03讲右值和移动究竟解决了什么问题

你好，我是吴咏炜。

从上一讲智能指针开始，我们已经或多或少接触了移动语义。本讲我们就完整地讨论一下移动语义和相关的概念。移动语义是 C++11 里引入的一个重要概念；理解这个概念，是理解很多现代 C++ 里的优化的基础。

值分左右

我们常常会说，C++ 里有左值和右值。这话不完全对。标准里的定义实际更复杂，规定了下面这些值类别（value categories）：

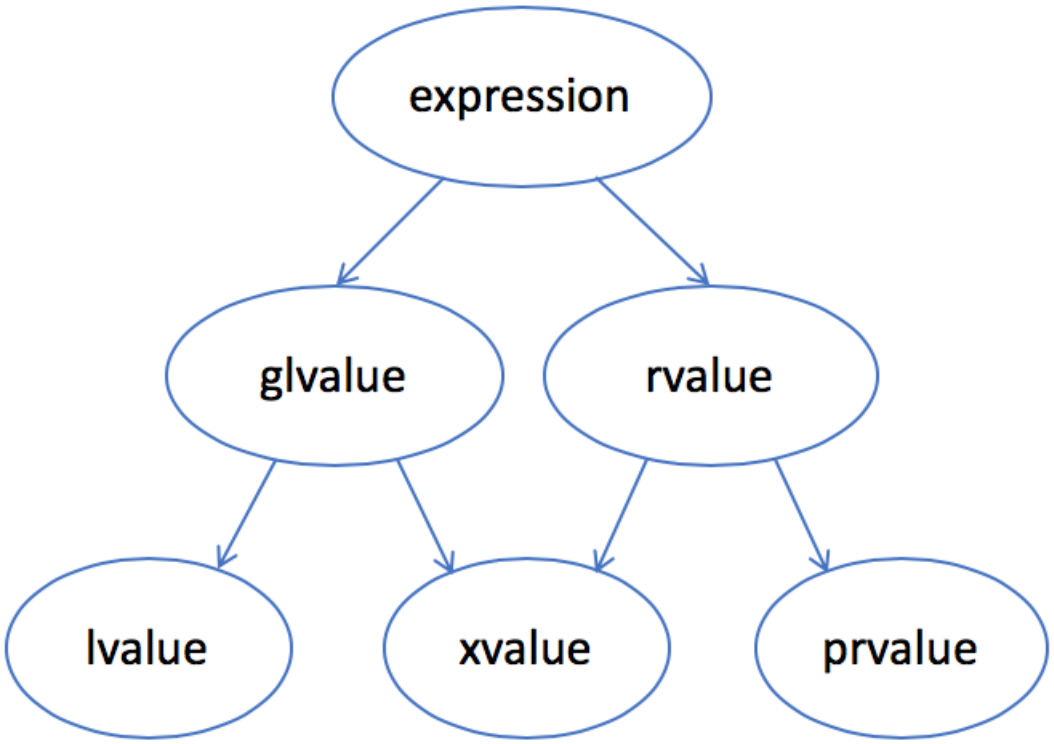

图1：C++ 表达式的值类别

我们先理解一下这些名词的字面含义：

- 一个 lvalue 是通常可以放在等号左边的表达式，左值
- 一个 rvalue 是通常只能放在等号右边的表达式，右值
- 一个 glvalue 是 generalized lvalue，广义左值
- 一个 xvalue 是 expiring lvalue，将亡值
- 一个 prvalue 是 pure rvalue，纯右值

还是有点晕，是吧？我们暂且抛开这些概念，只看其中两个：lvalue 和 prvalue。

左值 lvalue 是有标识符、可以取地址的表达式，最常见的情况有：

- 变量、函数或数据成员的名字
- 返回左值引用的表达式，如 `++x`、`x = 1`、`cout << ' '`
- 字符串字面量如 `"hello world"`

在函数调用时，左值可以绑定到左值引用的参数，如 `T&`。一个常量只能绑定到常左值引用，如 `const T&`。

反之，纯右值 prvalue 是没有标识符、不可以取地址的表达式，一般也称之为“临时对象”。最常见的情况有：

- 返回非引用类型的表达式，如 `x++`、`x + 1`、`make_shared<int>(42)`
- 除字符串字面量之外的字面量，如 `42`、`true`

在 C++11 之前，右值可以绑定到常左值引用（const lvalue reference）的参数，如 `const T&`，但不可以绑定到非常左值引用（non-const lvalue reference），如 `T&`。从 C++11 开始，C++ 语言里多了一种引用类型——右值引用。右值引用的形式是 `T&&`，比左值引用多一个 `&` 符号。跟左值引用一样，我们可以使用 `const` 和 `volatile` 来进行修饰，但最常见的情况是，我

们不会用 const 和 volatile 来修饰右值。本专栏就属于这种情况。

引入一种额外的引用类型当然增加了语言的复杂性，但也带来了很多优化的可能性。由于 C++ 有重载，我们就可以根据不同的引用类型，来选择不同的重载函数，来完成不同的行为。回想一下，在上一讲中，我们就利用了重载，让 smart_ptr 的构造函数可以有不同的行为：

```
template <typename U>
smart_ptr(const smart_ptr<U>& other) noexcept
{
  ptr_ = other.ptr_;
  if (ptr_) {
    other.shared_count_->add_count();
    shared_count_ =
      other.shared_count_;
  }
}
template <typename U>
smart_ptr(smart_ptr<U>&& other) noexcept
{
  ptr_ = other.ptr_;
  if (ptr_) {
    shared_count_ =
      other.shared_count_;
    other.ptr_ = nullptr;
  }
}
```

你可能会好奇，使用右值引用的第二个重载函数中的变量 other 算是左值还是右值呢？根据定义，other 是个变量的名字，变量有标识符、有地址，所以它还是一个左值——虽然它的类型是右值引用。

尤其重要的是，拿这个 other 去调用函数时，它匹配的也会是左值引用。也就是说，**类型是右值引用的变量是一个左值！**这点可能有点反直觉，但跟 C++ 的其他方面是一致的。毕竟对于一个右值引用的变量，你是可以取地址的，这点上它和左值完全一致。稍后我们再回到这个话题上来。

再看一下下面的代码：

```
smart_ptr<shape> ptr1{new circle()};
smart_ptr<shape> ptr2 = std::move(ptr1);
```

第一个表达式里的 new circle() 就是一个纯右值；但对于指针，我们通常使用值传递，并不关心它是左值还是右值。

第二个表达式里的 std::move(ptr) 就有趣点了。它的作用是把一个左值引用强制转换成一个右值引用，而并不改变其内容。从实用的角度，在我们这儿 std::move(ptr1) 等价于 static_cast<smart_ptr<shape>&&>(ptr1)。因此，std::move(ptr1) 的结果是指向 ptr1 的一个右值引用，这样构造 ptr2 时就会选择上面第二个重载。

我们可以把 `std::move(ptr1)` 看作是一个有名字的右值。为了跟无名的纯右值 prvalue 相区别，C++ 里目前就把这种表达式叫做 xvalue。跟左值 lvalue 不同，xvalue 仍然是不能取地址的——这点上，xvalue 和 prvalue 相同。所以，xvalue 和 prvalue 都被归为右值 rvalue。我们用下面的图来表示会更清楚一点：

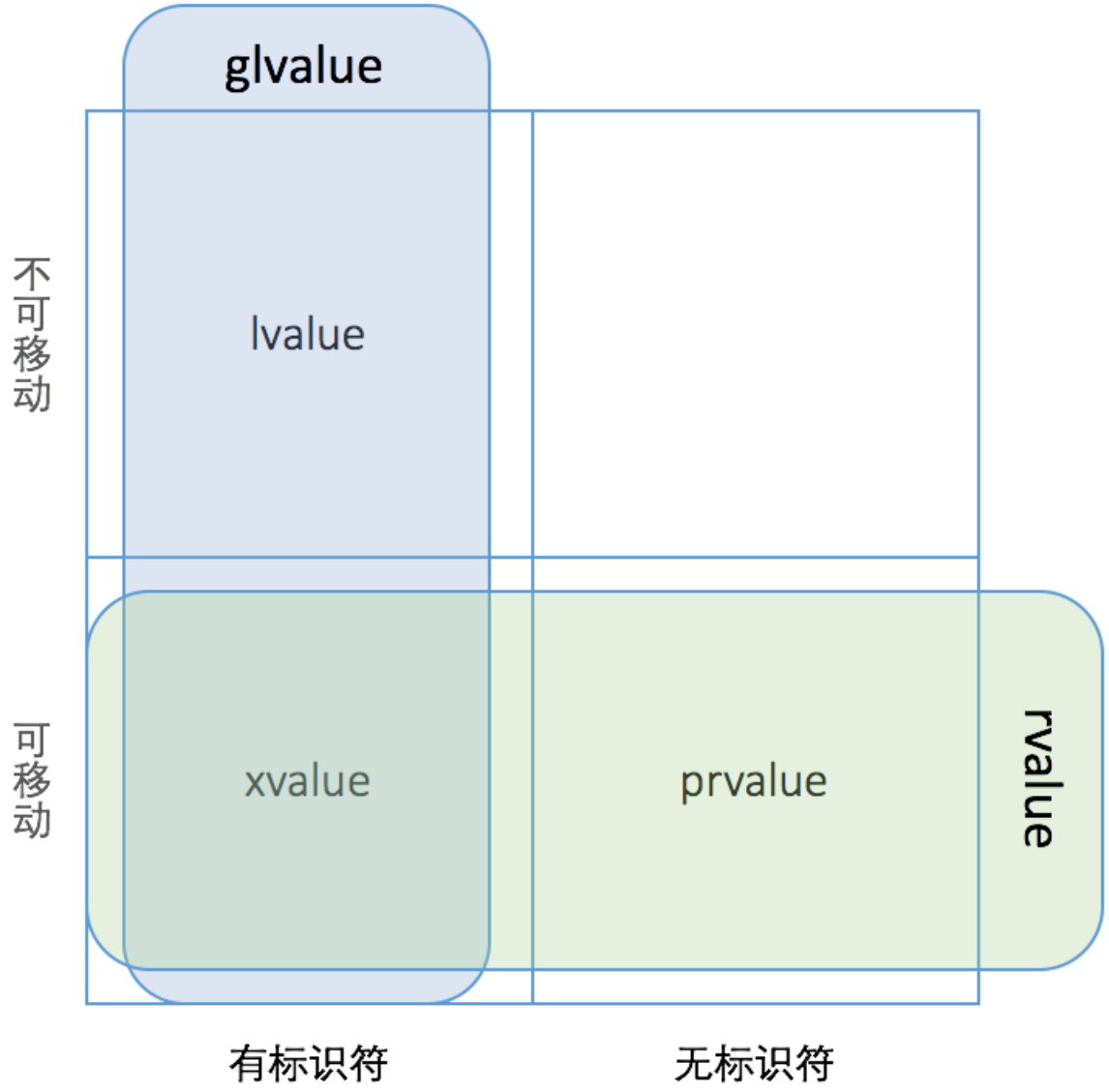

图2：换角度看的表达式值类别

另外请注意，“值类别”（value category）和“值类型”（value type）是两个看似相似、却毫不相干的术语。前者指的是上面这些左值、右值相关的概念，后者则是与引用类型（reference type）相对而言，表明一个变量是代表实际数值，还是引用另外一个数值。在 C++ 里，所有的原生类型、枚举、结构、联合、类都代表值类型，只有引用（`&`）和指针（`*`）才是引用类型。在 Java 里，数字等原生类型是值类型，类则属于引用类型。在 Python 里，一切类型都是引用类型。

生命周期和表达式类型

一个变量的生命周期在超出作用域时结束。如果一个变量代表一个对象，当然这个对象的生命周期也在那时结束。那临时对象（prvalue）呢？在这儿，C++ 的规则是：一个临时对象会在包含这个临时对象的完整表达式估值完成后、按生成顺序的逆序被销毁，除非有生命周期延长发生。我们先看一个没有生命周期延长的基本情况：

```
process_shape(circle(), triangle());
```

在这儿，我们生成了临时对象，一个圆和一个三角形，它们会在 process_shape 执行完成并生成结果对象后被销毁。

我们插入一些实际的代码，就可以演示这一行为：

```
#include <stdio.h>

class shape {
public:
  virtual ~shape() {}
};

class circle : public shape {
public:
  circle() { puts("circle()"); }
  ~circle() { puts("~circle()"); }
};

class triangle : public shape {
public:
  triangle() { puts("triangle()"); }
  ~triangle() { puts("~triangle()"); }
};

class result {
public:
  result() { puts("result()"); }
  ~result() { puts("~result()"); }
};

result
process_shape(const shape& shape1,
              const shape& shape2)
{
  puts("process_shape()");
  return result();
}

int main()
{
  puts("main()");
  process_shape(circle(), triangle());
  puts("something else");
}
```

输出结果可能会是（`circle` 和 `triangle` 的顺序在标准中没有规定）：

```
main()
```

```
circle()
triangle()
process_shape()
result()
~result()
~triangle()
~circle()
something else
```

目前我让 process_shape 也返回了一个结果，这是为了下一步演示的需要。你可以看到结果的临时对象最后生成、最先析构。

为了方便对临时对象的使用，C++ 对临时对象有特殊的生命周期延长规则。这条规则是：

> 如果一个 prvalue 被绑定到一个引用上，它的生命周期则会延长到跟这个引用变量一样长。

我们对上面的代码只要改一行就能演示这个效果。把 process_shape 那行改成：

```
result&& r = process_shape(
  circle(), triangle());
```

我们就能看到不同的结果了：

```
main()
circle()
triangle()
process_shape()
result()
~triangle()
~circle()
something else
~result()
```

现在 result 的生成还在原来的位置，但析构被延到了 main 的最后。

需要万分注意的是，这条生命期延长规则只对 prvalue 有效，而对 xvalue 无效。如果由于某种原因，prvalue 在绑定到引用以前已经变成了 xvalue，那生命期就不会延长。不注意这点的话，代码就可能会产生隐秘的 bug。比如，我们如果这样改一下代码，结果就不对了：

```
#include <utility>  // std::move
…
result&& r = std::move(process_shape(
  circle(), triangle()));
```

这时的代码输出就回到了前一种情况。虽然执行到 something else 那儿我们仍然有一个有效的变量 r，但它指向的对象已经不存在了，对 r 的解引用是一个未定义行为。由于 r 指向的是栈空间，通常不会立即导致程序崩溃，而会在某些复杂的组合

条件下才会引致问题……

对 C++ 的这条生命期延长规则，在后面讲到视图（view）的时候会十分有用。那时我们会看到，有些 C++ 的用法实际上会隐式地利用这条规则。

此外，参考资料 [5] 中提到了一个有趣的事实：你可以把一个没有虚析构函数的子类对象绑定到基类的引用变量上，这个子类对象的析构仍然是完全正常的——这是因为这条规则只是延后了临时对象的析构而已，不是利用引用计数等复杂的方法，因而只要引用绑定成功，其类型并没有什么影响。

移动的意义

上面我们谈了一些语法知识。就跟学外语的语法一样，这些内容是比较枯燥的。虽然这些知识有时有用，但往往要回过头来看的时候才觉得。初学之时，更重要的是理解为什么，和熟练掌握基本的用法。

对于 `smart_ptr`，我们使用右值引用的目的是实现移动，而实现移动的意义是减少运行的开销——在引用计数指针的场景下，这个开销并不大。移动构造和拷贝构造的差异仅在于：

- 少了一次 `other.shared_count_->add_count()` 的调用
- 被移动的指针被清空，因而析构时也少了一次 `shared_count_->reduce_count()` 的调用

在使用容器类的情况下，移动更有意义。我们可以尝试分析一下下面这个假想的语句（假设 `name` 是 `string` 类型）：

```
string result =
  string("Hello, ") + name + ".";
```

在 C++11 之前的年代里，这种写法是绝对不推荐的。因为它会引入很多额外开销，执行流程大致如下：

1. 调用构造函数 `string(const char*)`，生成临时对象 1；`"Hello, "` 复制 1 次。
2. 调用 `operator+(const string&, const string&)`，生成临时对象 2；`"Hello, "` 复制 2 次，`name` 复制 1 次。
3. 调用 `operator+(const string&, const char*)`，生成对象 3；`"Hello, "` 复制 3 次，`name` 复制 2 次，`"."` 复制 1 次。
4. 假设返回值优化能够生效（最佳情况），对象 3 可以直接在 `result` 里构造完成。
5. 临时对象 2 析构，释放指向 `string("Hello, ") + name` 的内存。
6. 临时对象 1 析构，释放指向 `string("Hello, ")` 的内存。

既然 C++ 是一门追求性能的语言，一个合格的 C++ 程序员会写：

```
string result = "Hello, ";
result += name;
result += ".";
```

这样的话，只会调用构造函数一次和 `string::operator+=` 两次，没有任何临时对象需要生成和析构，所有的字符串都只复制了一次。但显然代码就啰嗦多了——尤其如果拼接的步骤比较多的话。从 C++11 开始，这不再是必须的。同样上面那个单行的语句，执行流程大致如下：

1. 调用构造函数 `string(const char*)`，生成临时对象 1；`"Hello, "` 复制 1 次。
2. 调用 `operator+(string&&, const string&)`，直接在临时对象 1 上面执行追加操作，并把结果移动到临时对象 2；`name` 复制 1 次。

3. 调用 operator+(string&&, const char*)，直接在临时对象 2 上面执行追加操作，并把结果移动到 result；"." 复制 1 次。
4. 临时对象 2 析构，内容已经为空，不需要释放任何内存。
5. 临时对象 1 析构，内容已经为空，不需要释放任何内存。

性能上，所有的字符串只复制了一次；虽然比啰嗦的写法仍然要增加临时对象的构造和析构，但由于这些操作不牵涉到额外的内存分配和释放，是相当廉价的。程序员只需要牺牲一点点性能，就可以大大增加代码的可读性。而且，所谓的性能牺牲，也只是相对于优化得很好的 C 或 C++ 代码而言——这样的 C++ 代码的性能仍然完全可以超越 Python 类的语言的相应代码。

此外很关键的一点是，C++ 里的对象缺省都是值语义。在下面这样的代码里：

```
class A {
  B b_;
  C c_;
};
```

从实际内存布局的角度，很多语言——如 Java 和 Python——会在 A 对象里放 B 和 C 的指针（虽然这些语言里本身没有指针的概念）。而 C++ 则会直接把 B 和 C 对象放在 A 的内存空间里。这种行为既是优点也是缺点。说它是优点，是因为它保证了内存访问的局域性，而局域性在现代处理器架构上是绝对具有性能优势的。说它是缺点，是因为复制对象的开销大大增加：在 Java 类语言里复制的是指针，在 C++ 里是完整的对象。这就是为什么 C++ 需要移动语义这一优化，而 Java 类语言里则根本不需要这个概念。

一句话总结，移动语义使得在 C++ 里返回大对象（如容器）的函数和运算符成为现实，因而可以提高代码的简洁性和可读性，提高程序员的生产率。

所有的现代 C++ 的标准容器都针对移动进行了优化。

如何实现移动?

要让你设计的对象支持移动的话，通常需要下面几步：

- 你的对象应该有分开的拷贝构造和移动构造函数（除非你只打算支持移动，不支持拷贝——如 unique_ptr）。
- 你的对象应该有 swap 成员函数，支持和另外一个对象快速交换成员。
- 在你的对象的名空间下，应当有一个全局的 swap 函数，调用成员函数 swap 来实现交换。支持这种用法会方便别人（包括你自己在将来）在其他对象里包含你的对象，并快速实现它们的 swap 函数。
- 实现通用的 operator=。
- 上面各个函数如果不抛异常的话，应当标为 noexcept。这对移动构造函数尤为重要。

具体写法可以参考我们当前已经实现的 smart_ptr：

- smart_ptr 有拷贝构造和移动构造函数（虽然此处我们的模板构造函数严格来说不算拷贝或移动构造函数）。移动构造函数应当从另一个对象获取资源，清空其资源，并将其置为一个可析构的状态。

```
smart_ptr(const smart_ptr& other) noexcept
{
  ptr_ = other.ptr_;
  if (ptr_) {
    other.shared_count_
      ->add_count();
    shared_count_ =
      other.shared_count_;
  }
}
template <typename U>
smart_ptr(const smart_ptr<U>& other) noexcept
{
  ptr_ = other.ptr_;
  if (ptr_) {
    other.shared_count_
      ->add_count();
    shared_count_ =
      other.shared_count_;
  }
}
template <typename U>
smart_ptr(smart_ptr<U>&& other) noexcept
{
  ptr_ = other.ptr_;
  if (ptr_) {
    shared_count_ =
      other.shared_count_;
    other.ptr_ = nullptr;
  }
}
```

- smart_ptr 有 swap 成员函数。

```
void swap(smart_ptr& rhs) noexcept
{
  using std::swap;
  swap(ptr_, rhs.ptr_);
  swap(shared_count_,
       rhs.shared_count_);
}
```

- 有支持 `smart_ptr` 的全局 `swap` 函数。

```
template <typename T>
void swap(smart_ptr<T>& lhs,
          smart_ptr<T>& rhs) noexcept
{
  lhs.swap(rhs);
}
```

- `smart_ptr` 有通用的 `operator=` 成员函数。注意为了避免让人吃惊，通常我们需要将其实现成对 `a = a;` 这样的写法安全。下面的写法算是个小技巧，对传递左值和右值都有效，而且规避了 `if (&rhs != this)` 这样的判断。

```
  smart_ptr&
  operator=(smart_ptr rhs) noexcept
  {
    rhs.swap(*this);
    return *this;
  }
```

不要返回本地变量的引用

有一种常见的 C++ 编程错误，是在函数里返回一个本地对象的引用。由于在函数结束时本地对象即被销毁，返回一个指向本地对象的引用属于未定义行为。理论上来说，程序出任何奇怪的行为都是正常的。

在 C++11 之前，返回一个本地对象意味着这个对象会被拷贝，除非编译器发现可以做返回值优化（named return value optimization，或 NRVO），能把对象直接构造到调用者的栈上。从 C++11 开始，返回值优化仍可以发生，但在没有返回值优化的情况下，编译器将试图把本地对象移动出去，而不是拷贝出去。这一行为不需要程序员手工用 `std::move` 进行干预——使用 `std::move` 对于移动行为没有帮助，反而会影响返回值优化。

下面是个例子：

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

using namespace std;

class Obj {
public:
  Obj()
  {
    cout << "Obj()" << endl;
  }
  Obj(const Obj&)
  {
    cout << "Obj(const Obj&)"
```

```
    cout << "Obj(const Obj&)"
         << endl;
  }
  Obj(Obj&&)
  {
    cout << "Obj(Obj&&)" << endl;
  }
};

Obj simple()
{
  Obj obj;
  // 简单返回对象；一般有 NRVO
  return obj;
}

Obj simple_with_move()
{
  Obj obj;
  // move 会禁止 NRVO
  return std::move(obj);
}

Obj complicated(int n)
{
  Obj obj1;
  Obj obj2;
  // 有分支，一般无 NRVO
  if (n % 2 == 0) {
    return obj1;
  } else {
    return obj2;
  }
}

int main()
{
  cout << "*** 1 ***" << endl;
  auto obj1 = simple();
  cout << "*** 2 ***" << endl;
  auto obj2 = simple_with_move();
  cout << "*** 3 ***" << endl;
  auto obj3 = complicated(42);
}
```

输出通常为：

```
*** 1 ***
Obj()
*** 2 ***
Obj()
Obj(Obj&&)
*** 3 ***
Obj()
Obj()
Obj(Obj&&)
```

也就是，用了 `std::move` 反而妨碍了返回值优化。

引用坍缩和完美转发

最后讲一个略复杂、但又不得不讲的话题，引用坍缩（又称“引用折叠”）。这个概念在泛型编程中是一定会碰到的。我们今天既然讲了左值和右值引用，也需要一起讲一下。

我们已经讲了对于一个实际的类型 `T`，它的左值引用是 `T&`，右值引用是 `T&&`。那么：

1. 是不是看到 `T&`，就一定是个左值引用?
2. 是不是看到 `T&&`，就一定是个右值引用?

对于前者的回答是“是”，对于后者的回答为“否”。

关键在于，在有模板的代码里，对于类型参数的推导结果可能是引用。我们可以略过一些繁复的语法规则，要点是：

- 对于 `template <typename T> foo(T&&)` 这样的代码，如果传递过去的参数是左值，`T` 的推导结果是左值引用；如果传递过去的参数是右值，`T` 的推导结果是参数的类型本身。
- 如果 `T` 是左值引用，那 `T&&` 的结果仍然是左值引用——即 `type& &&` 坍缩成了 `type&`。
- 如果 `T` 是一个实际类型，那 `T&&` 的结果自然就是一个右值引用。

我们之前提到过，右值引用变量仍然会匹配到左值引用上去。下面的代码会验证这一行为：

```
void foo(const shape&)
{
  puts("foo(const shape&)");
}

void foo(shape&&)
{
  puts("foo(shape&&)");
}

void bar(const shape& s)
{
  puts("bar(const shape&)");
  foo(s);
}

void bar(shape&& s)
{
  puts("bar(shape&&)");
  foo(s);
}

int main()
{
  bar(circle());
}
```

输出为：

```
bar(shape&&)
foo(const shape&)
```

如果我们要让 bar 调用右值引用的那个 foo 的重载，我们必须写成：

```
foo(std::move(s));
```

或：

```
foo(static_cast<shape&&>(s));
```

可如果两个 bar 的重载除了调用 foo 的方式不一样，其他都差不多的话，我们为什么要提供两个不同的 bar 呢？

事实上，很多标准库里的函数，连目标的参数类型都不知道，但我们仍然需要能够保持参数的值类别：左值的仍然是左值，右

值的仍然是右值。这个功能在 C++ 标准库中已经提供了，叫 `std::forward`。它和 `std::move` 一样都是利用引用坍缩机制来实现。此处，我们不介绍其实现细节，而是重点展示其用法。我们可以把我们的两个 `bar` 函数简化成：

```
template <typename T>
void bar(T&& s)
{
  foo(std::forward<T>(s));
}
```

对于下面这样的代码：

```
circle temp;
bar(temp);
bar(circle());
```

现在的输出是：

```
foo(const shape&)
foo(shape&&)
```

一切如预期一样。

因为在 `T` 是模板参数时，`T&&` 的作用主要是保持值类别进行转发，它有个名字就叫“转发引用”（forwarding reference）。因为既可以是左值引用，也可以是右值引用，它也曾经被叫做“万能引用”（universal reference）。

内容小结

本讲介绍了 C++ 里的值类别，重点介绍了临时变量、右值引用、移动语义和实际的编程用法。由于这是 C++11 里的重点功能，你对于其基本用法需要牢牢掌握。

课后思考

留给你两道思考题：

1. 请查看一下标准函数模板 `make_shared` 的声明，然后想一想，这个函数应该是怎样实现的。
2. 为什么 `smart_ptr::operator=` 对左值和右值都有效，而且不需要对等号两边是否引用同一对象进行判断？

欢迎留言和我交流你的看法，尤其是对第二个问题。

参考资料

[1] cppreference.com, “Value categories”. https://en.cppreference.com/w/cpp/language/value_category

[1a] cppreference.com, “值类别”. https://zh.cppreference.com/w/cpp/language/value_category

[2] Anders Schau Knatten, “lvalues, rvalues, glvalues, prvalues, xvalues, help!”. https://blog.knatten.org/2018/03/09/lvalues-rvalues-glvalues-prvalues-xvalues-help/

[3] Jeaye, “Value category cheat-sheet”. https://blog.jeaye.com/2017/03/19/xvalues/

[4] Thomas Becker, “C++ rvalue references explained”. http://thbecker.net/articles/rvalue_references/section_01.html

[5] Herb Sutter, “GotW #88: A candidate for the ‘most important const’”. https://herbsutter.com/2008/01/01/gotw-88-a-candidate-for-the-most-important-const/

精选留言

hello world

一直有个问题想问老师，老师第一节也说了一些，但我还是有一些疑惑和焦虑希望老师能够解惑。
现在C++的应用范围貌似越来越窄，以前可能很多后台会用到，但是现在貌似后台都是go和java多一些，真正追求性能极致的貌似不多，现在也就一些人工智能方面和嵌入式方面会用到，但也是少数，真正要做一个深度学习框架的也不太多，即便是大企业也是个别几个部门，作为一个C++程序员比较迷茫，感觉这条路比较窄，未来的路怎么走呢？

2019-12-02 13:40

作者回复

把C++当作一种工具，也当作一种锻炼思维的方式，不要把它当成非用不可的圣器。把自己定位成“程序员”，而不是“C++程序员”。职业的将来方向定位成“软件架构师”、“开发主管”、“CTO”之类的角色，而不是“高级C++程序员”。

该用其他语言的时候用其他语言。要看到，即使不用C++，C++程序员在对优化的理解、对内存管理的理解等方面都是具有很大优势的。

2019-12-02 21:29

NEVER SETTLE

请教下老师，字符串字面量是左值，是不是在C++中 字符串其实是const char[N]，其实是个常量表达式，在内存中有明确的地址。

2019-12-02 19:51

作者回复

是。

2019-12-02 20:21

中年男子

第二题：
左值和右值都有效是因为构造参数时，如果是左值，就用拷贝构造构造函数，右值就用移动构造函数
无论是左值还是右值，构造参数时直接生成新的智能指针，因此不需要判断

2019-12-03 16:07

作者回复

理解满分。

2019-12-03 21:02

糖

又是看不懂的一节。。。老师讲的课程太深刻了。。。
1. 本来感觉自己还比较了解左右值的区别，但是，文中提到：一个 lvalue 是通常可以放在等号左边的表达式，左值，然后下面说：字符串字面量如 "hello world"，但字符串字面量貌似不可以放到等号左边，搞晕了。
2. 内存访问的局域性是指什么呢？又有何优势呢？老师能提供介绍的链接吗
3. 为何对于移动构造函数来讲不抛出异常尤其重要呢？
希望老师能指点一下

2019-12-02 11:28

作者回复

1. “通常”。字符串字面量是个继承自C的特殊情况。

2. 这个搜索一下就行。这是CPU的缓存结构决定的。

3. 其他类，尤其容器类，会期待移动构造函数无异常，甚至会在它有异常时选择拷贝构造函数，以保证强异常安全性。

2019-12-02 20:18

NEVER SETTLE

老师，留言有字数限制，我是接着上个留言来的，上面那个总结了下左值与右值，这个是右值引用的学习心得：
2、右值引用
针对以上的所说的“临时变量”，如何来“接收”它呢？
* 最直白的办法，就是直接用一个变量来“接收”
以x++为例：
```
void playVal(int y) {
cout << "y = " << y << ", (y).adrr = " << &y << endl;
}

int x = 0;
playVal(x++);
cout << "x = " << x << ", (x).adrr = " << &x << endl;
```

>运行结果：
y = 0, (y).adrr = 0x22fe20
x = 1, (x).adrr = 0x22fe4c

这是一个值传递过程，相当于 int y = x++，即x++生成的临时变量给变量y赋值，之后临时变量就“消失”，这里发生是一次拷贝。

如何避免发生拷贝呢?
通常做法是使用引用来“接收”，即引用传递。
上面说过，使用一个（常规）引用来“接收”一个临时变量，会报错：
```
void playVal(int& y)
```
> error : 非常量引用的初始值必须为左值

* 普遍的做法都是使用常量引用来“接收”临时变量（C++11之前）
```
void playVal(const int& y)
```
这里编译器做了处理： int tmp = x++; const int& y = tmp; 发生内存分配。
其实还是发生了拷贝。

* 使用右值引用来“接收”临时变量（C++11之后）
上面说过，“临时变量”是一个右值，所以这里可以使用右值引用来“接收”它
右值引用的形式是 T&& ：
```
void playVal(int&& y) {
cout << "y = " << y << ", (y).adrr = " << &y << endl;
}

int x = 0;
playVal(x++);
cout << "x = " << x << ", (x).adrr = " << &x << endl;
```

> 运行结果：

y = 0, (y).adrr = 0x22fe4c
x = 1, (x).adrr = 0x22fe48

这是一个（右值）引用传递的过程，相当于 int&& y = x++，这里的右值引用 y 直接“绑定”了“临时变量”，因为它就会有了命名，变成“合法”的，就不会“消失”。
注意：这里的变量 y 虽然是右值引用类型，但它是一个左值，可以正常对它取地址
（如上例所示）
2019-12-03 14:24

作者回复

再多说一句，如果每个人都这么让我来看笔记的话，我是不可能满足所有人的。只看你的，也对别人不公。在这儿回答（不重复的）问题则不同，问题一般是有共性的，回答之后，大家都能看到，都能从中受益。
2019-12-03 21:21

禾桃
"请查看一下标准函数模板 make_shared 的声明，然后想一想，这个函数应该是怎样实现的。"

template <class T, class... Args>
std::shared_ptr<T> make_shared (Args&&... args)
{
T* ptr = new T(std::forward<Args...>(args...));
return std::shared_ptr<T>(ptr);
}

我的考虑是：
make_shared声明里的（Args&&...） 是universal reference， 所以在函数体里用完美转发（std::forward）把参数出入T的构造函数, 以调用每个参数各自对用的构造函数（copy or move）。

肯定还有别的需要考量的地方，请指正。

谢谢！
2019-12-03 11:45

作者回复

对，最主要就是这点，用完美转发来正确调用构造函数。
2019-12-03 20:49

NEVER SETTLE
老师，我这个初学者看的比较慢，目前只看了右值与右值引用，下面是我总结了的学习心得，请您指点下：
**背景：
C++11为了支持移动操作，引用了新的引用类型-右值引用。
所谓右值引用就是绑定到右值的引用。
为了区分右值引用，引入左值引用概念，即常规引用。
那左值与右值是是什么？**

1、左值与右值

左值 lvalue 是有标识符、可以取地址的表达式
* 变量、函数或数据成员的名字
* 返回左值引用的表达式，如 ++x、x = 1、cout << ' '
* 字符串字面量如 "hello world"

表达式是不是左值，就看是否可以取地址，或者返回类型是否可以用（常规）引用来接收：

```
```

```
int x = 0;
cout << "(x).addr = " << &x << endl;
cout << "(x = 1).addr = " << &(x = 1) << endl; //x赋值1，返回x
cout << "(++x).addr = " << &++x << endl; //x自增1，返回x
```

> 运行结果：
(x).addr = 0x22fe4c
(x = 1).addr = 0x22fe4c
(++x).addr = 0x22fe4c

```
cout << "hello world = " << &("hello world") << endl;
```

> 运行结果：
hello world = 0x40403a

C++中的字符串字面量，可以称为字符串常量，表示为const char[N]，其实是地址常量表达式。
在内存中有明确的地址，不是临时变量。

```
cout << "cout << ' ' = " << &(cout << ' ') << endl;
```
> 运行结果：
cout << ' ' = 0x6fd0acc0

纯右值 prvalue 是没有标识符、不可以取地址的表达式，一般称为“临时对象”
* 返回非引用类型的表达式，如 x++、x + 1、make_shared(42)
* 除字符串字面量之外的字面量，如 42、true

```
//cout << "(x++).addr = " << &x++ << endl; //返回一个值为x的临时变量，再把x自增1
//cout << "(x + 1).addr = " << &(x + 1) << endl; //返回一个值为x+1的临时变量
//cout << "(42).addr = " << &(42) << endl; //返回一个值为42的临时变量
//cout << "(true).addr = " << &(true) << endl; //返回一个值为true的临时变量
```
> 编译出错：
每行代码报错：表达式必须为左值或函数指示符
因为以上表达式都返回的是“临时变量”，是不可以取地址的

2019-12-03 14:23

作者回复

你在学校的时候，都是让老师来看你的笔记记得好不好的吗?

对不起，如果有明确的问题，我可以回答。否则，我只能暂时忽略了。

2019-12-03 20:57

NEVER SETTLE

“返回左值引用的表达式，，如 x++、x + 1 ”不太清楚原因，后来我就试了下：

```
int x = 0;
cout << "(x).addr = " << &x << endl;
cout << "(x = 1).addr = " << &(x = 1) << endl;
cout << "(++x).addr = " << &++x << endl;
//cout << "(x++).addr = " << &x++ << endl;
```

> 运行结果：
(x).addr = 0x22fe4c
(x = 1).addr = 0x22fe4c
(++x).addr = 0x22fe4c
最后一行注释掉的代码报错：表达式必须为左值或函数指示符
2019-12-02 20:41

作者回复

对，x = 1 和 ++x 返回的都是对 x 的 int&。x++ 则返回的是 int。
2019-12-02 21:52

千鲤湖

老师，我把实例稍微改了下，
class Obj
{
public:
Obj()
{
std::cout << "Obj()" << std::endl;
}

Obj(const Obj&)
{
std::cout << "Obj(const Obj&)" << std::endl;
}

Obj(Obj&&)
{
std::cout << "Obj(Obj&&)" << std::endl;
}
};

void foo(const Obj&)
void foo(Obj&&)
void bar(const Obj& s)
void bar(Obj&& s)

int main()
{
bar(Obj());
}

构造函数内加了打印。

期望看到的结果是这样的

Obj()
Obj(&&)
bar(Obj&&)
Obj(const&)
foo(const Obj&)
可实际输出如下
Obj()
bar(&&)
foo(const &)

并没有期望中的移动构造和复制构造，这是为什么啊。
关于没有移动构造，我的理解是Obj()本来已经是个右值了，不必再构造。
可是想不通为什么没有了复制构造。

2019-12-04 18:00

作者回复

中间传的都是引用，没有拷贝或移动发生的。只有用Obj（而不是Obj&或Obj&&）作为参数类型才会发生拷贝或移动构造。

2019-12-04 20:51

安静的雨

Obj simple_with_move()
{
Obj obj;
// move 会禁止 NRVO
return std::move(obj);
}

move后不是类型转换到右值引用了吗？ 为啥返回值类型还是obj?

2019-12-03 10:13

作者回复

文中已经说了，禁止返回本地对象的引用。

需要生成一个 Obj，给了一个 Obj&&，不就是调用构造函数而已么。所以（看文中输出），就是多产生了一次Obj(Obj&&) 的调用。

2019-12-03 20:43

哇咔咔

老师你好，这段代码压测下来，发现左值引用没有性能的提升。压测时间对比是：
elapsed time: 1.2184s
elapsed time: 1.1857s

请问为什么呢?

#include <string>
#include <ctime>
#include <chrono>
#include <iostream>

void func1(std::string s)
{
}

void func2(const std::string &s)

```
{
}

void test2()
{
auto start = std::chrono::system_clock::now();
for (size_t i = 0; i < 20000000; i++)
{
func1(std::string("hello"));
}
auto end = std::chrono::system_clock::now();
std::chrono::duration<double> elapsed_seconds = end - start;
std::cout << "elapsed time: " << elapsed_seconds.count() << "s\n";

start = std::chrono::system_clock::now();
for (size_t i = 0; i < 20000000; i++)
{
func2(std::string("hello"));
}
end = std::chrono::system_clock::now();
elapsed_seconds = end - start;
std::cout << "elapsed time: " << elapsed_seconds.count() << "s\n";
}

int main()
{
test2();
}
```

2019-12-11 10:02

作者回复

因为移动发挥威力了……试试把 std::string("hello") 放到 test2 开头作为变量，然后后面使用这个变量。

2019-12-11 18:33

罗乾林

平时Java是主要使用语言，也来回答一下

1、make_shared 创建(new)新对象根据传入的值类别调用拷贝构造或移动构造,然后将新对象的指针给shared_ptr，其中我看见了_Types&&和forward

2、smart_ptr::operator= 中参数为值传递，会先调用smart_ptr的拷贝构造函数，生成了临时对象，然后调用swap，

因为生成了新对象所以对等号两边是否引用同一对象进行判断，也没意义了，但是a=a也会有临时对象的产生，有性能开销

有错误的方，望老师指正

2019-12-02 11:07

作者回复

2完全正确。1再想想。

2019-12-02 20:08

禾桃

“result&& r = process_shape(

circle(), triangle());

如果一个 prvalue 被绑定到一个引用上，它的生命周期则会延长到跟这个引用变量一样长。”

在上面这行代码执行完后，栈指针已经不再指向这个函数（process_shape)栈，换言之，这个栈所使用的内存可以被后续代码

使用。

您的例题中貌似也没有返回值优化，如果是的话，那个prvalue（result（））貌似被构建在了上面那个函数的栈内存上，这样的话，在函数栈被回收后，在函数栈上被构建的prvalue这个对象的周期是如何被延长的？难道这种情况下，函数栈没有被回收？

谢谢！

2019-12-19 08:46

作者回复

process_shape 返回的是对象，不是引用。结果不管有没有返回值优化（实际是有的），都是在调用者的栈上。没有生命期延长的话，执行完 process_shape 这一句，对象就销毁了。有生命期延长，则要到 r 的生命期结束时才销毁。

2019-12-19 13:43

张岩

```
int main()
{
A&& a = static_cast<A&&>(GetA());
cout<<"main end"<<endl;
a.func();
return 0;
}

g++ -g -std=c++11 construct.cpp ;./a.out
construct: 1
main end
func: 1
destruct: 1
```

老师您好，请问下，为什么上面的这种情况xvalue的生命周期被延长了。

2019-12-18 19:32

作者回复

xvalue 是有标识符的。这儿临时对象一直没有标识符，也就一直是 prvalue。

2019-12-18 23:10

微秒

老师，我对左值、右值、移动的区别很懵。可以简单的说下吗?

2019-12-13 15:56

作者回复

我觉得我做不到比文中更清楚了（否则我就写进去了）。

多看例子来体会一下吧……

2019-12-14 06:25

王小白白白

老师，string拼接那里，复制指的是?

2019-12-10 23:21

作者回复

内存复制，memcpy、strcpy 这样的操作。

2019-12-11 08:08

三味

看完了目前的06讲. 感觉最不好理解的还是这03讲. 关于xvalue还是感觉有些迷糊...

关于circle triangle shape result那一段的代码.

1. 我看上面描述的xvalue, 通常是使用std::move被强制转换为右值的值, 这么理解对不对?

2. 还有一个就是

result&& r = std::move(process_shape(circle(), triangle()));
这个r当然是个左值, 它指代的右值, 就是xvalue, 这么理解对不对?
3. 上面的这个xvalue, 究竟是什么时候完蛋的?
最后是从上面引申出来的问题:
4. std::move()函数的返回值是T&&, 所以用
T&& t = std::move(value);
多么自然的一件事情...为什么这么自然的表达式, 结果却是栈内存的错误解引用呢... C++为啥要这么去规定?
期待您的回复.

2019-12-09 15:12

作者回复

迷糊是正常的……我也啃了几遍才基本搞通。多看几遍吧。

1. 是。
2. 是。
3. 超出作用域就没了。在你的这个例子里，它的生命期跟对象的生命期没有关系。
4. 错误解引用是因为引用超出了变量的生命期。如果没有生命期延长，一个临时对象在当前语句执行结束即被销毁。你写的不会有问题，但把value替换成get_value()一般就会了。

2019-12-09 20:58

じJRisenづジ

老师您的github 是?

2019-12-05 11:38

作者回复

https://github.com/adah1972/

2019-12-05 17:56

宝林

这一节看了三遍，思路太跳跃了，不易读

2019-12-05 07:07

花晨少年

Obj simple()
{
Obj obj;
// 简单返回对象；一般有 NRVO
return obj;
}

auto obj1 = simple();

请问这个不应该是会调用一次构造和拷贝构造吗，因为函数返回对象Obj是用了函数内部的obj进行拷贝构造，而返回的这个对象被直接优化给了obj1而没有调用构造函数
那么如果没有优化应该是两次构造，一次拷贝构造才对啊，感觉

2019-12-05 00:18

作者回复

返回值优化在没有开启优化编译选项时也是可能发生的。编译器看到这种形式的代码直接统一处理了。

2019-12-05 09:27