lambda表达式

1、基本语法特性

```
[capture](params) opt -> retureType
{
    body;
};
```

其中capture是捕获列表，params是参数列表，opt是函数选项，retureType是返回值类型，body是函数体。

1.1、捕获列表capture

不能省略。捕获一定范围内的变量。

- [] 不捕捉任何变量
- [&] 捕获外部作用域中所有变量, 并作为引用在函数体内使用 (按引用捕获)
- [=] 捕获外部作用域中所有变量, 并作为副本在函数体内使用 (按值捕获)，拷贝的副本在匿名函数体内部是**只读的**
- [=, &bar] 按值捕获外部作用域中所有变量, 并按照引用捕获外部变量bar
- [bar] 按值捕获bar变量, 同时不捕获其他变量
- [&bar] 按引用捕获bar变量，同时不捕获其他变量
- [&，bar] 按引用捕获其他变量，同时按值捕获bar变量
- [this] 捕获当前类中的this指针。让lambda表达式拥有和当前类成员函数同样的访问权限；**可以修改类的成员变量，使用类的成员函数。**如果已经使用了 & 或者 =, 默认添加此选项

注意：如果是全局变量，可以不用捕获，直接使用。

下面看两个例子，看看lambda的使用方法。

```
class Example
{
public:
    void print(int x, int y)
    {
        auto x1 = [] {return _number; };                    // error
        auto x2 = [this] {return _number; };                // ok
        auto x3 = [this] {return _number + x + y; };        // error
        auto x4 = [this, x, y] {return _number + x + y; };  // ok
        auto x5 = [this] {return _number++; };              // ok
        auto x6 = [=] {return _number + x + y; };           // ok
        auto x7 = [&] {return _number + x + y; };           // ok
    }
    int _number = 100;
};

void test()
{
    int a = 10, b = 20;
    auto f1 = [] {return a; };                      // error
    auto f2 = [&] {return a++; };                   // ok
```

```
    auto f3 = [=] {return a; };                         // ok
    auto f4 = [=] {return a++; };                       // error
    auto f5 = [a] {return a + b; };                     // error
    auto f6 = [a, &b] {return a + (b++); };             // ok
    auto f7 = [=, &b] {return a + (b++); };             // ok
}
```

在匿名函数内部，需要通过lambda表达式的捕获列表控制如何捕获外部变量，以及访问哪些变量。默认状态下lambda表达式无法修改通过复制方式捕获外部变量，如果希望修改这些外部变量，需要通过引用的方式进行捕获。

1.2、参数列表params

和普通函数的参数列表一样，如果没有参数，参数列表可以省略不写。

```
auto f = [](){  return 10;  }   // 没有参数，参数列表为空
auto f = []{  return 10;  }     // 没有参数，参数列表省略不写
```

1.3、选项opt

可以省略。

mutable: 可以修改按值传递进来的拷贝（注意是能修改拷贝，而不是值本身）
exception: 指定函数抛出的异常，如抛出整数类型的异常，可以使用throw();

1.4、返回类型retureType

可以省略。通过返回值后置语法来定义的。一般情况下，不指定lambda表达式的返回值，编译器会根据return语句自动推导返回值的类型，但需要注意的是labmda表达式不能通过列表初始化自动推导出返回值类型。

```
//ok，可以自动推导出返回值类型
auto f = [](int i)
{
    return i;
}

//error,不能推导出返回值类型
auto f1 = []()
{
    return {1, 2};  // 基于列表初始化推导返回值，错误
}
```

1.5、函数体body

不能省略。但函数体可以为空

2、函数本质

使用lambda表达式捕获列表捕获外部变量，如果希望去按值捕获的外部变量，那么应该如何处理呢？这就需要使用mutable选项，被mutable修改是lambda表达式就算没有参数也要写明参数列表，并且可以去掉按值捕获的外部变量的只读（const）属性。

```
int a = 0;
auto f1 = [=] {return a++; };               // error, 按值捕获外部变量, a是只读的
auto f2 = [=]()mutable {return a++; };      // ok
```

最后再剖析一下为什么通过值拷贝的方式捕获的外部变量是只读的。

lambda表达式的类型在C++11中会被看做是一个带operator()的类，即仿函数。按照C++标准，lambda表达式的operator()默认是const的，一个const成员函数是无法修改成员变量值的。mutable选项的作用就在于取消operator()的const属性。

因为lambda表达式在C++中会被看做是一个仿函数，因此可以使用std::function和std::bind来存储和操作lambda表达式。

```
void  test()
{
    //包装可调用函数
    function<int(int)> f1 = [](int a) {return a; };

    //绑定可调用函数
    function<int(int)> f2 = bind([](int a) {return a; }, placeholders::_1);

    //函数调用
    cout << f1(100) << endl;
    cout << f2(200) << endl;
}
```

3、基本使用形式

```
void test()
{
    //匿名函数
    [](const string &name)
    {
        cout << "[] name = " << name << endl;
    }("lili");

    auto func2 = [](const string &name)->string
    {
        cout << "func2 name = " << name << endl;

        return name;
    };
    func2("lucy");

    pFunc func3 = [](const string &name)
    {
        cout << "func3 name = " << name << endl;
    };
    func3("welcome");

    function<void(const string &)> func4
        = [](const string &name)
        {
```

```
            cout << "func4 name = " << name << endl;
        };
    func4("wangdao");
}
```

4、捕获列表细究

```
void test()
{
    int num = 10;
    int age = 100;
    string name("wangdao");

    [](string value)
    {
        /* cout << "num = " << num << endl; */
        /* cout << "name = " << name<< endl; */
        cout << "gNum = " << gNum << endl;
        cout << "value = " << value << endl;
    }("lili");

    //值捕获
    [num, name](string value)
    {
        cout << "num = " << num << endl;
        cout << "name = " << name<< endl;
        cout << "value = " << value << endl;
    }("lili");

    //引用捕获
    cout << endl;
    [&num, &name](string value)
    {
        num = 100;
        name = "nice";
        cout << "num = " << num << endl;
        cout << "name = " << name<< endl;
        cout << "value = " << value << endl;
    }("wangdao");

    cout << "num = " << num
         << ", name = " << name << endl;

    cout << endl << endl;
    //全部用值捕获，name用引用捕获
    [=, &name]()
    {
        /* num = 1000;//error */
        name = "welcome";
        cout << "num = " << num << endl;
        cout << "age = " << age << endl;
        cout << "name = " << name << endl;
    }();

```

```
    cout << endl << endl;
    //全部引用捕获，name值捕获
    [&, name]()
    {
        num = 1000;//ok
        age = 100;//ok
        /* name = "welcome";//error */
        cout << "num = " << num << endl;
        cout << "age = " << age << endl;
        cout << "name = " << name << endl;
    }();
}
```

5、使用误区

不要捕获局部变量的引用

```
vector<function<void(const string &)>> vec;

void test()
{
    int num = 100;
    string name("wangdao");
    vec.push_back([&num, &name](const string &value){
                  cout << "num = " << num << endl;
                  cout << "name = " << name << endl;
                  cout << "value = " << value << endl;
                  });
}

void test2()
{
    for(auto func : vec)
    {
        func("wuhan");
    }
}
```