转 C++11 lambda表达式与函数对象

置顶　2017年06月02日 18:02:41　　阅读数：4216

C++ lambda表达式与函数对象

lambda 表达式是 C++11 中引入的一项新技术，利用 lambda 表达式可以编写内嵌的匿名函数，用以替换独立函数或者函数对象，并且使代码更可读。但是从本质上来讲， lambda 表达式只是一种语法糖，因为所有其能完成的工作都可以用其它稍微复杂的代码来实现。但是它简便的语法却给 C++ 带来了深远的影响。如果从广义上说， lamdba 表达式产生的是函数对象。在类中，可以重载函数调用运算符 () ，此时类的对象可以将具有类似函数的行为，我们称这些对象为函数对象（Function Object）或者仿函数（Functor）。相比 lambda 表达式，函数对象有自己独特的优势。下面我们开始具体讲解这两项黑科技。

lambda表达式

我们先从简答的例子开始，我们定义一个可以输出字符串的 lambda 表达式，表达式一般都是从方括号 [] 开始，然后结束于花括号 {} ，花括号里面就像定义函数那样，包含了 lamdba 表达式体：

```
// 定义简单的lambda表达式
auto basicLambda = [] { cout << "Hello, world!" << endl; };
// 调用
basicLambda();   // 输出：Hello, world!
```

上面是最简单的 lambda 表达式，没有参数。如果需要参数，那么就要像函数那样，放在圆括号里面，如果有返回值，返回类型要放在 -> 后面，即拖尾返回类型，当然你也可以忽略返回类型， lambda 会帮你自动推断出返回类型：

```
// 指明返回类型
auto add = [](int a, int b) -> int { return a + b; };
// 自动推断返回类型
auto multiply = [](int a, int b) { return a * b; };

int sum = add(2, 5);   // 输出：7
int product = multiply(2, 5);  // 输出：10
```

大家可能会想 lambda 表达式最前面的方括号的意义何在？其实这是 lambda 表达式一个很要的功能，就是闭包。这里我们先讲一下 lambda 表达式的大致原理：每当你定义一个 lambda 表达式后，编译器会自动生成一个匿名类（这个类当然重载了 () 运算符），我们称为闭包类型（closure type）。那么在运行时，这个 lambda 表达式就会返回一个匿名的闭包实例，其实一个右值。所以，我们上面的 lambda 表达式的结果就是一个个闭包。闭包的一个强大之处是其可以通过传值或者引用的方式捕捉其封装作用域内的变量，前面的方括号就是用来定义捕捉模式以及变量，我们又将其称为 lambda 捕捉块。看下面的例子：

```
int main()
{
    int x = 10;

    auto add_x = [x](int a) { return a + x; };  // 复制捕捉x
    auto multiply_x = [&x](int a) { return a * x; };  // 引用捕捉x
```

```
    cout << add_x(10) << " " << multiply_x(10) << endl;
    // 输出：20 100
    return 0;
}
```

当 lambda 捕捉块为空时，表示没有捕捉任何变量。但是上面的 add_x 是以复制的形式捕捉变量 x ，而 multiply 是以引用的方式捕捉 x 。前面讲过， lambda 表达式是产生一个闭包类，那么捕捉是回事？对于复制传值捕捉方式，类中会相应添加对应类型的非静态数据成员。在运行时，会用复制的值初始化这些成员变量，从而生成闭包。前面说过，闭包类也实现了函数调用运算符的重载，一般情况是：

```
class ClosureType
{
public:
    // ...
    ReturnType operator(params) const { body };
}
```

这意味着 lambda 表达式无法修改通过复制形式捕捉的变量，因为函数调用运算符的重载方法是 const 属性的。有时候，你想改动传值方式捕获的值，那么就要使用 mutable ，例子如下：

```
int main()
{
    int x = 10;

    auto add_x = [x](int a) mutable { x *= 2; return a + x; };  // 复制捕捉x

    cout << add_x(10) << endl; // 输出 30
    return 0;
}
```

这是为什么呢？因为你一旦将 lambda 表达式标记为 mutable ，那么实现的了函数调用运算符是非const属性的：

```
class ClosureType
{
public:
    // ...
    ReturnType operator(params) { body };
}
```

对于引用捕获方式，无论是否标记 mutable ，都可以在 lambda 表达式中修改捕获的值。至于闭包类中是否有对应成员， C++ 标准中给出的答案是：不清楚的，看来与具体实现有关。既然说到了深处，还有一点要注意： lambda 表达式是不能被赋值的：

```
auto a = [] { cout << "A" << endl; };
auto b = [] { cout << "B" << endl; };

a = b;   // 非法，lambda无法赋值
auto c = a;   // 合法，生成一个副本
```

你可能会想 a 与 b 对应的函数类型是一致的（编译器也显示是相同类型：lambda [] void () -> void），为什么不能相互赋值呢？因为禁用了赋值操作符：

```
ClosureType& operator=(const ClosureType&) = delete;
```

但是没有禁用复制构造函数，所以你仍然可以用一个 lambda 表达式去初始化另外一个 lambda 表达式而产生副本。并且 lambda 表达式也可以赋值给相对应的函数指针，这也使得你完全可以把 lambda 表达式看成对应函数类型的指针。

闲话少说，归入正题，捕获的方式可以是引用也可以是复制，但是具体说来会有以下几种情况来捕获其所在作用域中的变量：

- []：默认不捕获任何变量；
- [=]：默认以值捕获所有变量；
- [&]：默认以引用捕获所有变量；
- [x]：仅以值捕获x，其它变量不捕获；
- [&x]：仅以引用捕获x，其它变量不捕获；
- [=, &x]：默认以值捕获所有变量，但是x是例外，通过引用捕获；
- [&, x]：默认以引用捕获所有变量，但是x是例外，通过值捕获；
- [this]：通过引用捕获当前对象（其实是复制指针）；
- [*this]：通过传值方式捕获当前对象；

在上面的捕获方式中，注意最好不要使用 [=] 和 [&] 默认捕获所有变量。首先说默认引用捕获所有变量，你有很大可能会出现悬挂引用（Dangling references），因为引用捕获不会延长引用的变量的声明周期：

```
std::function<int(int)> add_x(int x)
{
    return [&](int a) { return x + a; };
}
```

因为参数 x 仅是一个临时变量，函数调用后就被销毁，但是返回的 lambda 表达式却引用了该变量，但调用这个表达式时，引用的是一个垃圾值，所以会产生没有意义的结果。你可能会想，可以通过传值的方式来解决上面的问题：

```
std::function<int(int)> add_x(int x)
{
    return [=](int a) { return x + a; };
}
```

是的，使用默认传值方式可以避免悬挂引用问题。但是采用默认值捕获所有变量仍然有风险，看下面的例子：

```
class Filter
{
public:
    Filter(int divisorVal):
        divisor{divisorVal}
```

```
    {}

    std::function<bool(int)> getFilter()
    {
        return [=](int value) {return value % divisor == 0; };
    }

private:
    int divisor;
};
```

这个类中有一个成员方法，可以返回一个 lambda 表达式，这个表达式使用了类的数据成员 divisor 。而且采用默认值方式捕捉所有变量。你可能认为这个 lambda 表达式也捕捉了 divisor 的一份副本，但是实际上大错特错。问题出现在哪里呢？因为数据成员 divisor 对 lambda 表达式并不可见，你可以用下面的代码验证：

```
// 类的方法，下面无法编译，因为divisor并不在lambda捕捉的范围
std::function<bool(int)> getFilter()
{
    return [divisor](int value) {return value % divisor == 0; };
}
```

那么原来的代码为什么能够捕捉到呢？仔细想想，原来每个非静态方法都有一个 this 指针变量，利用 this 指针，你可以接近任何成员变量，所以 lambda 表达式实际上捕捉的是 this 指针的副本，所以原来的代码等价于：

```
std::function<bool(int)> getFilter()
{
    return [this](int value) {return value % this->divisor == 0; };
}
```

尽管还是以值方式捕获，但是捕获的是指针，其实相当于以引用的方式捕获了当前类对象，所以 lambda 表达式的闭包与一个类对象绑定在一起了，这也很危险，因为你仍然有可能在类对象析构后使用这个 lambda 表达式，那么类似“悬挂引用”的问题也会产生。所以，采用默认值捕捉所有变量仍然是不安全的，主要是由于指针变量的复制，实际上还是按引用传值。

通过前面的例子，你还可以看到 lambda 表达式可以作为返回值。我们知道 lambda 表达式可以赋值给对应类型的函数指针。但是使用函数指针貌似并不是那么方便。所以 STL 定义在 <functional> 头文件提供了一个多态的函数对象封装 std::function ，其类似于函数指针。它可以绑定任何类函数对象，只要参数与返回类型相同。如下面的返回一个bool且接收两个int的函数包装器：

```
std::function<bool(int, int)> wrapper = [](int x, int y) { return x < y; };
```

而 lambda 表达式一个更重要的应用是其可以用于函数的参数，通过这种方式可以实现回调函数。其实，最常用的是在 STL 算法中，比如你要统计一个数组中满足特定条件的元素数量，通过 lambda 表达式给出条件，传递给 count_if 函数：

```
int value = 3;
vector<int> v {1, 3, 5, 2, 6, 10};

int count = std::count_if(v.beigin(), v.end(), [value](int x) { return x > value; });
```

再比如你想生成斐波那契数列，然后保存在数组中，此时你可以使用 generate 函数，并辅助 lambda 表达式：

```
vector<int> v(10);
int a = 0;
int b = 1;
std::generate(v.begin(), v.end(), [&a, &b] { int value = b; b = b + a; a = value; return value; });
// 此时v {1, 1, 2, 3, 5, 8, 13, 21, 34, 55}
```

此外， lambda 表达式还用于对象的排序准则：

```
class Person
{
public:
    Person(const string& first, const string& last):
        firstName{first}, lastName{last}
    {}

    Person() = default;

    string first() const { return firstName; }
    string last() const { return lastName; }
private:
    string firstName;
    string lastName;
};

int main()
{
    vector<Person> vp;
    // ... 添加Person信息

    // 按照姓名排序
    std::sort(vp.begin(), vp.end(), [](const Person& p1, const Person& p2)
    { return p1.last() < p2.last() || (p1.last() == p2.last() && p1.first() < p2.first()); });
        // ...
    return 0;
}
```

总之，对于大部分 STL 算法，可以非常灵活地搭配 lambda 表达式来实现想要的效果。

前面讲完了 lambda 表达式的基本使用，最后给出 lambda 表达式的完整语法：

```
// 完整语法
[ capture-list ] ( params ) mutable(optional) constexpr(optional)(c++17) exception attribute -> ret { body }

// 可选的简化语法
[ capture-list ] ( params ) -> ret { body }
[ capture-list ] ( params ) { body }
[ capture-list ] { body }
```

第一个是完整的语法，后面3个是可选的语法。这意味着 lambda 表达式相当灵活，但是照样有一定的限制，比如你使用了拖尾返回类型，那么就不能省略参数列表，尽管其可能是空的。针对完整的语法，我们对各个部分做一个说明：

- **capture-list**：捕捉列表，这个不用多说，前面已经讲过，记住它不能省略；
- **params**：参数列表，可以省略（但是后面必须紧跟函数体）；
- **mutable**：可选，将 `lambda` 表达式标记为 `mutable` 后，函数体就可以修改传值方式捕获的变量；
- **constexpr**：可选，C++17，可以指定 `lambda` 表达式是一个常量函数；
- **exception**：可选，指定 `lambda` 表达式可以抛出的异常；
- **attribute**：可选，指定 `lambda` 表达式的特性；
- **ret**：可选，返回值类型；
- **body**：函数执行体。

如果想了解更多，可以参考 cppreference lambda。

lambda新特性

在 `C++14` 中， `lambda` 又得到了增强，一个是泛型 `lambda` 表达式，一个是 `lambda` 可以捕捉表达式。这里我们对这两项新特点进行简单介绍。

lambda捕捉表达式

前面讲过， `lambda` 表达式可以按复制或者引用捕获在其作用域范围内的变量。而有时候，我们希望捕捉不在其作用域范围内的变量，而且最重要的是我们希望捕捉右值。所以 `C++14` 中引入了表达式捕捉，其允许用任何类型的表达式初始化捕捉的变量。看下面的例子：

```
// 利用表达式捕获，可以更灵活地处理作用域内的变量
int x = 4;
auto y = [&r = x, x = x + 1] { r += 2; return x * x; }();
// 此时 x 更新为6，y 为25

// 直接用字面值初始化变量
auto z = [str = "string"]{ return str; }();
// 此时z是const char* 类型，存储字符串 string
```

可以看到捕捉表达式扩大了 `lambda` 表达式的捕捉能力，有时候你可以用 `std::move` 初始化变量。这对不能复制只能移动的对象很重要，比如 `std::unique_ptr` ，因为其不支持复制操作，你无法以值方式捕捉到它。但是利用 `lambda` 捕捉表达式，可以通过移动来捕捉它：

```
auto myPi = std::make_unique<double>(3.1415);

auto circle_area = [pi = std::move(myPi)](double r) { return *pi * r * r; };
cout << circle_area(1.0) << endl; // 3.1415
```

其实用表达式初始化捕捉变量，与使用 `auto` 声明一个变量的机理是类似的。

泛型lambda表达式

从 `C++14` 开始， `lambda` 表达式支持泛型：其参数可以使用自动推断类型的功能，而不需要显示地声明具体类型。这就如同函数模板一样，参数要使用类型自动推断功能，只需要将其类型指定为 `auto` ，类型推断规则与函数模板一

样。这里给出一个简单例子：

```
auto add = [](auto x, auto y) { return x + y; };

int x = add(2, 3);   // 5
double y = add(2.5, 3.5);  // 6.0
```

函数对象

函数对象是一个广泛的概念，因为所有具有函数行为的对象都可以称为函数对象。这是一个高级抽象，我们不关心对象到底是什么，只要其具有函数行为。所谓的函数行为是指的是可以使用 () 调用并传递参数：

```
function(arg1, arg2, ...);   // 函数调用
```

这样来说， lambda 表达式也是一个函数对象。但是这里我们所讲的是一种特殊的函数对象，这种函数对象实际上是一个类的实例，只不过这个类实现了函数调用符 () ：

```
class X
{
public:
    // 定义函数调用符
    ReturnType operator()(params) const;

    // ...
};
```

这样，我们可以使用这个类的对象，并把它当做函数来使用：

```
X f;
// ...
f(arg1, arg2); // 等价于 f.operator()(arg1, arg2);
```

还是例子说话，下面我们定义一个打印一个整数的函数对象：

```
// T需要支持输出流运算符
template <typename T>
class Print
{
public:
    void operator()(T elem) const
    {
        cout << elem << ' ' ;
    }
};


int main()
{
    vector<int> v(10);
    int init = 0;
```

```
    std::generate(v.begin(), v.end(), [&init] { return init++; });

    // 使用for_each输出各个元素（送入一个Print实例）
    std::for_each(v.begin(), v.end(), Print<int>{});
    // 利用lambda表达式：std::for_each(v.begin(), v.end(), [](int x){ cout << x << ' ';});
    // 输出：0, 1, 2, 3, 4, 5, 6, 7, 8, 9
    return 0;
}
```

可以看到 `Print<int>` 的实例可以传入 `std::for_each` ，其表现可以像函数一样，因此我们称这个实例为函数对象。大家可能会想， `for_each` 为什么可以既接收 `lambda` 表达式，也可以接收函数对象，其实 `STL` 算法是泛型实现的，其不关心接收的对象到底是什么类型，但是必须要支持函数调用运算：

```
// for_each的类似实现
namespace std
{
    template <typename Iterator, typename Operation>
    Operation for_each(Iterator act, Iterator end, Operation op)
    {
        while (act != end)
        {
            op(*act);
            ++act;
        }
        return op;
    }
}
```

泛型提供了高级抽象，不论是 `lambda` 表达式、函数对象，还是函数指针，都可以传入 `for_each` 算法中。

本质上，函数对象是类对象，这也使得函数对象相比普通函数有自己的独特优势：

- **函数对象带有状态**：函数对象相对于普通函数是“智能函数”，这就如同智能指针相较于传统指针。因为函数对象除了提供函数调用符方法，还可以拥有其他方法和数据成员。所以函数对象有状态。即使同一个类实例化的不同的函数对象其状态也不相同，这是普通函数所无法做到的。而且函数对象是可以在运行时创建。
- **每个函数对象有自己的类型**：对于普通函数来说，只要签名一致，其类型就是相同的。但是这并不适用于函数对象，因为函数对象的类型是其类的类型。这样，函数对象有自己的类型，这意味着函数对象可以用于模板参数，这对泛型编程有很大提升。
- **函数对象一般快于普通函数**：因为函数对象一般用于模板参数，模板一般会在编译时会做一些优化。

这里我们看一个可以拥有状态的函数对象，其用于生成连续序列：

```
class IntSequence
{
public:
    IntSequence(int initVal) : value{ initVal } {}

    int operator()() { return ++value; }
private:
    int value;
```

```
};


int main()
{
    vector<int> v(10);
    std::generate(v.begin(), v.end(), IntSequence{ 0 });
    /*  lambda实现同样效果
        int init = 0;
        std::generate(v.begin(), v.end(), [&init] { return ++init; });
    */
    std::for_each(v.begin(), v.end(), [](int x) { cout << x << ' '; });
    //输出: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

    return 0;
}
```

可以看到，函数对象可以拥有一个私有数据成员，每次调用时递增，从而产生连续序列。同样地，你可以用 `lambda` 表达式实现类似的效果，但是必须采用引用捕捉方式。但是，函数对象可以实现更复杂的功能，而用 `lambda` 表达式则需要复杂的引用捕捉。考虑一个可以计算均值的函数对象：

```
class MeanValue
{
public:
    MeanValue(): num{0}, sum{0} {}

    void operator()(int e)
    {
        ++num;
        sum += num;
    }

    double value()
    { return static_cast<double>(sum) / static_cast<double>(num); }
private:
    int num;
    int sum;
};


int main()
{
    vector<int> v{ 1, 3, 5, 7 };
    MeanValue mv = std::for_each(v.begin(), v.end(), MeanValue{});
    cout << mv.value() << endl;  // output: 2.5

    return 0;
}
```

可以看到 `MeanValue` 对象中保存了两个私有变量 `num` 和 `sum` 分别记录数量与总和，最后可以通过两者计算出均值。 `lambda` 表达式也可以利用引用捕捉实现类似功能，但是会有点繁琐。这也算是函数对象独特的优势。

头文件 `<functional>` 中预定义了一些函数对象，如算术函数对象，比较函数对象，逻辑运算函数对象及按位函数对象，我们可以在需要时使用它们。比如 `less<>` 是 `STL` 排序算法中的默认比较函数对象，所以默认的排序结果是升序，但是如果你想降序排列，你可以使用 `greater<>` 函数对象：

```
vector<int> v{3, 4, 2, 9, 5};
// 升序排序
std::sort(v.begin(), v.end());  // output: 2, 3, 4, 5, 9
// 降序排列
std::sort(v.begin(), v.end(), std::greater<int>{}); // output: 9, 5, 4, 3, 2
```

更多有关函数对象的信息大家可以参考这里。

函数适配器

从设计模式来说，函数适配器是一种特殊的函数对象，是将函数对象与其它函数对象，或者特定的值，或者特定的函数相互组合的产物。由于组合特性，函数适配器可以满足特定的需求，头文件 `<functional>` 定义了几种函数适配器：

- std::bind(op, args...)：将函数对象op的参数绑定到特定的值args
- std::mem_fn(op)：将类的成员函数转化为一个函数对象
- std::not1(op), std::not2(op)：一元取反器和二元取反器

绑定器（binder）

绑定器 `std::bind` 是最常用的函数适配器，它可以将函数对象的参数绑定至特定的值。对于没有绑定的参数可以使用 `std::placeholers::_1`, `std::placeholers::_2` 等标记。我们从简单的例子开始，比如你想得到一个减去固定树的函数对象：

```
auto minus10 = std::bind(std::minus<int>{}, std::placeholders::_1, 10);
cout << minus10(20) << endl;  // 输出10
```

有时候你可以利用绑定器重新排列参数的顺序，下面的绑定器交换两个参数的位置：

```
// 逆转参数顺序
auto vminus = std::bind(std::minus<int>{}, std::placeholders::_2, std::placeholders::_1);
cout << vminus(20, 10) << endl;  // 输出-10
```

绑定器还可以互相嵌套，从而实现函数对象的组合：

```
// 定义一个接收一个参数，然后将参数加10再乘以2的函数对象
auto plus10times2 = std::bind(std::multiplies<int>{},
        std::bind(std::plus<int>{}, std::placeholders::_1, 10), 2);
cout << plus10times2(4) << endl; // 输出：  28

// 定义3次方函数对象
auto pow3 = std::bind(std::multiplies<int>{},
        std::bind(std::multiplies<int>{}, std::placeholders::_1, std::placeholders::_1),
        std::placeholders::_1);
cout << pow3(3) << endl;  // 输出：27
```

利用不同函数对象组合，函数适配器可以调用全局函数，下面的例子是不区分大小写来判断一个字符串是否包含一个特定的子串：

```
// 大写转换函数
char myToupper(char c)
{
    if (c >= 'a' && c <= 'z')
        return static_cast<char>(c - 'a' + 'A');
    return c;
}

int main()
{
    string s{ "Internationalization" };
    string sub{ "Nation" };

    auto pos = std::search(s.begin(), s.end(), sub.begin(), sub.end(),
                        std::bind(std::equal_to<char>{},
                            std::bind(myToupper, std::placeholders::_1),
                            std::bind(myToupper, std::placeholders::_2)));
    if (pos != s.end())
    {
        cout << sub << " is part of " << s << endl;
    }
    // 输出：Nation is part of Internationalization
    return 0;
}
```

注意绑定器默认是以传值方绑定参数，如果需要引用绑定值，那么要使用 `std::ref` 和 `std::cref` 函数，分别代表普通引用和const引用绑定参数：

```
void f(int& n1, int& n2, const int& n3)
{
    cout << "In function: " << n1 << ' ' << n2 << ' ' << n3 << '\n';
    ++n1;
    ++n2;
    // ++n3;  //无法编译
}

int main()
{
    int n1 = 1, n2 = 2, n3 = 3;
    auto boundf = std::bind(f, n1, std::ref(n2), std::cref(n3));
    n1 = 10;
    n2 = 11;
    n3 = 12;
    cout << "Before function: " << n1 << ' ' << n2 << ' ' << n3 << '\n';
    boundf();
    cout << "After function: " << n1 << ' ' << n2 << ' ' << n3 << '\n';
    //  Before function : 10 11 12
    //    In function : 1 11 12
    //    After function : 10 12 12

    return 0;
}
```

可以看到， `n1` 是以默认方式绑定到函数 `f` 上，故仅是一个副本，不会影响原来的 `n1` 变量，但是 `n2` 是以引用绑定的，绑定到 `f` 的参数与原来的 `n2` 相互影响， `n3` 是以const引用绑定的，函数 `f` 无法修改其值。

绑定器可以用于调用类中的成员函数：

```
class Person
{
public:
    Person(const string& n) : name{ n } {}
    void print() const { cout << name << endl; }
    void print2(const string& prefix) { cout << prefix << name << endl; }
private:
    string name;
};
int main()
{
    vector<Person> p{ Person{"Tick"}, Person{"Trick"} };
    // 调用成员函数print
    std::for_each(p.begin(), p.end(), std::bind(&Person::print, std::placeholders::_1));
    // 此处的std::placeholders::_1表示要调用的Person对象，所以相当于调用arg1.print()
    // 输出：Tick   Trick
    std::for_each(p.begin(), p.end(), std::bind(&Person::print2, std::placeholders::_1,
        "Person: "));
    // 此处的std::placeholders::_1表示要调用的Person对象，所以相当于调用arg1.print2("Person: ")
    // 输出：Person: Tick   Person: Trick

    return 0;
}
```

而且绑定器对虚函数也有效，你可以自己做一下测试。

前面说过， C++11 中 lambda 表达式无法实现移动捕捉变量，但是使用绑定器可以实现类似的功能：

```
vector<int> data{ 1, 2, 3, 4 };
auto func = std::bind([](const vector<int>& data) { cout << data.size() << endl; },
                          std::move(data));
func();  // 4
cout << data.size() << endl;  // 0
```

可以看到绑定器可以实现移动语义，这是因为对于左值参数，绑定对象是复制构造的，但是对右值参数，绑定对象是移动构造的。

std::mem_fn()适配器

当想调用成员函数时，你还可以使用 std::mem_fn 函数，此时你可以省略掉用于调用对象的占位符：

```
vector<Person> p{ Person{ "Tick" }, Person{ "Trick" } };
std::for_each(p.begin(), p.end(), std::mem_fn(&Person::print));
// 输出：Trick Trick
Person n{ "Bob" };
std::mem_fn(&Person::print2)(n, "Person: ");
// 输出：Person: Bob
```

所以，使用 std::men_fn 不需要绑定参数，可以更方便地调用成员函数。再看一个例子， std;:mem_fn 还可以调用成员变量：

```
class Foo
{
public:
    int data = 7;
    void display_greeting() { cout << "Hello, world.\n"; }
    void display_number(int i) { cout << "number: " << i << '\n'; }

};
int main()
{
    Foo f;
    // 调用成员函数
    std::mem_fn(&Foo::display_greeting)(f);  // Hello, world.
    std::mem_fn(&Foo::display_number)(f, 20);  // number: 20
    // 调用数据成员
    cout << std::mem_fn(&Foo::data)(f) << endl;  // 7

    return 0;
}
```

取反器 `std::not1` 与 `std::not2` 很简单，就是取函数对象的反结果，不过在 `C++17` 两者被弃用了，所以就不讲了。

References

[1] Marc Gregoire. Professional C++, Third Edition, 2016.
[2] cppreference
[3] 欧长坤(欧龙崎), 高速上手 C++ 11/14.
[4] Scott Meyers. Effective Modern C++, 2014.
[5] Nicolai M. Josuttis. The C++ Standard Library Second Edition, 2012.

文章标签： lambda 函数对象

个人分类： c++