**C++中闭包的简单实现**

01 Dec 2015

这个问题的起源是想把成员函数封装为回调函数，众所周知一个比较好的选择是tr1里的function和bind。

function对不同类型的函数指针进行统一的封装。例如：

int foo() {...}

std::tr1::function<int ()> bar = f;

bar();

如果函数带参数或者是成员函数，使用bind可以实现一个闭包的作用，将对象本身、参数都封装进来，返回一个function对象。例如：

struct X {

int foo(int data) {

std::cout << "X::foo " << data << std::endl;

return 0;

}

};//X

X x;

std::tr1::function<int (int)> f = std::tr1::bind(&X::foo, &x, std::tr1::placeholders::\_1);

f(1);

使用bind和function，可以很轻松的打平普通函数和C++成员函数在作为回调函数上使用的区别（静态成员函数等同于普通函数，因为即便对于普通函数，也可能存在namespace）。

**1. 那么思考一下，bind和function是如何实现的呢？**

回到闭包的定义，其实只要把函数指针、函数参数以及对象指针封装到一个结构里就可以了。这里主要面临两个问题：

1. 参数类型不同
2. 参数个数不同

解决方案也很简单：

1. 使用模板
2. 为可能的参数个数都定义一个闭包的结构体。  
   没错！就是可能有多少个参数，我们就定义多少个对应的闭包结构体。

**2 使用**

先来看下实现后的用法。  
最近正好在看protobuf的源码，在google/protobuf/stubs/common.h中有一个类似的实现。  
试着实现了一个简化的版本，先看下使用方法：

void foo() {

std::cout << "foo" << std::endl;

}

template<typename type>

void foo(type data) {

std::cout << "foo data=" << data << std::endl;

}

class Foo {

public:

void foo() {

std::cout << "Foo::foo" << std::endl;

}

template<typename type>

void foo1(type data) {

std::cout << "Foo::foo data=" << data << std::endl;

}

};//Foo

int main()

{

Foo f;

Closure\* closure;

std::cout << "address of object in main : " << &f << std::endl;

closure = NewCallback(foo);

closure->Run();

closure = NewCallback(&f, &Foo::foo);

closure->Run();

closure = NewCallback(foo<const char [7]>, "string");

closure->Run();

closure = NewCallback(&f, &Foo::foo1<double>, 100.0);

closure->Run();

return 0;

}

NewCallback函数返回一个new之后的closure,closure不需要手动delete。  
直接运行closure的Run方法就可以运行封装前的函数及对应参数。

**3 实现**

逐步看下是如何实现的。

**3.1 首先定义一个Closure基类:**

class Closure {

public:

Closure() {}

~Closure() {}

virtual void Run() = 0;

};//Closure

基类定义了一个纯虚函数Run，用于执行封装的函数，子类需要单独提供实现。

普通函数和成员函数需要定义两个不同的闭包子类

**3.2 先看下无参数情况下的普通函数对应的闭包：**

class FunctionClosure0 : public Closure {

public:

typedef void (\*FunctionType)();

FunctionClosure0(FunctionType f) :

\_f(f) {

}

~FunctionClosure0() {

}

virtual void Run() {

\_f();

delete this;

}

private:

FunctionType \_f;

};//FunctionClosure0

其中，FunctionType用于存储函数指针，注意对象本身通过NewCallback函数new而来，因此在Run方法里调用delete this清理自身，防止忘了delete而导致内存泄露。  
在protobuf里还提供了了NewPermanentCallback，使用该方法产生的Closure对象在调用Run方法时，不会调用delete this，实现上就是为Closure类增加了一个bool用于记录是否需要调用delete方法。

该Closure对应的NewCallback为：

Closure\* NewCallback(void (\*function)()) {

return new FunctionClosure0(function);

}

所有的NewCallback都返回指向Closure对象的指针，因此传入函数指针类型不同，但返回类型是相同的。

**3.3 接着看下无参数成员函数对应的闭包：**

template <typename Class>

class MethodClosure0 : public Closure {

public:

typedef void (Class::\*MethodType)();

MethodClosure0(Class\* object, MethodType m) :

\_object(object),

\_m(m) {

}

~MethodClosure0() {

}

virtual void Run() {

std::cout << "address of object in Run : " << \_object << std::endl;

(\_object->\*\_m)();

delete this;

}

private:

Class\* \_object;

MethodType \_m;

};//MethodClosure0

可以看到跟普通函数对应的闭包类差别不大，主要有：

1. 需要传入class模板
2. 函数定义不同，普通函数为typedef void (\*FunctionType)()，成员函数为typedef void (Class::\*MethodType)()
3. 构造函数需要传入class对应的对象指针，并保存为成员变量
4. 调用时使用(\_object->\*\_m)()的方式

对应的NewCallback:

template <typename Class>

Closure\* NewCallback(Class\* object, void (Class::\*method)()) {

return new MethodClosure0<Class>(object, method);

}

类似的，通过模板可以实现不同参数个数的闭包对象，每种情况对应需要实现两个闭包对象和两个NewCallback函数。

**3.4 例如对于单参数：**

template <typename Arg1>

class FunctionClosure1 : public Closure {

public:

typedef void (\*FunctionType)(Arg1);

FunctionClosure1(FunctionType f, Arg1 arg1) :

\_f(f),

\_arg1(arg1) {

}

~FunctionClosure1() {

}

virtual void Run() {

\_f(\_arg1);

delete this;

}

private:

FunctionType \_f;

Arg1 \_arg1;

};//FunctionClosure1

template <typename Class, typename Arg1>

class MethodClosure1 : public Closure {

public:

typedef void (Class::\*MethodType)(Arg1);

MethodClosure1(Class\* object, MethodType m, Arg1 arg1) :

\_object(object),

\_m(m),

\_arg1(arg1) {

}

~MethodClosure1() {

}

virtual void Run() {

std::cout << "address of object in Run : " << \_object << std::endl;

(\_object->\*\_m)(\_arg1);

delete this;

}

private:

Class\* \_object;

MethodType \_m;

Arg1 \_arg1;

};//MethodClosure1

template <typename Arg1>

Closure\* NewCallback(void(\*function)(Arg1), Arg1 arg1) {

return new FunctionClosure1<Arg1>(function, arg1);

}

template <typename Class, typename Arg1>

Closure\* NewCallback(Class\* object, void (Class::\*method)(Arg1), Arg1 arg1) {

return new MethodClosure1<Class, Arg1>(object, method, arg1);

}

区别就是增加了参数模板，以及在Closure子类里保存参数。  
实际使用中可以选择生成多少个参数对应的类和NewCallback，代码的生成是有规律可循的，有兴趣的朋友可以写个脚本实现下。

protobuf或者其他常见的关于closure的实现里要复杂一些，例如判断对象是否赋值过类似的需求。

完整的代码示例放在了[这里](https://gist.github.com/yingshin/e6f42dec075e5791c232)

**4. 继续思考**

完整的闭包考虑的问题比这篇文章里提到的要多得多，比如传入的对象指针如何才能保证在异步回调时没有被析构，如果已经被析构怎么样保证不出问题，这些就需要shared\_ptr/weak\_ptr登场了。  
又或者比如这么执行boost::function<void ()> function; function()会不会有问题？是否有判断空值的接口，自己来实现的话，这个接口是否有必要提供?

在chrome源码[bind.h](https://cs.chromium.org/chromium/src/base/bind.h?dr=CSs&q=bind.h&sq=package:chromium)里，bind的参数里会额外传入一个参数，取值为：base::Unretained(closure不拥有所有权，用于AddRef接口）, base::Owned(closure拥有所有权)等，另外通过使得Foo继承自RefCountedThreadSafe<Foo>或者内部维护一个WeakPtrFactory<Foo>对象等，实现了类似于弱回调的作用。同时chrome默认bind参数为const T&的形式，防止了参数的拷贝。也是值得思考的。有兴趣的同学可以看下[代码](https://cs.chromium.org/chromium/src/base/bind_helpers.h?dr=CSs&sq=package:chromium)和[文档](https://www.chromium.org/developers/coding-style/important-abstractions-and-data-structures#TOC-base::Callback-and-base::Bind-)