# [深入理解C++右值引用](https://www.cnblogs.com/kekec/p/10810507.html)

在C++中，常量、变量或表达式一定是左值（lvalue）或右值（rvalue）。

**左值**：非临时的（具名的，可在多条语句中使用，可以被取地址）。可以出现在等号的左边或右边。可分为非常量左值和常量左值。

|  |  |
| --- | --- |
| **类型** | **举例** |
| 非常量左值 | int ncLeft1 = 2;// ncLeft1为非常量左值  ++ncLeft1; // 前缀自增表达式返回值为非常量左值 |
| 常量左值 | const int cLeft1 = 2; // cLeft1为常量左值 |

**右值**：临时的（不具名的，只在当前语句中有效，不能取地址）。只能出现在等号的右边。可分为非常量右值和常量右值。

|  |  |
| --- | --- |
| **类型** | **举例** |
| 非常量右值 | int ncLeft1 = 1;  ncLeft1 + 1; // ncLeft1+1为非常量右值  ncLeft1++;// 后缀自增表达式返回值为非常量右值  add(1, 2); // int add(int,int)函数的返回值为非常量右值  [] {return 5; }(); // lambda表达式为非常量右值  CRect().GetWidth(); // 成员函数int GetWidth()返回值为非常量右值 |
| 常量右值 | const int& cLeftRef1 = 3; // 3为常量右值 |

**左值引用**：对左值的引用就是左值引用。可分为非常量左值引用和常量左值引用。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 引用类型 | 可被引用的类型 | | | | 注记 |
| 非常量左值 | 常量左值 | 非常量右值 | 常量右值 |
| 非常量左值引用 | // ncLeft1为非常量左值  int ncLeft1 = 2;  int& ncLeftRef1 = ncLeft1;  // 前缀自增表达式返回值为非常量左值  int& ncLeftRef2 = ++ncLeft1;  // ncLeftRef2为非常量左值  int& ncLeftRef3 = ncLeftRef2; | No | No | No | 该引用类型本身为非常量左值 |
| 常量左值引用 | // ncLeft1为非常量左值  int ncLeft1 = 2;  const int& cLeftRef1 = ncLeft1;  // 前缀自增表达式返回值为非常量左值  const int& cLeftRef2 = ++ncLeft1;  int& ncLeftRef1 = ++ncLeft1;  // ncLeftRef1为非常量左值  const int& cLeftRef3 = ncLeftRef1; | // cLeft1为常量左值  const int cLeft1 = 2;  const int& cLeftRef1 = cLeft1;  // cLeftRef1为常量左值  const int& cLeftRef2 = cLeftRef1; | int ncLeft1 = 1;  // ncLeft1+1为非常量右值  const int& cLeftRef1 = ncLeft1 + 1;  // 后缀自增表达式返回值为非常量右值  const int& cLeftRef2 = ncLeft1++;  // int add(int,int)函数的返回值为非常量右值  const int& cLeftRef3 = add(1, 2);  // lambda表达式为非常量右值  const int& cLeftRef4 = [] {return 5; }();  // 成员函数int GetWidth()返回值为非常量右值  const int& cLeftRef5 = CRect().GetWidth(); | // 3为常量右值  const int& cLeftRef1 = 3; | 该引用类型本身为常量左值 |

注：常量左值引用是“万能”的引用类型，可以绑定到所有类型的值，包括非常量左值、常量左值、非常量右值和常量右值。

**右值引用（Rvalue References）**：对右值的引用就是右值引用。可分为非常量右值引用和常量右值引用。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 引用类型 | 可被引用的类型 | | | | 注记 |
| 非常量左值 | 常量左值 | 非常量右值 | 常量右值 |
| 非常量右值引用 | No | No | int ncLeft1 = 1;  int&& ncRightRef1 = ncLeft1 + 1;   // 后缀自增表达式返回值为非常量右值  int&& ncRightRef2 = ncLeft1++;  // int add(int,int)函数的返回值为非常量右值  int&& ncRightRef3 = add(1, 2);  // lambda表达式为非常量右值  int&& ncRightRef4 = [] {return 5; }();  // 成员函数int GetWidth()返回值为非常量右值  int&& ncRightRef5 = CRect().GetWidth(); | No | 该引用类型本身为非常量左值 |
| 常量右值引用 | No | No | int ncLeft1 = 1;  // ncLeft1+1为非常量右值  const int&& cRightRef1 = ncLeft1 + 1;  // 后缀自增表达式返回值为非常量右值  const int&& cRightRef2 = ncLeft1++;  // int add(int,int)函数的返回值为非常量右值  const int&& cRightRef3 = add(1, 2);  // lambda表达式为非常量右值  const int&& cRightRef4 = [] {return 5; }();  // 成员函数int GetWidth()返回值为非常量右值  const int&& cRightRef5 = CRect().GetWidth(); | // 3为常量右值  const int&& cRightRef1 = 3; | 该引用类型本身为常量左值 |

为临时对象的右值，它的生命周期很短暂，一般在执行完当前这条表达式之后，就释放了。

通过将其赋值给右值引用，可以在不进行昂贵的拷贝操作的情况下被“续命”，让其生命周期与右值引用类型变量的生命周期一样长。

右值引用的两个基本特性：移动语义(Move Semantics)和完美转发(Perfect Forwarding)

**移动语义(Move Semantics)**

可将资源从一个对象转移到另一个对象；主要解决减少不必要的临时对象的创建、拷贝与销毁。

**移动构造函数**MyClass(Type&& a)：当构造函数参数是一个右值时，优先使用移动构造函数而不是拷贝构造函数MyClass(const Type& a)。

**移动赋值运算符**Type& operator = (Type&& a)：当赋值的是一个右值时，优先使用移动赋值而不是拷贝赋值运算符Type& operator = (const Type& a)。

[复制代码](javascript:void(0);)

#include <iostream>

#include <string>

#include <utility>

struct MyClass

{

std::string s;

MyClass(const char\* sz) : s(sz)

{

std::cout << "MyClass sz:" << sz << std::endl;

}

MyClass(const MyClass& o) : s(o.s)

{

std::cout << "copy construct!\n";

}

MyClass(MyClass&& o) noexcept : s(std::move(o.s))

{

std::cout << "move construct!\n";

}

MyClass& operator=(const MyClass& other) { // copy assign

std::cout << "copy assign!\n";

s = other.s;

return \*this;

}

MyClass& operator=(MyClass&& other) noexcept { // move assign

std::cout << "move assign!\n";

s = std::move(other.s);

return \*this;

}

static MyClass GetMyClassGo(const char\* sz)

{

MyClass o(sz); // 注意：可能会被NRVO优化掉

return o;

}

};

int main(int arg, char\* argv[])

{

MyClass a1("how");

MyClass a2("are");

a2 = a1; // copy assign 注：a1是一个左值

a2 = MyClass("you"); // move assign 注：MyClass("you")是一个右值

MyClass a3(a1); // copy construct 注：a1是一个左值

MyClass&& a4 = MyClass::GetMyClassGo("go"); // move construct 注：发生在MyClass::GetMyClassGo()内部

MyClass a5 = MyClass::GetMyClassGo("china"); // move construct两次 注：一次发生在MyClass::GetMyClassGo()内部；另一次发生在将返回值赋值给a5

return 0;

}

[复制代码](javascript:void(0);)

**使用std::move来实现移动语义**

将一个左值或右值强制转化为右值引用。   注：UE4中对应为MoveTemp模板函数

std::move（[en](https://en.cppreference.com/w/cpp/utility/move)  [chs](https://zh.cppreference.com/w/cpp/utility/move" \t "_blank)）并不会移动任何东西，只是将对象的状态或者所有权从一个对象转移到另一个对象。注：只是转移，没有内存的搬迁或者内存拷贝。

① 基本类型（如：int、double等）被std::move移动后，其数值不会发生变化

② 复合类型被std::move移动后，处于一个未定义，但有效的状态（大部分成员函数仍有意义）例如：标准库中的容器类对象被移动后，会变成空容器

**完美转发(Perfect Forwarding)**

针对模板函数，使用全能引用将一组参数原封不动的传递给另一个函数。

原封不动指：左值、右值、是否为const均不变。带来如下3方面好处：

① 保证左值、右值的属性

② 避免不必要的拷贝操作

③ 避免模版函数需要为左值、右值、是否为const的参数来实现不同的重载

**全能引用**（universal references、转发引用）是一种特殊的模板引用类型，采用右值引用的语法形式（但它并不是右值引用）。如：template <class T> void func(T&& t) {}

T&& t在发生自动类型推断的时候，它是未定的引用类型（universal references），T取决于传入的参数t是右值还是左值。右值经过T&&变为右值引用，而左值经过T&&变为左值引用。

std::move就是使用全能引用实现的。其定义如下：

[复制代码](javascript:void(0);)

template <typename T>

typename remove\_reference<T>::type&& move(T&& t)

{

return static\_cast<typename remove\_reference<T>::type &&>(t);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

std::remove\_reference功能为去除类型中的引用

std::remove\_reference<T &>::type ---> T

std::remove\_reference<T &&>::type ---> T

std::remove\_reference<T>::type ---> T

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

//原始的，最通用的版本

template <typename T> struct remove\_reference{

typedef T type; //定义T的类型别名为type

};

//部分版本特例化，将用于左值引用和右值引用

template <class T> struct remove\_reference<T&> //左值引用

{ typedef T type; }

template <class T> struct remove\_reference<T&&> //右值引用

{ typedef T type; }

[复制代码](javascript:void(0);)

① 当t为左值时，展开为：U&& move(U& t)    注：右值引用类型变量也是左值

② 当t为右值时，展开为：U&& move(U&& t)

最后，通过static\_cast<>进行强制类型转换返回右值引用。注：static\_cast之所以能使用类型转换，是通过remove\_refrence::type模板移除T&&，T&的引用，获取具体类型T（模板偏特化）。

**引用折叠**

规律：含左值引用就是左值引用，否则就是右值引用

|  |  |  |
| --- | --- | --- |
| 不同组合情况 | 声明类型 | 折叠类型 |
| 引用的引用 | &  & | & |
| 右值引用的引用 | &&  & | & |
| 引用的右值引用 | &  && | & |
| 右值引用的右值引用 | &&  && | && |

**使用std::forward实现参数的完美转发**。其定义如下（[en](https://en.cppreference.com/w/cpp/utility/forward)  [chs](https://zh.cppreference.com/w/cpp/utility/forward" \t "_blank)）：

[复制代码](javascript:void(0);)

template <typename T>

T&& forward(remove\_reference\_t<T>& arg) // forward an lvalue as either an lvalue or an rvalue

{

return static\_cast<T&&>(arg);

}

template <typename T>

T&& forward(remove\_reference\_t<T>&& arg) // forward an rvalue as an rvalue

{

static\_assert(!is\_lvalue\_reference\_v<T>, "bad forward call");

return static\_cast<T&&>(arg);

}

[复制代码](javascript:void(0);)

最后，通过static\_cast<>进行引用折叠，并强制类型转换后，实现原封不动转发参数。   注：UE4中对应为Forward模板函数

[复制代码](javascript:void(0);)

void bar(int& a, int&& b)

{

int c = a + b;

}

void func(int a, int&& b)

{

int c = a + b;

}

template <typename A, typename B>

void foo(A&& a, B&& b) { // a, b为左值引用或右值引用

bar(std::forward<A>(a), std::forward<B>(b)); // 在std::forward转发前后，参数a，b的类型完全不变

}

int main(int arg, char\* argv[])

{

int a = 10;

foo(a, 20); // 展开为void foo(int& a, int&& b)，经过std::forward完美转发后，会调用到void bar(int& a, int&& b)函数

func(std::forward<int>(a), std::forward<int&&>(30)); // 经过std::forward完美转发后，会调用到void func(int a, int&& b)函数

return 0;

}

[复制代码](javascript:void(0);)

**参考**

[Value categories, and references](https://docs.microsoft.com/en-us/windows/uwp/cpp-and-winrt-apis/cpp-value-categories)

[std::move原理实现与用法总结](https://blog.csdn.net/daaikuaichuan/article/details/88371948)

[Lvalues 和 Rvalues (C++)](https://docs.microsoft.com/zh-cn/cpp/cpp/lvalues-and-rvalues-visual-cpp?view=vs-2019)