**使用std::function作为函数入参**

# 1. 关于std::function()

在C语言的时代，我们可以使用函数指针来吧一个函数作为参数传递，这样我们就可以实现回调函数的机制。到了C++11以后在标准库里引入了std::function模板类，这个模板概括了函数指针的概念  
函数指针只能指向一个函数，而std::function对象可以代表任何可以调用的对象，比如说任何可以被当作函数一样调用的对象。  
当你创建一个函数指针的时候，你必须定义这个函数签名（表征这个函数的入参，返回值等信息）；同样的，当你创建一个std::function对象的时候，你也必须指定它所代表的可调用对象的函数签名。这一点可以通过std::function的模板参数来实现。  
举个例子来说，如果要定义一个std::function对象func，这个对象可以表示任何有如下函数签名的可调用对象的，

bool(const std::unique\_ptr<Widget>)&, // C++11里面用来比较两个

const std::unique\_ptr<Widget>&) //std::unique\_ptr<Widget>对象的函数签名

你可以这么写，

std::function<bool(const std::unique\_ptr<Widget>&,

const std::unique\_ptr<Widget>&)> func;

这是因为lambda表达式产生了可调用的对象，这个对象这里称做一个闭包（closure），可以保存在std::function对象里面。  
closure（闭包）的定义是，一个函数和它所引用的非本地变量（非lambda表达式内部定义的变量）的一个集合。

# 2. 使用std::function作为函数入参

## 2.1 基于传值的方式传递参数

参看下面一段代码，实现了一个注册回调函数的机制，

#include <fonctional>

void registerCallBack(std::function<void()>);

入参std::function<void()>是一个模板类对象，它可以用一个函数签名为void()的可调用对象来进行初始化；上述实现里面是一个传值调用。我们来看一下它的调用过程，

// 方法（A）

registerCallBack([=]{

.... // 回调函数的实现部分

})

这里使用了lambda表达式作为函数的入参，正如前面所说的lambda表达式会生成一个匿名的闭包（closure），基于这个闭包构造了一个std::function<void()>的对象，然后通过传值调用的方式把这个对象传递registerCallBack函数中使用。

# 2.2 基于引用的方式传递参数

当然我们还可以如下实现这个注册函数，入参通过const引用的方式传递，这里的引用必须是const的，这是因为调用registerCallBack函数的地方生成了一个临时的std::function()对象，是一个右值，否则编译会报错。

//方法(B)

void registerCallBack(std::function<void()> const&)

这两者的区别就在于，在registerCallBack函数内部怎么使用这个入参，如果只是简单的调用一下std::func()类，那么两种都没有问题，可能使用引用的效率更高；如果register函数内部需要保存这个std::func()，并用于以后使用，那么方法A直接保存没有问题，方法B就必须做一次拷贝，否则方法B中，当临时的对象销毁时，有可能出现引用悬空的问题。

# 2.3 传值方式下的std::function对象保存

如果我们要在registerCallBack函数内部保存这个传入的function对象，我们可以使用转移操作std::move，这样的效率更高，

class CallBackHolder

{

public:

void registerCallBack(std::function<void()> func)

{

callback = std::move(func);

}

private:

std::function<void()> callback;

}

# 3. 类的成员函数作为函数入参

类的成员函数都会默认有个隐藏的this指针，所以不像普通的函数直接作为入参就可以了。

## 3.1 使用std::bind()和std::function来实现

std::function是通用的多态函数封装器，它的实例可以存储、复制以及调用任何可以调用的目标：函数，lambda表达式/bind表达式或其他函数对象，还有指向成员函数指针和指向数据成员指针；  
std::bind接受一个函数（或者函数对象），生成一个重新组织的函数对象；  
看下面一个例子，classA提供了一个注册函数，用来注册一个回调函数

class classA

{

typedef std::function<void(int i)> callback\_t;

...

void registCb(callback\_t func)

{cbHandle = std::move(func);}

private:

callback\_t cbHandle;

};

另一个类classB需要注册自己的一个成员函数作为回调函数到classA中，这里就可以使用std::bind函数来实现，

class classB

{

public:

classB(classA& cA)

{

cA.registCb(std::bind(&classB::handle, this, std::placeholders::\_1));

}

};

* bind函数中显示的传递classB的this指针作为第一个参数给回调函数；
* std::placeholders:\_1代表一个占位符，用于回调函数显式的入参；

Effective Modern C++中专门有一节解释过std::bind的方式比较繁琐，并且有时侯会有一些局限性，所以在引入了lambda表达式后就可以用lambda表达式来替代std::bind实现函数回调注册。

## 3.2 使用lambda表达式实现

使用lambda表达式的方式可以简化这一个过程，参看如下一段代码，classB注册一个成员函数作为回调函数到classA中，classA会保存这个回调函数（std::function对象）到成员变量中，用于后面使用，

#include <iostream>

#include <functional>

#include <memory>

class classA

{

typedef std::function<void(int i)> callback\_t;

public:

classA() {}

~classA() {}

void handle(int i)

{

std::cout << "classA::handle" << std::endl;

cbHandle(i);

}

void registCb(callback\_t func)

{cbHandle = std::move(func);}

private:

callback\_t cbHandle;

};

class classB

{

public:

classB(classA& cA)

{

cA.registCb([this](int i){classB::handle(i);});

}

~classB() {}

void handle(int i)

{

std::cout << "classB, handle message" << i << std::endl;

}

};

int main()

{

classA testa;

classB testb(testa);

testa.handle(10);

}

* lambda表达式中捕获了classB的this指针
* 使用std::move的方式保存function对象到classA中

# [c++11 std::move() 的使用](https://www.cnblogs.com/SZxiaochun/p/8017349.html)

std::move函数可以以非常简单的方式将左值引用转换为右值引用。（左值、左值引用、右值、右值引用 参见：<http://www.cnblogs.com/SZxiaochun/p/8017475.html>）

通过std::move，可以避免不必要的拷贝操作。

std::move是为性能而生。

std::move是将对象的状态或者所有权从一个对象转移到另一个对象，只是转移，没有内存的搬迁或者内存拷贝。

如string类在赋值或者拷贝构造函数中会声明char数组来存放数据，然后把原string中的 char 数组被析构函数释放，如果a是一个临时变量，则上面的拷贝，析构就是多余的，完全可以把临时变量a中的数据直接 “转移” 到新的变量下面即可。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | #include <iostream>  #include <utility>  #include <vector>  #include <string>  int main()  {      std::string str = "Hello";      std::vector<std::string> v;      //调用常规的拷贝构造函数，新建字符数组，拷贝数据      v.push\_back(str);      std::cout << "After copy, str is \"" << str << "\"\n";      //调用移动构造函数，掏空str，掏空后，最好不要使用str      v.push\_back(std::move(str));      std::cout << "After move, str is \"" << str << "\"\n";      std::cout << "The contents of the vector are \"" << v[0]                                           << "\", \"" << v[1] << "\"\n";  } |

# [左值、左值引用、右值、右值引用](https://www.cnblogs.com/SZxiaochun/p/8017475.html)

#### 1、左值和右值的概念

         左值是可以放在赋值号左边可以被赋值的值；左值必须要在内存中有实体；  
         右值当在赋值号右边取出值赋给其他变量的值；右值可以在内存也可以在CPU寄存器。  
         一个对象被用作右值时，使用的是它的内容(值)，被当作左值时，使用的是它的地址**。**

#### 2、引用

        引用是C++语法做的优化，引用的本质还是靠指针来实现的。引用相当于变量的别名。

        引用可以改变指针的指向，还可以改变指针所指向的值。

        引用的基本规则：

1. 声明引用的时候必须初始化，且一旦绑定，不可把引用绑定到其他对象；即引用必须初始化，不能对引用重定义**；**
2. 对引用的一切操作，就相当于对原对象的操作。

#### 3、左值引用和右值引用

    3.1 左值引用  
         左值引用的基本语法：type &引用名 = 左值表达式；  
  
    3.2 右值引用

        右值引用的基本语法type &&引用名 = 右值表达式；

        右值引用在企业开发人员在代码优化方面会经常用到。

        右值引用的“&&”中间不可以有空格。