一文读懂C++右值引用和std::move

腾讯技术工程
已认证帐号

作者：rickonji 冀铭哲

> C++11引入了右值引用，有一定的理解成本，工作中发现不少同事对右值引用理解不深，认为右值引用性能更高等等。本文从实用角度出发，用尽量通俗易懂的语言讲清左右值引用的原理，性能分析及其应用场景，帮助大家在日常编程中用好右值引用和std::move。

1. 什么是左值、右值

首先不考虑引用以减少干扰，可以从2个角度判断：左值**可以取地址、位于等号左边**；而右值**没法取地址，位于等号右边。**

```
int a = 5;
```

- a可以通过 & 取地址，位于等号左边，所以a是左值。
- 5位于等号右边，5没法通过 & 取地址，所以5是个右值。

再举个例子：

```
struct A {
    A(int a = 0) {
        a_ = a;
    }

    int a_;
};

A a = A();
```

- 同样的，a可以通过 & 取地址，位于等号左边，所以a是左值。
- A()是个临时值，没法通过 & 取地址，位于等号右边，所以A()是个右值。

可见左右值的概念很清晰，有地址的变量就是左值，没有地址的字面值、临时值就是右值。

2. 什么是左值引用、右值引用

引用本质是别名，可以通过引用修改变量的值，传参时传引用可以避免拷贝，其实现原理和指针类似。 个人认为，引用出现的本意是为了降低C语言指针的使用难度，但现在指针+左右值引用共同存在，反而大大增加了学习和理解成本。

2.1 左值引用

左值引用大家都很熟悉，**能指向左值，不能指向右值的就是左值引用**：

```
int a = 5;
int &ref_a = a; // 左值引用指向左值，编译通过
int &ref_a = 5; // 左值引用指向了右值，会编译失败
```

引用是变量的别名，由于右值没有地址，没法被修改，所以左值引用无法指向右值。

但是，const左值引用是可以指向右值的：

```
const int &ref_a = 5;  // 编译通过
```

const左值引用不会修改指向值，因此可以指向右值，这也是为什么要使用 `const &` 作为函数参数的原因之一，如 `std::vector` 的 `push_back` ：

```
void push_back (const value_type& val);
```

如果没有 `const`， `vec.push_back(5)` 这样的代码就无法编译通过了。

2.2 右值引用

再看下右值引用，右值引用的标志是 `&&` ，顾名思义，右值引用专门为右值而生，**可以指向右值，不能指向左值**：

```
int &&ref_a_right = 5; // ok

int a = 5;
int &&ref_a_left = a; // 编译不过，右值引用不可以指向左值

ref_a_right = 6; // 右值引用的用途：可以修改右值
```

2.3 对左右值引用本质的讨论

下边的论述比较复杂，也是本文的核心，对理解这些概念非常重要。

2.3.1 右值引用有办法指向左值吗?

有办法， `std::move` ：

```
int a = 5; // a是个左值
int &ref_a_left = a; // 左值引用指向左值
int &&ref_a_right = std::move(a); // 通过std::move将左值转化为右值，可以被右值引用指向

cout << a; // 打印结果：5
```

在上边的代码里，看上去是左值a通过std::move移动到了右值ref_a_right中，那是不是a里边就没有值了？并不是，打印出a的值仍然是5。

`std::move` 是一个非常有迷惑性的函数，不理解左右值概念的人们往往以为它能把一个变量里的内容移动到另一个变量，**但事实上std::move移动不了什么，唯一的功能是把左值强制转化为右值**，让右值引用可以指向左值。其实现等同于一个类型转换： `static_cast<T&&>(lvalue)` 。 所以，**单纯的std::move(xxx)不会有性能提升**，std::move的使用场景在第三章会讲。

同样的，右值引用能指向右值，本质上也是把右值提升为一个左值，并定义一个右值引用通过std::move指向该左值：

```
int &&ref_a = 5;
ref_a = 6;
```

等同于以下代码：

```
int temp = 5;
int &&ref_a = std::move(temp);
ref_a = 6;
```

2.3.2 左值引用、右值引用本身是左值还是右值?

被声明出来的左、右值引用都是左值。 因为被声明出的左右值引用是有地址的，也位于等号左边。仔细看下边代码：

```
// 形参是个右值引用
void change(int&& right_value) {
    right_value = 8;
}

int main() {
    int a = 5; // a是个左值
    int &ref_a_left = a; // ref_a_left是个左值引用
    int &&ref_a_right = std::move(a); // ref_a_right是个右值引用

    change(a); // 编译不过，a是左值，change参数要求右值
    change(ref_a_left); // 编译不过，左值引用ref_a_left本身也是个左值
    change(ref_a_right); // 编译不过，右值引用ref_a_right本身也是个左值

    change(std::move(a)); // 编译通过
    change(std::move(ref_a_right)); // 编译通过
    change(std::move(ref_a_left)); // 编译通过

    change(5); // 当然可以直接接右值，编译通过

    cout << &a << ' ';
    cout << &ref_a_left << ' ';
    cout << &ref_a_right;
    // 打印这三个左值的地址，都是一样的
}
```

看完后你可能有个问题，std::move会返回一个右值引用 `int &&` ，它是左值还是右值呢？ 从表达式 `int &&ref = std::move(a)` 来看，右值引用 `ref` 指向的必须是右值，所以move返回的 `int &&` 是个右值。所以右值引用既可能是左值，又可能是右值吗？ 确实如此：**右值引用既可以是左值也可以是右值，如果有名称则为左值，否则是右值。**

或者说：**作为函数返回值的 && 是右值，直接声明出来的 && 是左值。** 这同样也符合第一章对左值，右值的判定方式：其实引用和普通变量是一样的， `int &&ref = std::move(a)` 和 `int a = 5` 没有什么区别，等号左边就是左值，右边就是右值。

最后，从上述分析中我们得到如下结论：

1. **从性能上讲，左右值引用没有区别，传参使用左右值引用都可以避免拷贝。**
2. **右值引用可以直接指向右值，也可以通过std::move指向左值；而左值引用只能指向左值(const左值引用也能指向右值)。**
3. **作为函数形参时，右值引用更灵活。虽然const左值引用也可以做到左右值都接受，但它无法修改，有一定局限性。**

```
void f(const int& n) {
    n += 1; // 编译失败，const左值引用不能修改指向变量
}

void f2(int && n) {
    n += 1; // ok
}

int main() {
    f(5);
    f2(5);
}
```

3. 右值引用和std::move的应用场景

按上文分析，`std::move`只是类型转换工具，不会对性能有好处；右值引用在作为函数形参时更具灵活性，看上去还是挺鸡肋的。他们有什么实际应用场景吗？

3.1 实现移动语义

在实际场景中，右值引用和std::move被广泛用于在STL和自定义类中**实现移动语义，避免拷贝，从而提升程序性能。** 在没有右值引用之前，一个简单的数组类通常实现如下，有`构造函数`、`拷贝构造函数`、`赋值运算符重载`、`析构函数`等。深拷贝/浅拷贝在此不做讲解。

```
class Array {
public:
    Array(int size) : size_(size) {
        data = new int[size_];
    }

    // 深拷贝构造
    Array(const Array& temp_array) {
        size_ = temp_array.size_;
        data_ = new int[size_];
        for (int i = 0; i < size_; i ++) {
            data_[i] = temp_array.data_[i];
```

```
        }
    }

    // 深拷贝赋值
    Array& operator=(const Array& temp_array) {
        delete[] data_;

        size_ = temp_array.size_;
        data_ = new int[size_];
        for (int i = 0; i < size_; i ++) {
            data_[i] = temp_array.data_[i];
        }
    }

    ~Array() {
        delete[] data_;
    }

public:
    int *data_;
    int size_;
};
```

该类的拷贝构造函数、赋值运算符重载函数已经通过使用左值引用传参来避免一次多余拷贝了，但是内部实现要深拷贝，无法避免。 这时，有人提出一个想法：是不是可以提供一个 `移动构造函数` ，把被拷贝者的数据移动过来，被拷贝者后边就不要了，这样就可以避免深拷贝了，如：

```
class Array {
public:
    Array(int size) : size_(size) {
        data = new int[size_];
    }

    // 深拷贝构造
    Array(const Array& temp_array) {
        ...
    }

    // 深拷贝赋值
    Array& operator=(const Array& temp_array) {
        ...
    }

    // 移动构造函数，可以浅拷贝
    Array(const Array& temp_array, bool move) {
        data_ = temp_array.data_;
```

```
        size_ = temp_array.size_;
        // 为防止temp_array析构时delete data，提前置空其data_
        temp_array.data_ = nullptr;
    }


    ~Array() {
        delete [] data_;
    }

public:
    int *data_;
    int size_;
};
```

这么做有2个问题：

- 不优雅，表示移动语义还需要一个额外的参数(或者其他方式)。
- 无法实现！ `temp_array` 是个const左值引用，无法被修改，所以 `temp_array.data_ = nullptr;` 这行会编译不过。当然函数参数可以改成非const： `Array(Array& temp_array, bool move){...}` ，这样也有问题，由于左值引用不能接右值， `Array a = Array(Array(), true);` 这种调用方式就没法用了。

可以发现左值引用真是用的很不爽，**右值引用的出现解决了这个问题**，在STL的很多容器中，都实现了以**右值引用为参数**的 `移动构造函数` 和 `移动赋值重载函数` ，或者其他函数，最常见的如std::vector的 `push_back` 和 `emplace_back` 。参数为左值引用意味着拷贝，为右值引用意味着移动。

```
class Array {
public:
    ......

    // 优雅
    Array(Array&& temp_array) {
        data_ = temp_array.data_;
        size_ = temp_array.size_;
        // 为防止temp_array析构时delete data，提前置空其data_
        temp_array.data_ = nullptr;
    }


public:
    int *data_;
    int size_;
};
```

如何使用：

```
// 例1：Array用法
int main(){
    Array a;

    // 做一些操作
    .....

    // 左值a，用std::move转化为右值
    Array b(std::move(a));
}
```

3.2 实例：vector::push_back使用std::move提高性能

```
// 例2：std::vector和std::string的实际例子
int main() {
    std::string str1 = "aacasxs";
    std::vector<std::string> vec;

    vec.push_back(str1); // 传统方法，copy
    vec.push_back(std::move(str1)); // 调用移动语义的push_back方法，避免拷贝，str1会失去
    vec.emplace_back(std::move(str1)); // emplace_back效果相同，str1会失去原有值
    vec.emplace_back("axcsddcas"); // 当然可以直接接右值
}

// std::vector方法定义
void push_back (const value_type& val);
void push_back (value_type&& val);

void emplace_back (Args&&... args);
```

在vector和string这个场景，加个 `std::move` 会调用到移动语义函数，避免了深拷贝。

除非设计不允许移动，STL类大都支持移动语义函数，即 `可移动的` 。 另外，编译器会**默认**在用户自定义的 `class` 和 `struct` 中生成移动语义函数，但前提是用户没有主动定义该类的 `拷贝构造` 等函数(具体规则自行百度哈)。 **因此，可移动对象在<需要拷贝且被拷贝者之后不再被需要>的场景，建议使用 `std::move` 触发移动语义，提升性能。**

```
moveable_objecta = moveable_objectb;
改为：
moveable_objecta = std::move(moveable_objectb);
```

还有些STL类是 `move-only` 的，比如 `unique_ptr` ，这种类只有移动构造函数，因此只能移动(转移内部对象所有权，或者叫浅拷贝)，不能拷贝(深拷贝):

```
std::unique_ptr<A> ptr_a = std::make_unique<A>();

std::unique_ptr<A> ptr_b = std::move(ptr_a); // unique_ptr只有‘移动赋值重载函数‘，参数

std::unique_ptr<A> ptr_b = ptr_a; // 编译不通过
```

std::move本身只做类型转换，对性能无影响。 我们可以在自己的类中实现移动语义，避免深拷贝，充分利用右值引用和std::move的语言特性。

4. 完美转发 std::forward

和 `std::move` 一样，它的兄弟 `std::forward` 也充满了迷惑性，虽然名字含义是转发，但他并不会做转发，同样也是做类型转换.

与move相比，forward更强大，move只能转出来右值，forward都可以。

> std::forward<T>(u)有两个参数：T与 u。 a. 当T为左值引用类型时，u将被转换为T类型的左值； b. 否则u将被转换为T类型右值。

举个例子，有main，A，B三个函数，调用关系为： `main->A->B` ，建议先看懂*2.3节对左右值引用本身是左值还是右值的讨论*再看这里：

```
void B(int&& ref_r) {
    ref_r = 1;
}

// A、B的入参是右值引用
// 有名字的右值引用是左值，因此ref_r是左值
void A(int&& ref_r) {
    B(ref_r);  // 错误，B的入参是右值引用，需要接右值，ref_r是左值，编译失败

    B(std::move(ref_r)); // ok，std::move把左值转为右值，编译通过
    B(std::forward<int>(ref_r));  // ok，std::forward的T是int类型，属于条件b，因此会把r
}

int main() {
    int a = 5;
    A(std::move(a));
}
```

例2：

```
void change2(int&& ref_r) {
    ref_r = 1;
}

void change3(int& ref_l) {
    ref_l = 1;
}

// change的入参是右值引用
// 有名字的右值引用是 左值，因此ref_r是左值
void change(int&& ref_r) {
    change2(ref_r);  // 错误，change2的入参是右值引用，需要接右值，ref_r是左值，编译失败

    change2(std::move(ref_r)); // ok，std::move把左值转为右值，编译通过
    change2(std::forward<int &&>(ref_r));  // ok，std::forward的T是右值引用类型(int &&

    change3(ref_r); // ok，change3的入参是左值引用，需要接左值，ref_r是左值，编译通过
    change3(std::forward<int &>(ref_r)); // ok，std::forward的T是左值引用类型(int &)，
    // 可见，forward可以把值转换为左值或者右值
}

int main() {
    int a = 5;
    change(std::move(a));
}
```

上边的示例在日常编程中基本不会用到，`std::forward`最主要运于模版编程的参数转发中，想深入了解需要学习`万能引用(T &&)`和`引用折叠(eg:& && → ?)`等知识，本文就不详细介绍这些了。

如有错误，请指正！

编辑于 2020-12-11 10:29

C++11 指针（编程） C / C++