Lambda表达式

C++在C++11标准中引入了lambda表达式，一般用于定义匿名函数，使得代码更加灵活简洁。

lambda表达式与普通函数类似，也有参数列表、返回值类型和函数题，只是它的定义方式更加简洁，并且可以在函数内部定义

什么是Lambda表达式

最常见的lambda表达式写法如下：

Lambda函数示例 C++

```cpp
auto plus = [] (int v1, int v2) -> int { return v1 + v2; }
int sum = plus(1, 2);
```

一般情况下我们不会这么用，更多的时候，我们都是和STL的一些算法结合使用，例如排序和打印

C++

```cpp
// 将std::vector<Item>类型的动态数组升序排列
    std::sort(vec.begin(), vec.end(),
        [] (const Item& v1, const Item& v2) { return v1.a < v2.a; });

// 打印std::vector<Item>类型的动态数组的成员
    std::for_each(vec.begin(), vec.end(),
        [] (const Item& item) { std::cout << item.a << " " << item.b << std
::endl; });
```

Lambda表达式的语法

[*捕获*] (*形参*) *lambda说明符* *约束*(可选) { *函数体* }	(1)	
[*捕获*] { *函数体* }	(2)	(C++23 前)
[*捕获*] *lambda说明符* { *函数体* }	(2)	(C++23 起)
[*捕获*] < *模板形参* > *约束*(可选) (*形参*) *lambda说明符* *约束*(可选) { *函数体* }	(3)	(C++20 起)
[*捕获*] < *模板形参* > *约束*(可选) { *函数体* }	(4)	(C++20 起) (C++23 前)
[*捕获*] < *模板形参* > *约束*(可选) *lambda说明符* { *函数体* }	(4)	(C++23 起)

1) 完整声明。
2) 省略形参列表：函数不接收实参，如同形参列表是 ()。
3) 与 1) 相同，但指定泛型 lambda 并显式提供模板形参列表。
4) 与 2) 相同，但指定泛型 lambda 并显式提供模板形参列表。

捕获：lambda可以把上下文变量以值或者引用的方式捕获，在body中直接使用

模版形参：模版形参列表，为泛型lambda提供各个模版形参的名字，在C++20引入

形参：形参列表，在C++14之后可以使用auto类型

lambda说明符：由**说明符**、**异常说明**、**属性**和**尾随返回类型**按照顺序组成，每个组分均非必需

说明符：可选的说明符序列。不提供说明符时复制补货的对象在lambda体内时const的

1. mutable：允许函数体修改复制捕获的对象，以及调用它们的非const函数
2. 其它的参数见cppreference，不是很常用

异常说明：为闭包类型的 `operator()` 提供动态异常说明或noexcept说明符

属性：为闭包类型的函数调用运算符或运算符模版的类型提供属性说明

尾随返回类型： `-> 返回类型` ，其中**返回类型**指定返回类型，若没有则由编译器推倒

约束：向闭包类型的 `operator()` 添加约束

捕获列表

捕获列表是Lambda表达式的灵魂，几种常用的捕获方式：

`[]`：什么也不捕获

`[=]`：按值的方式捕获所有变量

`[&]`：按引用的方式捕获所有变量

`[=, &a]`：除了变量a按照引用方式捕获之外，按照值的方式捕获所有局部变量

`[&, a]`：除了变量a按值的方式捕获之外，按照引用的方式捕获所有局部变量

`[a, &b]`：以值的方式捕获a，引用的方式捕获b

`[this]`：在成员函数中，也可以直接捕获this指针(`[=]` 和 `[&]` 也会捕获this指针)

捕获列表使用示例 C++

```cpp
#include <iostream>

int main()
{
    int a = 3;
    int b = 5;

    // 按值来捕获
    auto func1 = [a] { std::cout << a << std::endl; };
    func1();

    // 按值来捕获
    auto func2 = [=] { std::cout << a << " " << b << std::endl; };
    func2();

    // 按引用来捕获
    auto func3 = [&a] { std::cout << a << std::endl; };
    func3();

    // 按引用来捕获
    auto func4 = [&] { std::cout << a << " " << b << std::endl; };
    func4();
}
```

编译器如何解析Lambda表达式

编译器会把我们写的lambda表达式翻译成一个类，并重载 `operator()` 来实现

值捕获

值捕获Lambda函数 C++

```cpp
int x = 1; int y = 2;
auto plus = [=] (int a, int b) -> int { return x + y + a + b; };
int c = plus(1, 2);
```

编译器会将其翻译为：

值捕获Lambda函数对应的编译器翻译结果 C++

```cpp
class LambdaClass
{
public:
    LambdaClass(int xx, int yy)
    : x(xx), y(yy) {}

    int operator () (int a, int b) const
    {
        return x + y + a + b;
    }

private:
    int x;
    int y;
}

int x = 1; int y = 2;
LambdaClass plus(x, y);
int c = plus(1, 2);
```

值捕获时，编译器会把捕获到的值作为类的成员变量以值的方式传递。如果所有的参数都是值捕获的方式，那么生成的 `operator()` 函数是 `const函数` ，无法修改捕获的值。如果想要修改lambda表达式外部的变量，需要加上 `mutable` 关键字：

值捕获Lambda函数使用mutable关键字 C++

```cpp
int x = 1; int y = 2;
auto plus = [=] (int a, int b) mutable -> int { x++; return x + y + a + b;
};
int c = plus(1, 2);
```

引用捕获

引用捕获Lambda函数 C++

```cpp
int x = 1; int y = 2;
auto plus = [&] (int a, int b) -> int { x++; return x + y + a + b;};
int c = plus(1, 2);
```

编译器的翻译结果：

引用捕获Lambda函数对应的编译器翻译结果 C++

```cpp
class LambdaClass
{
public:
    LambdaClass(int& xx, int& yy)
    : x(xx), y(yy) {}

    int operator () (int a, int b)
    {
        x++;
        return x + y + a + b;
    }

private:
    int &x;
    int &y;
};
```

值捕获和引用捕获的比较

我们发现引用捕获和值捕获有几个不同的地方

1. 参数以引用的方式进行传递
2. 引用捕获在函数体内修改变量，会直接改变Lambda表达式外部的变量
3. `operator()` 函数不是 `const` 的

所以我们把Lambda表达式的各个部分和类的各个成分对应起来就是如下关系：

1. **捕获**：对应LamdbaClass类的**private成员**
2. **形参**：对应LambdaClass类的成员函数的 `operator()` 的形参列表
3. `mutable` **关键字**：对应LambdaClass类成员函数的 `operator()` 的const属性，但是**只有在捕获列表捕获的参数不含有引用捕获的情况下才生效**，因为捕获列表只要包括引用捕获，那么 `operator()` 函数就一定是非const函数
4. **返回类型**：对应LambdaClass类成员函数 `operator()` 的返回类型

5. **函数体**：对应LambdaClass类成员函数 `operator()` 的函数体