C++ lambda的演化

0. 前言

本书是bfilipek.com上两篇文章的更新版：

Lambdas: From C++11 to C++20, Part 1

Lambdas: From C++11 to C++20, Part 2

本书介绍lambda表达式，我们将从C++03开始，一路进入最新的C++标准。

- C++11——早期。你将了解到lambda表达式的组成以及一些技巧。这是最长的一章，因为我们要讲很多内容。
- C++14——更新。lambda被加入标准以后，我们发现有很多地方可以优化。

- C++17——更多改进，特别是处理this指针，以及允许constexpr。
- C++20——在这一章中我们将一瞥未来。

反馈

如果你发现任何错误，请让我们知晓！可以给bartlomiej.filipek AT bfilipek.com（将AT替换为@）发邮件，或在Leanpub的反馈页面留下反馈。

这本书中的代码基于Creative Commons协议。

本书中的很多代码可在Wandbox在线编译器中运行，正文中有相应链接。这里还有一个在线编译器列表：在线C++编译器列表。

关于作者

Bartłomiej Filipek是一位拥有超过11年专业经验的C++软件开发者。他于2010年毕业于波兰克拉科夫市雅盖隆大学并获得计算机科学硕士学位，现供职于Xara。

Bartek自2011年开始在他的网站bfilipek.com写博客。早期的主题围绕图形编程，现在的博文聚焦核心C++。他也是克拉科夫C++用户组的协办者。你还可以在CppCast episode上收听他讲的C++17、博客与文字处理。

自2018年10月，Bartek成为直接与ISO/IEC JTC 1/SC 22（C++标准委员会）协作的波兰国家机构的一位C++专家。同月，他获得第一个2019/2020年度微软MVP头衔。Bartek同时还是C++17 In Detail的作者。

译者的话

本书围绕lambda表达式的主题，从C++03讲到C++20，花了近万字篇幅（中文）。时间跨度长而又紧扣主题，这样纵向地介绍一项具体技术的文章是不多的。读者们不仅可以从中学到知识，更应该从这一语言特性的演化中发现背后的道理。

这是我第一次翻译完整的文章。英语和中文对同一内容的表达方式有所不同，阅读英语原文让我更好地理解作者的意图，从而更有效地学习知识；有些很容易理解的英语表达，想翻译成中文却不太容易；逐字逐句的阅读也让我注意到了许多原本可能忽略的细节——这些感受是我翻译过程中最大的收获。

初次翻译，经验不足；如有疏漏，烦请指正。

1. C++03中的LAMBDA

从早期的标准库开始，std::sort之类的算法就接受可调用对象作为参数，用容器中的每个对象调用它。然而，在C++03中这只能是函数指针与仿函数，例如：（点击代码标题在线运行代码）

基础的打印仿函数

```
#include <algorithm>
#include <iostream>
#include <vector>

struct PrintFunctor {
    void operator()(int x) const {
```

```
        std::cout << x << std::endl;
    }
};
int main() {
    std::vector<int> v;
    v.push_back(1);
    v.push_back(2);
    std::for_each(v.begin(), v.end(), PrintFunctor());
}
```

这个例子定义了一个带有operator()的简单仿函数。

函数指针是无状态的，而仿函数可以包含一些状态。一个例子是数调用次数：

有状态的仿函数

```
#include <iostream>
#include <algorithm>
#include <vector>

struct PrintFunctor {
    PrintFunctor(): numCalls(0) { }

    void operator()(int x) const {
        std::cout << x << '\n';
        ++numCalls;
    }

    mutable int numCalls;
};

int main() {
    std::vector<int> v;
    v.push_back(1);
    v.push_back(2);
    PrintFunctor visitor = std::for_each(v.begin(), v.end(), PrintFunctor());
    std::cout << "num calls: " << visitor.numCalls << '\n';
}
```

在上面的例子中，我们用一个成员变量来数调用运算符被调用的次数。由于调用运算符是const的，我们必须用一个mutable变量。

我们还可以从主调作用域中"捕获"变量——在仿函数中创建一个成员变量并在构造函数中初始化。

带有捕获变量的仿函数

```
#include <algorithm>
#include <iostream>
#include <string>
#include <vector>

struct PrintFunctor {
    PrintFunctor(const std::string& str): strText(str), numCalls(0) { }

    void operator()(int x) const {
        std::cout << strText << x << '\n';
        ++numCalls;
```

```
    }

    std::string strText;
    mutable int numCalls;
};

int main() {
    std::vector<int> v;
    v.push_back(1);
    v.push_back(2);
    const std::string introText("Elem: ");
    PrintFunctor visitor = std::for_each(v.begin(), v.end(),
PrintFunctor(introText));
    std::cout << "num calls: " << visitor.numCalls << '\n';
}
```

在迭代（遍历）中，PrintFunctor接受一个额外的参数以初始化成员变量，然后这个变量在调用运算符中被使用。

问题

正如所见，仿函数很强大。它是一个单独的类，你可以自如地设计它。但问题是你必须在不同于算法调用处的另一个作用域中写一个单独的函数或仿函数。

一个可能的方案是，你可以写局部仿函数类——既然C++总是支持这种句法。然而这样并不可以……看这段代码：

局部仿函数

```
int main() {
    struct PrintFunctor {
        void operator()(int x) const {
            std::cout << x << std::endl;
        }
    };

    std::vector<int> v;
    std::for_each(v.begin(), v.end(), PrintFunctor());
}
```

在GCC中用-std=c++98编译，你会得到以下错误：

error: template argument for

'template<class _IIter, class _Funct> _Funct

std::for_each(_IIter, _IIter, _Funct)'

uses local type 'main()::PrintFunctor'

通常来说，在C++98/03中你不能用局部类型来实例化一个模板。

新特性的动机

在C++11中，委员会放宽了对局部变量实例化模板的限制，你可以在离使用处更近的地方写仿函数。

但C++11也提供了另一个方案：让编译器帮开发者写这样的小仿函数会怎样？这意味着新的句法，我们可以“就地”创建仿函数，使代码更干净简洁。这就是“lambda表达式”的起源！

在C++11的最终草案N3337中，我们可以看到单独的一节描述lambda：[expr.prim.lambda]。我们将在下一章中介绍这个新特性。

2. C++11中的LAMBDA

棒极了！C++委员会听取了C++03开发者的建议，从C++11开始我们有了lambda表达式！lambda很快成为了现代C++最突出的特性之一。

我们可以在C++11最终草案N3337中阅读lambda的特性，有单独的一节：[expr.prim.lambda]。

我认为lambda被以一个聪明的方式加入了语言。它们使用相同的句法，但编译器将它扩展为实际的类。在增强类型语言中，这种方法给我们带来很多好处（但有时也有坏处）。

在这一章中你将学到：

- Lambda的基本句法
- 如何捕获变量
- 如何捕获成员变量
- Lambda的返回类型是什么
- 什么是闭包
- 向函数指针的转换
- IIFE

让我们开始吧！

句法

这是一个基本的代码实例，同时也展示了相应的局部仿函数对象。

第一个lambda与一个相应的仿函数

```
#include <iostream>
#include <algorithm>
#include <vector>

int main()
{
    struct
    {
        void operator()(int x) const
        {
            std::cout << x << std::endl;
        }
    } someInstance;

    std::vector<int> v;
    v.push_back(1);
    v.push_back(2);
    std::for_each(v.begin(), v.end(), someInstance);
    std::for_each(v.begin(), v.end(), [] (int x) { std::cout << x << std::endl;
});
}
```

在这个例子中编译器将：

```
[](int x) { std::cout << x << '\n'; }
```

转换为这样（简化形式）：

```
struct{

    voidoperator()(intx) const{

        std::cout << x << '\n';

    }

} someInstance;
```

lambda表达式的句法：

[] () { code; }

^ ^ ^

| | |

| | 可选：mutable、异常、尾置返回.....

| |

| 可选：形参列表

|

lambda引导与捕获列表

在开始之前，先看一些定义。[expr.prim.lambda#2]：

*lambda*表达式的求值结果是纯右值临时变量，这个临时变量称为闭包对象。

以及[expr.prim.lambda#3]：

*lambda*表达式的类型（也就是闭包对象的类型）是一个独一无二的、无名的非联合体类型，称为闭包类型。

一些lambda表达式的例子：

```
[]{} // 最简单的lambda

[](float f, int a) { return a*f; }

[](MyClass t) -> int { auto a = t.compute(); return a; }

[](int a, int b) { return a < b; }

[x](int a, int b) mutable { return a < b; ++x; }
```

LAMBDA的类型

由于编译器为每个lambda生成一个独一无二的类名，我们没法提前知道它的类型。这就是你为什么必须用auto（或decltype）来推断类型。

```
auto myLambda = [](int a) -> double { return 2.0 * a; }
```

以及，[expr.prim.lambda]：

与***lambda***表达式相关联的闭包类型有删除的默认构造函数与拷贝构造函数。

这就是为什么你不能写：

```
auto foo = [&x, &y]() { ++x; ++y; };

decltype(foo) fooCopy;
```

GCC对这段代码给出以下错误：

error: use of deleted function 'main()::<lambda()>::()'

decltype(foo) fooCopy;

^~~

note: a lambda closure type has a deleted default constructor

另一个问题是，如果你有两个lambda：

```
auto firstLam = [](int x) { return x*2; };

auto secondLam = [](int x) { return x*2; };
```

它们的类型是不同的！即使"背后的代码"是相同的……总之编译器被要求为每个lambda声明独一无二的无名类型。

然而，你可以拷贝lambda：

拷贝lambda

```
#include <type_traits>

int main() {
    auto firstLam = [](int x) { return x*2; };
    auto secondLam = firstLam;
    static_assert(std::is_same_v<decltype(firstLam), decltype(secondLam)>);
}
```

拷贝lambda也同时拷贝了它的状态。在有变量捕获时这很重要，闭包类型会把捕获变量作为成员域存储。

预见未来

在C++20中无状态的lambda可以被默认构造和赋值。

调用运算符

lambda函数体中写的代码被"翻译"为对应闭包类型的operator()中的代码。它默认是一个const inline方法，你可以在参数声明子句后指明mutable以改变这一限定：

```
auto myLambda = [](int a) mutable { std::cout << a; }
```

对于空捕获列表的lambda，const方法没有问题，但当你要从局部作用域捕获变量时，情况就不同了。捕获子句是下一节的主题：

捕获

[]不只是引导一个lambda，它也包含了捕获变量列表，称为“捕获子句”。

通过捕获变量，你在闭包类型中创建了那个变量的拷贝作为成员，然后你可以在lambda函数体中存取它。在C++03那一章中我们为PrintFunctor做过一件类似的事。在那个类中，我们加入了一个成员变量std::string strText;，由构造函数初始化。

捕获的基本句法：

- [&]——引用捕获所有声明在可触及作用域中的自动存取期限变量；
- [=]——值捕获，值被拷贝（拷贝捕获）；
- [x, &y]——显式地值捕获x、引用捕获y。

例如：

捕获变量

```
std::string str {"Hello World"};
auto foo = [str]() { std::cout << str << '\n'; };
foo();
```

在上面的lambda中，编译器可能生成这样的局部仿函数：

可能的编译器生成的仿函数，单个变量

```
struct _unnamedLambda {
    _unnamedLambda(std::string s) : str(s) { }
    void operator() const {
        std::cout << str << '\n';
    }
    std::string str;
};
```

从概念上讲，传入构造函数的变量在lambda声明时被使用。标准[expr.prim.lambda#21]中有更精确的描述：

当lambda表达式被求值时，拷贝捕获的实体被用于直接初始化结果闭包类型中对应的非static数据成员。

上面展示的可能的构造函数（_unnamedLambda）只是演示用途，编译器可能用不同的实现，并且不会暴露出来。

以引用方式捕获两个变量

```
int x = 1, y = 1;
std::cout << x << " " << y << std::endl;
auto foo = [&x, &y]() { ++x; ++y; };
foo();
std::cout << x << " " << y << std::endl;
```

对于上面的lambda，编译器可能生成这样的局部仿函数：

可能的编译器生成的仿函数，两个引用

```
struct _unnamedLambda {
    _unnamedLambda(int& a, int& b) : x(a), y(b) { }
    void operator() const {
        ++x; ++y;
    }
    int& x;
    int& y;
};
```

由于我们引用捕获x和y，闭包类型也会包含引用成员变量。

注意

值捕获变量的值在lambda被定义时的值——不是在使用时！引用捕获变量的值是在lambda被使用时的值——不是在定义时。

尽管写[=]或[&]方便，因为它捕获所有自动存储期限的变量，但显式捕获变量更加清楚，并且编译器会就不想要的效果给你警告（参见全局与静态变量）。你也可以在Scott Meyers的《Effective Modern C++》中条款31“避免默认捕获方式”中阅读更多。

注意

C++闭包不会延长捕获引用的生命周期。要确保捕获的变量在lambda调用时仍存在。

Mutable

operator()默认为const，你不能在lambda函数体中修改捕获的变量。如果要改变这种行为，需要在参数列表后加上mutable关键字：

拷贝捕获两个变量

```
int x = 1, y = 1;
std::cout << x << " " << y << std::endl;
auto foo = [x, y]() mutable { ++x; ++y; };
foo();
std::cout << x << " " << y << std::endl;
```

在上面的例子中，我们可以改变x和y的值。当然，由于他们只是父级作用域中x和y的拷贝，foo调用之后他们没有新的值。

另一方面，如果你以引用捕获，在非mutable的lambda中，你不能重新绑定引用，但你可以改变被引用的变量。

引用捕获一个变量

```
int x = 1;
std::cout << x << '\n';
auto foo = [&x]() { ++x; };
foo();
std::cout << x << '\n';
```

在上面的例子中，lambda不是mutable的，但它可以改变被引用的值。

捕获全局变量

如果有一个全局变量，然后你在lambda中用[=]，你可能觉得全局变量被值捕获了……然而并没有。

捕获全局变量

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <memory>

int global = 10;

int main()
{
    std::cout << global << std::endl;
    auto foo = [=] () mutable { ++global; };
    foo();
    std::cout << global << std::endl;
    [] { ++global; } ();
    std::cout << global << std::endl;
    [global] { ++global; } ();
}
```

只有自动存取期限的变量才可以被捕获。GCC甚至会给出以下警告：

warning: capture of variable 'global' with non-automatic storage duration

这个警告只会在你显式捕获全局变量时出现，所以如果你用[=]，编译器也帮不了你。

Clang编译器更好，它产生一个错误：（去看看）

error: 'global' cannot be captured because it does not have automatic storage d\

uration

捕获静态变量

与捕获全局变量类似，对于静态变量的情况，你会得到相同结果：

捕获静态变量

```
#include <iostream>

void bar()
{
    static int static_int = 10;
    std::cout << static_int << std::endl;
    auto foo = [=] () mutable { ++static_int; };
    foo();
    std::cout << static_int << std::endl;
    [] { ++static_int; } ();
    std::cout << static_int << std::endl;
    [static_int] { ++static_int; } ();
}

int main()
{
```

```
    bar();
}
```

输出是：

10

11

12

同样地，警告只会在你显式捕获静态变量时出现，如果你用[=]，编译器帮不上忙。

捕获成员变量与this

在类方法中，情况就有些复杂了：

捕获成员变量时的错误

```
#include <iostream>

struct Baz {
    void foo() {
        auto lam = [s]() { std::cout << s; };
        lam();
    }

    std::string s;
};

int main() {
    Baz b;
    b.foo();
}
```

代码试图捕获成员变量s，但编译器会给出一个错误：

In member function 'void Baz::foo()':

error: capture of non-variable 'Baz::s'

error: 'this' was not captured for this lambda function

...

为了解决这个问题，你必须捕获this指针，然后才可以存取成员变量。我们可以将代码改为：

```
struct Baz {
    void foo() {
        auto lam = [this]() { std::cout << s; };
        lam();
    }
    std::string s;
};
```

编译器不再产生错误。

你可以用[=]或[&]来捕获this（效果是相同的！），但请注意我们捕获的是this指针，所以存取的将是成员变量，而不是它的拷贝。

在C++11（甚至C++14）中你不能写：

```
auto lam = [*this]() { std::cout << s; };
```

来捕获对象的拷贝。

如果你在一个方法的上下文中使用lambda，捕获this一切安好，但对于更复杂的情况如何呢？你知道以下代码会发生什么吗？

从方法返回lambda

```
#include <iostream>
#include <functional>

struct Baz
{
    std::function<void()> foo()
    {
        return [=] { std::cout << s << std::endl; };
    }

    std::string s;
};

int main()
{
   auto f1 = Baz{"ala"}.foo();
   auto f2 = Baz{"ula"}.foo();
   f1();
   f2();
}
```

代码声明了一个Baz对象然后调用foo()。请注意foo()返回一个捕获类成员的lambda（存储在std::function中）。

由于使用临时对象，我们无法确定调用f1和f2时会发生什么。这是一个悬空引用问题，会导致未定义行为，类似于：

```
struct Bar {
    std::string const& foo() const { return s; };
    std::string s;
};
auto&& f1 = Bar{"ala"}.foo(); // 悬空引用
```

同样地，如果你显式捕获（[s]）：

```
std::function<void()> foo()
{
    return [s] { std::cout << s << std::endl; };
}
```

总之，由于lambda可以作用于对象生存周期之外（outlive the object），捕获this看起来有些狡猾。在异步调用或多线程环境中，这种情况就会发生。

我们将在C++17一章中回到这一话题。

仅可移动对象

如果你有一个只能移动不能拷贝的对象（比如std::unique_ptr），你不能把它作为捕获变量移动到lambda中。值捕获是不行的，所以你只能引用捕获，然而这不会转交所有权，而且多半不是你想要的。

```
std::unique_ptr<int> p(new int{10});
auto foo = [p] () {}; // 不能通过编译……
```

保留const

如果你捕获一个const变量，它的常量性被保留：（测试）

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <memory>
#include <iostream>
#include <functional>
#include <type_traits>


int main()
{
    int const x = 10;
    auto foo = [x] () mutable {
        std::cout << std::is_const<decltype(x)>::value << std::endl;
        x = 11;
    };
    foo();
    // remove_reference_t<decltype(x)> x = x;
}
```

返回类型

在C++11中，你可以跳过lambda的尾置返回类型，编译器会帮你推断。

起初，返回类型推断仅限于只有一个return语句的lambda，但由于更实用的版本实现起来没有问题，这个限制很快就被放宽了。参见C++标准核心语言缺陷报告与接受的问题。

所以从C++11开始，只要你的return语句都是同一个类型的，编译器就能推断返回类型。

如果所有return语句返回一个表达式并且在左值-右值转换（7.1 [conv.lval]）、数组-指针转换（7.2 [conv.array]），以及函数-指针转换（7.3 [conv.func]）后返回类型相同，（返回类型就是）这个相同的类型。

```
auto baz = [] () {
    int x = 10;
    if ( x < 20)
        return x * 1.1;
    else
        return x * 2.1;
};
```

(在线运行)

这个lambda中有两个return语句，但它们都返回double，所以编译器能推断类型。在C++14中lambda的返回类型更新为适用于普通函数的auto类型推断规则。

IIFE——立即调用函数表达式

在之前的例子中，我总是先定义一个lambda，然后用闭包对象调用它。但你也可以立即调用它：

```
int x = 1, y = 1;
[&]() { ++x; ++y; }(); // <-- 调用()
std::cout << x << " " << y << std::endl;
```

这样的表达式可用于初始化一个复杂的const对象：

```
const auto val = []() { /* 几行代码... */ }();
```

我在这篇博客里面写了更多相关内容：利用IIFE进行复杂初始化。

转换为函数指针

如果一个lambda没有捕获，则：

没有捕获的lambda表达式的闭包类型有一个public的、非virtual的、隐式的向函数指针的const转换函数，此函数指针与闭包类型的函数调用运算符有相同参数与返回类型。这种转换返回的值应该是一个函数的地址，当调用时与调用闭包类型的函数调用运算符有相同的效果。

换言之，你可以将没有捕获的lambda转换为函数指针，比如：

转换为函数指针

```
#include <iostream>

void callWith10(void(* bar)(int))
{
    bar(10);
}

int main()
{
    struct
    {
        using f_ptr = void(*)(int);
        void operator()(int s) const { return call(s); }
        operator f_ptr() const { return &call; }

    private:
        static void call(int s) { std::cout << s << std::endl; };
    } baz;

    callWith10(baz);
    callWith10([=](int x) { std::cout << x << std::endl; });
}
```

小结

在这一章中，你学到了如何创建与使用lambda表达式。我介绍了句法、捕获子句、lambda的类型，等等。

lambda表达式是现代C++最值得注意的标志之一。在更多的使用案例中，开发者们发现了改进lambda的可能性。这就是为什么你现在可以看向下一章，了解委员会在C++14中加入的更新。

3. C++14中的LAMBDA

C++为lambda表达式加入了两个重要的改进：

- 带有初始化的捕获
- 泛型lambda

此外，标准还更新了一些规则，比如：

- lambda的默认参数
- auto作为返回类型

这些特性可以解决C++11中存在的部分问题。

你可以在N4140和[expr.prim.lambda]中阅读特性。

LAMBDA的默认参数

在C++14中你可以在闭包函数调用中使用默认参数。这是个小特性，但让lambda更像普通函数。

带有默认参数的lambda

```
#include <iostream>

int main() {
    auto lam = [](int x = 10) { std::cout << x << '\n'; };
    lam();
    lam(100);
    return 0;
}
```

有趣的是GCC和Clang从C++11开始就支持这个特性了。

返回类型

在C++14中lambda的返回类型推断更新为与函数auto推断规则一致。

[expr.prim.lambda#4]：

lambda的返回类型是auto，如果提供尾置返回类型可以替换，或按照[dcl.spec.auto]由return语句推导。

如果你有多个return语句，它们必须推断出相同类型：

```
auto foo = [] (int x) {
    if (x < 0)
        return x * 1.1f; // float!
    else
        return x * 2.1; // double!
};
```

上面的代码不能通过编译，因为第一个return语句返回float，而第二个推断出double。

另一个与返回类型相关的重要概念是我们可以不再使用std::function来返回lambda！编译器会推导出正确的闭包类型：

```
auto CreateMulLambda(int x) {
    return [x](int param) { return x * param; };
}
auto lam = CreateMulLambda(10);
```

带初始化的捕获

来看更大的更新！

在lambda表达式中你可以捕获变量：编译器扩展捕获句法，在闭包类型中创建成员变量。现在，在C++14中，你可以创建新的成员变量，在捕获子句中初始化它们。然后你可以在lambda中使用这些变量。例如：

简单的带初始化的捕获

```
int main() {
    int x = 10;
    int y = 11;
    auto foo = [z = x+y]() { std::cout << z << '\n'; };
    foo();
}
```

在上面的例子中，编译器会生成一个新的成员变量，用x+y初始化它。所以从概念上讲，它解析为：

```
struct _unnamedLambda {
    void operator()() const {
        std::cout << z << '\n';
    }
    int z;
} someInstance;
```

当lambda表达式被求值时，z将被用x+y直接初始化。

这个特性能解决一些问题，比如仅可移动类型。我们来回看这个问题。

移动

在之前的C++11中，你不能值捕获一个std::unique_ptr。而现在，我们可以将对象移动到闭包类型的成员中：

捕获一个仅可移动对象

```
#include <memory>
int main(){
    std::unique_ptr<int> p(new int{10});
    auto foo = [x=10] () mutable { ++x; };
    auto bar = [ptr=std::move(p)] {};
    auto baz = [p=std::move(p)] {};
}
```

多亏了初始化，你才能给std::unique_ptr赋以合适的值。

优化

另一个想法是把捕获初始化作为一种优化技术。与其在每次调用lambda的时候计算一些值，我们可以仅在初始化中计算一次：

为lambda创建字符串

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <memory>
#include <iostream>
#include <functional>

struct Baz
{
    auto foo()
    {
        return [s=s] { std::cout << s << std::endl; };
    }

    std::string s;
};

int main()
{
   auto f1 = Baz{"ala"}.foo();
   auto f2 = Baz{"ula"}.foo();
   f1();
   f2();
}
```

上面你的代码展示了两次std::find_if调用。第一次我们没有捕获任何东西，仅仅将输入的值与"foo"s+"bar"s作比较。每次lambda被调用时都会创建一个临时变量用于存储两个字符串的和（连接）。第二次std::find_if调用使用了优化：我们创建一个捕获变量p，计算一次两字符串的和，然后我们在lambda函数体中安全地使用它。

捕获成员变量

初始化可以捕获成员变量。我们可以捕获成员变量的拷贝，无需担心悬空引用问题。例如：

捕获成员变量

```
```

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <memory>
#include <iostream>
#include <functional>

struct Baz
{
    auto foo()
    {
        return [s=s] { std::cout << s << std::endl; };
    }

    std::string s;
};

int main()
{
   auto f1 = Baz{"ala"}.foo();
   auto f2 = Baz{"ula"}.foo();
   f1();
   f2();
}
```

在foo()中通过拷贝进闭包类型捕获了一个成员变量。此外，我们还对整个方法的返回类型推导用了auto（在之前的C++11中我们可以用std::function）。

泛型LAMBDA

lambda的另一个重要改进是泛型lambda。从C++14开始你可以写：

```
auto foo = [](auto x) { std::cout << x << '\n'; };
foo(10);
foo(10.1234);
foo("hello world");
```

注意lambda的一个参数是auto x。这等价于在闭包类型中使用模板声明函数调用运算符：

```
struct {
    template<typename T>
    void operator()(T x) const {
        std::cout << x << '\n';
    }
} someInstance;
```

译者注

写模板的时候总会有一个问题：模板声明放在哪一层？在泛型lambda的问题中，是写模板类还是写类的模板方法？

由于lambda表达式返回一个闭包类型的对象，它必须是一个确定的类型，然而此时并不知道将来会以什么参数调用它，甚至可以用不同类型参数调用，因此泛型lambda的闭包类型一定是一个类，其中含有模板方法，而不是模板类的一系列实例。

在泛型lambda 中你不仅可以用auto x，也能像其他auto变量一样添加修饰符。

当类型推断困难的时候，泛型lambda非常有用，例如：

正确的std::map迭代类型

```
std::map<std::string, int> numbers {
    { "one", 1 }, {"two", 2 }, { "three", 3 }
};
// 每次函数入口都是std::pair<const std::string, int>的拷贝！
std::for_each(std::begin(numbers), std::end(numbers),
    [](const std::pair<std::string, int>& entry) {
        std::cout << entry.first << " = " << entry.second << '\n';
    }
);
```

我有犯声明错误吗？enrty是否具有正确的类型？

恐怕没有吧，因为std::map的value_type是std::pair<const Key, T>。所以我的代码会执行额外的字符串拷贝。

这个问题可以由auto解决：

```
std::for_each(std::begin(numbers), std::end(numbers),
    [](auto& entry) {
        std::cout << entry.first << " = " << entry.second << '\n';
    }
);
```

BONUS——利用LAMBDA放宽限制

目前，把重载函数传入标准库算法（或任何需要可调用对象的东西）是不行的：

调用重载函数

```
// 两个重载：
void foo(int) {}
void foo(float) {}
int main() {
    std::vector<int> vi;
    std::for_each(vi.begin(), vi.end(), foo);
}
```

在GCC 9（主线版本）中我们得到以下错误：

error: no matching function for call to

for_each(std::vector::iterator, std::vector::iterator,

)

 std::for_each(vi.begin(), vi.end(), foo);

^^^^^

然而，一个技巧是我们可以用lambda，然后调用所需要的重载函数。一个基本的形式是，对于简单的值类型，对于我们的两个函数，可以写这样的代码：

```
std::for_each(vi.begin(), vi.end(), [](auto x) { return foo(x); });
```

在最通用的形式中，我们需要多打一点字

```
#define LIFT(foo) \
  [](auto&&... x) \
    noexcept(noexcept(foo(std::forward<decltype(x)>(x)...))) \
   -> decltype(foo(std::forward<decltype(x)>(x)...)) \
  { return foo(std::forward<decltype(x)>(x)...); }
```

好复杂的代码啊……不是吗？:) 我们来试着解释它：

我们创建了一个泛型lambda，然后转发所有得到的参数。为了正确地定义，我们需要指明noexcept和返回类型。这就是为什么我们必须复制调用代码——为了得到正确的类型。

这个LIFT宏可以在任何支持C++14的编译器中工作。

小结

正如本章所述，C++14带来了一些lambda表达式的关键改进。自C++14开始你可以在lambda作用域内定义新的变量，也可以在模板代码中有效地使用它们。在下一章中我们将进入带来更多更新的C++17！

4. C++17中的LAMBDA

标准（出版前的草案）N4659以及lambda一节：[expr.prim.lambda]。

C++17为lambda表达式加入了两个重要的改进：

- constexpr lambda
- 捕获*this

这些特性对你来说意味着什么？一起来看吧。

CONSTEXPR LAMBDA表达式

自C++17开始，如果可行，标准将lambda类型的operator()隐式地定义为constexpr。摘自expr.prim.lambda #4：

函数调用运算符是constexpr函数，当对应的lambda表达式的参数声明字句后面有constexpr，或它满足constexpr函数的要求。

例如：

```
constexpr auto Square = [] (int n) { return n*n; }; // 隐式constexpr
static_assert(Square(2) == 4);
```

回忆一下，C++17中constexpr函数有以下规则：

- 不能是***virtual***；
- 返回类型是字面值类型；
- 参数都是字面值类型；
- 函数体为***=delete***，***=default***，或不含有以下内容的复合表达式
    - *asm*定义、

- *goto*语句、
- 标识符标签、
- *try*语句块，或
- 非字面值类型，或静态或线程存储期限，或没有初始化的变量的定义

来看个更实际的例子：

constexpr lambda

```
#include <array>

template<typename Range, typename Func, typename T>
constexpr T SimpleAccumulate(const Range& range, Func func, T init) {
    for (auto &&obj : range) { // begin/end are constexpr
        init += func(obj);
    }
    return init;
}

int main() {
    constexpr std::array arr{ 1, 2, 3 }; // clas deduction...

    // with constexpr lambda
    static_assert(SimpleAccumulate(arr, [](int i) {
            return i * i;
        }, 0) == 14);

    return arr[0];
}
```

这段代码写了一个constexpr lambda，然后把它传给简单的算法SimpleAccumulate。这个算法还用到了一些C++17元素：除了std::array可以constexpr外，std::begin和std::end（在基于范围的循环中使用）也可以成为constexpr，使整段代码都可以在编译器执行。

当然，还有更多。你还可以捕获变量（假设它们都是常量表达式）：

constexpr lambda，捕获

```
constexpr int add(int const& t, int const& u) {
    return t + u;
}
int main() {
    constexpr int x = 0;
    constexpr auto lam = [x](int n) { return add(x, n); };
    static_assert(lam(10) == 10);
}
```

有一个有趣的情况，就是当你不“传”你捕获的参数时，像这样：

```
constexpr int x = 0;
constexpr auto lam = [x](int n) { return n + x };
```

在这种情况下，在Clang，我们可能得到以下警告：

warning: lambda capture 'x' is not required to be captured for this use

这可能是因为每处使用的x都可以被就地替换（除非你把它传走或取地址）。

但请告诉我你是否知道这种行为的官方规则。我只找到（cppreference）（我在草案中没找到）（译者注：expr.const#2.12有一些相关内容）：

lambda表达式可以不捕获就读取变量的值，如果它是非volatile整值或枚举类型且已被常量表达式初始化，或是constexpr且没有mutable成员。

迎接未来

在C++20中我们将有constexpr标准算法，甚至一些容器，所以在那样的环境中comstexpr lambda会变得很方便。你的运行期版本和constexpr（编译期）版本看起来会是一样的！

概括来讲：constexpr lambda允许你混合模板编程，并多半会缩短代码。

我们来看第二个自C++17起可用的重要特性：

捕获THIS

你还记得当我们想捕获类成员时候的问题吗？

我们默认地捕获了this（作为一个指针！），这就是为什么临时变量离开作用域以后我们会遇到麻烦……我们可以用带初始化的捕获来解决，就像我在C++14一章中描述的那样。

但现在，在C++17中我们有另一种方式。我们可以捕获*this的拷贝：

捕获*this

```
#include <iostream>
struct Baz {
    auto foo() {
        return [*this] { std::cout << s << std::endl; };
    }
    std::string s;
};

int main() {
    auto f1 = Baz{"ala"}.foo();
    auto f2 = Baz{"ula"}.foo();
    f1();
    f2();
}
```

通过初始化捕获需要的成员变量解决了临时变量可能导致的错误，但当我们想调用类型方法的时候不能这么做，例如：

捕获*this以调用方法

```
struct Baz {
    auto foo() {
        return [this] { print(); };
    }
    void print() const { std::cout << s << '\n'; }
    std::string s;
};
```

在C++14中一个更安全的方法是用初始化捕获*this：

```
auto foo() {
    return[self=*this] { self.print(); };
}
```

但C++17中有一个更清晰的写法：

```
auto foo() {
    return[*this] { print(); };
}
```

还有一点：当你在成员函数中写[=]时，this被隐式捕获了！

指南

好，我们应该捕获[this]或[*this]，为什么这个选择很重要呢？

在大多数情况下，当你在类作用域中工作时，[this]（或[&]）是很好的。没有多余的拷贝，在你的对象很大时这尤其必要。

当你真的想要拷贝，或lambda可能超出对象的作用域时，你可以考虑[*this]。

在异步或并行执行中，为了避免数据竞争，这更加重要。同时，在异步或多线程执行模式下，lambda可能超出对象作用域，this指针不再有效。

小结

在这一章中你看到了C++17将两个重要的元素结合起来：constexpr与lambda。现在你可以在constexpr上下文中使用lambda了！另外C++17标准还解决了捕获this的问题。

在下一章中，我们将一瞥C++20带来的未来。

5. C++20中的未来

我们来一瞥C++20带来的改变。

在这一章中你将了解到：

- C++20改变了什么
- 捕获this的新方法
- 模板lambda是什么

快速概览

在C++20中我们将有以下特性：

- 允许[=, this]作为lambda捕获（P0409R2），废弃通过[=]隐式捕获this（P0806）
- lambda初始化捕获中的包展开：...args = std::move(args){}（P0780）
- static、thread_local与lambda捕获中的结构化绑定（P1091）
- 模板lambda（与concept）（P0428R2）
- 简化隐式lambda捕获（P0588R1）
- 可默认构造与复制的无状态lambda（P0624R2）
- 不求值上下文中的lambda

新加入的特性大多“清理”了lambda的使用，并允许更高级的用法。比如用P1091你可以捕获结构化绑定。

捕获this得到澄清。在C++20中如果你在方法中捕获[=]，你会得到警告：（试试看）

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <memory>
#include <iostream>
#include <functional>

struct Baz
{
    std::function<void()> foo()
    {
        return [=] { std::cout << s << std::endl; };
    }

    std::string s;
};

int main()
{
   auto f1 = Baz{"ala"}.foo();
   auto f2 = Baz{"ula"}.foo();
   f1();
   f2();
}
```

GCC 9：

warning: implicit capture of 'this' via '[=]' is deprecated in C++20

之所以有这个警告是因为即使用[=]你也会捕获this指针。所以最好显式写出你想要什么：[=, this]或[=, *this]。

还有些与高级用法相关的改变，比如不求值上下文与无状态lambda可默认构造。

有了这两个改进，你可以写：

```
std::map<int, int, decltype([](int x, int y) { return x > y; })> map;
```

注意

C++20标准已经特性完整，所以我们不用再期待有什么关于lambda的新特性了。但即使已选出的元素也可能稍微改动，所以应该视上面的列表为进行中而不是过时。

我们来看看一个有趣的特性：模板lambda。

模板LAMBDA

C++14中我们有泛型lambda，声明为auto的参数是模板参数。

对于一个lambda：

```
[](auto x) { x; }
```

编译器生成一个对应以下模板方法的函数调用运算符：

```
template<typename T>
void operator(T x) { x; }
```

但是我们不能改变这个模板参数而使用"真正的"模板参数，而这在C++20中是可行的。比如，我们如何限制lambda只能接受某种类型的std::vector?

我们可以写一个泛型lambda:

```
auto foo = [](auto& vec) {
        std::cout<< std::size(vec) << '\n';
        std::cout<< vec.capacity() << '\n';
    };
```

但如果你用int实参调用它（像foo(10);），你会得到一些难以阅读的错误:

prog.cc: In instantiation of 'main()::<lambda(const auto:1&)> [with auto:1 = in\

t]':

prog.cc:16:11: required from here

prog.cc:11:30: error: no matching function for call to 'size(const int&)'

 11 | std::cout<< std::size(vec) << '\n';

在C++20中我们可以写： (在线编译)

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <memory>
#include <iostream>
#include <functional>

struct Baz
{
    std::function<void()> foo()
    {
        return [=] { std::cout << s << std::endl; };
    }

    std::string s;
};

int main()
{
   auto f1 = Baz{"ala"}.foo();
   auto f2 = Baz{"ula"}.foo();
   f1();
   f2();
}
```

以上lambda解析为一个模板函数调用运算符:

```
template<typename T>
void operator(std::vector<T> const& s) { ... }
```

模板参数出现在捕获子句[]之后。

如果你用int（foo(10);）调用它，你会得到一条可读的信息：

note: mismatched types 'const std::vector' and 'int'

在上面这个例子中，编译器可以就lambda接口不匹配给我们警告，而不是在函数体中。

另一个重要的地方是在泛型lambda中你只有变量而没有其模板类型。如果你要存取它，你必须用decltype(x)（对于参数为(auto x)的lambda），这让代码有些冗长复杂。例如（引用P0428中的代码）：

```
auto f = [](auto const& x) {
    using T = std::decay_t<decltype(x)>;
    T copy = x;
    T::static_function();
    using Iterator = typename T::iterator;
}
```

现在可以写成：

```
auto f = []<typename T>(T const& x) {
    T::static_function();
    T copy = x;
    using Iterator = typename T::iterator;
}
```

小结

在这一章中，你看到了lambda的一些改变。lambda是现代C++的一个稳定特性，所以多数新元素与高级用法相关，比如不求值上下文或捕获结构化绑定。还有一些"扩展"，比如模板lambda。在大多数情况下泛型lambda就够用了，但对更高级的场景，你可能想要显式声明模板参数。

参考文献

- C++11 - [expr.prim.lambda]
- C++14 - [expr.prim.lambda]
- C++17 - [expr.prim.lambda]
- Lambda Expressions in C++ | Microsoft Docs
- Demystifying C++ lambdas - Sticky Bits - Powered by FeabhasSticky Bits – Powered by Feabhas
- The View from Aristeia: Lambdas vs. Closures
- Simon Brand - Passing overload sets to functions
- Jason Turner - C++ Weekly - Ep 128 - C++20's Template Syntax For Lambdas
- Jason Turner - C++ Weekly - Ep 41 - C++17's constexpr Lambda Support