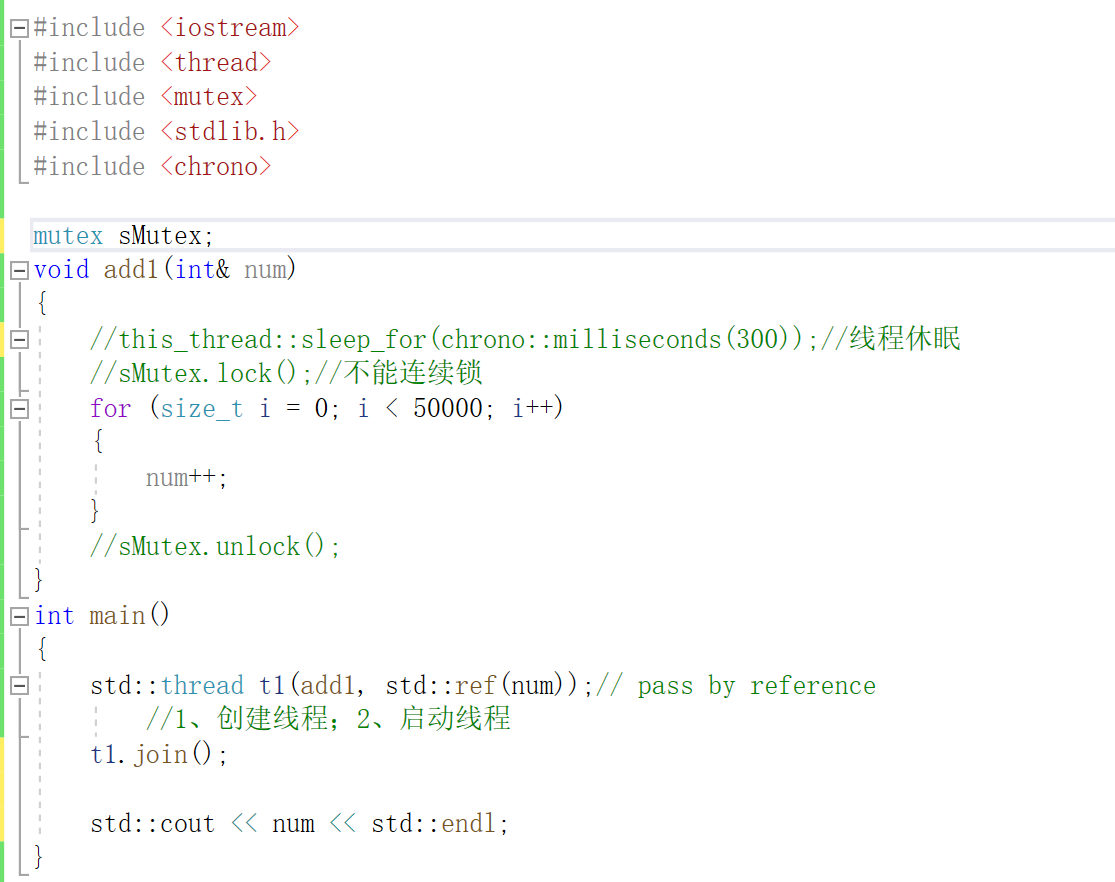
1 std::thread 入口函数pass by reference需要用std:: ref(实参)



其实也可以不使用std:: ref(实参)，不过形参必须用const修饰

## 2 使用detach带来的问题

2.1 主线程执行完，子线程未执行应用的操作。

int detachTest()

{

this\_thread::sleep\_for(chrono::seconds(1));

char data[50] = "Writing to the file";

ofstream testWrite("C:/CodeRepository/CppPractice/stdThreadTest/11.txt", ios::out); //以文本模式打开out.txt备写

if (!testWrite) {

cout << "error opening destination file." << endl;

return 0;

}

testWrite << data << endl;

testWrite.close();

return 0;

}

2.2 临时对象造成的问题。

std::thread 入口函数为仿函数或者成员函数

#include <iostream>

#include <thread>

#include <mutex>

#include <shared\_mutex>

#include <stdlib.h>

#include <chrono>

using namespace std;

int num = 0;

mutex sMutex;

class MyFunctor

{

public:

MyFunctor(int& i) :mi(i) {};

~MyFunctor() = default;;

void operator()(const int& a, const int& b) const {

cout << mi << endl;

for (size\_t i = 0; i < 50000; i++)

{

sMutex.lock();

num++;

sMutex.unlock();

}

}

private:

int& mi;

};

int main()

{

int tem = 5;

std::thread t1(MyFunctor(tem), 1, 2);//仿函数

std::thread t2(MyFunctor(tem), 1, 2);

std::thread t3(MyFunctor(tem), 1, 2);

std::thread t4(MyFunctor(tem), 1, 2);

t1.detach();

t2.detach();

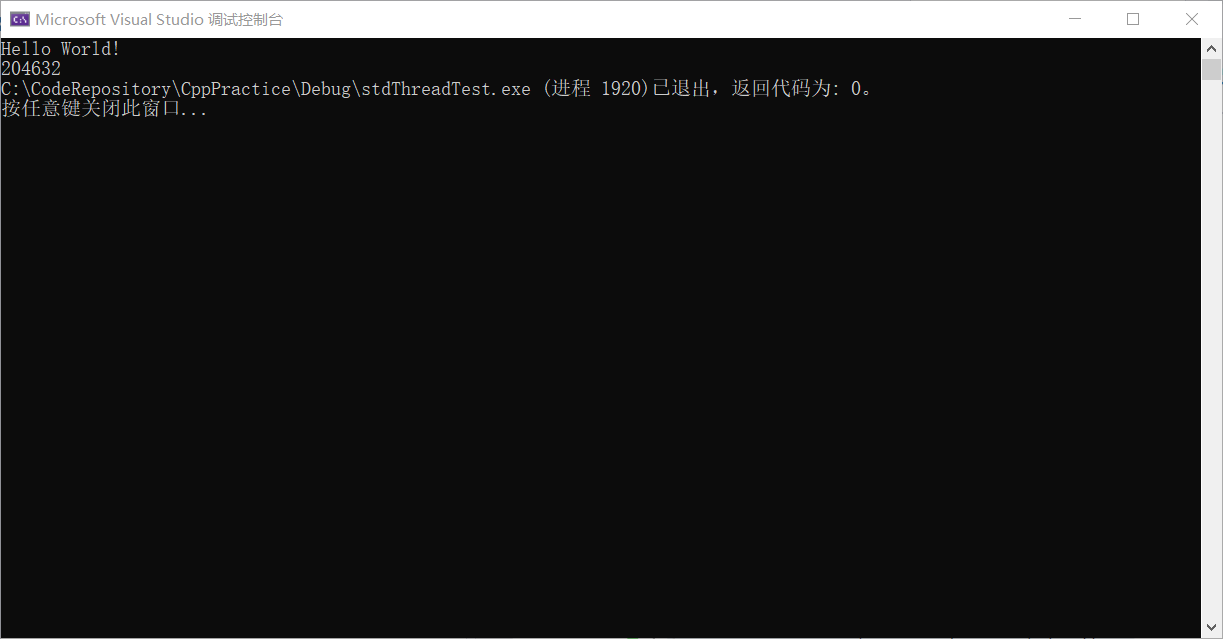
t3.detach();

t4.detach();

std::cout << "Hello World!\n";

}

某次执行结果



原因分析：

构造函数传引用，主函数执行完tmp地址对应的值具有随机性

注：猜测pass by reference需要用std:: ref(实参)的原因，

3待解决问题

# 3.1

int num = 0;

mutex sMutex;

class MyFunctor

{

public:

MyFunctor(int& i) :mi(i) {

std::cout << "Hello 1 World!\n";

};

//MyFunctor(MyFunctor& oth) :mi(oth.mi) {

// std::cout << "Hello 2 World!\n";

//};

//~MyFunctor() { "Hello 111 World!\n"; };

void operator()(const int& a, const int& b) const {

cout << mi << endl;

for (size\_t i = 0; i < 50000; i++)

{

sMutex.lock();

num++;

sMutex.unlock();

}

}

private:

int& mi;

};

int main()

{

int tem = 5;

std::thread t1(MyFunctor(tem),1,2);//仿函数

t1.join();

std::cout << "Hello World!\n";

}

编译通过

# 3.2

int num = 0;

mutex sMutex;

class MyFunctor

{

public:

MyFunctor(int& i) :mi(i) {

std::cout << "Hello 1 World!\n";

};

MyFunctor(MyFunctor& oth) :mi(oth.mi) {

std::cout << "Hello 2 World!\n";

};

~MyFunctor() { std::cout << "Hello 111 World!\n"; };

void operator()(const int& a, const int& b) const {

cout << mi << endl;

for (size\_t i = 0; i < 50000; i++)

{

sMutex.lock();

num++;

sMutex.unlock();

}

}

private:

int& mi;

};

int main()

{

int tem = 5;

MyFunctor f(tem);

std::thread t1(f,1,2);//仿函数

t1.join();

std::cout << "Hello World!\n";

}

编译通过

# 3．3

int num = 0;

mutex sMutex;

class MyFunctor

{

public:

MyFunctor(int& i) :mi(i) {

std::cout << "Hello 1 World!\n";

};

MyFunctor(MyFunctor& oth) :mi(oth.mi) {

std::cout << "Hello 2 World!\n";

};

~MyFunctor() { std::cout << "Hello 111 World!\n"; };

void operator()(const int& a, const int& b) const {

cout << mi << endl;

for (size\_t i = 0; i < 50000; i++)

{

sMutex.lock();

num++;

sMutex.unlock();

}

}

private:

int& mi;

};

int main()

{

int tem = 5;

std::thread t1(MyFunctor(tem),1,2);//仿函数

t1.join();

std::cout << "Hello World!\n";

}

无法成功编译

# 3.1、3. 2 编译成功，3.3编译不通过

3.3区别3.1自己写了拷贝构造函数和析构函数。

3.3区别3.2自己写了使用了临时对象。

# 问

# 多线程中

# 1临时对象扮演什么角色

# 2对应临时对象其类的拷贝构造函数该怎么写

尼玛，拷贝构造函数写错了。

MyFunctor(MyFunctor& oth) :mi(oth.mi) {

std::cout << "Hello 2 World!\n";

};

错

正确写法

MyFunctor(const MyFunctor& oth) :mi(oth.mi) {

std::cout << "Hello 2 World!\n";

};

原因：

std::thread t1(MyFunctor(tem),1,2);//仿函数

MyFunctor(tem)是一个临时对象，是右值，

而MyFunctor(MyFunctor& oth) 没有const，必须接受一个左值引用。

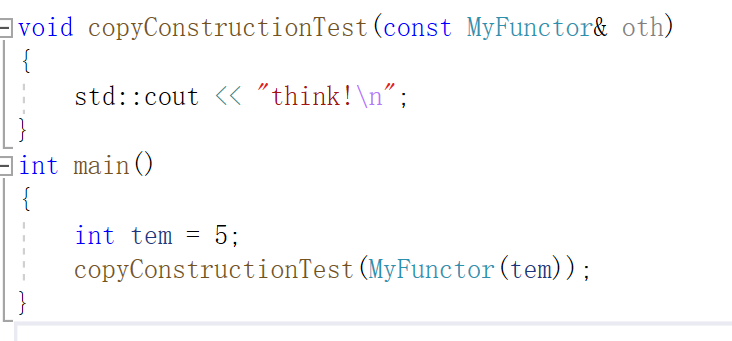
MyFunctor(const MyFunctor& oth) 有const，它可以接受左值或

右值引用。

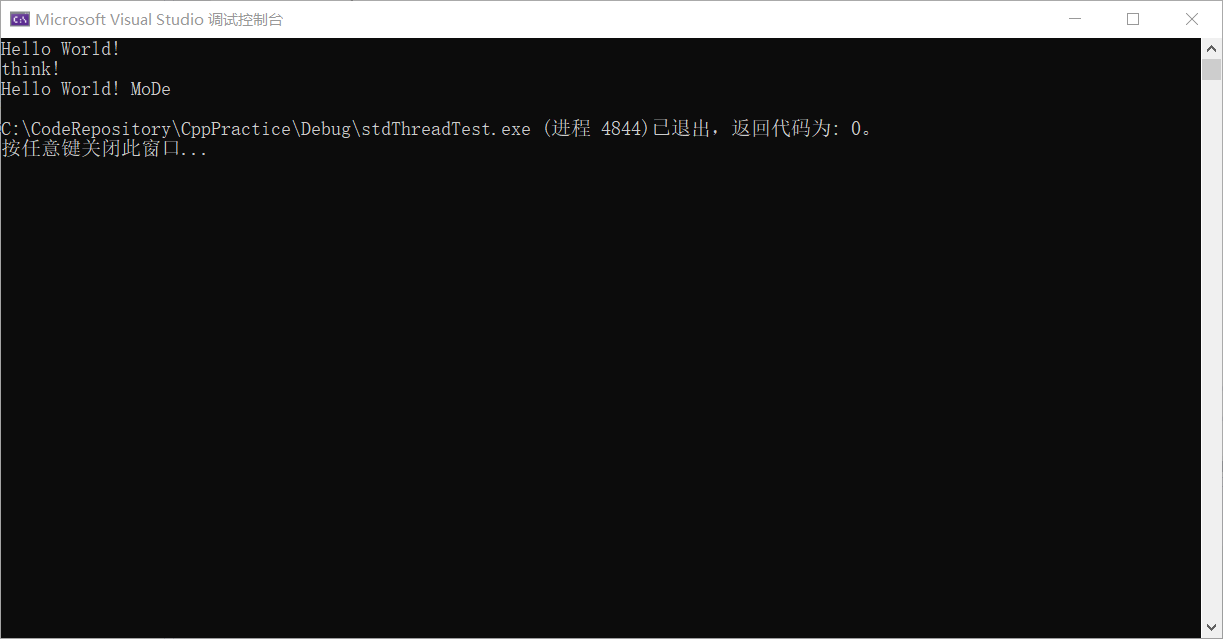
当接受的形参（非const限定的引用）只能接受左值，此时传右值，会出现 非常量引用的初始值必须为左值 错误

# 4.1线程传参问题（引用传参）

函数作为普通函数调用



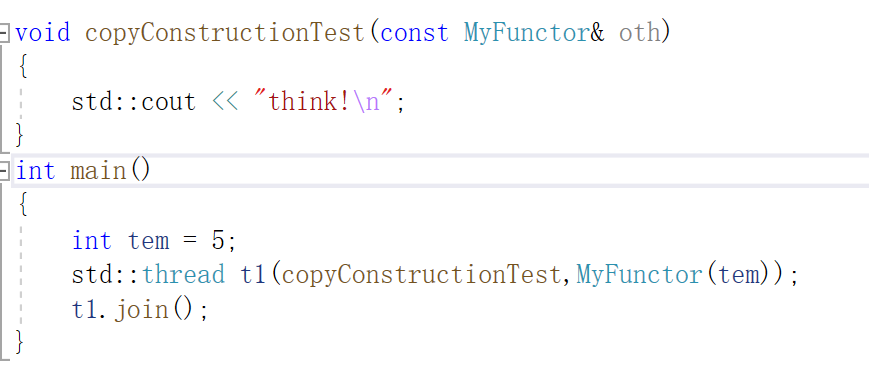
执行结果

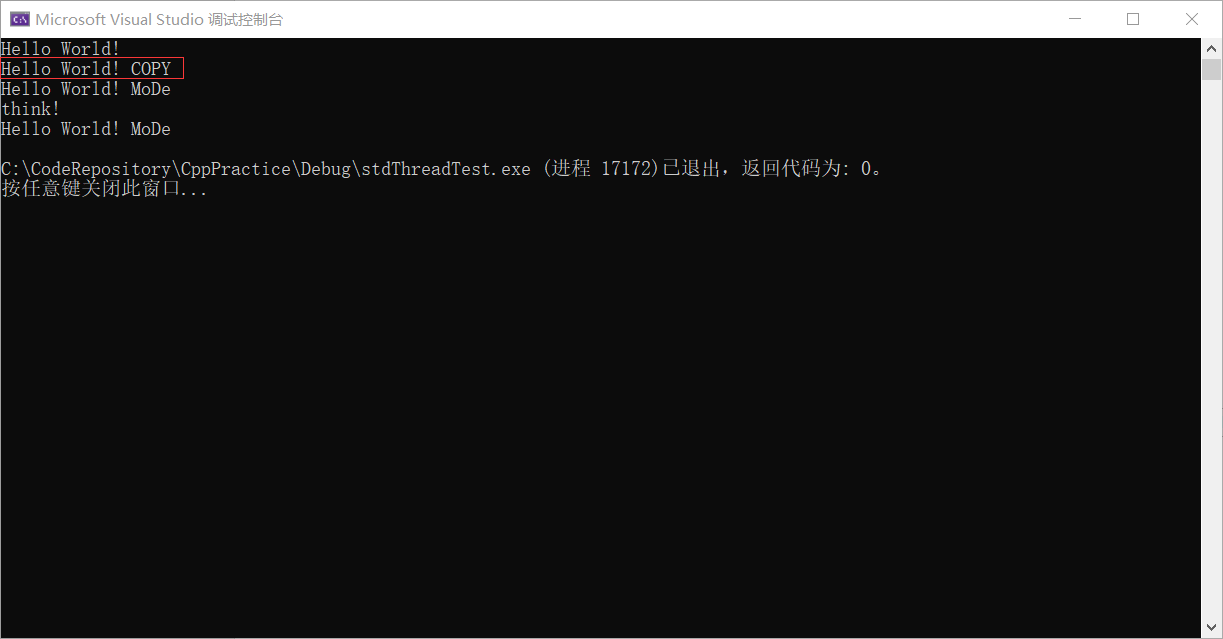


没有执行了拷贝构造函数。

作为线程入口函数

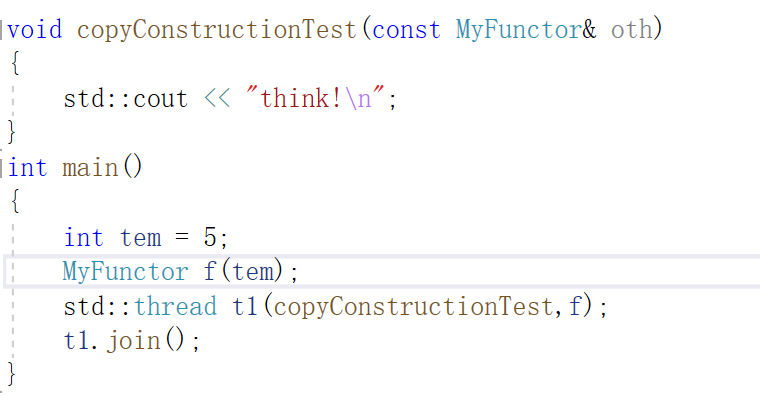
零时对象

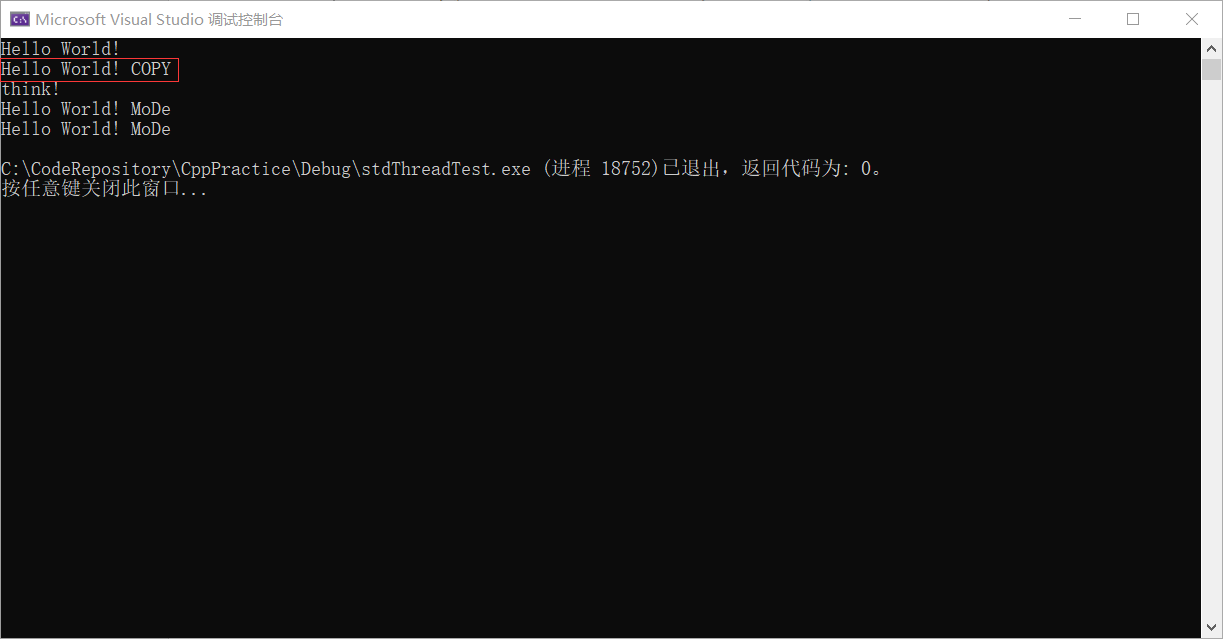




执行了拷贝构造函数。

非零时对象





执行了拷贝构造函数。

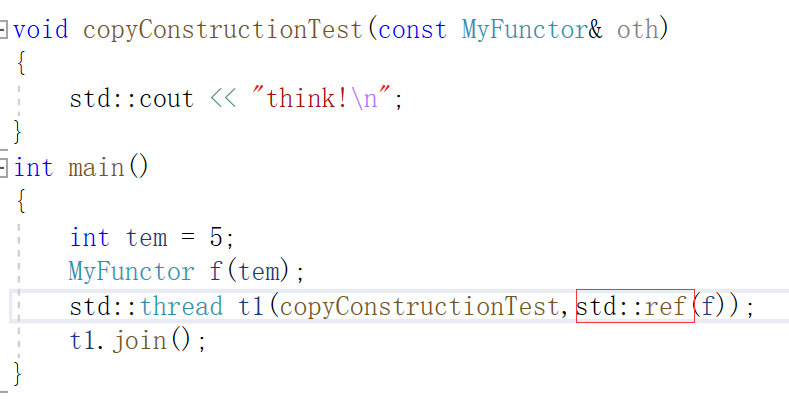
# 虽然是用引用接受但其实还是拷贝了

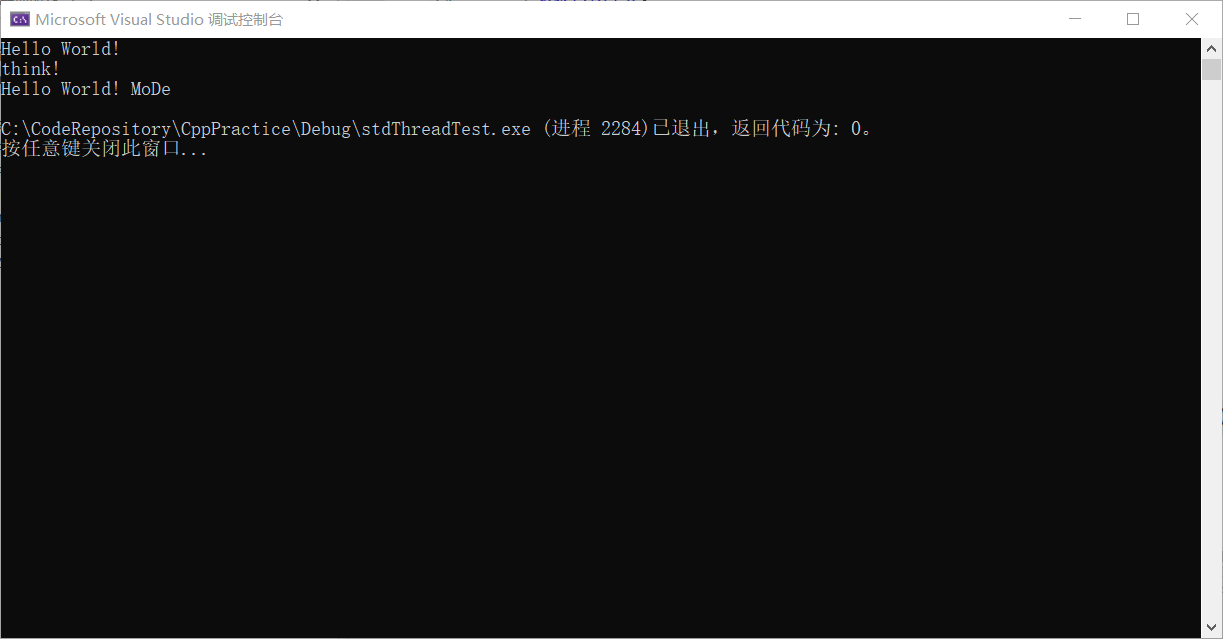
# 注：此时形参使用const限定，实参没有用std::ref()。

# 3形参未const限定，实参没有用std::ref()。

# 编译不通过。

# 4 使用std::ref()





没有执行了拷贝构造函数。

# 此时真的是传引用.

# 4.2线程传参问题（指针传参）

# 此时是真的传地址，没有拷贝动作

# 结论：使用detach()分离子线程，入口函数传参不要使用引用和传地址。

# 4.3线程传参问题（形参和实参间存在隐式转换）

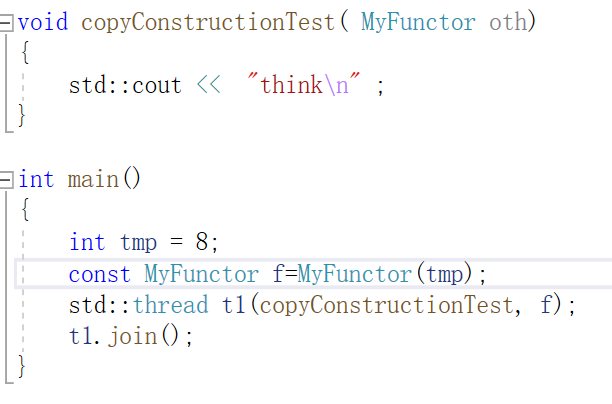
# 结论：形参和实参间存在隐式转换不要使用detach()分离子线程。

原因：此时实参依附与主线程，使用detach()，

可能存在一种情况主线程先执行完，

形参和实参间存在隐式转换还没开始，实参被回收，转换无法继续，程序出现异常，因此避免参数中进行隐式转换。

# 4.3线程传参问题（pass by value）

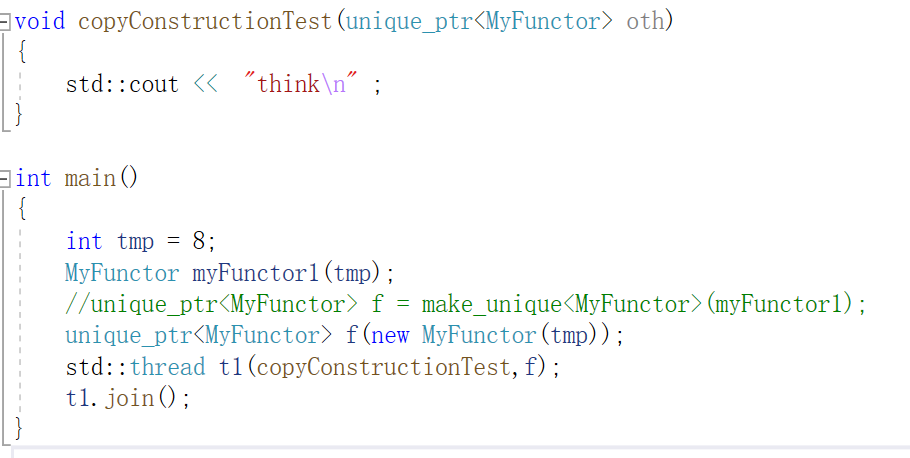




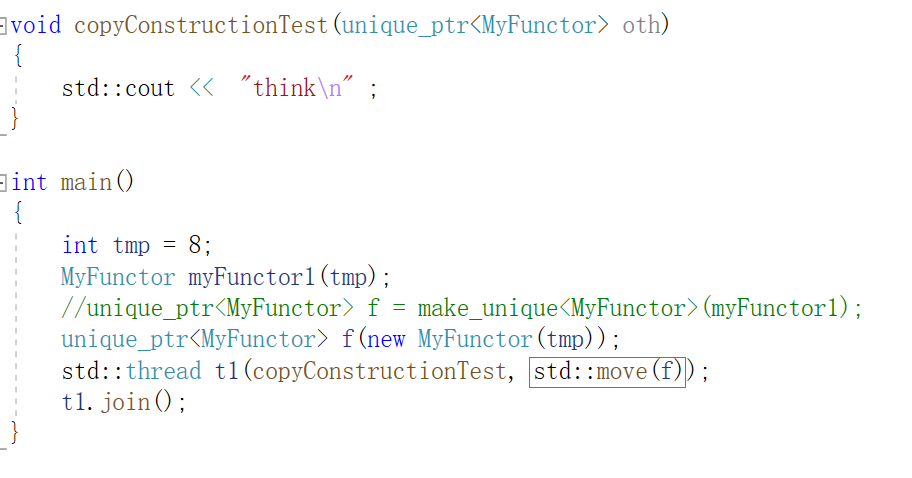
拷贝构造函数执行了两遍

# 4.4线程传参问题（传智能指针）

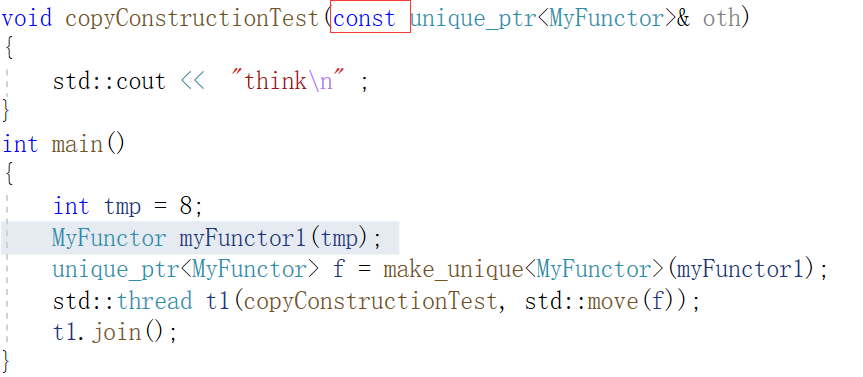
unique\_ptr



编译不通过



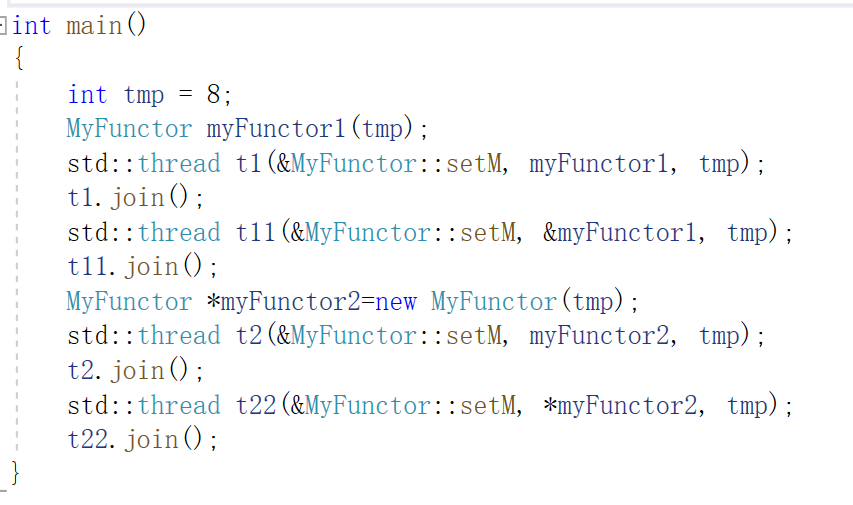
正确

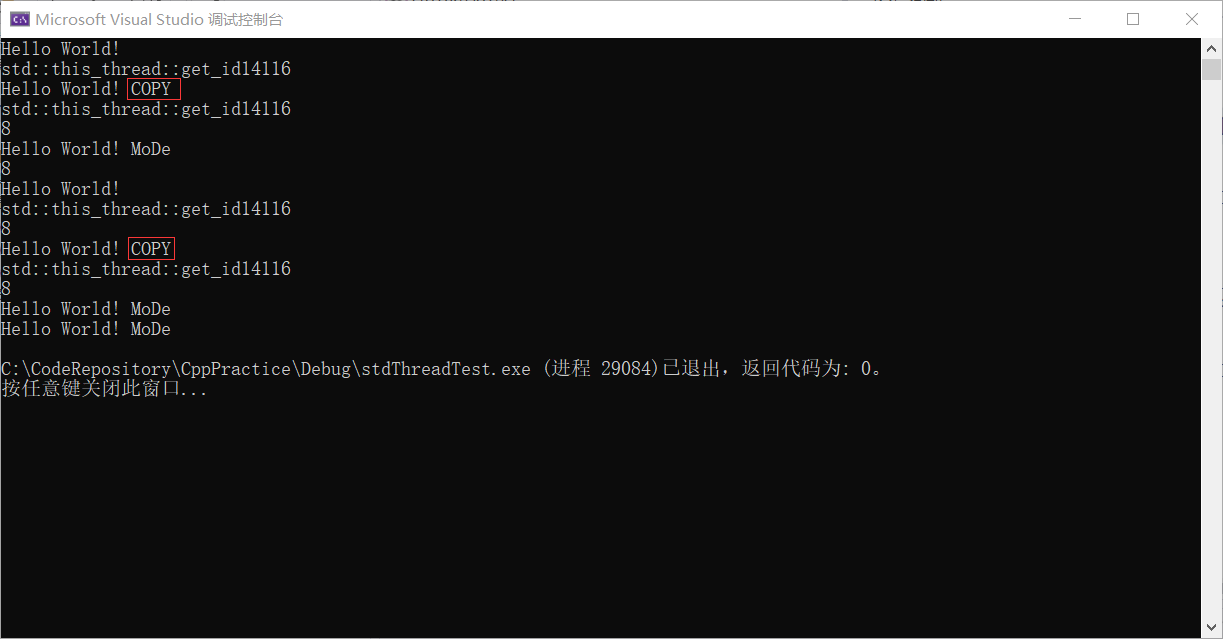


形参为引用要用const 修饰的原因，使用是std::move后对象就成了右值。

# 5使用类的成员函数作为线程的入口函数

需要传对象或对象的地址





执行结果

当传值时会copy对象作为参

# unique\_lock构造函数

**1 unique\_lock<mutex> uniqueLock(sMutex);**

**一个参数，指示sMutex未被lock，需要在uniqueLock的构造函数中lock；**

**2 unique\_lock<mutex> uniqueLock(sMutex，adopt\_lock\_t());**

**第二个参数为adopt\_lock\_t类型，指示sMutex已经被lock，不要在uniqueLock的构造函数中lock；**

**3 unique\_lock<mutex> uniqueLock(sMutex，defer\_lock\_t());**

**第二个参数为defer\_lock\_t类型，指示sMutex未被lock，不要在uniqueLock的构造函数中lock,**

**在后续代码中可以调用uniqueLock提供的lock操作mutex；**

# condition\_variable的使用

**wait(unique\_lock<mutex>& \_Lck,（可调用对象）)**

**如果可调用对象返回false那么wait()将解锁互斥量，并阻塞到本行，**

**直到其他线程调用notify\_one()或者notify\_all()为止。**

**如果可调用对象返回true那么wait()直接返回，执行接下来的操作。**

**wait(unique\_lock<mutex>& \_Lck)**

**相当于上面的可调用对象返回false。**

**当wait()被notify\_one()或者notify\_all()唤醒后，就不断尝试对获取锁(即加锁)，**

**如果失败，则继续尝试，直到成功，再判断第二参数的返回值，进行之后的操作。**

**如果wait()没有第二参数，则相当于第二参数的返回值为true；**

**对于notify\_one()和notify\_all()的理解。**

**notify\_one()和notify\_all()能够唤醒某个正在wait()的线程，如果线程此时不是在wait()，**

**那么此次唤醒无效。**

**example：**

#include <iostream>

#include <thread>

#include <mutex>

#include <chrono>

#include <memory>

#include <list>

using namespace std;

int num = 0;

mutex sMutex;

condition\_variable cond;

class MyFunctor

{

public:

MyFunctor(int& i) :mi(i) {

std::cout << "Hello World!\n";

std::cout << "std::this\_thread::get\_id" << std::this\_thread::get\_id() << std::endl;

};

MyFunctor(const MyFunctor& oth) :mi(oth.mi) {

std::cout << "Hello World! COPY\n";

std::cout << "std::this\_thread::get\_id" << std::this\_thread::get\_id() << std::endl;

};

~MyFunctor() { std::cout << "Hello World! MoDe\n"; };

void operator()(const int& a, const int& b)

{

cout << mi << endl;

for (size\_t i = 0; i < 50000; i++)

{

sMutex.lock();

num++;

sMutex.unlock();

}

}

void setM(int m)

{

mi = m;

cout << mi << endl;

}

int getM()

{

return mi;

}

private:

int& mi;

};

list<MyFunctor> slist;

void pushList()

{

while (true)

{

unique\_lock<mutex> uniqueLock(sMutex);

slist.push\_back(std::move(MyFunctor(num)));

num++;

cout << "size" << slist.size() << endl;

uniqueLock.unlock();

cond.notify\_one();

this\_thread::sleep\_for(chrono::milliseconds(100));

}

}

void popList()

{

while (true)

{

unique\_lock<mutex> uniqueLock(sMutex);

cond.wait(uniqueLock, []()-> bool {

if (slist.size() < 1)

return false;

return true;

}

);

MyFunctor f = slist.front();

slist.pop\_front();

cout << "member" << f.getM() << endl;

cout << "size" << slist.size() << endl;

this\_thread::sleep\_for(chrono::milliseconds(100));

}

}

int main()

{

vector<thread>threadVector;

for (size\_t i = 0; i < 5; i++)

{

threadVector.push\_back(thread(popList));

}

this\_thread::sleep\_for(chrono::milliseconds(6000));

for (size\_t i = 0; i < 3; i++)

{

threadVector.push\_back(thread(pushList));

}

for (size\_t i = 0; i < 8; i++)

{

threadVector[i].join();

}

}

# std::async()函数模板

启动一个异步任务。

**example：**

#include <iostream>

#include <future>

using namespace std;

int add(int& number, int& count)

{

for (size\_t i = 0; i < count; i++)

{

number++;

}

return number;

}

int calculate( int (\*add)(int& number, int& count), int& number, int& count)

{

return add(number, count);

}

int main()

{

int number = 10;

int count = 100;

std::future<int> ret=std::async(std::launch::deferred,calculate, add, std::ref(number), std::ref(count));

cout<< ret.get()<<endl;

}

注：

1、线程入口函数calculate()参数为引用，实参传递需加std::ref()。

2、类模板 std::future 提供访问异步操作结果的机制，这个结果可能不能马上获取，但是在不久的将来（即线程执行结束）可以获取。

3、使用std::future 对象的get()方法，如果线程已经执行完了。就返回结果，如果线程未执行完就等待线程执行完。

4、std::future 对象的get()方法只能调用一次。

5、上面的std::asyn(std::launch::deferred,calculate, add, std::ref(number), std::ref(count));第一个参数的类型为std::launch。

enum launch {

async = 0x1,

deferred = 0x2

};

async表示创建线程并开始执行线程入口函数。

deferred表示线程入口函数延迟到调用std::future 对象的get()或wait()方法才执行。这种情况下不调用get()或wait()方法线程入口函数将不执行。调用get()或wait()方法入口函数执行，但此时没有创建线程，只是调用了这个入口函数。

不写第一个参数默认为launch::async | launch::deferred，

可能创建新线程并立即执行；也可能不创建新线程被延迟到调用get()或wait()才执行。系统根据资源使用情况来选择不同的策略。

如何判断系统采取哪种策略？

通过返回值std::future提供的wait\_for()方法的返回值来判断。

std::future\_status wait\_for( const std::chrono::duration<Rep,Period>& timeout\_duration ) const;

future\_status::deferred 要计算结果的函数仍未启动

future\_status::ready 结果就绪(结果已返回)

future\_status::timeout 已经过时限(还未等到结果)。

## example：

std::future<int>ret=std::async(func);

future\_status status=ret.wait\_for(std::chrono::seconds(0));

if (status == std::future\_status::deferred)

{

ret.get();

}

else if (status == std::future\_status::timeout)

{

}

else if (status == std::future\_status::ready) {

}

# std::packaged\_task类模板

类模板 std::packaged\_task 包装任何*[可调用](https://zh.cppreference.com/w/cpp/named_req/Callable" \o "cpp/named req/Callable)*[(](https://zh.cppreference.com/w/cpp/named_req/Callable" \o "cpp/named req/Callable)*[Callable](https://zh.cppreference.com/w/cpp/named_req/Callable" \o "cpp/named req/Callable)*[)](https://zh.cppreference.com/w/cpp/named_req/Callable" \o "cpp/named req/Callable)对象

使用有点奇怪

**example：**

#include <iostream>

#include <future>

#include <chrono>

using namespace std;

int add(int& number, int& count)

{

for (size\_t i = 0; i < count; i++)

{

number++;

}

this\_thread::sleep\_for(chrono::milliseconds(800));

return number;

}

int calculate( int (\*add)(int& number, int& count), int& number, int& count)

{

cout << this\_thread::get\_id() << endl;

return add(number, count);

}

int main()

{

int number = 10;

int count = 100;

std::packaged\_task<int (int (\*)(int& number, int& count), int& number, int& count)>task(calculate);

·//声明

std::future<int> ret1 = task.get\_future();

task(add, std::ref(number), std::ref(count));

·//使用

cout<<ret1.get() << endl;

}

没有创建新线程。

两行可以调换顺序。

**example：**

#include <iostream>

#include <future>

#include <chrono>

using namespace std;

int add(int& number, int& count)

{

for (size\_t i = 0; i < count; i++)

{

number++;

}

this\_thread::sleep\_for(chrono::milliseconds(800));

return number;

}

int calculate( int (\*add)(int& number, int& count), int& number, int& count)

{

cout << this\_thread::get\_id() << endl;

return add(number, count);

}

int main()

{

int number = 10;

int count = 100;

cout << this\_thread::get\_id() << endl;

std::packaged\_task<int (int (\*)(int& number, int& count), int& number, int& count)>task(calculate);

std::future<int> ret1 = task.get\_future();

std::thread t(std::move(task), add, std::ref(number), std::ref(count));

cout<<ret1.get() << endl;

t.join();

}

创建新线程。

两行不能调换顺序，否则程序崩溃。

使用std::thread t(std::ref(task), add, std::ref(number), std::ref(count));创建线程两行可以调换顺序。

原因：使用了std::move，task的所有权变了。

int main()

{

int number = 10;

int count = 100;

cout << this\_thread::get\_id() << endl;

std::packaged\_task<int (int (\*)(int& number, int& count), int& number, int& count)>task(calculate);

std::future<void> ret1 = async(std::ref(task), add, std::ref(number), std::ref(count));

std::future<int> ret = task.get\_future();

cout<<ret.get() << endl;

}

std::packaged\_task重载了(),因此也是个可调用对象。

但是 void operator()(\_ArgTypes... \_Args) { // call the function object

if (\_MyPromise.\_Is\_ready()) {

\_Throw\_future\_error(make\_error\_code(future\_errc::promise\_already\_satisfied));

}

\_MyStateManagerType& \_State = \_MyPromise.\_Get\_state\_for\_set();

\_MyStateType\* \_Ptr = static\_cast<\_MyStateType\*>(\_State.\_Ptr());

\_Ptr->\_Call\_immediate(\_STD forward<\_ArgTypes>(\_Args)...);

}

返回值是void,因此需要使用task.get\_future()来获取task包装的可调用对象的返回值（如果有）。

**example（by** <https://zh.cppreference.com/w/cpp/thread/packaged_task>**）：**

#include <iostream>

#include <cmath>

#include <thread>

#include <future>

#include <functional>

// 避免对 std::pow 重载集消歧义的独有函数

int f(int x, int y) { return [std::pow](http://zh.cppreference.com/w/cpp/numeric/math/pow)(x,y); }

void task\_lambda()

{

std::packaged\_task<int(int,int)> task([](int a, int b) {

return [std::pow](http://zh.cppreference.com/w/cpp/numeric/math/pow)(a, b);

});

[std::future](http://zh.cppreference.com/w/cpp/thread/future)<int> result = task.get\_future();

task(2, 9);

[std::cout](http://zh.cppreference.com/w/cpp/io/cout) << "task\_lambda:**\t**" << result.get() << '**\n**';

}

void task\_bind()

{

std::packaged\_task<int()> task([std::bind](http://zh.cppreference.com/w/cpp/utility/functional/bind)(f, 2, 11));

[std::future](http://zh.cppreference.com/w/cpp/thread/future)<int> result = task.get\_future();

task();

[std::cout](http://zh.cppreference.com/w/cpp/io/cout) << "task\_bind:**\t**" << result.get() << '**\n**';

}

void task\_thread()

{

std::packaged\_task<int(int,int)> task(f);

[std::future](http://zh.cppreference.com/w/cpp/thread/future)<int> result = task.get\_future();

[std::thread](http://zh.cppreference.com/w/cpp/thread/thread) task\_td(std::move(task), 2, 10);

task\_td.join();

[std::cout](http://zh.cppreference.com/w/cpp/io/cout) << "task\_thread:**\t**" << result.get() << '**\n**';

}

int main()

{

task\_lambda();

task\_bind();

task\_thread();

}

# std::packaged\_task

包装可调用对象测试

#include <iostream>

#include <future>

#include <chrono>

using namespace std;

class MyFunctor

{

public:

MyFunctor(int& i) :mi(i) {

std::cout << "Hello World!\n";

std::cout << "std::this\_thread::get\_id" << std::this\_thread::get\_id() << std::endl;

};

MyFunctor(const MyFunctor& oth) :mi(oth.mi) {

std::cout << "Hello World! COPY\n";

std::cout << "std::this\_thread::get\_id" << std::this\_thread::get\_id() << std::endl;

};

~MyFunctor() { std::cout << "Hello World! MoDe\n"; };

int operator()(const int& a, const int& b)

{

cout << mi << endl;

return a + b;

}

void setM(int m)

{

mi = m;

cout << mi << endl;

}

int getM()

{

return mi;

}

private:

int& mi;

};

正确写法1

int main()

{

int number = 10;

int count = 100;

cout << this\_thread::get\_id() << endl;

MyFunctor f(number);

std::packaged\_task<int(int& number, int& count)>task(f);

task(number, number);

auto ret1 = task.get\_future();

cout<<ret1.get() << endl;

}

注：值传递调用了两次拷贝构造函数。

正确写法2

int main()

{

int number = 10;

int count = 100;

cout << this\_thread::get\_id() << endl;

std::packaged\_task<int(int& number, int& count)>task(std::move(MyFunctor(number)));

task(number, number);

auto ret1 = task.get\_future();

cout<<ret1.get() << endl;

}

注：使用std::move居然也调用了两次拷贝构造函数。

错误写法1

int main()

{

int number = 10;

int count = 100;

cout << this\_thread::get\_id() << endl;

std::packaged\_task<int(int& number, int& count)>task(std::move(MyFunctor(number)));

task(number, number).get\_future();

}

说明task(number, number)是把参数传给std::packaged\_task包装的可调用对象，并调用了；这并不是调用了构造函数产生了临时对象，task才是一个对象。

错误写法2

int main()

{

int number = 10;

int count = 100;

cout << this\_thread::get\_id() << endl;

std::packaged\_task<int(int& number, int& count)>task( MyFunctor(number));

task(number, number);

}

这个并不是临时对象的原因。

std::packaged\_task<int(int& number, int& count)>task( (MyFunctor(number)) );

在不改变其他任何地方，构造临时对象加的()居然就对了。

原因：

std::packaged\_task<int(int& number, int& count)>task2(MyFunctor(number));

被理解为一个函数

std::packaged\_task<int(int& number, int& count)>

函数返回值；

task2 函数名；

MyFunctor(number) 参数类型和形参。

正确写法：

std::packaged\_task<int(int& number, int& count)>task2(MyFunctor(number))

{

std::packaged\_task<int(int& number, int& count)>task([](int& number, int& count)->int

{

return number;

});

return task;

}

int main()

{

int number = 10;

int count = 100;

std::packaged\_task<int(int& number, int& count)>task((MyFunctor(number)));

task(number, count);

auto ret1 = task.get\_future();

std::packaged\_task<int(int& number, int& count)>task2(MyFunctor(number));//函数声明，可以不要。

std::packaged\_task<int(int& number, int& count)>task1 = task2(MyFunctor(number));

task1(number, number);

auto ret2 = task1.get\_future();

cout<< ret2.get() << endl;

}

错误写法

int main()

{

int number = 10;

int count = 100;

future<int> ret2 = task2(MyFunctor(number)).get\_future();

cout << ret2.get() << endl;

}

task2(MyFunctor(number))调用task2生成了一个std::packaged\_task对象，但是没有执行包裹的可调用对象。

future<int> ret2没有东西，ret2.get()出错。

|  |
| --- |
| std::promise类模板 类模板 std::promise 提供存储值或异常的设施，之后通过 std::promise  对象所创建的 [std::future](https://zh.cppreference.com/w/cpp/thread/future) 对象异步获得结果。  注意 std::promise 只应当使用一次。 |

即std::promise可以保存一个值，将来使用。

**example：**

void add(std::promise<int>& value, int& number, int& count)

{

std::cout << "std::this\_thread::get\_id" << std::this\_thread::get\_id() << std::endl;

for (size\_t i = 0; i < count; i++)

{

number++;

}

value.set\_value(number);

}

int main()

{

std::promise<int> value;

int number = 100;

int count = 111000000;

std::cout << "std::this\_thread::get\_id" << std::this\_thread::get\_id() << std::endl;

std::thread t(add, std::ref(value), std::ref(number), std::ref(count));

std::future<int>fut = value.get\_future();

cout << fut.get()<<endl;

t.join();

}

# std::atomic

example：

std::atomic<int>g\_number = 0;

void add()

{

for (size\_t i = 0; i < 10000000; i++)

{

g\_number++;

g\_number += 1;

//g\_number = g\_number+ 1;

}

}

int main()

{

vector< std::thread>vecThreads;

for (size\_t i = 0; i < 5; i++)

{

vecThreads.push\_back(std::thread(add));

}

for (size\_t i = 0; i < 5; i++)

{

vecThreads[i].join();

}

cout << g\_number << endl;

}

、是原子操作

不是原子操作，数据会乱。