Item 12:使用override声明重载函数

条款12:使用override声明重载函数

在C++面向对象的世界里，涉及的概念有类，继承，虚函数。这个世界最基本的概念是派生类的虚函数重写基类同名函数。令人遗憾的是虚函数重写可能一不小心就错了。给人感觉语言的这一部分设计观点是墨菲定律不是用来遵守的，只是值得尊敬的。

鉴于"重写"听起来像"重载"，尽管两者完全不相关，下面就通过一个派生类和基类来说明什么是虚函数重写:

```
class Base {
public:
    virtual void doWork(); // 基类虚函数
    …
};
class Derived: public Base {
    public:
    virtual void doWork(); // 重写Base::doWork(这里"virtual"是可以省略的)
    …
};
std::unique_ptr<Base> upb =          // 创建基类指针
    std::make_unique<Derived>();     // 指向派生类对象
                                     // 关于std: : make_unique请
…                                    // 参见Item1
upb->doWork();  // 通过基类指针调用doWork
                // 实际上是派生类的doWork
                // 函数被调用
```

要想重写一个函数，必须满足下列要求:

- 基类函数必须是`virtual`
- 基类和派生类函数名必须完全一样（除非是析构函数
- 基类和派生类函数参数必须完全一样
- 基类和派生类函数常量性(constness)必须完全一样
- 基类和派生类函数的返回值和异常说明(exception specifications)必须兼容
  除了这些C++98就存在的约束外，C++11又添加了一个:
- 函数的引用限定符（reference qualifiers）必须完全一样。成员函数的引用限定符是C++11很少抛头露脸的特性，所以如果你从没听过它无需惊讶。它可以限定成员函数只能用于左值或者右值。成员函数不需要`virtual`也能使用它们:

```
class Widget {
public:
    …
    void doWork() &; //只有*this为左值的时候才能被调用
    void doWork() &&; //只有*this为右值的时候才能被调用
};
…
Widget makeWidget();     // 工厂函数（返回右值）
Widget w;                // 普通对象（左值）
…
w.doWork(); // 调用被左值引用限定修饰的Widget::doWork版本
            // (即Widget::doWork &)
```

```
makeWidget().doWork(); // 调用被右值引用限定修饰的Widget::doWork版本
                       // (即Widget::doWork &&)
```

后面我还会提到引用限定符修饰成员函数，但是现在，只需要记住如果基类的虚函数有引用限定符，派生类的重写就必须具有相同的引用限定符。如果没有，那么新声明的函数还是属于派生类，但是不会重写父类的任何函数。

这么多的重写需求意味着哪怕一个小小的错误也会造成巨大的不同。
代码中包含重写错误通常是有效的，但它的意图不是你想要的。因此你不能指望当你犯错时编译器能通知你。比如，下面的代码是完全合法的，咋一看，还很有道理，但是它包含了非虚函数重写。你能识别每个case的错误吗，换句话说，为什么派生类函数没有重写同名基类函数?

```
class Base {
public:
    virtual void mf1() const;
    virtual void mf2(int x);
    virtual void mf3() &;
    void mf4() const;
};
class Derived: public Base {
public:
    virtual void mf1();
    virtual void mf2(unsigned int x);
    virtual void mf3() &&;
    void mf4() const;
};
```

需要一点帮助吗?

- `mf1`在基类声明为`const`,但是派生类没有这个常量限定符
- `mf2`在基类声明为接受一个`int`参数，但是在派生类声明为接受`unsigned int`参数
- `mf3`在基类声明为左值引用限定，但是在派生类声明为右值引用限定
- `mf4`在基类没有声明为虚函数
  你可能会想，“哎呀，实际操作的时候，这些warnings都能被编译器探测到，所以我不需要担心。”可能你说的对，也可能不对。就我目前检查的两款编译器来说，这些代码编译时没有任何warnings，即使我开启了输出所有warnings（其他编译器可能会为这些问题的部分输出warnings，但不是全部）

由于正确声明派生类的重写函数很重要，但很容易出错，C++11提供一个方法让你可以显式的将派生类函数指定为应该是基类重写版本：将它声明为`override`。还是上面那个例子，我们可以这样做:

```
class Derived: public Base {
    public:
    virtual void mf1() override;
    virtual void mf2(unsigned int x) override;
    virtual void mf3() && override;
    virtual void mf4() const override;
};
```

代码不能编译，当然了，因为这样写的时候，编译器会抱怨所有与重写有关的问题。这也是你想要的，以及为什么要在所有重写函数后面加上`override`。使用`override`的代码编译时看起来就像这样（假设我们的目的是重写基类的所有函数）：

```
class Base {
public:
```

```
    virtual void mf1() const;
    virtual void mf2(int x);
    virtual void mf3() &;
    virtual void mf4() const;
};
class Derived: public Base {
public:
    virtual void mf1() const override;
    virtual void mf2(int x) override;
    virtual void mf3() & override;
    void mf4() const override; // 可以添加virtual，但不是必要
};
```

注意在这个例子中`mf4`有别于之前，它在`Base`中的声明有`virtual`修饰，所以能正常工作。
大多数和重写有关的错误都是在派生类引发的，但也可能是基类的不正确导致。

比起让编译器（译注：通过warnings）告诉你"将要"重写实际不会重写，不如给你的派生类成员函数全都加上`override`。如果你考虑修改修改基类虚函数的函数签名，`override`还可以帮你评估后果。
如果派生类全都用上`override`，你可以只改变基类函数签名，重编译系统，再看看你造成了多大的问题（即，多少派生类不能通过编译），然后决定是否值得如此麻烦更改函数签名。没有重写，你只能寄希望于完善的单元测试，因为，正如我们所见，派生类虚函数本想重写基类，但是没有，编译器也没有探测并发出诊断信息。

C++既有很多关键字，C++11引入了两个上下文关键字(contextual keywords),`override`和`final`（向虚函数添加`final`可以防止派生类重写。`final`也能用于类，这时这个类不能用作基类）。
这两个关键字的特点是它们是保留的，它们只是位于特定上下文才被视为关键字。对于`override`，它只在成员函数声明结尾处才被视为关键字。这意味着如果你以前写的代码里面已经用过**override**这个名字，那么换到C++11标准你也无需修改代码：

```
class Warning { // potential legacy class from C++98
public:
    …
void override(); // C++98和C++11都合法
};
```

关于`override`想说的就这么多，但对于成员函数引用限定(reference qualifiers)还有一些内容。我之前承诺我会在后面提供更多的关于它们的资料，现在就是"后面"了。
如果我们想写一个函数只接受左值实参，我们的声明可以包含一个左值引用形参：

```
void doSomething(Widget& w); // 只接受左值Widget对象
```

如果我们想写一个函数只接受右值实参，我们的声明可以包含一个右值引用形参：

```
void doSomething(Widget&& w); // 只接受右值Widget对象
```

成员函数的引用限定可以很容易的区分哪个成员函数被对象调用（即`*this`）。它和在成员函数声明尾部添加一个`const`暗示该函数的调用者（即`*this`）是`const`很相似。
对成员函数添加引用限定不常见，但是可以见。
举个例子，假设我们的`Widget`类有一个`std::vector`数据成员，我们提供一个范围函数让客户端可以直接访问它：

```
class Widget {
public:
    using DataType = std::vector<double>; // 参见Item
    …
    DataType& data() { return values; }
    …
private:
    DataType values;
};
```

这是最具封装性的设计，只给外界保留一线光。但先把这个放一边，思考一下下面的客户端代码：

```
Widget w;
…
auto vals1 = w.data(); // 拷贝w.values到vals1
```

Widget::data函数的返回值是一个左值引用（准确的说是`std::vector<double>&`），
因为左值引用是左值，`vals1`从左值初始化，因此它由`w.values`拷贝构造而得，就像注释说的那样。
现在假设我们有一个创建`Widgets`的工厂函数，

```
Widget makeWidget();
```

我们想用`makeWidget`返回的`std::vector`初始化一个变量：

```
auto vals2 = makeWidget().data(); // 拷贝Widget里面的值到vals2
```

再说一次，`Widgets::data`返回的是左值引用，还有，左值引用是左值。所以，我们的对象(vals2)又得从Widget里的values拷贝构造。这一次，`Widget`是`makeWidget`返回的临时对象（即右值），所以将其中的`std::vector`进行拷贝纯属浪费。最好是移动，但是因为`data`返回左值引用，C++的规则要求编译器不得不生成一个拷贝。
我们需要的是指明当`data`被右值`Widget`对象调用的时候结果也应该是一个右值。
现在就可以使用引用限定写一个重载函数来达成这一目的：

```
class Widget {
public:
    using DataType = std::vector<double>;
    …
    DataType& data() &              // 对于左值Widgets,
    { return values; }              // 返回左值
    DataType data() &&              // 对于右值Widgets,
    { return std::move(values); }   // 返回右值
    …
private:
    DataType values;
};
```

注意`data`重载的返回类型是不同的，左值引用重载版本返回一个左值引用，右值引用重载返回一个临时对象。这意味着现在客户端的行为和我们的期望相符了：

```
auto vals1 = w.data();              //调用左值重载版本的Widget::data，拷贝构造vals1
auto vals2 = makeWidget().data();   //调用右值重载版本的Widget::data，移动构造vals2
```

这真的很nice，但别被这结尾的暖光照耀分心以致忘记了该条款的中心。这个条款的中心是只要你在派生类声明想要重写基类虚函数的函数，就加上`override`。

记住：

- 为重载函数加上`override`
- 成员函数限定让我们可以区别对待左值对象和右值对象（即`*this`）