区分通用引用与右值引用

据说，真相使人自由，然而在特定的环境下，一个精心挑选的谎言也同样使人解放。这一节就是这样一个谎言。因为我们在和软件打交道，然而，让我们避开“谎言(lie)”这个词，不妨说，本节包含了一种“**抽象**(abstraction)”。

为了声明一个指向某个类型T的右值引用(Rvalue Reference), 你写下了`T&&`。由此，一个合理的假设是，当你看到一个`T&&`出现在源码中，你看到的是一个右值引用。唉，事情并不如此简单:

```
void f(Widget&& param);      //右值引用
Widget&& var1 = Widget();    //右值引用
auto&& var2 = var1;          //不是右值引用

template <typename T>
void f(std::vector<T>&& param); //右值引用

template <typename T>
void f(T&& param);           //不是右值引用
```

事实上，`T&&`有两种不同的意思。第一种，当然是右值引用。这种引用表现得正如你所期待的那样: 它们只绑定到右值上，并且它们主要的存在原因就是为了声明某个对象可以被移动。

`T&&`的第二层意思，是它既可以是一个右值引用，也可以是一个左值引用。这种引用在源码里看起来像右值引用（也即`T&&`),但是它们可以表现得它们**像是**左值引用(也即`T&`)。它们的二重性(dual nature)使它们既可以绑定到右值上(就像右值引用)，也可以绑定到左值上(就像左值引用)。 此外，它们还可以绑定到常量(const)和非常量(non-const)的对象上，也可以绑定到`volatile`和`non-volatile`的对象上，甚至可以绑定到即`const`又`volatile`的对象上。它们可以绑定到几乎任何东西。这种空前灵活的引用值得拥有自己的名字。我把它叫做**通用引用**(universal references)。（注: Item 25解释了`std::forward`几乎总是可以应用到通用引用上，并且在这本书即将出版之际，一些C++社区的成员已经开始将这种通用引用称之为**转发引用**(forwarding references))。

在两种情况下会出现通用引用。最常见的一种是函数模板参数，正如在之前的示例代码中所出现的例子:

```
    template <typename T>
    void f(T&& param);       //param是一个通用引用
```

第二种情况是`auto`声明符，包含从以上示例中取得的这个例子:

```
    auto&& val2 = var1;      //var2是一个通用引用
```

这两种情况的共同之处就是都存在**类型推导(type deduction)**。在模板`f`的内部，参数`param`的类型需要被推导，而在变量`var2`的声明中，`var2`的类型也需要被推导。同以下的例子相比较(同样来自于上面的示例代码)，下面的例子不带有类型推导。如果你看见`T&&`不带有类型推导，那么你看到的就是一个右值引用。

```
void f(Widget&& param);          //没有类型推导
                                 //param是一个右值引用
Widget&& var1 = Widget();        //没有类型推导
                                 //var1是一个右值引用
```

因为通用引用是引用，所以他们必须被初始化。一个通用引用的初始值决定了它是代表了右值引用还是左值引用。如果初始值是一个右值，那么通用引用就会是对应的右值引用，如果初始值是一个左值，那么通用引用就会是一个左值引用。对那些是函数参数的通用引用来说，初始值在调用函数的时候被提供:

```
template <typename T>
void f(T&& param);              //param是一个通用引用

Widget w;
f(w);                           //传递给函数f一个左值;参数param的类型
                                //将会是Widget&,也即左值引用

f(std::move(w));                //传递给f一个右值;参数param的类型会是
                                //Widget&&,即右值引用
```

对一个通用引用而言，类型推导是必要的,但是它还不够。声明引用的格式必须正确，并且这种格式是被限制的。它必须是准确的`T&&`。再看看之前我们已经看过的代码示例:

```
template <typename T>
void f(std::vector<T>&& param);     //param是一个右值引用
```

当函数`f`被调用的时候，类型`T`会被推导（除非调用者显式地指定它，这种边缘情况我们不考虑）。但是参数`param`的类型声明并不是`T&&`，而是一个`std::vector<T>&&`。这排除了参数`param`是一个通用引用的可能性。`param`因此是一个右值引用——当你向函数`f`传递一个左值时，你的编译器将会开心地帮你确认这一点:

```
std::vector<int> v;
f(v);                   //错误！不能将左值绑定到右值引用
```

即使是出现一个简单的`const`修饰符，也足以使一个引用失去成为通用引用的资格:

```
    template <typename T>
    void f(const T&& param);    //param是一个右值引用
```

如果你在一个模板里面看见了一个函数参数类型为`T&&`,你也许觉得你可以假定它是一个通用引用。错！这是由于在模板内部并不保证一定会发生类型推导。考虑如下`push_back`成员函数，来自`std::vector`:

```
template <class T,class Allocator = allocator<T>> //来自C++标准
    class vector
    {
        public:
        void push_back(T&& x);
        ...
    }
```

`push_back`函数的参数当然有资格成为一个通用引用，然而，在这里并没有发生类型推导。因为`push_back`在一个特有(particular)的`vector`实例化(instantiation)之前不可能存在，而实例化`vector`时的类型已经决定了`push_back`的声明。也就是说，

```
    std::vector<Widget> v;
```

将会导致`std::vector`模板被实例化为以下代码:

```
    class vector<Widget , allocagor<Widget>>
    {
        public:
        void push_back(Widget&& x);              // 右值引用
    }
```

现在你可以清楚地看到，函数`push_back`不包含任何类型推导。`push_back`对于`vector<T>`而言(有两个函数——它被重载了)总是声明了一个类型为指向`T`的右值引用的参数。

相反，`std::vector`内部的概念上相似的成员函数`emplace_back`，却确实包含类型推导:

```
template <class T,class Allocator = allocator<T>> //依旧来自C++标准
    class vector
    {
        public:
        template <class... Args>
        void emplace_back(Args&&... args);
        ...
    }
```

这儿，类型参数(type parameter)`Args`是独立于`vector`的类型参数之外的，所以`Args`会在每次`emplace_back`被调用的时候被推导(Okay,`Args`实际上是一个参数包(parameter pack),而不是一个类型参数，但是为了讨论之利，我们可以把它**当作**是一个类型参数)。

虽然函数`emplace_back`的类型参数被命名为`Args`，但是它仍然是一个通用引用，这补充了我之前所说的，通用引用的格式必须是`T&&`。 没有任何规定必须使用名字`T`。举个例子，如下模板接受一个通用引用，但是格式(`type&&`)是正确的，并且参数`param`的类型将会被推导（重复一次，不考虑边缘情况，也即当调用者明确给定参数类型的时候)。

```
template <typename MyTemplateType>          //param是通用引用
void someFunc(MyTemplateType&& param);
```

我之前提到，类型为`auto`的变量可以是通用引用。更准确地说，类型声明为`auto&&`的变量是通用引用，因为会发生类型推导，并且它们满足正确的格式要求(`T&&`)。`auto`类型的通用引用不如模板函数参数中的通用引用常见，但是它们在`C++11`中常常突然出现。而它们在`C++14`中出现地更多，因为`C++14`的匿名函数表达式(lambda expressions)可以声明`auto&&`类型的参数。举个例子，如果你想写一个`C++14`标准的匿名函数，来记录任意函数调用花费的时间，你可以这样:

```
    auto timeFuncInvocation =
    [](auto&& func, auto&&... params)           //C++14标准
    {
        start timer;
        std::forward<decltype(func)>(func)(     //对参数params调用func
            std::forward<delctype(params)>(params)...
        );
        stop timer and record elapsed time;
    };
```

如果你对位于匿名函数里的`std::forward<decltype(blah blah blah)>`反应是"What the ....!", 这只代表着你可能还没有读 Item 33。别担心。在本节，重要的事是匿名函数声明的`auto&&`类型的参数。`func`是一个通用引用，可以被绑定到任何可被调用的对象，无论左值还是右值。`args`是**0个**或者多个通用引用（也就是说，它是个通用引用参数包（a universal reference parameter pack)），它可以绑定到任意数目、任意类型的对象上。
多亏了`auto`类型的通用引用，函数`timeFuncInvocation`可以对**近乎任意**(pretty-much any)函数进行计时。(如果你想知道*任意(any)*和*近乎任意(pretty-much any*的区别，往后翻到 Item 30)。

牢记整个本小节——通用引用的基础——是一个谎言,uhh,一个“抽象”。隐藏在其底下的真相被称为"**引用折叠(reference collapsing)**",小节Item 28致力于讨论它。但是这个真相并不降低该抽象的有用程度。区分右值引用和通用引用将会帮助你更准确地阅读代码（"究竟我眼前的这个`T&&`是只绑定到右值还是可以绑定任意对象呢？"），并且，当你在和你的合作者交流时，它会帮助你避免歧义（"在这里我在用一个通用引用，而非右值引用")。它也可以帮助你弄懂Item 25和26,它们依赖于右值引用和通用引用的区别。所以，拥抱这份抽象，陶醉于它吧。就像牛顿的力学定律（本质上不正确），比起爱因斯坦的相对论（这是真相）而言，往往更简单，更易用。所以这份通用引用的概念，相较于穷究引用折叠的细节而言，是更合意之选。

记住：

- 如果一个函数模板参数的类型为`T&&`，并且`T`需要被推导得知，或者如果一个对象被声明为`auto&&`，这个参数或者对象就是一个通用引用。
- 如果类型声明的形式不是标准的`type&&`，或者如果类型推导没有发生，那么`type&&`代表一个右值引用。
- 通用引用，如果它被右值初始化，就会对应地成为右值引用;如果它被左值初始化，就会成为左值引用。