Ite 17:理解特殊成员函数的生成

条款 17:理解特殊成员函数函数的生成

在C++术语中，特殊成员函数是指C++自己生成的函数。C++98有四个：默认构造函数函数，析构函数，拷贝构造函数，拷贝赋值运算符。这些函数仅在需要的时候才生成，比如某个代码使用它们但是它们没有在类中声明。默认构造函数仅在类完全没有构造函数的时候才生成。（防止编译器为某个类生成构造函数，但是你希望那个构造函数有参数）生成的特殊成员函数是隐式public且inline，除非该类是继承自某个具有虚函数的类，否则生成的析构函数是非虚的。

但是你早就知道这些了。好吧好吧，都说古老的历史：美索不达米亚，商朝，FORTRAN,C++98。但是时代改变了，C++生成特殊成员的规则也改变了。要留意这些新规则，因为用C++高效编程方面很少有像它们一样重要的东西需要知道。

C++11特殊成员函数俱乐部迎来了两位新会员：移动构造函数和移动赋值运算符。它们的签名是：

```
class Widget {
public:
    ...
    Widget(Widget&& rhs);
    Widget& operator=(Widget&& rhs);
    ...
};
```

掌控它们生成和行为的规则类似于拷贝系列。移动操作仅在需要的时候生成，如果生成了，就会对非static数据执行逐成员的移动。那意味着移动构造函数根据`rhs`参数里面对应的成员移动构造出新部分，移动赋值运算符根据参数里面对应的非static成员移动赋值。移动构造函数也移动构造基类部分（如果有的话），移动赋值运算符也是移动赋值基类部分。

现在，当我对一个数据成员或者基类使用移动构造或者移动赋值时，没有任何保证移动一定会真的发生。逐成员移动，实际上，更像是逐成员移动**请求**，因为对不可移动类型使用移动操作实际上执行的是拷贝操作。逐成员移动的核心是对对象使用**std::move**，然后函数决议时会选择执行移动还是拷贝操作。**Item 23**包括了这个操作的细节。本章中，简单记住如果支持移动就会逐成员移动类成员和基类成员，如果不支持移动就执行拷贝操作就好了。

两个拷贝操作是独立的：声明一个不会限制编译器声明另一个。所以如果你声明一个拷贝构造函数，但是没有声明拷贝赋值运算符，如果写的代码用到了拷贝赋值，编译器会帮助你生成拷贝赋值运算符重载。同样的，如果你声明拷贝赋值运算符但是没有拷贝构造，代码用到拷贝构造编译器就会生成它。上述规则在C++98和C++11中都成立。

如果你声明了某个移动函数，编译器就不再生成另一个移动函数。这与复制函数的生成规则不太一样：两个复制函数是独立的，声明一个不会影响另一个的默认生成。这条规则的背后原因是，如果你声明了某个移动函数，就表明这个类型的移动操作不再是“逐一移动成员变量”的语义，即你不需要编译器默认生成的移动函数的语义，因此编译器也不会为你生成另一个移动函数。

再进一步，如果一个类显式声明了拷贝操作，编译器就不会生成移动操作。这种限制的解释是如果声明拷贝操作就暗示着默认逐成员拷贝操作不适用于该类，编译器会明白如果默认拷贝不适用于该类，移动操作也可能是不适用的。

这是另一个方向。声明移动操作使得编译器不会生成拷贝操作。（编译器通过给这些函数加上delete来保证，参见Item11）。比较，如果逐成员移动对该类来说不合适，也没有理由指望逐成员考吧操作是合适的。听起来会破坏C++98的某些代码，因为C++11中拷贝操作可用的条件比C++98更受限，但事实并非如此。C++98的代码没有移动操作，因为C++98中没有移动对象这种概念。只有一种方法能让老代码

使用用户声明的移动操作，那就是使用C++11标准然后添加这些操作， 并在享受这些操作带来的好处同时接受C++11特殊成员函数生成规则的限制。

也许你早已听过*Rule of Three*规则。这个规则告诉我们如果你声明了拷贝构造函数，拷贝赋值运算符，或者析构函数三者之一，你应该也声明其余两个。它来源于长期的观察，即用户接管拷贝操作的需求几乎都是因为该类会做其他资源的管理，这也几乎意味着1）无论哪种资源管理如果能在一个拷贝操作内完成，也应该在另一个拷贝操作内完成2）类析构函数也需要参与资源的管理（通常是释放）。通常意义的资源管理指的是内存（如STL容器会动态管理内存），这也是为什么标准库里面那些管理内存的类都声明了“the big three”：拷贝构造，拷贝赋值和析构。

Rule of Three带来的后果就是只要出现用户定义的析构函数就意味着简单的逐成员拷贝操作不适用于该类。接着，如果一个类声明了析构也意味着拷贝操作可能不应该自定生成，因为它们做的事情可能是错误的。在C++98提出的时候，上述推理没有得倒足够的重视，所以C++98用户声明析构不会左右编译器生成拷贝操作的意愿。C++11中情况仍然如此，但仅仅是因为限制拷贝操作生成的条件会破坏老代码。

Rule of Three规则背后的解释依然有效，再加上对声明拷贝操作阻止移动操作隐式生成的观察，使得C++11不会为那些有用户定义的析构函数的类生成移动操作。所以仅当下面条件成立时才会生成移动操作：

- 类中没有拷贝操作
- 类中没有移动操作
- 类中没有用户定义的析构

有时，类似的规则也会扩展至移动操作上面，因为现在类声明了拷贝操作，C++11不会为它们自动生成其他拷贝操作。这意味着如果你的某个声明了析构或者拷贝的类依赖自动生成的拷贝操作，你应该考虑升级这些类，消除依赖。假设编译器生成的函数行为是正确的（即逐成员拷贝类数据是你期望的行为），你的工作很简单，C++11的`=default`就可以表达你想做的：

```
class Widget {
   public:
    ...
    ~Widget();
    ...
    Widget(const Widget&) = default;
    Widget&
    operator=(const Widget&) = default; // behavior is OK
    ...
};
```

这种方法通常在多态基类中很有用，即根据继承自哪个类来定义接口。多态基类通常有一个虚析构函数，因为如果它们非虚，一些操作（比如对一个基类指针或者引用使用delete或者typeid）会产生未定义或错误结果。除非类继承自一个已经是virtual的析构函数，否则要想析构为虚函数的唯一方法就是加上virtual关键字。通常，默认实现是对的，`=default`是一个不错的方式表达默认实现。然而用户声明的析构函数会抑制编译器生成移动操作，所以如果该类需要具有移动性，就为移动操作加上`=default`。声明移动会抑制拷贝生成，所以如果拷贝性也需要支持，再为拷贝操作加上`=default`：

```
class Base {
public:
  virtual ~Base() = default;
  Base(Base&&) = default;
  Base& operator=(Base&&) = default;
  Base(const Base&) = default;
  Base& operator=(const Base&) = default;
  ...
};
```

实际上，就算编译器乐于为你的类生成拷贝和移动操作，生成的函数也如你所愿，你也应该手动声明它们然后加上`=default`。这看起来比较多余，但是它让你的意图更明确，也能帮助你避免一些微妙的bug。比如，你有一个字符串哈希表，即键为整数id，值为字符串，支持快速查找的数据结构：

```
 class StringTable {
   public:
     StringTable() {}
     ...
   private:
     std::map<int, std::string> values;
};
```

假设这个类没有声明拷贝操作，没有移动操作，也没有析构，如果它们被用到编译器会自动生成。没错，很方便。

后来需要在对象构造和析构中打日志，增加这种功能很简单：

```
class StringTable {
public:
  StringTable()
  { makeLogEntry("Creating StringTable object"); }

  ~StringTable()
  { makeLogEntry("Destroying StringTable object"); }
  ...
 Item 17 | 113
private:
     std::map<int, std::string> values;    // as before
};
```

看起来合情合理，但是声明析构有潜在的副作用：它阻止了移动操作的生成。然而，拷贝操作的生成是不受影响的。因此代码能通过编译，运行，也能通过功能（译注：即打日志的功能）测试。功能测试也包括移动功能，因为即使该类不支持移动操作，对该类的移动请求也能通过编译和运行。这个请求正如之前提到的，会转而由拷贝操作完成。它因为着对**StringTable**对象的移动实际上是对对象的拷贝，即拷贝里面的`std::map<int, std::string>`对象。拷贝`std::map<int, std::string>`对象很可能比移动慢几个数量级。简单的加个析构就引入了极大的性能问题！对拷贝和移动操作显式加个`=default`，问题将不再出现。

受够了我喋喋不休的讲述C++11拷贝移动规则了吧，你可能想知道什么时候我才会把注意力转入到剩下两个特殊成员函数，默认构造和析构。现在就是时候了，但是只有一句话，因为它们几乎没有改变：它们在C++98中是什么样，在C++11中就是什么样。

C++11对于特殊成员函数处理的规则如下：

- 默认构造函数：和C++98规则相同。仅当类不存在用户声明的构造函数时才自动生成。
- 析构函数：基本上和C++98相同；稍微不同的是现在析构默认**noexcept**（参见Item14）。和C++98一样，仅当基类析构为虚函数时该类析构才为虚函数。
- 拷贝构造函数：和C++98运行时行为一样：逐成员拷贝非static数据。仅当类没有用户定义的拷贝构造时才生成。如果类声明了移动操作它就是**delete**。当用户声明了拷贝赋值或者析构，该函数不再自动生成。
- 拷贝赋值运算符：和C++98运行时行为一样：逐成员拷贝赋值非static数据。仅当类没有用户定义的拷贝赋值时才生成。如果类声明了移动操作它就是**delete**。当用户声明了拷贝构造或者析构，该函数不再自动生成。
- 移动构造函数和移动赋值运算符：都对非static数据执行逐成员移动。仅当类没有用户定义的拷贝操作，移动操作或析构时才自动生成。

注意没有成员函数模版阻止编译器生成特殊成员函数的规则。这意味着如果**Widget**是这样：

```
class Widget {
  ...
  template<typename T>
  Widget(const T& rhs);

  template<typename T>
  Widget& operator=(const T& rhs); ...
};
```

编译器仍会生成移动和拷贝操作（假设正常生成它们的条件满足），即使可以模板实例化产出拷贝构造和拷贝赋值运算符的函数签名。（当T为Widget时）。很可能你会决定这是一个不值得承认的边缘情况，但是我提到它是有道理的，Item16将会详细讨论它可能带来的后果。

记住：

- 特殊成员函数是编译器可能自动生成的函数：默认构造，析构，拷贝操作，移动操作。
- 移动操作仅当类没有显式声明移动操作，拷贝操作，析构时才自动生成。
- 拷贝构造仅当类没有显式声明拷贝构造时才自动生成，并且如果用户声明了移动操作，拷贝构造就是delete。拷贝赋值运算符仅当类没有显式声明拷贝赋值运算符时才自动生成，并且如果用户声明了移动操作，拷贝赋值运算符就是delete。当用户声明了析构函数，拷贝操作不再自动生成。