对于std::forward的auto&&形参使用decltype

泛型lambda(generic lambdas)是C++14中最值得期待的特性之一——因为在lambda的参数中可以使用auto关键字。这个特性的实现是非常直截了当的：即在闭包类中的operator()函数是一个函数模版。例如存在这么一个lambda：

```
auto f = [](auto x){ return func(normalize(x)); };
```

对应的闭包类中的函数调用操作符看来就变成这样：

```
class SomeCompilerGeneratedClassName { public:
  template<typename T>
  auto operator()(T x) const
  { return func(normalize(x)); }
  ...
};
```

在这个样例中，lambda对变量`x`做的唯一一件事就是把它转发给函数`normalize`。如果函数`normalize`对待左值右值的方式不一样，这个lambda的实现方式就不大合适了，因为即使传递到lambda的实参是一个右值，lambda传递进去的形参总是一个左值。

实现这个lambda的正确方式是把`x`完美转发给函数`normalize`。这样做需要对代码做两处修改。首先，x需要改成通用引用，其次，需要使用`std::forward`将`x`转发到函数`normalize`。实际上的修改如下：

```
auto f = [](auto&& x)
    { return func(normalize(std::forward<???>(x))); };
```

在理论和实际之间存在一个问题：你传递给`std::forward`的参数是什么类型，就决定了上面的`???`该怎么修改。

一般来说，当你在使用完美转发时，你是在一个接受类型参数为`T`的模版函数里，所以你可以写`std::forward<T>`。但在泛型lambda中，没有可用的类型参数`T`。在lambda生成的闭包里，模版化的`operator()`函数中的确有一个`T`，但在lambda里却无法直接使用它。

前面item28解释过在传递给通用引用的是一个左值，那么它会变成左值引用。传递的是右值就会变成右值引用。这意味着在这个lambda中，可以通过检查`x`的类型来检查传递进来的实参是一个左值还是右值，decltype就可以实现这样的效果。传递给lambda的是一个左值，`decltype(x)`就能产生一个左值引用；如果传递的是一个右值，`decltype(x)`就会产生右值引用。

Item28也解释过在调用`std::forward`，传递给它的类型类型参数是一个左值引用时会返回一个左值；传递的是一个非引用类型时，返回的是一个右值引用，而不是常规的非引用。在前面的lambda中，如果x绑定的是一个左值引用，`decltype(x)`就能产生一个左值引用；如果绑定的是一个右值，`decltype(x)`就会产生右值引用，而不是常规的非引用。

在看一下Item28中关于`std::forward`的C++14实现：

```
template<typename T> // in namespace
T&& forward(remove_reference_t<T>& param) // std
{
    return static_cast<T&&>(param);
}
```

如果用户想要完美转发一个Widget类型的右值时，它会使用Widget类型（非引用类型）来示例化`std::forward`，然后产生以下的函数：

```
Widget&& forward(Widget& param)
{                                                                    //
instantiation of
    return static_cast<Widget&&>(param); // std::forward when
}                                                                    // T is
Widget
```

思考一下如果用户代码想要完美转发一个Widget类型的右值，但没有遵守规则将T指定为非引用类型，而是将T指定为右值引用，这回发生什么？思考将T换成`Widget`如何，在`std::forward`实例化、应用了`remove_reference_t`后，音乐折叠之前，这是产生的代码：

```
Widget&& && forward(Widget& param)        // instantiation of
{                                                                  //
std::forward when
    return static_cast<Widget&& &&>(param); // T is Widget&&
}                                                                        //
(before reference-collapsing)
```

应用了引用折叠之后，代码会变成：

```
Widget&& forward(Widget& param)        // instantiation of
{                                         // std::forward when
    return static_cast<Widget&&>(param); // T is Widget&&
}                                         // (before reference-collapsing)
```

对比发现，用一个右值引用去实例化`std::forward`和用非引用类型去实例化产生的结果是一样的。

那是一个很好的消息，引用当传递给lambda形参x的是一个右值实参时，`decltype(x)`可以产生一个右值引用。前面已经确认过，把一个左值传给lambda时，`decltype(x)`会产生一个可以传给`std::forward`的常规类型。而现在也验证了对于右值，把`decltype(x)`产生的类型传递给`std::forward`的类型参数是非传统的，不过它产生的实例化结果与传统类型相同。所以无论是左值还是右值，把`decltype(x)`传递给`std::forward`都能得到我们想要的结果，因此lambda的完美转发可以写成：

```
auto f =
    [](auto&& param)
    {
        return
          func(normalize(std::forward<decltype(pram)>(param)));
    };
```

再加上6个点，就可以让我们的lambda完美转发接受多个参数了，因为C++14中的lambda参数是可变的：

```
auto f =
    [](auto&&... params)
    {
        return
          func(normalized(std::forward<decltype(params)>(params)...));
    };
```

要谨记的是:

- 对`auto&&`参数使用`decltype`来（`std::forward`）转发参数;