C++ Lambda 编译器实现原理

作者：Private
原文链接：https://ld246.com/article/1545790136502
来源网站：链滴
许可协议：署名-相同方式共享 4.0 国际 (CC BY-SA 4.0)

Lambda 表达式语法

Lambda表达式完整的格式如下：

```
[捕获列表] (形参列表) mutable 异常列表-> 返回类型
{
    函数体
}
```

各项的含义：

1. **捕获列表**：捕获外部变量，捕获的变量可以在函数体中使用，可以省略，即不捕获外部变量。
2. **形参列表**：和普通函数的形参列表一样。可省略，即无参数列表
3. **mutable**：mutable关键字，如果有，则表示在函数体中可以修改捕获变量，根据具体需求决定是需要省略。
4. **异常列表**：noexcept / throw(...),和普通函数的异常列表一样，可省略，即代表可能抛出任何类型异常。
5. **返回类型**：和函数的返回类型一样。可省略，如省略，编译器将自动推导返回类型。
6. **函数体**：代码实现。可省略，但是没意义。

使用示例

```
void LambdaDemo()
{
    int a = 1;
    int b = 2;
    auto lambda = [a, b](int x, int y)mutable throw() -> bool
    {
        return a + b > x + y;
    };
    bool ret = lambda(3, 4);
}
```

编译器实现原理

编译器实现**lambda表达式**大致分为一下几个步骤

1. 创建 **lambda类**，实现构造函数，使用 lambda 表达式的函数体重载 **operator()**（所以 lambd 表达式 也叫匿名函数对象）
2. 创建lambda对象
3. 通过对象调用 **operator()**

编译器将lambda表达式翻译后的代码：

```
class lambda_xxxx
{
private:
```

```
    int a;
    int b;
public:
    lambda_xxxx(int _a, int _b) :a(_a), b(_b)
    {
    }
    bool operator()(int x, int y) throw()
    {
        return a + b > x + y;
    }
};
void LambdaDemo()
{
    int a = 1;
    int b = 2;
    lambda_xxxx lambda = lambda_xxxx(a, b);
    bool ret = lambda.operator()(3, 4);
}
```

其中，类名 **lambda_xxxx** 的 xxxx 是为了防止命名冲突加上的。

lambda_xxxx 与 lambda表达式 的对应关系

1. lambda表达式中的 **捕获列表**，对应 lambda_xxxx 类的 **private 成员**
2. lambda表达式中的 **形参列表**，对应 lambda_xxxx 类成员函数 **operator() 的形参列表**
3. lambda表达式中的 **mutable**，对应 lambda_xxxx 类成员函数 **operator() 的常属性 const**，是否是 **常成员函数**
4. lambda表达式中的 **返回类型**，对应 lambda_xxxx 类成员函数 **operator() 的返回类型**
5. lambda表达式中的 **函数体**，对应 lambda_xxxx 类成员函数 **operator() 的函数体**

另外，lambda 表达 捕获列表的捕获方式，也影响 对应 lambda_xxxx 类的 private 成员 的类型

1. 值捕获：private 成员 的类型与捕获变量的类型一致
2. 引用捕获：private 成员 的类型是捕获变量的引用类型

不捕获任何外部变量

如果lambda表达式不捕获任何外部变量，在特定的情况下，会有额外的代码生成。

其中，特定情况是指：有 **lambda_xxxx 类** 到 **函数指针** 的类型转换

如以下代码

```
typedef int(_stdcall *Func)(int);
int Test(Func func)
{
    return func(1);
}
void LambdaDemo()
{
    Test([](int i) {
        return i;
```

```
    });
}
```

Test函数接受一个函数指针作为参数，并调用这个函数指针。

实际调用Test时，传入的参数却是一个Lambda表达式，所以这里有一个类型的隐式转换

lambda_xxxx => 函数指针。

上面已经提到，Lambda表达式就是一个lambda_xxxx 类的匿名对象，与函数指针之间按理说不应该在转换，但是上述代码却没有问题。

其问题关键在于，上述代码中，lambda表达式没有捕获任何外部变量，即lambda_xxxx 类没有任何员变量，在operator() 中也就不会用到任何成员变量，也就是说，operator() 虽然是个成员函数，它不依赖 this 就可以调用。

因为不依赖 this，所以 一个lambda_xxxx 类的匿名对象与函数指针之间就存在转换的可能。

大致过程如下：

1. 在lambda_xxxx 类中生成一个 **静态函数**，静态函数的函数签名**与 operator() 一致**，在这个静函数中，通过一个**空指针**去调用该类的 operator()

2.在lambda_xxxx 重载与函数指针的类型转换操作符，在这个函数中，返回第1步中静态函数的地址。

上述代码在编译器的翻译后代码如下：

```
typedef int(_stdcall *Func)(int);

class lambda_xxxx
{
private:
    //没有捕获任何外部变量，所有没有成员
public:
        /*...省略其他代码...*/
    int operator()(int i)
    {
        return i;
    }
    static int _stdcall lambda_invoker_stdcall(int i)
    {
        return ((lambda_xxxx *)nullptr)->operator()(i);
    }

    operator Func() const
    {
        return &lambda_invoker_stdcall;
    }
};

int Test(Func func)
{
    return func(1);
}
void LambdaDemo()
{
```

```
    auto lambda = lambda_xxxx ();
    Func func = lambda.operator Func();
    Test(func);
}
```

上述代码只是以 **__stdcall** 调用约定的函数指针举例，实际使用时，对于不同调用约定，会生成对应本的静态函数和类型转换函数

以上结论的正确性可以通过**显式调用**lambda的转换函数与反汇编来证明

```
void LambdaDemo()
{
    auto lambda = [](int i) {return i;};
    Func func = lambda.operator Func();
    Test(func);
}
```

- 转为函数指针

```
    Func func = lambda.operator Func();
00B01790  lea         ecx,[lambda]
00B01793  call        <lambda_6de3db419a642e8a3c5a238fa0e4791f>::operator int (__stdcall*)(int) (0B016E0h)
00B01798  mov         dword ptr [func],eax
```

- 将静态函数 lambda_invoker_stdcall 地址作为 类型转换函数的返回值

```
00B01703  mov         eax,offset <lambda_6de3db419a642e8a3c5a238fa0e4791f>::<lambda_invoker_stdcall> (0B01690h)
```

- 静态函数 lambda_invoker_stdcall 中，使用 0 作为 this 调用 operator()

```
003916B2  xor         ecx,ecx
003916B4  call        <lambda_6de3db419a642e8a3c5a238fa0e4791f>::operator() (0391720h)
```