

16 | 函数对象和lambda：进入函数式编程

2020-01-01 吴咏炜

现代C++实战30讲

[进入课程 >](#)



讲述：吴咏炜

时长 14:28 大小 13.26M



你好，我是吴咏炜。

本讲我们将介绍函数对象，尤其是匿名函数对象——lambda 表达式。今天的内容说难不难，但可能跟你的日常思维方式有较大的区别，建议你一定要试验一下文中的代码（使用 xeus-cling 的同学要注意：xeus-cling 似乎不太喜欢有 lambda 的代码 😞；遇到有问题时，还是只能回到普通的编译执行方式了）。

C++98 的函数对象

函数对象（function object）[\[1\]](#) 自 C++98 开始就已经被标准化了。从概念上来说，函数对象是一个可以被当作函数来用的对象。它有时也会被叫做 functor，但这个术语在范畴论

里有着完全不同的含义，还是不用为妙——否则玩函数式编程的人可能会朝着你大皱眉头的。

下面的代码定义了一个简单的加 n 的函数对象类（根据一般的惯例，我们使用了 `struct` 关键字而不是 `class` 关键字）：

```
1 struct adder {
2     adder(int n) : n_(n) {}
3     int operator()(int x) const
4     {
5         return x + n_;
6     }
7 private:
8     int n_;
9 };
```

 复制代码

它看起来相当普通，唯一有点特别的地方就是定义了一个 `operator()`，这个运算符允许我们像调用函数一样使用小括号的语法。随后，我们可以定义一个实际的函数对象，如 C++11 形式的：

```
1 auto add_2 = adder(2);
```

 复制代码

或 C++98 形式的：

```
1 adder add_2(2);
```

 复制代码

得到的结果 `add_2` 就可以当作一个函数来用了。你如果写下 `add_2(5)` 的话，就会得到结果 7。

C++98 里也定义了少数高阶函数：你可以传递一个函数对象过去，结果得到一个新的函数对象。最典型的也许是目前已经从 C++17 标准里移除的 `bind1st` 和 `bind2nd` 了（在 `<functional>` 头文件中提供）：

[复制代码](#)

```
1 auto add_2 = bind2nd(plus<int>(), 2);
```

这样产生的 `add_2` 功能和前面相同，是把参数 2 当作第二个参数绑定到函数对象 `plus<int>`（它的 `operator()` 需要两个参数）上的结果。当然，`auto` 在 C++98 里是没有的，结果要赋给一个变量就有点别扭了，得写成：

[复制代码](#)

```
1 binder2nd<plus<int> > add_2(  
2     plus<int>(), 2);
```

因此，在 C++98 里我们通常会直接使用绑定的结果：

[复制代码](#)

```
1 #include <algorithm>  
2 #include <functional>  
3 #include <vector>  
4 using namespace std;  
5  
6 vector v{1, 2, 3, 4, 5};  
7 transform(v.begin(), v.end(),  
8           v.begin(),  
9           bind2nd(plus<int>(), 2));
```

上面的代码会将容器里的每一项数值都加上 2（`transform` 函数模板在 `<algorithm>` 头文件中提供）。可以验证结果：

[复制代码](#)


```
1 v
```

```
{ 3, 4, 5, 6, 7 }
```

函数的指针和引用

除非你用一个引用模板参数来捕捉函数类型，传递给一个函数的函数实参会退化成为一个函数指针。不管是函数指针还是函数引用，你也都可以当成函数对象来用。

假设我们有下面的函数定义：

 复制代码

```
1 int add_2(int x)
2 {
3     return x + 2;
4 };
```

如果我们有下面的模板声明：

 复制代码

```
1 template <typename T>
2 auto test1(T fn)
3 {
4     return fn(2);
5 }
6
7 template <typename T>
8 auto test2(T& fn)
9 {
10    return fn(2);
11 }
12
13 template <typename T>
14 auto test3(T* fn)
15 {
16    return (*fn)(2);
17 }
```

当我们拿 `add_2` 去调用这三个函数模板时，`fn` 的类型将分别被推导为 `int (*)(int)`、`int (&)(int)` 和 `int (*)(int)`。不管我们得到的是指针还是引用，我们都可以直接拿它当普通的函数用。当然，在函数指针的情况下，我们直接写 `*value` 也可以。因而上面三个函数拿 `add_2` 作为实参调用的结果都是 4。


很多接收函数对象的地方，也可以接收函数的指针或引用。但在个别情况下，需要通过函数对象的类型来区分函数对象的时候，就不能使用函数指针或引用了——原型相同的函数，

它们的类型也是相同的。

Lambda 表达式

Lambda 表达式 [2] 是一个源自阿隆佐·邱奇（Alonzo Church）——艾伦·图灵（Alan Turing）的老师——的术语。邱奇创立了 λ 演算 [3]，后来被证明和图灵机是等价的。

我们先不看数学上的 λ 表达式，看一下上一节给出的代码在使用 lambda 表达式时可以如何简化。

 复制代码

```
1 auto add_2 = [](int x) {  
2     return x + 2;  
3 };
```

显然，定义 `add_2` 不再需要定义一个额外的类型了，我们可以直接写出它的定义。理解它只需要注意下面几点：

Lambda 表达式以一对中括号开始（中括号中是可以有内容的；稍后我们再说）


跟函数定义一样，我们有参数列表

跟正常的函数定义一样，我们会有一个函数体，里面会有 `return` 语句

Lambda 表达式一般不需要说明返回值（相当于 `auto`）；有特殊情况需要说明时，则应使用箭头语法的方式（参见 [🔗\[第 8 讲\]](#)）：`[](int x) -> int { ... }`

每个 lambda 表达式都有一个全局唯一的类型，要精确捕捉 lambda 表达式到一个变量中，只能通过 `auto` 声明的方式

当然，我们想要定义一个通用的 `adder` 也不难：

 复制代码

```
1 auto adder = [](int n) {  
2     return [n](int x) {  
3         return x + n;  
4     };  
5 };
```

这次我们直接返回了一个 lambda 表达式，并且中括号中写了 `n` 来捕获变量 `n` 的数值。这个函数的实际效果和前面的 `adder` 函数对象完全一致。也就是说，捕获 `n` 的效果相当于在一个函数对象中用成员变量存储其数值。

纯粹为了满足你可能有的好奇心，上面的 `adder` 相当于这样一个 λ 表达式：

$$\text{adder} = \lambda n. (\lambda x. (+ x n))$$

如果你去学 Lisp 或 Scheme 的话，你就会发现这些语言和 λ 表达式几乎是一一映射了。在 C++ 里，表达虽然稍微啰嗦一点，但也比较接近了。用我上面的 `adder`，就可以得到类似于函数式编程语言里的 currying [4] 的效果——把一个操作（此处是加法）分成几步来完成。没见过函数式编程的，可能对下面的表达式感到奇怪吧：

```
1 auto seven = adder(2)(5);
```

 复制代码

不过，最常见的情况是，写匿名函数就是希望不需要起名字。以前面的把所有容器元素值加 2 的操作为例，使用匿名函数可以得到更简洁可读的代码：

```
1 transform(v.begin(), v.end(),  
2          v.begin(),  
3          [](int x) {  
4              return x + 2;  
5          });
```

 复制代码

到了可以使用 `ranges`（已在 C++20 标准化）的时候，代码可以更短、更灵活。这个我们就留到后面再说了。

一个 lambda 表达式除了没有名字之外，还有一个特点是你立即进行求值。这就使得我们可以把一段独立的代码封装起来，达到更干净、表意的效果。

先看一个简单的例子：

```
1 [](int x) { return x * x; }(3)
```

这个表达式的结果是 3 的平方 9。即使这个看似无聊的例子，都是有意义的，因为它免去了我们定义一个 constexpr 函数的必要。只要能满足 constexpr 函数的条件，一个 lambda 表达式默认就是 constexpr 函数。

另外一种用途是解决多重初始化路径的问题。假设你有这样的代码：

```
1 Obj obj;
2 switch (init_mode) {
3 case init_model:
4     obj = Obj(...);
5     break;
6 case init_mode2:
7     obj = Obj(...);
8     break;
9 ...
10 }
```

这样的代码，实际上是调用了默认构造函数、带参数的构造函数和（移动）赋值函数：既可能有性能损失，也对 Obj 提出了有默认构造函数的额外要求。对于这样的代码，有一种重构意见是把这样的代码分离成独立的函数。不过，有时候更直截了当的做法是用一个 lambda 表达式来进行改造，既可以提升性能（不需要默认函数或拷贝 / 移动），又让初始化部分显得更清晰：

```
1 auto obj = [init_mode]() {
2     switch (init_mode) {
3     case init_model:
4         return Obj(...);
5         break;
6     case init_mode2:
7         return Obj(...);
8         break;
9     ...
10 }
11 }();
```


变量捕获

现在我们来细看一下 lambda 表达式中变量捕获的细节。

变量捕获的开头是可选的默认捕获符 `=` 或 `&`，表示会自动按值或按引用捕获用到的本地变量，然后后面可以跟（逗号分隔）：

本地变量名标明对其按值捕获（不能在默认捕获符 `=` 后出现；因其已自动按值捕获所有本地变量）

`&` 加本地变量名标明对其按引用捕获（不能在默认捕获符 `&` 后出现；因其已自动按引用捕获所有本地变量）

`this` 标明按引用捕获外围对象（针对 lambda 表达式定义出现在一个非静态类成员内的情况）；注意默认捕获符 `=` 和 `&` 号可以自动捕获 `this`（并且在 C++20 之前，在 `=` 后写 `this` 会导致出错）

`*this` 标明按值捕获外围对象（针对 lambda 表达式定义出现在一个非静态类成员内的情况；C++17 新增语法）

变量名 `=` 表达式 标明按值捕获表达式的结果（可理解为 `auto` 变量名 `=` 表达式）

`&`变量名 `=` 表达式 标明按引用捕获表达式的结果（可理解为 `auto&` 变量名 `=` 表达式）

从工程的角度，大部分情况不推荐使用默认捕获符。更一般化的一条工程原则是：**显式的代码比隐式的代码更容易维护**。当然，在这条原则上走多远是需要权衡的，你也不愿意写出非常啰嗦的代码吧？否则的话，大家就全部去写 C 了。

一般而言，按值捕获是比较安全的做法。按引用捕获时则需要更小心些，必须能够确保被捕获的变量和 lambda 表达式的生命期至少一样长，并在有下面需求之一时才使用：

需要在 lambda 表达式中修改这个变量并让外部观察到

需要看到这个变量在外部被修改的结果

这个变量的复制代价比较高

如果希望以移动的方式来捕获某个变量的话，则应考虑 `变量名 = 表达式` 的形式。表达式可以返回一个 `prvalue` 或 `xvalue`，比如可以是 `std::move` (需移动捕获的变量)。

上一节我们已经见过简单的按值捕获。下面是一些更多的演示变量捕获的例子。


按引用捕获：

 复制代码

```
1 vector<int> v1;
2 vector<int> v2;
3 ...
4 auto push_data = [&](int n) {
5     // 或使用 [&v1, &v2] 捕捉
6     v1.push_back(n);
7     v2.push_back(n)
8 };
9
10 push_data(2);
11 push_data(3);
```

这个例子很简单。我们按引用捕获 `v1` 和 `v2`，因为我们需要修改它们的内容。

按值捕获外围对象：

 复制代码

```
1 #include <chrono>
2 #include <iostream>
3 #include <sstream>
4 #include <thread>
5
6 using namespace std;
7
8 int get_count()
9 {
10     static int count = 0;
11     return ++count;
12 }
13
14 class task {
15 public:
16     task(int data) : data_(data) {}
17     auto lazy_launch()
18     {
```

```

19     return
20     [*this, count = get_count]()
21     mutable {
22         ostringstream oss;
23         oss << "Done work " << data_
24             << " (No. " << count
25             << ") in thread "
26             << this_thread::get_id()
27             << '\n';
28         msg_ = oss.str();
29         calculate();
30     };
31 }
32 void calculate()
33 {
34     this_thread::sleep_for(100ms);
35     cout << msg_;
36 }
37
38 private:
39     int data_;
40     string msg_;
41 };
42
43 int main()
44 {
45     auto t = task{37};
46     thread t1{t.lazy_launch()};
47     thread t2{t.lazy_launch()};
48     t1.join();
49     t2.join();
50 }

```

这个例子稍复杂，演示了好几个 lambda 表达式的特性：

`mutable` 标记使捕获的内容可更改（缺省不可更改捕获的值，相当于定义了 `operator() (...) const`）；

`[*this]` 按值捕获外围对象（`task`）；

`[count = get_count()]` 捕获表达式可以在生成 lambda 表达式时计算并存储等号后表达式的结果。

这样，多个线程复制了任务对象，可以独立地进行计算。请自行运行一下代码，并把 `*this` 改成 `this`，看看输出会有什么不同。

泛型 lambda 表达式


函数的返回值可以 `auto`，但参数还是要一一声明的。在 lambda 表达式里则更进一步，在参数声明时就可以使用 `auto`（包括 `auto&&` 等形式）。不过，它的功能也不那么神秘，就是给你自动声明了模板而已。毕竟，在 lambda 表达式的定义过程中是没法写 `template` 关键字的。

还是拿例子说话：

 复制代码

```
1 template <typename T1,  
2         typename T2>  
3 auto sum(T1 x, T2 y)  
4 {  
5     return x + y;  
6 }
```

跟上面的函数等价的 lambda 表达式是：

 复制代码

```
1 auto sum = [](auto x, auto y)  
2 {  
3     return x + y;  
4 }
```

是不是反而更简单了？😁

你可能要问，这么写有什么用呢？问得好。简单来说，答案是可组合性。上面这个 `sum`，就跟标准库里的 `plus` 模板一样，是可以传递给其他接受函数对象的函数的，而 `+` 本身则不行。下面的例子虽然略有点无聊，也可以演示一下：

 复制代码

```
1 #include <array>      // std::array  
2 #include <iostream>    // std::cout/endl  
3 #include <numeric>     // std::accumulate  
4  
5 using namespace std;  
6
```

```

7  int main()
8  {
9      array a{1, 2, 3, 4, 5};
10     auto s = accumulate(
11         a.begin(), a.end(), 0,
12         [](auto x, auto y) {
13             return x + y;
14         });
15     cout << s << endl;
16 }

```

虽然函数名字叫 `accumulate`——累加——但它的行为是通过第四个参数可修改的。我们把上面的加号 `+` 改成星号 `*`，上面的计算就从从 1 加到 5 变成了算 5 的阶乘了。

bind 模板

我们上面提到了 `bind1st` 和 `bind2nd` 目前已经从 C++ 标准里移除。原因实际上有两个：

- 它的功能可以被 `lambda` 表达式替代

- 有了一个更强大的 `bind` 模板 [\[5\]](#)

拿我们之前给出的例子：

```

1  transform(v.begin(), v.end(),
2           v.begin(),
3           bind2nd(plus<int>(), 2));

```

 复制代码

现在我们可以写成：

```

1  using namespace std::
2  placeholders; // for _1, _2...
3  transform(v.begin(), v.end(),
4           v.begin(),
5           bind(plus<>(), _1, 2));

```

 复制代码


原先我们只能把一个给定的参数绑定到第一个参数或第二个参数上，现在则可以非常自由地适配各种更复杂的情况！当然，`bind` 的参数数量，必须是第一个参数（函数对象）所需的参数数量加一。而 `bind` 的结果的参数数量则没有限制——你可以无聊地写出 `bind(plus<>(), _1, _3)(1, 2, 3)`，而结果是 4（完全忽略第二个参数）。

你可能会问，它的功能是不是可以被 `lambda` 表达式替代呢。回答是“是”。对 `bind` 只需要稍微了解一下就好——在 C++14 之后的年代里，已经没有什么地方必须要使用 `bind` 了。

function 模板

每一个 `lambda` 表达式都是一个单独的类型，所以只能使用 `auto` 或模板参数来接收结果。在很多情况下，我们需要使用一个更方便的通用类型来接收，这时我们就可以使用 `function` 模板 [6]。`function` 模板的参数就是函数的类型，一个函数对象放到 `function` 里之后，外界可以观察到的就只剩下它的参数、返回值类型和执行效果了。注意 `function` 对象的创建还是比较耗资源的，所以请你只在用 `auto` 等方法解决不了问题的时候使用这个模板。

下面是个简单的例子。

 复制代码

```
1 map<string, function<int(int, int)>>
2   op_dict{
3     {"+",
4      [](int x, int y) {
5        return x + y;
6      }},
7     {"-",
8      [](int x, int y) {
9        return x - y;
10      }},
11     {"*",
12      [](int x, int y) {
13        return x * y;
14      }},
15     {"/",
16      [](int x, int y) {
17        return x / y;
18      }},
19  };
```

这儿，由于要把函数对象存到一个 `map` 里，我们必须使用 `function` 模板。随后，我们就可以用类似于 `op_dict.at("+")(1, 6)` 这样的方式来使用 `function` 对象。这种方式对表达式的解析处理可能会比较有用。

内容小结

在这一讲中，我们了解了函数对象和 `lambda` 表达式的基本概念，并简单介绍了 `bind` 模板和 `function` 模板。它们在泛型编程和函数式编程中都是重要的基础组成部分，你应该熟练掌握。

课后思考

请：

1. 尝试一下，把文章的 `lambda` 表达式改造成完全不使用 `lambda`。
2. 体会一下，`lambda` 表达式带来了哪些表达上的好处。

欢迎留言和我分享你的想法。

参考资料

[1] Wikipedia, “Function object” . [🔗 https://en.wikipedia.org/wiki/Function_object](https://en.wikipedia.org/wiki/Function_object)

[1a] 维基百科, “函数对象” . [🔗 https://zh.wikipedia.org/zh-cn/ 函数对象](https://zh.wikipedia.org/zh-cn/函数对象)

[2] Wikipedia, “Anonymous function” . [🔗 https://en.wikipedia.org/wiki/Anonymous_function](https://en.wikipedia.org/wiki/Anonymous_function)

[2a] 维基百科, “匿名函数” . [🔗 https://zh.wikipedia.org/zh-cn/ 匿名函数](https://zh.wikipedia.org/zh-cn/匿名函数)

[3] Wikipedia, “Lambda calculus” .
[🔗 https://en.wikipedia.org/wiki/Lambda_calculus](https://en.wikipedia.org/wiki/Lambda_calculus)

[3a] 维基百科, “λ演算” . <https://zh.wikipedia.org/zh-cn/λ演算>

[4] Wikipedia, “Currying” . <https://en.wikipedia.org/wiki/Currying>

[4a] 维基百科, “柯里化” . <https://zh.wikipedia.org/zh-cn/柯里化>

[5] cppreference.com, “std::bind” .

<https://en.cppreference.com/w/cpp/utility/functional/bind>

[5a] cppreference.com, “std::bind” .

<https://zh.cppreference.com/w/cpp/utility/functional/bind>

[6] cppreference.com, “std::function” .

<https://en.cppreference.com/w/cpp/utility/functional/function>

[6a] cppreference.com, “std::function” .

<https://zh.cppreference.com/w/cpp/utility/functional/function>

点击查看 

打卡学习 C++ 拒绝从入门到放弃



PC 端用户扫码参与



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

上一篇 15 | constexpr: 一个常态的世界

下一篇 17 | 函数式编程: 一种越来越流行的编程范式

精选留言 (11)

写留言



总统老唐

2020-01-01

2020第一课, 吴老师新年好

展开

作者回复: 谢谢 🙏。在这儿也顺祝所有的同学们新年好! 🎉🎊🎊



3



Encoded Star

2020-01-09

函数指针和引用这个模块中

当我们拿 add_2 去调用这三个函数模板时, fn 的类型将分别被推导为 `int (*)(int)`、`int (&)(int)` 和 `int (*)(int)`。

第一个和第三个都是 `int (*)(int)` 第一个是不是 `int (int)`

展开

作者回复: 不是。你漏看了这句话:

“除非你用一个引用模板参数来捕捉函数类型, 传递给一个函数的函数实参会退化成为一个函数指针。”



1



空气

2020-01-04

吴老师, 我在工作中很经常用到function。文中讲到function对象的创建比较耗资源, 能否介绍一下原因, 或者可以参考哪些资料? 确实要使用的话, 是否有必要使用共享指针管理来减轻复制和转移消耗?

如果lambda的推导类型不是function, 那是什么类型呢? 和function有什么区别?

展开

作者回复: 你如果不是频繁创建 function 对象的话, 关系也不大吧。我觉得多考虑移动就行了。除非性能测试工具报告瓶颈就在这儿了, 用智能指针去优化不太值 (毕竟需要修改使用的代码)。

每个 lambda 都有自己的独特类型, 每次定义相当于编译器帮你产生了一个函数对象 (就像这一讲里定义的那些函数对象一样)。

具体如何实现, 我倒没读到过相关的文章。你可以网上搜搜看, 或者阅读标准库里的源码。



橙子888

2020-01-04

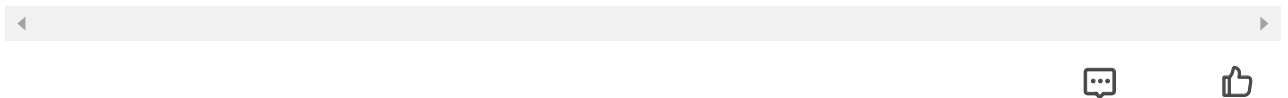
最近项目里使用到了libgo这个C++写的协程库, 示例代码中用到了好多老师今天讲的知识点:

```
void foo()
{
    printf("function pointer\n");...
```

展开 ∨

作者回复: libgo 我没有任何使用经验, 不过, 看起来它和大部分库实现的协程一样, 都是 stackful coroutine。我第 30 讲会讲的是会进入 C++20 的 stackless coroutine。

每个 libgo 的协程都有自己的独立栈空间, 因此, 协程唤起和休眠时都需要进行栈切换。无栈协程则跟唤起者使用同一个栈。有栈的协程实现另外还有 libco、Boost.Coroutine2 等。



李亮亮

2020-01-02

Microsoft Visual Studio Community 2019 版本 16.4.2, 语言标准: C++17 例子编译不过, 水平又菜, 不会改。

展开 ∨

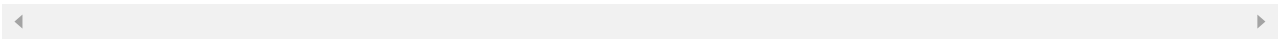
作者回复: 是说有线程的那个例子吗? 我刚又试了, 没问题的。

你是不是没有设定语言标准为 c++17? 但你评论里又说设了.....不设是确实不行的。

我编译的命令是:

```
cl /EHsc /std:c++17 test16.cpp
```

如果你遇到错误了，又不贴出错误信息，别人也没法帮你啊.....



5



廖熊猫

2020-01-02

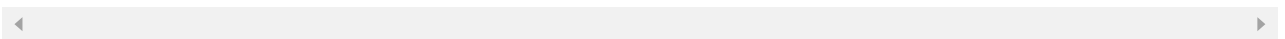
老师新年快乐。

lambda表达式大概是生成了一个匿名的struct吧，实现了operator(), 捕获的话对应struct上的字段。

展开 ▾

作者回复: 新年快乐。

对，概念上就是这样。



tt

2020-01-02

1、感觉lambda表达式就是C++中的闭包。

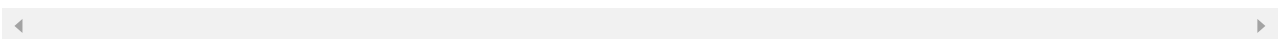
2、lambda表达式可以立即进行求职，这一点和JavaScript里的立即执行函数（Immediately Invoked Function Expression, IIFE）一样。在JavaScript里，它是用来解决作用域缺陷的。...

展开 ▾

作者回复: 对，就是闭包。

Stackful 协程见 Boost.Coroutine2。Stackless 协程已经进入 C++20，第 30 讲讨论。🤖

新年快乐！



禾桃

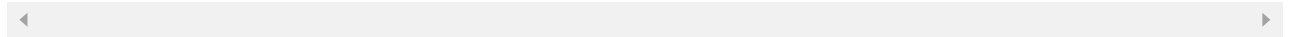
2020-01-01

"请自行运行一下代码，并把 *this 改成 this，看看输出会有什么不同。"

```
int get_count()
{
    static int count = 0;...
```

展开 ▾

作者回复: 就是让你想一想的呀。提示: 按引用捕获的后果。



6



罗乾林

2020-01-01

编译器遇到lambda 表达式时，产生一个匿名的函数对象，各种捕获相当于按值或者按引用设置给匿名对象的成员字段。

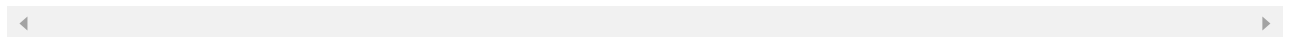
不对的地方，望老师指正。

对function<int(int, int)>这货怎么实现的比较好奇，大多数模板参数都是类型，做的都是类型推导，这货居然是int(int, int)

展开 ▾

作者回复: lambda表达式的理解没啥问题。

int(int, int) 也是一个类型：一个接受两个整数参数、返回一个整数的函数。function 的主要复杂性，应该是需要处理函数、函数指针、函数对象等各种情况。函数对象的大小不确定，因而 function 需要在堆上分配内存。operator() 我记得相当于一个虚函数调用的复杂度。

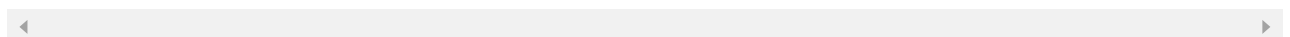


hello world

2020-01-01

请问老师后续会讲关于类对象及虚函数表相关知识吗，这块比较薄弱

作者回复: 不会。谈这个的书和文章够多了。



viper

2020-01-01

老师，为什么上面会说用add_2去调用那三模版函数返回值都是2，不该是4吗？

作者回复: 谢谢反馈。已更正。

