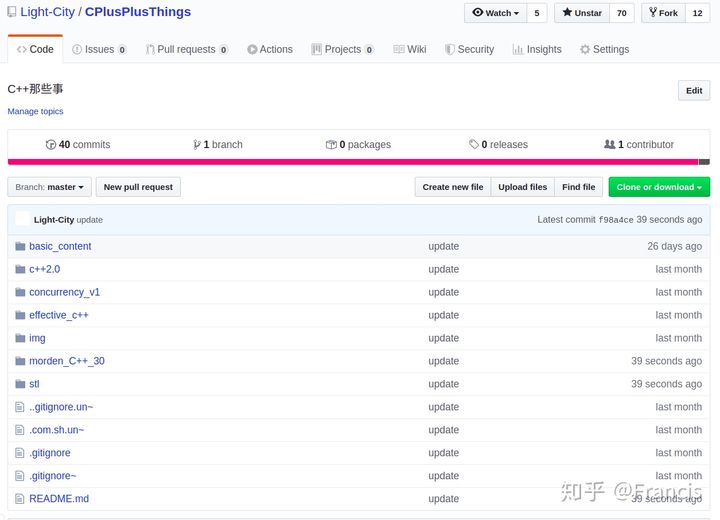
## **现代C++之万能引用、完美转发、引用折叠**

## **0.导语**

**个人公众号：guangcity，文章中代码存放在个人github仓库中，《C++那些事》**



[https://github.com/Light-City/CPlusPlusThings​github.com/Light-City/CPlusPlusThings](https://link.zhihu.com/?target=https://github.com/Light-City/CPlusPlusThings" \t "https://zhuanlan.zhihu.com/p/_blank)

不知道大家有没有听说过Scott Meyers，这位是C++的一位巨佬，知名的《Effective C++》与《Effective Modern C++》都是他编写的，bla bla...

这篇文章是翻译isocpp上的一篇文章以及学习极客时间第三讲的内容、stackoverflow等网站整合起来的，全文1w多字，还望收藏下来慢慢体会。

在谈到万能引用这里,Scott Meyers在本文中详细阐述了所谓的万能引用，同时也在《Effective Modern C++》中详细阐述，大家可以去看看，好了，开始正文。

T&& Doesn’t Always Mean “Rvalue Reference”  
by Scott Meyers

## **1.问题引入**

可能右值引用(rvalue references)是C++11里面最重要的新特性了。移动语义和完美转发都建立在它的基础之上。(如果你不熟悉右值引用基础, 移动语义, 或是完美转发, 再继续阅读本文之前你可能需要先看看 [Thomas Becker’s overview](https://link.zhihu.com/?target=http://thbecker.net/articles/rvalue_references/section_01.html" \t "https://zhuanlan.zhihu.com/p/_blank) )。

[http://thbecker.net/articles/rvalue\_references/section\_01.html](https://link.zhihu.com/?target=http://thbecker.net/articles/rvalue_references/section_01.html" \t "https://zhuanlan.zhihu.com/p/_blank)

从语法上来看，声明右值引用看起来和声明"普通"的引用(现在被称为左值引用(lvalue references))很像，只不过你需要用&&而不是&。下面这个函数需要一个类型为rvalue-reference-to-Widget:的参数:

**void** **f**(Widget**&&** param);

假设右值引用是使用&&声明的，那么假设类型声明中出现&& 表示右值引用似乎是合理的。事实并非如此:

Widget**&&** var1 **=** someWidget; *// here, “&&” means rvalue reference* **auto&&** var2 **=** var1; *// here, “&&” does not mean rvalue reference* **template<typename** T**>void** f(std**::**vector**<**T**>&&** param); *// here, “&&” means rvalue reference* **template<typename** T**>void** f(T**&&** param); *// here, “&&”does not mean rvalue reference*

在本文当中，我会对类型声明中 “&&” 可能具有的两种含义进行阐释，讲解如何区分它们，并且会引入一个新术语以便在交流的时候清楚的表明在当前说的“&&”是哪种含义。正确的区分这两种含义非常重要，因为如果你看到“&&”就以为是右值引用的话，你可是会误读很多c++11代码的。

## **2.引入万能引用**

这个问题的本质实际上是，类型声明当中的“&&”有的时候意味着rvalue reference，但有的时候意味着rvalue reference *或者* lvalue reference。因此，源代码当中出现的 “&&” 有可能是 “&” 的意思，即是说，语法上看着像rvalue reference (“&&”)，但实际上却代表着一个lvalue reference (“&”)。在这种情况下，此种引用比lvalue references 或者 rvalue references都要来的更灵活。

Rvalue references只能绑定到右值上，lvalue references除了可以绑定到左值上，在**某些条件**下还可以绑定到右值上。[1] 这里某些条件绑定右值为：常左值引用绑定到右值，非常左值引用不可绑定到右值！

例如：

string **&**s **=** "asd"; *// error***const** string **&**s **=** "asd"; *// ok*

规则简化如下：

左值引用 {左值}

右值引用 {右值}

常左值引用 {右值}

相比之下，声明中带 “&&” 的，可能是lvalue references 或者 rvalue references 的引用可以绑定到任何东西上。这种引用灵活也忒灵活了，值得单独给它们起个名字。我称它们为 *universal references*(万能引用或转发引用、通用引用)。

拓展：在资料[6]中提到了const的重要性!

例如：

string **f**() { **return** "abc"; }​**void** g() {

**const** string **&**s **=** f(); *// still legal?* cout **<<** s **<<** endl;}

上面g函数中合法？

答案是合法的，原因是s是个左值，类型是常左值引用，而f()是个右值，前面提到常左值引用可以绑定到右值！所以合法，当然把const去掉，便是不合法！

## **3.万能引用出现场合**

到底 “&&” 什么时候才意味着一个universal reference呢(即，代码当中的“&&”实际上可能是 “&”)，具体细节还挺棘手的，所以这些细节我推迟到后面再讲。现在，我们还是先集中精力研究下下面的经验原则，因为你在日常的编程工作当中需要牢记它：

If a variable or parameter is declared to have type **T&&** for some **deduced type** T, that variable or parameter is a *universal reference*.  
如果一个变量或者参数被声明为**T&&**，其中T是**被推导**的类型，那这个变量或者参数就是一个*universal reference*。

**"T需要是一个被推导类型"**这个要求限制了universal references的出现范围。

在实践当中，几乎所有的universal references都是函数模板的参数。因为auto声明的变量的类型推导规则本质上和模板是一样的，所以使用auto的时候你也可能得到一个universal references。

这些在生产代码中并不常见，但我在本文里给出了一些例子，因为由auto声明的universal reference看着没有模板的那么啰嗦。在本文的[Nitty Gritty Details section](https://link.zhihu.com/?target=https://isocpp.org/blog/2012/11/universal-references-in-c11-scott-meyers%23NittyGrittyDetails" \t "https://zhuanlan.zhihu.com/p/_blank)，

[https://isocpp.org/blog/2012/11/universal-references-in-c11-scott-meyers#NittyGrittyDetails](https://link.zhihu.com/?target=https://isocpp.org/blog/2012/11/universal-references-in-c11-scott-meyers%23NittyGrittyDetails" \t "https://zhuanlan.zhihu.com/p/_blank)

我会讲解说明使用typedef和decltype的时候也可能会出现universal references，但在我们讲解这些繁琐的细节之前，我们可以暂时认为universal references只会出现在模板参数和由auto声明的变量当中。

一个universal reference必须具有形如T&&，这个约束比它看起来要重要得多，但是我们稍后再对这一点进行详细的研究。现在，就先把这个约束记在脑子里吧。

和所有的引用一样，你必须对universal references进行初始化，而且正是universal reference的initializer决定了它到底代表的是lvalue reference 还是 rvalue reference:

* 如果用来初始化universal reference的表达式是一个左值，那么universal reference就变成lvalue reference。
* 如果用来初始化universal reference的表达式是一个右值，那么universal reference就变成rvalue reference。

上述可以根据下面代码例子理解：

**template<typename** T**>void** f(T**&&** param);

假设你是initializer：

**int** a;f(a); *// 传入左值,那么上述的T&& 就是lvalue reference,也就是左值引用绑定到了左值*f(1); *// 传入右值,那么上述的T&& 就是rvalue reference,也就是右值引用绑定到了左值*

## **4.理解左值与右值**

### **4.1 精简版**

**只有在你能区分左值和右值的前提下**，这个信息才有用。想要对这些术语进行精确定义是一件很难的事(c++11标准基本上是通过举例来说明一个表达式是否是一个lvalue还是rvalue的)，但实践当中，下面的定义就足够了。

* 如果你可以对一个表达式取地址，那这个表达式就是个lvalue。
* 如果一个表达式的类型是一个lvalue reference (例如, T& 或 const T&, 等.)，那这个表达式就是一个lvalue。
* 其它情况，这个表达式就是一个rvalue。从概念上来讲(通常实际上也是这样)，rvalue对应于临时对象，例如函数返回值或者通过隐式类型转换得到的对象，大部分字面值(e.g., 10 and 5.3)也是rvalues。

### **4.2 完整版**

实际上，上述不太完整，标准里的定义实际更复杂，规定了下面这些值类别（value categories）：



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

左值(lvalue)

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
* 返回左值引用的表达式  
  如 ++x、x = 1、cout << ' '  
  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; // error  
  cout << "(cout << ' ').addr=" << &(cout << ' ') << endl;
* **字符串字面量**是左值，而且是不可被更改的左值。字符串字面量并不具名，但是可以用&取地址所以也是左值。  
  如 "hello",在c++中是 char const [6] 类型，而在c中是 char [6] 类型  
  cout << "(\"hello\").addr=" << &("hello") << endl;
* 如果一个表达式的类型是一个lvalue reference (例如, T& 或 const T&, 等.)，那这个表达式就是一个lvalue。

纯右值(prvalue)

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
* 除字符串字面量之外的字面量如 42、true

将亡值(xvalue)

* 隐式或显式调用函数的结果，该函数的返回类型是对所返回对象类型的右值引用

**int&&** f(){

**return** 3;}

**int** **main**(){

f(); *// The expression f() belongs to the xvalue category, because f() return type is an rvalue reference to object type.*

**return** 0;}

* 对对象类型右值引用的转换

**int** **main**(){

**static\_cast<int&&>**(7); *// The expression static\_cast<int&&>(7) belongs to the xvalue category, because it is a cast to an rvalue reference to object type.* std**::**move(7); *// std::move(7) is equivalent to static\_cast<int&&>(7).*

**return** 0;}

* 类成员访问表达式，指定非引用类型的非静态数据成员，其中对象表达式是xvalue

**struct** **As**{

**int** i;};

As**&&** f(){

**return** **As**();}

**int** **main**(){

f().i; *// The expression f().i belongs to the xvalue category, because As::i is a non-static data member of non-reference type, and the subexpression f() belongs to the xvlaue category.*

**return** 0;}

xvalue有标识符，所以也被称为lvalue。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。如下所示：

\_有标识符\_ \_无标识符号\_

/ X \

/ / \ \

| l | x | pr |

\ \ / /

\\_\_\_\_\_\_\_\_\_X\_\_\_\_\_\_\_\_/

gl r

### **4.3 生命周期延长**

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。临时对象生命周期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。

（1）无生命周期延长

**#include** <iostream>**using** **namespace** std;**class** **shape** {**public:**

shape() { cout **<<** "shape" **<<** endl; }

**virtual** **~**shape() {

cout **<<** "~shape" **<<** endl;

}};**class** **circle** **:** **public** shape {**public:**

circle() { cout **<<** "circle" **<<** endl; }

**~**circle() {

cout **<<** "~circle" **<<** endl;

}};**class** **triangle** **:** **public** shape {**public:**

triangle() { cout **<<** "triangle" **<<** endl; }

**~**triangle() {

cout **<<** "~triangle" **<<** endl;

}};**class** **rectangle** **:** **public** shape {**public:**

rectangle() { cout **<<** "rectangle" **<<** endl; }

**~**rectangle() {

cout **<<** "~rectangle" **<<** endl;

}};**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**() {

process\_shape(circle(), triangle());}

输出：

shapetriangleshapecircleprocess\_shape()result()**~**result()**~**circle**~**shape**~**triangle**~**shape

先构造triangle,在构造circle,这两个都继承自shape，所以前面都会先构造shape,后面依次，析构的时候最后构造的，最先析构。

（2）有生命周期延长

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。**

result **&&**r **=** process\_shape(circle(), triangle());

输出结果如下：

shapetriangleshapecircleprocess\_shape()result()**~**circle**~**shape**~**triangle**~**shape**~**result()

result析构被延到最后了。

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：

result **&&**r **=** std**::**move(process\_shape(circle(), triangle()));

输出结果回到无延迟的结果了。

### **4.4 生命周期延长应用**

生命周期延长可以被应用在析构函数上，当我们想要去继承某个基类的时候，这个基类往往会被声明为virtual，当不声明的话，子类便不会得到析构。如果想让这个子类对象的析构仍然是完全正常，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。

例如：

**class** **Base** {**public:**

Base() {

cout **<<** "Base()" **<<** endl;

}

**~**Base() {

cout **<<** "~Base()" **<<** endl;

}};

**class** **Derived** **:** **public** Base {**public:**

Derived() {

cout **<<** "Derived()" **<<** endl;

}

**~**Derived() {

cout **<<** "~Derived()" **<<** endl;

}};

Base **\***b1 **=** **new** Derived;**delete** b1;cout**<<**endl;Derived d;Base **&**b2 **=**d;

输出：

Base()Derived()**~**Base()

Base()Derived()**~**Derived()**~**Base()

大家可以发现，当把子类绑定到基类的时候，子类析构正常了，这便是生命周期延长的应用。

## **5.区分万能引用**

回头再来看下本文开头的代码:

Widget**&&** var1 **=** someWidget;**auto&&** var2 **=** var1;

你可以对var1取址，所以var1是一个lvalue。var2的类型声明是auto&&，所以它就是一个universal reference，并且因为它会被var1 (一个lvalue)初始化，进而，var2就变成了一个lvalue reference。如果草草略过这段代码，你可能就会以为var2是rvalue reference, 类型声明当中的 "&&" 会误导你得出这个结论。但实际上，当一个universal reference开始被lvalue初始化的时候，var2就变成了lvalue reference。就好像我们是这么声明var2的:

Widget**&** var2 **=** var1;

正如前面所说的，如果一个表达式的类型是lvalue reference，它就是lvalue。我们来看看下面这个例子:

std**::**vector**<int>** v;...**auto&&** val **=** v[0]; *// val becomes an lvalue reference (see below)*

val是universal reference，并且被v[0]初始化，即是说用调用std::vector<int>::operator[]的结果来初始化。这个函数返回vector元素的lvalue reference。[2]

因为所有的lvalue references都是lvalues，并且这个lvalue被用来初始化val，val就变成了lvalue reference，即使它的类型声明看起来像是rvalue reference。

我前面说universal reference在函数模板的参数中最常见。我们再来看看本文开头时给出的模板:

template<typename T>

void f(T&& param); // “&&” might mean rvalue reference

调用：

f(10); *// 10 is an rvalue***int** x **=** 10;f(x); *// x is an lvalue*

第一行：param 被字面值10初始化，因为你不能对字面值取址，所以10是一个rvalue。这就意味着上面对f的调用当中，universal reference param被一个rvalue初始化，所以 param 就变成了rvalue reference – 具体来讲，就是 int&&。

第三行：param被变量 x 初始化，因为你能对 x 取址，所以x是个lvalue。这就是说，这里对f的调用，universal reference param被一个lvalue初始化，因此param就变成lvalue reference – 准确的说，就是int&。

param 实质上就是一个*universal reference*。

还记得*只有在发生类型推导*的时候 “&&” 才代表 universal reference 吗。如果没有类型推导，就没有universal reference。这种时候，类型声明当中的“&&”总是代表着rvalue reference。因此:

**template<typename** T**>void** f(T**&&** param); *// deduced parameter type ⇒ type deduction;* *// && ≡ universal reference* **template<typename** T**>class** **Widget** {

...

Widget(Widget**&&** rhs); *// fully specified parameter type ⇒ no type deduction;* ... *// && ≡ rvalue reference*};

**template<typename** T1**>class** **Gadget** {

...

**template<typename** T2**>**

Gadget(T2**&&** rhs); *// deduced parameter type ⇒ type deduction;* ... *// && ≡ universal reference*};

**void** **f**(Widget**&&** param); *// fully specified parameter type ⇒ no type deduction;* *// && ≡ rvalue reference*

上面的例子没什么好说的。在每一个例子当中，如果你看到T&& (其中T是模板参数)，那这里就有类型推导，那T&&就是universal reference。如果你看到 “&&” 跟在一个具体的类型名后面 (e.g., Widget&&)，那它就是个rvalue reference。

我前面说过声明引用的时候必须用 “T&&”的形式才能获得一个universal reference。这个一个很重要的信息。再看看这段代码:

**template<typename** T**>void** f(std**::**vector**<**T**>&&** param); *// “&&” means rvalue reference*

这里，我们同时有类型推导和一个带“&&”的参数，但是参数确不具有 “T&&” 的形式，而是 “std::vector<t>&&”。其结果就是，参数就只是一个普通的rvalue reference，而不是universal reference。 Universal references只以 “T&&”的形式出现！即便是仅仅加一个const限定符都会使得“&&”不再被解释为universal reference:

**template<typename** T**>void** f(**const** T**&&** param); *// “&&” means rvalue reference*

现在， “T&&” 正是universal reference所需要的形式。这不是说你的模板参数非得要用T:

**template<typename** MyTemplateParamType**>void** f(MyTemplateParamType**&&** param); *// “&&” means universal reference*

有的时候你可以在函数模板的声明中看到T&&，但却没有发生类型推导。来看下std::vector的 push\_back 函数:[3]

**template** **<class** **T**, **class** **Allocator** **=** allocator**<**T**>** **>class** **vector** {**public:**

...

**void** push\_back(T**&&** x); *// fully specified parameter type ⇒ no type deduction;* ... *// && ≡ rvalue reference*};

这里, T 是模板参数, 并且push\_back接受一个``T&&, 但是这个参数却不是universal reference! 这怎么可能?

如果我们看看push\_back在类外部是如何声明的，这个问题的答案就很清楚了。我会假装std::vector的 Allocator 参数不存在，因为它和我们的讨论无关。我们来看看没Allocator参数的std::vector::push\_back:

**template** **<class** **T>void** vector**<**T**>::**push\_back(T**&&** x);

push\_back不能离开std::vector<T>这个类而独立存在。但如果我们有了一个叫做std::vector<T>的类，我们就已经知道了T是什么东西，那就没必要推导T。

举个例子可能会更好。如果我这么写:

Widget **makeWidget**(); *// factory function for Widget*std**::**vector**<**Widget**>** vw;...Widget w;vw.push\_back(makeWidget()); *// create Widget from factory, add it to vw*

代码中对 push\_back 的使用会让编译器实例化类 std::vector<Widget> 相应的函数。这个push\_back 的声明看起来像这样:

**void** std**::**vector**<**Widget**>::**push\_back(Widget**&&** x);

看到了没? 一旦我们知道了类是 std::vector<Widget>，push\_back的参数类型就完全确定了: 就是Widget&&。这里完全不需要进行任何的类型推导。

对比下 std::vector 的emplace\_back，它看起来是这样的:

**template** **<class** **T**, **class** **Allocator** **=** allocator**<**T**>** **>class** **vector** {**public:**

...

**template** **<class**... **Args>**

**void** emplace\_back(Args**&&**... args); *// deduced parameter types ⇒ type deduction;* ... *// && ≡ universal references*};

emplace\_back 看起来需要多个参数(Args和args的声明当中都有...)，但重点是每一个参数的类型都需要进行推导。函数的模板参数 Args 和类的模板参数T无关，所以即使我知道这个类具体是什么，比如说，std::vector<Widget>，但我们还是不知道emplace\_back的参数类型是什么。

我们看下在类std::vector<Widget>外面声明的 emplace\_back会更清楚的表明这一点 (我会继续忽略 Allocator 参数):

**template<class**... **Args>void** std**::**vector**<**Widget**>::**emplace\_back(Args**&&**... args);

## **6.表达式的左右值性与类型无关**

“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。

一个表达式的lvalueness (左值性)或者 rvalueness (右值性)和它的类型无关。

来看下 int。可以有lvalue的int (e.g., 声明为int的变量)，还有rvalue的int (e.g., 字面值10)。用户定义类型Widget等等也是一样的。

一个Widget对象可以是lvalue(e.g., a Widget 变量) 或者是rvalue (e.g., 创建Widget的工程函数的返回值)。

表达式的类型不会告诉你它到底是个lvalue还是rvalue。因为表达式的 lvalueness 或 rvalueness 独立于它的类型，我们就可以有一个 *lvalue*，但它的类型确是 *rvalue reference*，也可以有一个 *rvalue reference* 类型的 *rvalue* :

Widget **makeWidget**(); *// factory function for Widget* Widget**&&** var1 **=** makeWidget() *// var1 is an lvalue, but* *// its type is rvalue reference (to Widget)* Widget var2 **=** **static\_cast<**Widget**&&>**(var1); *// the cast expression yields an rvalue, but* *// its type is rvalue reference (to Widget)*

var1类别是左值，但它的类型是右值引用。static\_cast<Widget&&>(var1)表达式是个右值，但它的类型是右值引用。

把 lvalues (例如 var1) 转换成 rvalues 比较常规的方式是对它们调用std::move，所以 var2 可以像这样定义:

Widget var2 **=** std**::**move(var1); *// equivalent to above*

我最初的代码里使用 static\_cast 仅仅是为了显示的说明这个表达式的类型是个rvalue reference (Widget&&)。rvalue reference 类型的具名变量和参数是 lvalues。(你可以对他们取地址。)

我们再来看下前面提到的 Widget 和 Gadget 模板:

**template<typename** T**>class** **Widget** {

...

Widget(Widget**&&** rhs); *// rhs’s type is rvalue reference,* ... *// but rhs itself is an lvalue*};

**template<typename** T1**>class** **Gadget** {

...

**template** **<typename** T2**>**

Gadget(T2**&&** rhs); *// rhs is a universal reference whose type will* ... *// eventually become an rvalue reference or*}; *// an lvalue reference, but rhs itself is an lvalue*

在 Widget 的构造函数当中, rhs 是一个rvalue reference，**前面提到，右值引用只能被绑定到右值上**，所以我们知道它被绑定到了一个rvalue上面(i.e., 因此我们需要传递了一个rvalue给它)， 但是 rhs 本身是一个 lvalue，所以，当我们想要用到这个被绑定在 rhs 上的rvalue 的 rvalueness 的时候，我们就需要把 rhs 转换回一个rvalue。之所以我们想要这么做，是因为我们想将它作为一个移动操作的source，这就是为什么我们用 std::move将它转换回一个 rvalue。

类似地，Gadget 构造函数当中的rhs 是一个 universal reference,，所以它可能绑定到一个 lvalue 或者 rvalue 上，但是无论它被绑定到什么东西上，rhs 本身还是一个 lvalue。

如果它被绑定到一个 rvalue 并且我们想利用这个rvalue 的 rvalueness， 我们就要重新将 rhs 转换回一个rvalue。如果它被绑定到一个lvalue上，当然我们就不想把它当做 rvalue。

**一个绑定到universal reference上的对象可能具有 lvalueness 或者 rvalueness，正是因为有这种二义性**，所以催生了std::forward: 如果一个本身是 lvalue 的 universal reference 如果绑定在了一个 rvalue 上面，就把它重新转换为rvalue。函数的名字 (“forward”) 的意思就是，我们希望在传递参数的时候，可以保存参数原来的lvalueness 或 rvalueness，即是说把参数转发给另一个函数。

## **7.引用折叠和完美转发**

### **7.1 引用折叠之本质细节**

这个问题的核心是，C++11当中的一些构造会弄出来引用的引用，而C++不允许出现引用的引用。如果代码当中显示的出现了一个引用的引用，那代码就是不合法的:

Widget w1;...Widget**&** **&** w2 **=** w1; *// error! No such thing as “reference to reference”*

但是，有些情况下，在你对类型进行操作的时候可能会搞出来引用的引用，编译器如果对这种情况报错是不对的。我们从C++98/C++03标准的时候就知道这件事了。

在对一个 universal reference 的模板参数进行类型推导时候，同一个类型的 lvalues 和 rvalues 被推导为稍微有些不同的类型。具体来说，类型T的lvalues被推导为T&(i.e., lvalue reference to T)，而类型T的 rvalues 被推导为 T。(注意，虽然 lvalue 会被推导为lvalue reference，但 rvalues 却不会被推导为 rvalue references!) 我们来看下分别用rvalue和lvalue来调用一个接受universal reference的模板函数时会发生什么:

**template<typename** T**>void** f(T**&&** param);

...

**int** x;

...

f(10); *// invoke f on rvalue*f(x); *// invoke f on lvalue*

当用rvalue 10调用 f 的时候, T被推导为 int，实例化的 f 看起来像这样:

**void** **f**(**int&&** param); *// f instantiated from rvalue*

这里一切都OK。但是当我们用lvalue x 来调用 f 的时候，T 被推导为int&，而实例化的 f 就包含了一个引用的引用:

**void** **f**(**int&** **&&** param); *// initial instantiation of f with lvalue*

因为这里出现了引用的引用，这实例化的代码乍一看好像不合法，但是像– “f(x)” –这么写代码是完全合理的。为了避免编译器对这个代码报错，C++11引入了一个叫做“引用折叠”(reference collapsing)的规则来处理某些像模板实例化这种情况下带来的"引用的引用"的问题。

因为有两种类型的引用 (lvalue references 和 rvalue references)，那"引用的引用"就有四种可能的组合: lvalue reference to lvalue reference, lvalue reference to rvalue reference, rvalue reference to lvalue reference, 以及 rvalue reference to rvalue reference。

引用折叠只有两条规则:

* 一个 rvalue reference to an rvalue reference 会变成 (“折叠为”) 一个 rvalue reference.
* 所有其他种类的"引用的引用" (i.e., 组合当中含有lvalue reference) 都会折叠为 lvalue reference.

在用lvalue实例化 f 时，应用这两条规则，会生成下面的合法代码，编译器就是这样处理这个函数调用的:

**void** **f**(**int&** param); *// instantiation of f with lvalue after reference collapsing*

上面的内容精确的说明了一个 universal reference 是如何在经过类型推导和引用折叠之后，可以变为一个 lvalue reference的。实际上，universal reference 其实只是一个身处于引用折叠背景下的rvalue reference。

**当一个变量本身的类型是引用类型的时候，这里就有点难搞了。这种情况下，类型当中所带的引用就被忽略了**。例如:

**int** x;

...

**int&&** r1 **=** 10; *// r1’s type is int&&* **int&** r2 **=** x; *// r2’s type is int&*

在调用模板函数 f 的时候 r1 和 r2 的类型都被当做 int。这个扒掉引用的行为，和"universal references 在类型推导期间，lvalues 被推导为 T& ，rvalues 被推导为"T" 这条规则无关。所以，这么调用模板函数的时候:

**template<typename** T**>void** f(T **&&**param) {

**static\_assert**(std**::**is\_lvalue\_reference**<**T**>::**value, "T& is lvalue reference");

cout **<<** "T& is lvalue reference" **<<** endl;}

**int** **main**() {

**int** x;

**int** **&&**r1 **=** 10;

**int** **&**r2 **=** x;

f(r1);

f(r2);}

r1 和r2 的类型都被推导为 int&。这是为啥呢?

首先，r1 和 r2 的引用部分被去掉了(留下的只是 int)。然后，因为它们都是 lvalues 所以当调用 f，对 universal reference 参数进行类型推导的时候，得到的类型都是int&。

我前面已经说过，引用折叠只发生在“像是模板实例化这样的场景当中”。 声明auto变量是另一个这样的场景。推导一个universal reference的 auto 变量的类型，在本质上和推导universal reference的函数模板参数是一样的，所以类型 T 的lvalue被推导为 T&，类型 T 的rvalue被推导为 T。我们再来看一下本文开头的实例代码:

Widget**&&** var1 **=** someWidget; *// var1 is of type Widget&& (no use of auto here)* **auto&&** var2 **=** var1; *// var2 is of type Widget& (see below)*

var1 的类型是 Widget&&，但是它的 reference-ness 在推导 var2 类型的时候被忽略了;var1 这时候就被当做 Widget。

因为它是个lvalue，所以初始化一个universal reference(var2)的时候，var1 的类型就被推导成Widget&。在 var2 的定义当中将 auto 替换成Widget& 会生成下面的非法代码:

Widget**&** **&&** var2 **=** var1; *// note reference-to-reference*

而在引用折叠之后，就变成了:

Widget**&** var2 **=** var1; *// var2 is of type Widget&*

还有第三种发生引用折叠的场景，就是形成和使用 typedef 的时候。看一下这样一个类模板，

**template<typename** T**>class** **Widget** {

**typedef** T**&** LvalueRefType;

...};**int** **main**() {

Widget**<int&>** w;}

根据引用折叠的规则：

* 一个 rvalue reference to an rvalue reference 会变成 (“折叠为”) 一个 rvalue reference.
* 所有其他种类的"引用的引用" (i.e., 组合当中含有lvalue reference) 都会折叠为 lvalue reference.

我们知道T会被推导为lvalue reference，因此结果肯定是lvalue reference,对应于上述规则，我们来通过代码验证。

**template<typename** T**>class** **Widget** {

**typedef** T**&** LvalueRefType;

**typedef** T**&&** RvalueRefType;**public:**

**void** judge() {

**static\_assert**(std**::**is\_lvalue\_reference**<**LvalueRefType**>::**value, "LvalueRefType & is lvalue reference");

**static\_assert**(std**::**is\_lvalue\_reference**<**RvalueRefType**>::**value, "RvalueRefType & is lvalue reference");

cout **<<** "LvalueRefType and RvalueRefType is lvalue reference" **<<** endl;

}};**int** **main**() {

Widget**<int&>** w;}

输出：

LvalueRefType and RvalueRefType is lvalue reference

如果我们在应用引用的上下文中使用这个typedef，例如:

void f(Widget<int&>::LvalueRefType&& param);

在对 typedef 扩展之后会产生非法代码:

**void** **f**(**int&** **&&** param);

但引用折叠这时候又插了一脚进来，所以最终的声明会是这样:

**void** **f**(**int&** param);

最后还有一种场景会有引用折叠发生，就是使用 decltype。和模板和 auto 一样，decltype 对表达式进行类型推导时候可能会返回 T 或者 T&，然后decltype 会应用 C++11 的引用折叠规则。

好吧， decltype 的类型推导规则其实和模板或者 auto 的类型推导不一样。这里的细节过于晦涩，所以就放在 [Further Information section](https://link.zhihu.com/?target=http://isocpp.org/blog/2012/11/universal-references-in-c11-scott-meyers%23FurtherInformation" \t "https://zhuanlan.zhihu.com/p/_blank) 里讲解，但是我们需要注意这样一个区别，即 decltype 对一个具名的、非引用类型的变量，会推导为类型 T (i.e., 一个非引用类型)，在相同条件下，模板和 auto 却会推导出 T&。

还有一个重要的区别就是decltype 进行类型推导只依赖于 decltype 的表达式; 用来对变量进行初始化的表达式的类型(如果有的话)会被忽略。因此:

Widget w1, w2;

**auto&&** v1 **=** w1;

**decltype**(w1)**&&** v2 **=** w2;

v1本身是左值，根据auto&&知道为万能引用，因此v1被推导为指向w1的左值引用。

w2是左值，decltype(w1)推导为Widget，因此v2为右值引用，根据右值引用只能绑定到右值，这里却给了一个左值，因此不能编译！

### **7.2 示例与使用**

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，那 T&& 的结果仍然是左值引用——即 T& && 坍缩成了T&
* 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。那 T&& 的结果自然就是一个右值引用。

例如：

**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：

bar(shape**&&**)foo(**const** shape**&**）

bar中传入的是右值，调用bar的&&重载函数，但是对于void bar(shape&& s)来说，s本身是一个lvalue，所以在foo(s)后，仍旧调用的是&重载函数。

如果想要调用foo(shape&&)，可以：

foo(std**::**move(s)

或者：

foo(**static\_cast<**shape**&&>**(s)

再考虑下面这个例子：

**void** **foo**(**const** shape**&**){

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

puts("foo(shape&&)");}**template** **<typename** T**>void** bar(T**&&** s){

foo(std**::**forward**<**T**>**(s));}**int** **main**() {

circle temp;

bar(temp);

bar(circle());}

输出：

foo(**const** shape**&**)foo(shape**&&**)

上面提到过**一个绑定到universal reference上的对象可能具有 lvalueness 或者 rvalueness，正是因为有这种二义性**,所以催生了std::forward: 如果一个本身是 lvalue 的 universal reference 如果绑定在了一个 rvalue 上面，就把它重新转换为rvalue。函数的名字 (“forward”) 的意思就是，我们希望在传递参数的时候，可以保存参数原来的lvalueness 或 rvalueness，即是说把参数转发给另一个函数。

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）。

### **7.3 std::move()与std::forward()源码剖析**

在分析std::move()与std::forward()之前，先看看remove\_reference，下面是remove\_reference的实现：

**template<typename** \_Tp**>struct** **remove\_reference**{ **typedef** \_Tp type; };

*// 特化版本***template<typename** \_Tp**>struct** **remove\_reference<**\_Tp**&>**{ **typedef** \_Tp type; };

**template<typename** \_Tp**>struct** **remove\_reference<**\_Tp**&&>**{ **typedef** \_Tp type; };

remove\_reference的作用是去除T中的引用部分，只获取其中的类型部分。无论T是左值还是右值，最后只获取它的类型部分。

std::forward源码剖析

* 转发左值

**template<typename** \_Tp**>constexpr** \_Tp**&&**forward(**typename** std**::**remove\_reference**<**\_Tp**>::**type**&** \_\_t) **noexcept**{ **return** **static\_cast<**\_Tp**&&>**(\_\_t); }

先获得类型type，定义\_t为左值引用的左值变量，通过static\_cast进行强制转换。\_Tp&&会发生引用折叠，当\_Tp推导为左值引用，则折叠为\_Tp& &&，即\_Tp&，推导为右值引用，则为本身\_Tp&&,所以froward返回值与static\_cast处都为\_Tp&&。

* 转发右值

**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);}

不同与转发左值，\_t为右值引用的左值变量，除此之外中间加了一个断言，表示当不是左值的时候，也就是右值，才进行static\_cast转换。

std::move源码剖析

**template<typename** \_Tp**>** **constexpr** **typename** std**::**remove\_reference**<**\_Tp**>::**type**&&** move(\_Tp**&&** \_\_t) **noexcept** {

**return** **static\_cast<typename** std**::**remove\_reference**<**\_Tp**>::**type**&&>**(\_\_t); }

看完上述的std::forward，再看这个，就很简单了，参数处根据模板推导，得出左值引用与右值引用，所以\_\_t可能是\_Tp&或者\_Tp&&。std::move的功能是：

* 传递的是左值，推导为左值引用，仍旧static\_cast转换为右值引用。
* 传递的是右值，推导为右值引用，仍旧static\_cast转换为右值引用。

在返回处，直接范围右值引用类型即可。还是通过renive\_reference获得\_Tp类型，然后直接type&&即可。

所以std::remove\_reference<\_Tp>::type&&，就是一个右值引用，我们就知道了std::move干的事情了。

小结：

* 在《Effective Modern C++》中建议：**对于右值引用使用std::move，对于万能引用使用std::forward。**
* std::move()与std::forward()都仅仅做了类型转换而已。真正的移动操作是在移动构造函数或者移动赋值操作符中发生的。
* std::move()可以应用于左值(普通的变量int这些使用move与不使用move效果一样)，但这么做要谨慎。因为一旦“移动”了左值，就表示当前的值不再需要了，如果后续使用了该值，产生的行为是未定义。

最后给个练手的例子[5]：

**void** **overloaded**( **int** **const** **&**arg ) { std**::**cout **<<** "by lvalue\n"; }**void** **overloaded**( **int** **&&** arg ) { std**::**cout **<<** "by rvalue\n"; }

**template<** **typename** t **>***/\* "t &&" with "t" being template param is special, and adjusts "t" to be (for example) "int &" or non-ref "int" so std::forward knows what to do. \*/***void** forwarding( t **&&** arg ) {

std**::**cout **<<** "via std::forward: ";

overloaded( std**::**forward**<** t **>**( arg ) );

std**::**cout **<<** "via std::move: ";

overloaded( std**::**move( arg ) ); *// conceptually this would invalidate arg* std**::**cout **<<** "by simple passing: ";

overloaded( arg );}

**int** **main**() {

std**::**cout **<<** "initial caller passes rvalue:\n";

forwarding( 5 );

std**::**cout **<<** "initial caller passes lvalue:\n";

**int** x **=** 5;

forwarding( x );}

输出：

initial caller passes rvalue:via std**::**forward: by rvaluevia std**::**move: by rvalueby simple passing: by lvalueinitial caller passes lvalue:via std**::**forward: by lvaluevia std**::**move: by rvalueby simple passing: by lvalue

## **8.不要返回本地变量的引用**

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。

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。

例如：

**#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&)"

**<<** 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：

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

## **9.总结**

（1）在类型声明当中， “&&” 要不就是一个 rvalue reference ，要不就是一个 *universal reference* – 一种可以解析为lvalue reference或者rvalue reference的引用。对于某个被推导的类型T，universal references 总是以 T&& 的形式出现。

（2）*引用折叠*是 会让 universal references (其实就是一个处于引用折叠背景下的rvalue references ) 有时解析为 lvalue references 有时解析为 rvalue references 的根本机制。引用折叠只会在一些特定的可能会产生"引用的引用"场景下生效。 这些场景包括模板类型推导，auto 类型推导， typedef 的形成和使用，以及decltype 表达式。

（3）std::move与std::forward本质都是static\_cast转换，**对于右值引用使用std::move，对于万能引用使用std::forward。**std::move解决的问题是对于一个本身是左值的右值引用变量需要绑定到一个右值上，所以需要使用一个能够传递右值的工具，而std::move就干了这个事。而std::forward解决的问题是一个绑定到universal reference上的对象可能具有 lvalueness 或者 rvalueness，正是因为有这种二义性，所以催生了std::forward: 如果一个本身是 左值 的 万能引用如果绑定在了一个 右边值 上面，就把它重新转换为右值。函数的名字 (“forward”) 的意思就是。**我们希望在传递参数的时候，可以保存参数原来的lvalueness 或 rvalueness，即是说把参数转发给另一个函数。**

（4）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。

## **10.补充**

* 参考资料

[1] I discuss rvalues and their counterpart, lvalues, later in this article. The restriction on lvalue references binding to rvalues is that such binding is permitted only when the lvalue reference is declared as a reference-to-const, i.e., a const T&.

[2] I’m ignoring the possibility of bounds violations. They yield undefined behavior.

[3] std::vector::push\_back is overloaded. The version shown is the only one that interests us in this article.

【4】[c++ - What are rvalues, lvalues, xvalues, glvalues, and prvalues? - Stack Overflow](https://stackoverflow.com/questions/3601602/what-are-rvalues-lvalues-xvalues-glvalues-and-prvalues)

【5】. [c++ - What's the difference between std::move and std::forward - Stack Overflow](https://stackoverflow.com/questions/9671749/whats-the-difference-between-stdmove-and-stdforward)

【6】[GotW #88: A Candidate For the “Most Important const” – Sutter’s Mill (herbsutter.com)](https://herbsutter.com/2008/01/01/gotw-88-a-candidate-for-the-most-important-const/?important-const/)

【7】[Universal References in C++11 -- Scott Meyers : Standard C++ (isocpp.org)](https://isocpp.org/blog/2012/11/universal-references-in-c11-scott-meyers)

【8】pdf文稿