第 3 章 语言运行期的强化 现代 C++ 教程: 高速上手 C++ 11/14/17/20

https://changkun.de/modern-cpp/zh-cn/03-runtime/index.html

None Thu Jul, 22 15:43

3.1 Lambda 表达式

Lambda 表达式是现代 C++ 中最重要的特性之一，而 Lambda 表达式，实际上就是提供了一个类似匿名函数的特性，而匿名函数则是在需要一个函数，但是又不想费力去命名一个函数的情况下去使用的。这样的场景其实有很多很多，所以匿名函数几乎是现代编程语言的标配。

基础

Lambda 表达式的基本语法如下：

```
[捕获列表](参数列表) mutable(可选) 异常属性 -> 返回类型 {
// 函数体
}
```

上面的语法规则除了 `[捕获列表]` 内的东西外，其他部分都很好理解，只是一般函数的函数名被略去，返回值使用了一个 `->` 的形式进行（我们在上一节前面的尾返回类型已经提到过这种写法了）。

所谓捕获列表，其实可以理解为参数的一种类型，Lambda 表达式内部函数体在默认情况下是不能够使用函数体外部的变量的，这时候捕获列表可以起到传递外部数据的作用。根据传递的行为，捕获列表也分为以下几种：

1. 值捕获

与参数传值类似，值捕获的前提是变量可以拷贝，不同之处则在于，被捕获的变量在 Lambda 表达式被创建时拷贝，而非调用时才拷贝：

```
void lambda_value_capture() {
    int value = 1;
    auto copy_value = [value] {
        return value;
    };
    value = 100;
    auto stored_value = copy_value();
    std::cout << 'stored_value = ' << stored_value << std::endl;


}
```

2. 引用捕获

与引用传参类似，引用捕获保存的是引用，值会发生变化。

```
void lambda_reference_capture() {
    int value = 1;
    auto copy_value = [&value] {
        return value;
    };
    value = 100;
    auto stored_value = copy_value();
    std::cout << 'stored_value = ' << stored_value << std::endl;


}
```

3. 隐式捕获

手动书写捕获列表有时候是非常复杂的，这种机械性的工作可以交给编译器来处理，这时候可以在捕获列表中写一个 `&` 或 `=` 向编译器声明采用引用捕获或者值捕获.

总结一下，捕获提供了 Lambda 表达式对外部值进行使用的功能，捕获列表的最常用的四种形式可以是:

- [] 空捕获列表
- [name1, name2, ...] 捕获一系列变量
- [&] 引用捕获, 让编译器自行推导引用列表
- [=] 值捕获, 让编译器自行推导值捕获列表

4. 表达式捕获

> 这部分内容需要了解后面马上要提到的右值引用以及智能指针

上面提到的值捕获、引用捕获都是已经在外层作用域声明的变量，因此这些捕获方式捕获的均为左值，而不能捕获右值。

C++14 给与了我们方便，允许捕获的成员用任意的表达式进行初始化，这就允许了右值的捕获， 被声明的捕获变量类型会根据表达式进行判断，判断方式与使用 `auto` 本质上是相同的:

```
#include <iostream>
#include <utility>

int main() {
    auto important = std::make_unique<int>(1);
    auto add = [v1 = 1, v2 = std::move(important)](int x, int y) -> int {
        return x+y+v1+(*v2);
    };
    std::cout << add(3,4) << std::endl;
    return 0;
}
```

在上面的代码中，important 是一个独占指针，是不能够被 '=' 值捕获到，这时候我们可以将其转移为右值，在表达式中初始化。

泛型 Lambda

上一节中我们提到了 `auto` 关键字不能够用在参数表里，这是因为这样的写法会与模板的功能产生冲突。但是 Lambda 表达式并不是普通函数，所以 Lambda 表达式并不能够模板化。这就为我们造成了一定程度上的麻烦：参数表不能够泛化，必须明确参数表类型。

幸运的是，这种麻烦只存在于 C++11 中，从 C++14 开始，Lambda 函数的形式参数可以使用 `auto` 关键字来产生意义上的泛型：

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

add(1, 2);
add(1.1, 2.2);
```

3.2 函数对象包装器

这部分内容虽然属于标准库的一部分，但是从本质上来看，它却增强了 C++ 语言运行时的能力，这部分内容也相当重要，所以放到这里来进行介绍。

`std::function`

Lambda 表达式的本质是一个和函数对象类型相似的类类型（称为闭包类型）的对象（称为闭包对象），当 Lambda 表达式的捕获列表为空时，闭包对象还能够转换为函数指针值进行传递，例如：

```
#include <iostream>

using foo = void(int);
void functional(foo f) {
    f(1);
}

int main() {
    auto f = [](int value) {
        std::cout << value << std::endl;
    };
    functional(f);
    f(1);
    return 0;
}
```

上面的代码给出了两种不同的调用形式，一种是将 Lambda 作为函数类型传递进行调用，而另一种则是直接调用 Lambda 表达式，在 C++11 中，统一了这些概念，将能够被调用的对象的类型，统一称之为可调用类型。而这种类型，便是通过 `std::function` 引入的。

C++11 `std::function` 是一种通用、多态的函数封装，它的实例可以对任何可以调用的目标实体进行存储、复制和调用操作，它也是对 C++ 中现有的可调用实体的一种类型安全的包裹（相对来说，函数指针的调用不是类型安全的），换句话说，就是函数的容器。当我们有了函数的容器之后便能够更加方便的将函数、函数指针作为对象进行处理。例如：

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

int foo(int para) {
    return para;
}

int main() {

    std::function<int(int)> func = foo;

    int important = 10;
    std::function<int(int)> func2 = [&](int value) -> int {
        return 1+value+important;
    };
    std::cout << func(10) << std::endl;
    std::cout << func2(10) << std::endl;
}
```

`std::bind` 和 `std::placeholder`

而 `std::bind` 则是用来绑定函数调用的参数的，它解决的需求是我们有时候可能并不一定能够一次性获得调用某个函数的全部参数，通过这个函数，我们可以将部分调用参数提前绑定到函数身上成为一个新的对象，然后在参数齐全后，完成调用。例如：

```
int foo(int a, int b, int c) {
    ;
}
int main() {

    auto bindFoo = std::bind(foo, std::placeholders::_1, 1,2);

    bindFoo(1);
}
```

提示：注意 `auto` 关键字的妙用。有时候我们可能不太熟悉一个函数的返回值类型，但是我们却可以通过 `auto` 的使用来规避这一问题的出现。

3.3 右值引用

右值引用是 C++11 引入的与 Lambda 表达式齐名的重要特性之一。它的引入解决了 C++ 中大量的历史遗留问题，消除了诸如 `std::vector`、`std::string` 之类的额外开销，也才使得函数对象容器 `std::function` 成为了可能。

左值、右值的纯右值、将亡值、右值

要弄明白右值引用到底是怎么一回事，必须要对左值和右值做一个明确的理解。

左值(lvalue, left value)，顾名思义就是赋值符号左边的值。准确来说，左值是表达式（不一定是赋值表达式）后依然存在的持久对象。

右值(rvalue, right value)，右边的值，是指表达式结束后就不再存在的临时对象。

而 C++11 中为了引入强大的右值引用，将右值的概念进行了进一步的划分，分为：纯右值、将亡值。

纯右值(prvalue, pure rvalue)，纯粹的右值，要么是纯粹的字面量，例如 `10`，`true`；要么是求值结果相当于字面量或匿名临时对象，例如 `1+2`。非引用返回的临时变量、运算表达式产生的临时变量、原始字面量、Lambda 表达式都属于纯右值。

需要注意的是，字符串字面量只有在类中才是右值，当其位于普通函数中是左值。例如：

```
class Foo {
        const char*&& right = 'this is a rvalue';
public:
        void bar() {
            right = 'still rvalue';
        }
};

int main() {
    const char* const &left = 'this is an lvalue';
}
```

将亡值(xvalue, expiring value)，是 C++11 为了引入右值引用而提出的概念（因此在传统 C++ 中， 纯右值和右值是同一个概念），也就是即将被销毁、却能够被移动的值。

将亡值可能稍有些难以理解，我们来看这样的代码：

```
std::vector<int> foo() {
    std::vector<int> temp = {1, 2, 3, 4};
    return temp;
}

std::vector<int> v = foo();
```

在这样的代码中，就传统的理解而言，函数 `foo` 的返回值 `temp` 在内部创建然后被赋值给 `v`， 然而 `v` 获得这个对象时，会将整个 `temp` 拷贝一份，然后把 `temp` 销毁，如果这个 `temp` 非常大， 这将造成大量额外的开销（这也就是传统 C++ 一直被诟病的问题）。在最后一行中， `v` 是左值、 `foo()` 返回的值就是右值（也是纯右值）。但是， `v` 可以被别的变量捕获到， 而 `foo()` 产生的那个返回值作为一个临时值，一旦被 `v` 复制后，将立即被销毁，无法获取、也不能修改。 而将亡值就定义了这样一种行为：临时的值能够被识别、同时又能够被移动。

在 C++11 之后，编译器为我们做了一些工作，此处的左值 `temp` 会被进行此隐式右值转换，等价于 `static_cast<std::vector<int> &&>(temp)` ，进而此处的 `v` 会将 `foo` 局部返回的值进行移动。 也就是后面我们将会提到的移动语义。

右值引用和左值引用

要拿到一个将亡值，就需要用到右值引用： `T &&` ，其中 `T` 是类型。 右值引用的声明让这个临时值的生命周期得以延长、只要变量还活着，那么将亡值将继续存活。

C++11 提供了 `std::move` 这个方法将左值参数无条件的转换为右值， 有了它我们就能够方便的获得一个右值临时对象，例如：

```
#include <iostream>
#include <string>

void reference(std::string& str) {
    std::cout << '左值' << std::endl;
}
void reference(std::string&& str) {
    std::cout << '右值' << std::endl;
}

int main()
{
    std::string lv1 = 'string,';

    std::string&& rv1 = std::move(lv1);
    std::cout << rv1 << std::endl;

    const std::string& lv2 = lv1 + lv1;

    std::cout << lv2 << std::endl;

    std::string&& rv2 = lv1 + lv2;
    rv2 += 'Test';
    std::cout << rv2 << std::endl;

    reference(rv2);

    return 0;
}
```

`rv2` 虽然引用了一个右值，但由于它是一个引用，所以 `rv2` 依然是一个左值。

注意，这里有一个很有趣的历史遗留问题，我们先看下面的代码：

```
#include <iostream>

int main() {

    const int &b = std::move(1);

    std::cout << a << b << std::endl;
}
```

第一个问题，为什么不允许非常量引用绑定到非左值？这是因为这种做法存在逻辑错误：

```
void increase(int & v) {
    v++;
}
void foo() {
    double s = 1;
    increase(s);
}
```

由于 `int&` 不能引用 `double` 类型的参数，因此必须产生一个临时值来保存 `s` 的值，从而当 `increase()` 修改这个临时值时，从而调用完成后 `s` 本身并没有被修改。

第二个问题，为什么常量引用允许绑定到非左值？原因很简单，因为 Fortran 需要。

移动语义

传统 C++ 通过拷贝构造函数和赋值操作符为类对象设计了拷贝/复制的概念，但为了实现对资源的移动操作，调用者必须使用先复制、再析构的方式，否则就需要自己实现移动对象的接口。试想，搬家的时候是把家里的东西直接搬到新家去，而不是将所有东西复制一份（重买）再放到新家、再把原来的东西全部扔掉（销毁），这是非常反人类的一件事情。

传统的 C++ 没有区分『移动』和『拷贝』的概念，造成了大量的数据拷贝，浪费时间和空间。右值引用的出现恰好就解决了这两个概念的混淆问题，例如：

```
#include <iostream>
class A {
public:
    int *pointer;
    A():pointer(new int(1)) {
        std::cout << '构造' << pointer << std::endl;
    }
    A(A& a):pointer(new int(*a.pointer)) {
        std::cout << '拷贝' << pointer << std::endl;
    }
    A(A&& a):pointer(a.pointer) {
        a.pointer = nullptr;
        std::cout << '移动' << pointer << std::endl;
    }
    ~A(){
        std::cout << '析构' << pointer << std::endl;
        delete pointer;
    }
};

A return_rvalue(bool test) {
    A a,b;
    if(test) return a;
    else return b;
}
int main() {
    A obj = return_rvalue(false);
    std::cout << 'obj:' << std::endl;
    std::cout << obj.pointer << std::endl;
    std::cout << *obj.pointer << std::endl;
    return 0;
}
```

在上面的代码中：

1. 首先会在 `return_rvalue` 内部构造两个 `A` 对象，于是获得两个构造函数的输出；
2. 函数返回后，产生一个将亡值，被 `A` 的移动构造（`A(A&&)`）引用，从而延长生命周期，并将这个右值中的指针拿到，保存到了 `obj` 中，而将亡值的指针被设置为 `nullptr`，防止了这块内存区域被销毁。

从而避免了无意义的拷贝构造，加强了性能。再来看看涉及标准库的例子：

```
#include <iostream> // std::cout
#include <utility> // std::move
#include <vector> // std::vector
#include <string> // std::string

int main() {

std::string str = 'Hello world.';
std::vector<std::string> v;


    v.push_back(str);

    std::cout << 'str: ' << str << std::endl;




    v.push_back(std::move(str));

    std::cout << 'str: ' << str << std::endl;

    return 0;
}
```

完美转发

前面我们提到了，一个声明的右值引用其实是一个左值。这就为我们进行参数转发（传递）造成了问题：

```
void reference(int& v) {
    std::cout << '左值' << std::endl;
}
void reference(int&& v) {
    std::cout << '右值' << std::endl;
}
template <typename T>
void pass(T&& v) {
    std::cout << '普通传参:';
    reference(v);
}
int main() {
    std::cout << '传递右值:' << std::endl;
    pass(1);

    std::cout << '传递左值:' << std::endl;
    int l = 1;
    pass(l);

    return 0;
}
```

对于 `pass(1)` 来说，虽然传递的是右值，但由于 `v` 是一个引用，所以同时也是左值。 因此 `reference(v)` 会调用 `reference(int&)`，输出『左值』。 而对于`pass(l)`而言， `l`是一个左值，为什么会成功传递给 `pass(T&&)` 呢?

这是基于**引用坍缩规则**的：在传统 C++ 中，我们不能够对一个引用类型继续进行引用，但 C++ 由于右值引用的出现而放宽了这一做法，从而产生了引用坍缩规则，允许我们对引用进行引用，既能左引用，又能右引用。但是却遵循如下规则：

函数形参类型	实参参数类型	推导后函数形参类型
T&	左引用	T&
T&	右引用	T&
T&&	左引用	T&
T&&	右引用	T&&

因此，模板函数中使用 `T&&` 不一定能进行右值引用，当传入左值时，此函数的引用将被推导为左值。 更准确的讲，**无论模板参数是什么类型的引用，当且仅当实参类型为右引用时，模板参数才能被推导为右引用类型**。 这才使得 `v` 作为左值的成功传递。

完美转发就是基于上述规律产生的。所谓完美转发，就是为了让我们在传递参数的时候，保持原来的参数类型（左引用保持左引用，右引用保持右引用）。 为了解决这个问题，我们应该使用 `std::forward` 来进行参数的转发（传递）：

```
#include <iostream>
#include <utility>
void reference(int& v) {
    std::cout << '左值引用' << std::endl;
}
void reference(int&& v) {
    std::cout << '右值引用' << std::endl;
}
template <typename T>
void pass(T&& v) {
    std::cout << '              普通传参: ';
    reference(v);
    std::cout << '       std::move 传参: ';
    reference(std::move(v));
    std::cout << '    std::forward 传参: ';
    reference(std::forward<T>(v));
    std::cout << 'static_cast<T&&> 传参: ';
    reference(static_cast<T&&>(v));
}
int main() {
    std::cout << '传递右值:' << std::endl;
    pass(1);

    std::cout << '传递左值:' << std::endl;
    int v = 1;
    pass(v);

    return 0;
}
```

输出结果为:

```
传递右值:
              普通传参: 左值引用
       std::move 传参: 右值引用
    std::forward 传参: 右值引用
static_cast<T&&> 传参: 右值引用
传递左值:
              普通传参: 左值引用
       std::move 传参: 右值引用
    std::forward 传参: 左值引用
static_cast<T&&> 传参: 左值引用
```

无论传递参数为左值还是右值，普通传参都会将参数作为左值进行转发，所以 `std::move` 总会接受到一个左值，从而转发调用了 `reference(int&&)` 输出右值引用。

唯独 `std::forward` 即没有造成任何多余的拷贝，同时**完美转发**(传递)了函数的实参给了内部调用的其他函数。

`std::forward` 和 `std::move` 一样，没有做任何事情，`std::move` 单纯的将左值转化为右值，`std::forward` 也只是单纯的将参数做了一个类型的转换，从现象上来看，`std::forward<T>(v)` 和 `static_cast<T&&>(v)` 是完全一样的。

读者可能会好奇，为何一条语句能够针对两种类型的返回对应的值，我们再简单看一看 `std::forward` 的具体实现机制，`std::forward` 包含两个重载：

```
template<typename _Tp>
constexpr _Tp&& forward(typename std::remove_reference<_Tp>::type& __t) noexcept
{ return static_cast<_Tp&&>(__t); }

template<typename _Tp>
constexpr _Tp&& forward(typename std::remove_reference<_Tp>::type&& __t) noexcept
{
    static_assert(!std::is_lvalue_reference<_Tp>::value, 'template argument'
        ' substituting _Tp is an lvalue reference type');
    return static_cast<_Tp&&>(__t);
}
```

在这份实现中，`std::remove_reference` 的功能是消除类型中的引用，而 `std::is_lvalue_reference` 用于检查类型推导是否正确，在 `std::forward` 的第二个实现中检查了接收到的值确实是一个左值，进而体现了坍缩规则。

当 `std::forward` 接受左值时，`_Tp` 被推导为左值，而所以返回值为左值；而当其接受右值时，`_Tp` 被推导为 右值引用，则基于坍缩规则，返回值便成为了 `&& + &&` 的右值。可见 `std::forward` 的原理在于巧妙的利用了模板类型推导中产生的差异。

这时我们能回答这样一个问题：为什么在使用循环语句的过程中，`auto&&` 是最安全的方式？因为当 `auto` 被推导为不同的左右引用时，与 `&&` 的坍缩组合是完美转发。

总结

本章介绍了现代 C++ 中最为重要的几个语言运行时的增强，其中笔者认为本节中提到的所有特性都是值得掌握的：

1. Lambda 表达式
2. 函数对象容器 std::function
3. 右值引用

进一步阅读的参考文献

- Bjarne Stroustrup, C++ 语言的设计与演化