", &std::vector<int>::push_back)
];
```

需要为类注册析构函数吗？

不用，类的析构函数会在对象被回收时由 luabind 调用。注意 Lua 需要有对象的所有权才能回收。如果你将对象传递给 C++ 并放弃了所有权（使用 adopt 策略），它将不是由 Lua 所有，就不会被回收。

Fatal Error C1063 compiler limit : VC中编译栈溢出

VC6.5报很多警告上来，如果你的代码报了很多这种警告，只要使用 pragma 指令压制这个警告，可以解决这个问题。

在Windows平台将luabind链接为dll时崩溃

当你构建luabind，lua以及你的工程时，要确保使用运行时动态链接（即dll）。

我不能注册使用非const参数的的函数

因为没有办法获取Lua值的引用。看一下14.3 out_value和14.4 pure_out_value策略。

20 已知问题

- 不能把中间内含额外0字符的字符串作为 C++ 的成员的名字。
- 如果注册的类中有两个相同名字，相同签名的函数，目前不会报错。结果使用的是最后注册的那个。
- VC7中，类名不能取为test。

- 注册函数后又将其重命名，错误消息中仍然使用之前的名字。
- Luabind不支持类的虚继承。转换是通过静态指针偏移实现的。

21 Acknowledgments

Daniel Wallin 和Arvid Norberg 编写。版权所有2003。保留所有权利。
Evan Wies 贡献了测试，数不清的bug报告和有关特性的点子。
Luabind库从Dave Abrahams的 Boost.Python 中获取了极多的灵感.