Item28：理解引用折叠

Item23中指出，当参数传递给模板函数时，模板参数的类型是左值还是右值被推导出来。但是并没有提到只有当参数被声明为通用引用时，上述推导才会发生，但是有充分的理由忽略这一点：因为通用引用是Item24中才提到。回过头来看，通用引用和左值/右值编码意味着：

```
template<typename T>
void func(T&& param);
```

被推导的模板参数T将根据被传入参数类型被编码为左值或者右值。

编码机制是简单的。当左值被传入时，T被推导为左值。当右值被传入时，T被推导为非引用（请注意不对称性：左值被编码为左值引用，右值被编码为非引用），因此：

```
Widget widgetFactory(); // function returning rvalue
Widget w; // a variable(an lvalue)
func(w);  // call func with lvalue; T deduced to be Widget&
func(widgetFactory()); // call func with rvalue; T deduced to be Widget
```

上面的两种调用中，Widget被传入，因为一个是左值，一个是右值，模板参数T被推导为不同的类型。正如我们很快看到的，这决定了通用引用成为左值还是右值，也是`std::forward`的工作基础。

在我们更加深入`std::forward`和通用引用之前，必须明确在C++中引用的引用是非法的。不知道你是否尝试过下面的写法，编译器会报错：

```
int x;
...
auto& & rx = x;  //error! can't declare reference to reference
```

考虑下，如果一个左值传给模板函数的通用引用会发生什么：

```
template<typename T>
void func(T&& param);

func(w); // invoke func with lvalue; T deduced as Widget&
```

如果我们把推导出来的类型带入回代码中看起来就像是这样：

```
void func(Widget& && param);
```

引用的引用！但是编译器没有报错。我们从Item24中了解到因为通用引用param被传入一个左值，所以param的类型被推导为左值引用，但是编译器如何采用T的推导类型的结果，这是最终的函数签名？

```
void func(Widget& param);
```

答案是引用折叠。是的，禁止你声明引用的引用，但是编译器会在特定的上下文中使用，包括模板实例的例子。当编译器生成引用的引用时，引用折叠指导下一步发生什么。

存在两种类型的引用（左值和右值），所以有四种可能的引用组合（左值的左值，左值的右值，右值的右值，右值的左值）。如果一个上下文中允许引用的引用存在（比如，模板函数的实例化），引用根据规则折叠为单个引用：

> 如果任一引用为左值引用，则结果为左值引用。否则（即，如果引用都是右值引用），结果为右值引用

在我们上面的例子中，将推导类型Widget&替换模板func会产生对左值引用的右值引用，然后引用折叠规则告诉我们结果就是左值引用。

引用折叠是`std::forward`工作的一种关键机制。就像Item25中解释的一样，`std::forward`应用在通用引用参数上，所以经常能看到这样使用：

```
template<typename T>
void f(T&& fParam)
{
  ...  // do some work
  someFunc(std::forward<T>(fParam)); // forward fParam to someFunc
}
```

因为fParam是通用引用，我们知道参数T的类型将在传入具体参数时被编码。`std::forward`的作用是当传入参数为右值时，即T为非引用类型，才将fParam（左值）转化为一个右值。

`std::forward`可以这样实现：

```
template<typename T>
T&& forward(typename remove_reference<T>::type& param)
{
  return static_cast<T&&>(param);
}
```

这不是标准库版本的实现（忽略了一些接口描述），但是为了理解`std::forward`的行为，这些差异无关紧要。

假设传入到f的Widget的左值类型。T被推导为Widget&，然后调用`std::forward`将初始化为`std::forward<Widget&>`。带入到上面的`std::forward`的实现中：

```
Widget& && forward(typename remove_reference<Widget&>::type& param)
{
  return static_cast<Widget& &&>(param);
}
```

`std::remove_reference<Widget&>::type`表示Widget（查看Item9），所以`std::forward`成为：

```
Widget& && forward(Widget& param)
{
  return static_cast<Widget& &&>(param);
}
```

根据引用折叠规则，返回值和static_cast可以化简，最终版本的`std::forward`就是

```
Widget& forward(Widget& param)
{
  return static_cast<Widget&>(param);
}
```

正如你所看到的，当左值被传入到函数模板f时，`std::forward`转发和返回的都是左值引用。内部的转换不做任何事，因为param的类型已经是`Widget&`，所以转换没有影响。左值传入会返回左值引用。通过定义，左值引用就是左值，因此将左值传递给`std::forward`会返回左值，就像说的那样，完美转发。

现在假设一下，传递给f的是一个`Widget`的右值。在这个例子中，T的类型推导就是Widget。内部的`std::forward`因此转发`std::forward<Widget>`，带入回`std::forward`实现中：

```
Widget&& forward(typename remove_reference<Widget>::type& param)
{
  return static_cast<Widget&&>(param);
}
```

将`remove_reference`引用到非引用的类型上还是相同的类型，所以化简如下

```
Widget&& forward(Widget& param)
{
  return static_cast<Widget&&>(param);
}
```

这里没有引用的引用，所以不需要引用折叠，这就是最终版本。

从函数返回的右值引用被定义为右值，因此在这种情况下，`std::forward`会将f的参数fParam（左值）转换为右值。最终结果是，传递给f的右值参数将作为右值转发给someFunc，完美转发。

在C++14中，`std::remove_reference_t`的存在使得实现变得更简单：

```
template<typename T>  // C++ 14; still in namepsace std
T&& forward(remove_reference_t<T>& param)
{
  return static_cast<T&&>(param);
}
```

引用折叠发生在四种情况下。**第一**，也是最常见的就是模板实例化。**第二**，是auto变量的类型生成，具体细节类似模板实例化的分析，因为类型推导基本与模板实例化雷同（参见Item2）。考虑下面的例子：

```
template<typename T>
void func(T&& param);
Widget widgetFactory(); // function returning rvalue
Widget w; // a variable(an lvalue)
func(w); // call func with lvalue; T deduced to be Widget&
func(widgetFactory()); // call func with rvalue; T deduced to be Widget
```

在auto的写法中，规则是类似的：`auto&& w1 = w;`初始化`w1`为一个左值，因此为auto推导出类型`Widget&`。带回去就是`Widget& && w1 = w`，应用引用折叠规则，就是`Widget& w1 = w`,结果就是`w1`是一个左值引用。

另一方面，`auto&& w2 = widgetFactory();`使用右值初始化`w2`，非引用带回`Widget&& w2 = widgetFactory()`。没有引用的引用，这就是最终结果。

现在我们真正理解了Item24中引入的通用引用。通用引用不是一种新的引用，它实际上是满足两个条件下的右值引用：

- **通过类型推导将左值和右值区分**。T类型的左值被推导为&类型，T类型的右值被推导为T

- **引用折叠的发生**

通用引用的概念是有用的，因为它使你不必一定意识到引用折叠的存在，从直觉上判断左值和右值的推导即可。

我说了有四种情况会发生引用折叠，但是只讨论了两种：模板实例化和auto的类型生成。**第**三，是使用typedef和别名声明（参见Item9），如果，在创建或者定义typedef过程中出现了引用的引用，则引用折叠就会起作用。举例子来说，假设我们有一个Widget的类模板，该模板具有右值引用类型的嵌入式typedef：

```
template<typename T>
class Widget {
public:
  typedef T&& RvalueRefToT;
  ...
};
```

假设我们使用左值引用实例化Widget：

```
Widget<int&> w;
```

就会出现

```
typedef int& && RvalueRefToT;
```

引用折叠就会发挥作用：

```
typedef int& RvalueRefToT;
```

这清楚表明我们为typedef选择的name可能不是我们希望的那样：RvalueRefToT是左值引用的typedef，当使用Widget被左值引用实例化时。

最后，**也是第四**种情况是，decltype使用的情况，如果在分析decltype期间，出现了引用的引用，引用折叠规则就会起作用（关于decltype，参见Item3）

需要记住的事

- 引用折叠发生在四种情况：模板实例化；auto类型推导；typedef的创建和别名声明；decltype
- 当编译器生成了引用的引用时，结果通过引用折叠就是单个引用。有左值引用就是左值引用，否则就是右值引用
- 通用引用就是通过类型推导区分左值还是右值，并且引用折叠出现的右值引用