by Waldemar Celes, Ariel Manzur.

tolua++中文参考手册【完整翻译】

官网：http://www.codenix.com/~tolua/tolua++.html

翻 译： **ChildhoodAndy**

BLOG: childhood.logdown.com

EMAIL：**dabing1022@gmail.com**

译者前言： 由于工作接触quick-cocos2d-x的缘故，接触到了luabinding。对luabinding的理解在一定程度上会加深对quick框架的理解（也包括cocos2dx的lua绑定），在学习的过程中便涉及到了tolua++。先到官网上大概看了下，有个参考手册，谷歌百度之，现在网络存在的版本有两种，一种是谷歌机器人直接翻译，非常生涩，一种是其他朋友自己翻译的，但不完整，其中的翻译也不是很理想，所以萌生了自己翻译的这么一个想法。我不能保证这个完整版没有错误，但会尽我最大的努力去还原作者的本意。如果读者发现部分的翻译有问题，非常欢迎在我的博客留言或者给我发送email邮件，我会认真阅读，如果正确我将会更新翻译，以便呈现在大家面前的是一个准确描述的译本。

---

tolua++是tolua的扩展版本，是一款能够集成C/C++与lua代码的工具。在面向C++方面，tolua++包含了一些新的特性比如：

- 支持 `std::string` 作为基本类型（这个可以由一个命令行选项关闭）
- 支持类模板

以及其他的特性还有一些bug的修复。

tolua这款工具，极大的简化了C/C++代码与lua代码的集成。基于一个干净的头文件（或者从实际头文件中提取），tolua会自动生成从lua访问C/C++功能的绑定代码。

tolua如何工作

要使用tolua，我们需要创建一个package文件（译者注：pkg文件），即一个从C/C++实际头文件整理后的头文件，列举出我们想导出到lua环境中的那些常量、变量、函数、类以及方法。然后tolua会解析该文件并且创建自动绑定C/C++代码到lua的C/C++文件。如果将创建的文件同我们的应用链接起来，我们就可以从lua中访问指定的C/C++代码。

我们从一些例子开始。如果我们指定下面的类似C头文件作为输入给tolua：

```
#define FALSE 0
#define TRUE 1

enum {
 POINT = 100,
 LINE,
 POLYGON
}
Object* createObejct (int type);
void drawObject (Object* obj, double red, double green, dou
ble blue);
int isSelected (Object* obj);
```

就会自动创建一个绑定上面代码到lua的C文件。因此，在lua代码里， 我们可以访问C代码。举个例子：

```
...
myLine = createObject(LINE)
...
if isSelected(myLine) == TRUE then
  drawObject(myLine, 1.0, 0.0, 0.0);
else
  drawObject(myLine, 1.0, 1.0, 1.0);
end
...
```

另外，考虑下面类似C++头文件

```
#define FALSE 0
#define TRUE 1
class Shape
{
  void draw (void);
  void draw (double red, double green, double blue);
  int isSelected (void);
};
class Line : public Shape
{
 Line (double x1, double y1, double x2, double y2);
 ~Line (void);
};
```

如果tolua输入加载该文件，就会自动生成一个C++文件，从而为我们提供lua层访问C++层所需要的对应的代码。因此，以下的的lua代码是有效的：

```
...
myLine = Line:new (0,0,1,1)
...
if myLine:isSelected() == TRUE then
 myLine:draw(1.0,0.0,0.0)
else
 myLine:draw()
end
...
myLine:delete()
...
```

传给tolua的package文件并不是真正的C/C++头文件，而是清理过的版本。tolua并没有实现对C/C++代码的完全解析，但是却能够解析暴露给lua的功能的声明。通常头文件可以被包括进package文件里，tolua将会提取出用户指定的代码以用于解析头文件。

如何使用tolua

tolua由两部分代码组成：可执行程序和静态库（an executable and a library）。可执行程序用于解析，读入package文件，然后输出C/C++代码，该代码提供了从lua层访问C/C++层的绑定。如果package文件是与C++类似的代码（例如包括类的定义），就会生成一份C++代码。如果package文件是与C类

似的代码（例如不包括类），就会生成一份C代码。tolua可接受一些选项。运行 `tolua -h` 显示当前可接受的选项。例如，要解析一个名为 `myfile.pkg` 生成一个名为 `myfile.c` 的绑定代码，我们需要输入：

```
tolua -o myfile.c myfile.pkg
```

产生的代码必须被编译并且和应用程序进行链接，才能提供给Lua进行访问。每个被解析的文件代表导出到lua的package。默认情况下，package的名称就是是输入文件的根名称（例子中为myfile），用户可以指定一个不同的名称给package：

```
tolua -n pkgname -o myfile.c myfile.pkg
```

package还应当被明确初始化。为了从C/C++代码中初始化package，我们需要声明和调用初始化函数。初始化函数被定义为：

```
int tolua_pkgname_open (lua_State*);
```

其中pkgname是被绑定package的名字。如果我们使用的是C++，我们可以选择自动初始化：

```
tolua -a -n pkgname -o myfile.c myfile.pkg
```

在这种情况下，初始化函数会被自动调用。然而，如果我们计划使用多个 `Lua State` ，自动初始化就行不通了，因为静态变量初始化的顺序在C++里没有定义。

tolua生成的绑定代码使用了一系列tolua库里面的函数。因此，这个库同样需要被链接到应用程序中。 `tolua.h` 也是有必须要编译生成的代码。

应用程序无需绑定任何package文件也可以使用tolua的面向对象框架。在这种情况下，应用程序必须调用tolua初始化函数（此函数被任何package文件初始化函数调用）：

```
int tolua_open (void);
```

基本概念

使用tolua的第一步就是创建package文件。我们从真正的头文件入手，将想要暴露给lua的特性转换成tolua可以理解的格式。tolua能够理解的格式就是一些简

单的C/C++声明。我们从下面几个方面来讨论：

文件包含

一个package文件可以包含另外的package文件。一般格式是：

`$pfile "include_file"`

一个package文件也可以包含常规的C/C++头文件，使用 `hfile` 或者 `cfile` 命令：

`$cfile "example.h"`

在这种情况下，tolua将会提取出被 `tolua_begin` 和 `tolua_end` 所封闭的代码，或者 `tolua_export` 所在的单行。以下面C++代码为例：

```
#ifndef EXAMPLE_H
#define EXAMPLE_H

class Example { // tolua_export

private:

    string name;
    int number;

public:

    void set_number(int number);

    //tolua_begin

    string get_name();
    int get_number();
};
// tolua_end

#endif
```

在这个例子中，不被tolua支持的代码（类的私有部分），还有 `set_number` 函数被留在了package文件之外。

最后，lua文件可以被包含进package文件中，使用 `$lfile`：

```
$lfile "example.lua"
```

tolua++新特性：从tolua++1.0.4版本以后，还有一种方式来包含源文件就是用 `ifile`：

```
$ifile "filename"
```

`ifile` 允许在文件名之后附带额外的可选的参数，举个例子：

```
$ifile "widget.h", GUI
$ifile "vector.h", math, 3d
```

`ifile` 默认会将整个文件原封不动的包含进去，但是文件的内容和额外的参数通过 `include_file_hook` 函数才能被包含进package文件中。

基本类型

tolua会自动映射C/C++的基本类型到lua的基本类型。

- `char`，`int`，`float` 和 `double` 类型被映射为lua中的 `number`
- `char*` 被映射为lua中的 `string`
- `void*` 被映射为lua中的 `userdata`

C/C++中的数据类型前面可能有修饰语（如unsigned, static, short, const等等）。然而我们要注意到tolua会忽略基本类型前面的修饰语 `const`。因此，我们给lua传递一个基本类型常量然后再从lua中传递回给C/C++代码，常量到非常量的转换会被悄悄的完成。

C/C++函数也可以对lua对象进行明确的操作。`lua_Object` 被认为是一个基本类型，任何lua值都符合。

tolua++新特性： C++中的 `string` 类型同样被认为是基本类型，会被当作值传递给lua(使用c_str()方法)。这个功能可以使用命令行选项 `-S` 进行关闭。

用户自定义类型

在package文件里的所有其他类型都会被认为是用户自定义类型。它们会映射到lua的userdata类型。lua只能存储指向用户自定义类型的指针；但是，tolua会自

动采取措施来处理引用和值。例如，如果一个函数或方法返回一个用户定义类型的值，当这个值返回给lua的时候，tolua会分配一个克隆对象，同时会设置垃圾收集标记，以便在lua不再使用该对象时会自动释放。

对于用户定义类型，常量是被保留的，因此将用户自定义类型的非常量作为常量传递给一个函数时，会产生类型不匹配的错误。

NULL和nil

C/C++的NULL或0指针映射到lua中的nil类型。反之，nil却可以被指定为任何C/C++指针类型。这对任何类型都有效：`char*`，`void*`以及用户自定义类型指针。

Typedefs

真实头文件的包含

绑定常量

tolua支持两种绑定常量的方式：`define`和`enum`。

- `define`通常的格式是：`#define NAME [ VALUE ]`

上面的VALUE是可选的。如果这样的代码出现在被解析的文件中，tolua会将NAME作为lua的全局变量，该全局变量是C/C++的常量，值为VALUE。这里只接受数字常量。

tolua++新特性：所有的其他预处理指令会被忽略。

- `enum`的一般格式：

```
enum {
  NAME1 [ = VALUE1 ] ,
  NAME2 [ = VALUE2 ] ,
  ...
  NAMEn [ = VALUEn ]
};
```

同样的，tolua会创建一系列全局变量，命名为`NAMEi`，对应着各自的值。

绑定外部变量

全局外部变量也可以被导出。在清理的头文件中可以这样指定：

```
[extern] type var;
```

tolua会将这样的声明绑定到lua全局变量中。因此，我们可以自然的访问C/C++变量。如果变量不是常量，我们也可以在lua中赋予该变量新的值。全局数组也可以绑定到lua中。数组可以是任何类型。数组对应的lua类型是以数字作为下标的 `table` 。但是，要注意的是lua中下标1对应C/C++中的下标0。数组一定要定长。例如：

```
double v[10]
```

tolua++新特性：外部变量可以使用 `tolua_readonly` 修饰符。

绑定函数

函数也被指定为传统的C/C++声明：

```
type funcname (type1 par1[, type2 par2[,...typeN parN]]);
```

返回值类型是 `void` ，表示没有返回值。函数也可以没有参数。这种情况下， `void` 可以被指定在参数列表的位置处。参数的类型必须遵循发布的规则。tolua创建一个lua函数绑定着C/C++函数。当从lua层调用函数时，参数类型必须对应C/C++的类型，否则tolua会产生一个错误并报告错误的参数。如果参数名是省略的，tolua会自定命名，但是类型必须是基本类型或者是以前使用过的用户定义类型。

数组

tolua同时支持以数组作为参数的函数或方法。关于数组有一件很美妙的事情就是，当数组映射到lua的table后，当C/C++函数改变了数组里面的值，会实时更新到lua的table中。

数组的大小必须预先定义好。例如：

```
void func (double a[3]);
```

在tolua里，这是一个有效的函数声明。在lua里调用这个函数如：

```
p = {1.0,1.5,8.6}
func (p)
```

数组长度不一定是一个常数表达式，也可以是在运行时能计算出结果的表达式。例如：

```
void func (int n, int m, double image[n*m]);
```

这个是合法的。因为表达式 `n*m` 在函数的绑定范围是有效的。但是必须考虑到，tolua使用动态分配的方式绑定这个函数，这一方式会降低性能。

尽管数组大小确定，但是更重要的是意识到，所有数组传递给实际C/C++函数是使用局部变量的方式进行传递。因此，如果C/C++函数想要保留这个数组供以后使用，绑定代码不会正确工作。

重载函数

tolua支持重载函数。两个同名函数是基于映射到lua的参数类型来区分的。因此

```
void func (int a);
void func (double a);
```

在C/C++中这代表两个不同的函数，然而在tolua里它们是同一个函数，因为int和double类型都映射到lua中的number。

另一个棘手的情况是，当遇到指针时。假设：

```
void func (char* s);
void func (void* p);
void func (Object1* ptr);
void func (Object2* prt);
```

尽管在C++中这是4个不同的函数，但是映射到lua里面的声明： `func(nil)` 却匹配了上面全部。

重要的是，tolua决定哪个函数将在运行时被调用，尝试映射到那些提供的函数。tolua第一次会尝试调用最后一个指定的函数；如果失败，tolua接着尝试前一个函数。这个过程一直持续到找到一个能成功运行的代码或者到第一个函

数。因为这个原因，不匹配的错误信息的发出，总是基于第一个函数的规格（the mismatching error message, when it occurs, is based on the first function specification）。当性能很重要时，我们应该把最常用的函数放在最后一个，因为它总是被第一个尝试调用。

tolua支持C的重载函数。

参数默认值

函数最后几个参数可以带关联默认值。在这种情况下，如果函数参数不足，默认值就会被设定。指定默认值的格式如同在C++代码使用的一样：

```
type funcname (..., typeN-1 parN-1 [= valueN-1], typeN parN
 [= valueN]);
```

tolua实现这个功能并没有用任何C++机制，所以，它也可以用来绑定C函数。

我们也可以对数组的元素指定默认值（尽管不能为数组本身指定默认值）。例如：

```
void func (int a[5]=0);
```

上述代码会将数组元素的默认值设置为0，因此可以在lua中直接调用而无需初始化。

对于lua对象类型(`lua_Object`)，tolua定义了一个常量用来指定nil为默认值：

```
void func (lua_Object lo = TOLUA_NIL);
```

tolua++新特性：C++类构造函数作为默认参数是有效的。例如：

```
void set_color(const Color& color = Color(0,0,0));
```

多个返回值

在lua中，一个函数可以有多个返回值。tolua使用这个特性来模拟值的引用传递。如果一个函数的参数是一个指针或者基本类型的引用或者是用户定义类型的指针或指针的引用，tolua接受对应类型的传入和返回，除了常规的函数返回

值，如果还有的话，附带更新过的参数值。

例如，考虑用于交换两数值的C函数：

```
void swap (double* x, double* y);
```

或者

```
void swap (double& x, double& y);
```

如果在package文件中声明此函数，tolue绑定该函数为输入两个参数，返回两个值。所以，正确的Lua代码为：

```
x,y = swap(x,y)
```

如果不使用输入值，lua会自动使用默认参数值调用函数而不需要指定它们。

```
void getBox (double* xmin=0, double* xmax=0, double* ymin=0
, double* ymax=0);
```

lua代码：

```
xmin, xmax, ymin, ymax = getBox()
```

以用户定义类型为例：

```
void update (Point** p);
```

或者

```
void update (Point*& p);
```

绑定结构域

用户定义类型可以很好的被tolua绑定。对于每一个变量或函数类型，不符合基本类型， tolua自动创建一个标签的用户数据代表的C/C++类型。 如果类型对应结构，Lua可以通过下标直接访问结构里面的内容，indexing a variable that holds an object of such a type. 在C代码，这些类型通常是用typedef定义的：

typedef struct [name] { type1 fieldname1; type2 fieldname2; ... typeN fieldnameN; } typename; 如果这段代码正在被插入到package文件中，tolua允许任何拥有类型对象名的对象反问任何索引列出的变量的字段名字（If such a code is inserted in the package file being processed, tolua allows any variable that holds an object of type typename to access any listed field indexing the variable by the field name.）。例如，如果var持有对象, var.fieldnamei可以访问fieldnamei里面的东西。 块内的数组同样被映射。 typedef struct { int x[10]; int y[10]; } Example; Binding classes and methods Tolua支持C++的类定义。事实上，tolua能很自然的处理单一继承和多态性。以下各小节将会介绍什么可以暴露给lua，在类定义的时候（The subsections below describe what can be exported by a class definition.）。

常量、变量、函数重命名

当导出常量，变量和函数（类成员函数或者不是）时，我们可以重命名它们，这样，它们就会从C/C++副本中用不同的名字得到绑定。为了实现这点，我们在字符 `@` 后跟上lua中要引用的名字即可。例如：

```
extern int cvar @ lvar;

#define CNAME @ LNAME

enum {
  CITEM1 @ LITEM1,
  CITEM2 @ LITEM2,
  ...
};

void cfunc @ lfunc (...);

class T
{
   double cfield @ lfield;
   void cmeth @ lmeth (...);
   ...
};
```

在这种情况下，全局变量 `cvar` 在lua中用 `lvar` 标识，不断 服务商 通过LNAME，等等。请注意，类不能更名，因为他们代表在C类型 此功能允许重命

名功能，在C超载，因为我们可以选择出口同一Lua中命名了两个不同的C函数： 无效glVertex3d @ glVertex（双十，双Y，双ž = 0.0）; 无效glVertexdv @ glVertex（双V [3] = 0.0）; 重命名类型 可以重命名的类型使用 $命名 指令对包装的文件，使用的格式： $命名real_name @ new_name 对命名参数可Lua中模式。例如： $命名^ _ + @ $命名hash_map“字符串，矢量”@ StringHash 第一个例子将删除所有在所有类型的开始强调，第二个将重命名模板类型hash_map“字符串，矢量”的StringHash。一旦改名，每个只能由他们的访问类型的新名称，例如，Lua中的表： StringHash：新的（）

存储附加字段

最后，重要的是要知道，即使持有C/C++对象的变量其实是将lua中的 `userdata` ，tolua创建了一种机制，使得我们能够在这些对象上存储附加的字段。也就是说，这些对象可以看作为常规的lua表。

```
obj = ClassName:new()
obj.myfield = 1  -- 尽管ClassName中并没有myField字段
```

这样的构造是可能的，因为，如果需要的话， tolua会自动创建一个lua表，并与该对象关联。所以说，该对象可以存储没有映射到C/C++的附加字段，但实际上附加字段是存储在复合表中。lua程序员用统一的方式访问C/C++功能和这些额外的字段。请注意，事实上，这些额外的字段名如果和C/C++对象的字段名相同的话，会覆盖的C/C++的字段或者方法。

tolua++附加功能

多个变量声明

同一类型的多个变量可以一次性声明，如:

```
float x,y,z;
```

这将会创建3个不同的浮点型变量。确保逗号之间没有任何空格，否则会解析错误。

tolua_readonly

任何变量声明都可以使用 `tolua_readonly` 修饰符来确保变量只读，即使该变量为非常量类型。例如:

```
class Widget {

    tolua_readonly string name;
};
```

这个特性可以用来'hack'一些不支持的东西如 `operator->` 。考虑下面这个pkg文件:

```
$hfile "node.h"
$#define __operator_arrow operator->()
$#define __get_name get_name()
```

`node.h` 文件内容如下:

```
template class<T>
class Node { // tolua_export

private:
    string name;
    T* value;

public:

    T* operator->() {return value;};
    string get_name() {return name;};

    // tolua_begin

    #if 0
    TOLUA_TEMPLATE_BIND(T, Vector3D)

    tolua_readonly __operator_arrow @ p;
    tolua_readonly __get_name @ name;
    #endif


    Node* next;
    Node* prev;

    void set_name(string p_name) {name = p_name;};

    Node();
};
// tolua_end
```

虽然这样对头文件的处理不是很漂亮，但是却解决了下面几个问题：

- `operator->()` 方法可以通过在lua中使用对象的变量 `p` 来调用
- `get_name()` 方法可以通过在lua中使用对象的变量 `name` 来调用

lua用法举例：

```
node = Node_Vector3D_:new_local()
-- do something with the node here --
print("node name is "..node.name)
print("node value is ".. node.p.x ..", ".. node.p.y ..", ".
. node.p.z)
```

tolua++会忽略掉所有的预处理命令（除了#define), 但 `node.h` 仍然是一个有效的C++头文件，同时也是tolua的一个有效的源，这样就避免了维护2个不同的文件，即使是具有不一样的功能的对象。

像变量一样去重命名函数的功能特性可能会被扩展进将来的版本中。

在命令行中定义值

从版本1.0.92开始，命令行选项 `-E` 在tolua++运行的时候允许我们传值给luaState，就像GCC的 `-D` 一样。例如：

```
$ tolua++ -E VERSION=5.1 -E HAVE_ZLIB package.pkg > package
_bind.cpp
```

上述命令将会在全局表 `_extra_parameters` 中添加两个字段，字符串值为“5.1”的“VERSION”，还有布尔值为true的“HAVE_ZLIB”。就目前而言，我们还不能使用这些值，除非用户自定义脚本。

使用C++ typeid

从版本1.0.92开始，命令行选项 `-t` 变得可用。这将会产生一个对空宏 `Mtolua_typeid` 的调用的列表，该列表包含了C++ `type_infoobject` 和区分其他类型的名字。例如，如果你的package文件中绑定了2个类， `Foo` 和 `Bar` ,使用 `-t` 会有如下输出：

```
#ifndef Mtolua_typeid
#define Mtolua_typeid(L,TI,T)
#endif
 Mtolua_typeid(tolua_S,typeid(Foo), "Foo");
 Mtolua_typeid(tolua_S,typeid(Bar), "Bar");
```

`Mtolua_typename` 的实现留给读者作为练习。

导出实用函数

tolua本身为lua导出了一些实用的函数，包括了面向对象的框架。tolua使用的package文件如下：

```
module tolua
{
 char* tolua_bnd_type @ type (lua_Object lo);
 void tolua_bnd_takeownership @ takeownership (lua_Object l
o);
 void tolua_bnd_releaseownership @ releaseownership (lua_Ob
ject lo);
 lua_Object tolua_bnd_cast @ cast (lua_Object lo, char* typ
e);
 void tolua_bnd_inherit @ inherit (lua_Object table, lua_Ob
ject instance);

/* for lua 5.1 */
 void tolua_bnd_setpeer @ setpeer (lua_Object object, lua_O
bject peer_table);
 void tolua_bnd_getpeer @ getpeer (lua_Object object);
}
```

tolua.type(var)

返回一个代表对象类型的字符串。例如， `tolua.type(tolua)` 返回字符串 `table` ， `tolua.type(tolua.type)` 返回 `cfunction` 。类似的，如果 `var` 是一个存储着用户自定义类型 `T` 的变量， `tolua.type(var)` 将会返回 `const T` 或者 `T` , 取决于它是否是一个常量引用。

tolua.takeownership(var)

接管对象引用变量的所有权，意味着当对象的所有引用都没有的话，对象本身将会被lua删除。

tolua.releaseownership(var)

释放对象引用变量的所有权。

tolua.cast(var, type)

改变 `var` 的元表以便使得它成为 `type` 类型。 `type` 须是一个完整的C类型对象（包括命名空间等）字符串。

tolua.inherit(table, var)

（tolua++新特性）tolua++会将 `table` 作为对象和 `var` 具有一样的类型，当必要的时候使用 `var` 。举个例子，考虑下这个方法：

```
void set_parent(Widget* p_parent);
```

一个lua对象可以这么使用：

```
local w = Widget()
local lua_widget = {}
tolua.inherit(lua_widget, w)

set_parent(lua_widget);
```

注意这只是使得 `table` 能够在必要的时候被识别为 `Widget` 类型。为了能够访问Widget的方法，你必须要实现你自己的对象系统。下面是个简单的例子：

```
lua_widget.show = Widget.show

lua_widget:show()
-- 调用Widget的show方法
-- table作为self，因为 lua_widget继承于一个widget实例
-- tolua++会把self识别为Widget，并且调用方法
```

当然，一个更好的办法就是为lua对象增加 `__index` 元方法。

嵌入Lua代码

tolua允许我们将lua代码嵌入到生成的C/C++代码中。要做到这一点，它对指定的lua代码进行编译，并且在生成的代码中产生一个C常量字符串，存储下来相应的字节码。当package文件被打开的时候，字符串被执行。嵌入的lua代码格式如下：

```
$[

嵌入的lua代码
...

$]
```

以下面的.pkg摘录为例：

```
/* Bind a Point class */
class Point
{
 Point (int x, int y);
 ~Point ();
 void print ();
 ...
} CPoint;

$[

-- Create a Point constructor
function Point (self)
 local cobj = CPoint:new(self.x or 0, self.y or 0)
 tolua.takeownership(cobj)
 return cobj
end

$]
```

绑定这样的代码之后，我们可以在lua中这么写：

```
p = Point{ x=2, y=3 }
p:print()
...
```