Item 14:如果函数不抛出异常请使用noexcept

条款 14:如果函数不抛出异常请使用noexcept

在C++98中，异常说明（exception specifications）是喜怒无常的野兽。你不得不写出函数可能抛出的异常类型，如果函数实现有所改变，异常说明也可能需要修改。改变异常说明会影响客户端代码，因为调用者可能依赖原版本的异常说明。编译器不会为函数实现，异常说明和客户端代码中提供一致性保障。大多数程序员最终都认为不值得为C++98的异常说明如此麻烦。

在C++11标准化过程中，大家一致认为异常说明真正有用的信息是一个函数是否会抛出异常。非黑即白，一个函数可能抛异常，或者不会。这种"可能-绝不"的二元论构成了C++11异常说的基础，从根本上改变了C++98的异常说明。（C++98风格的异常说明也有效，但是已经标记为deprecated（废弃））。在C++11中，无条件的**noexcept**保证函数不会抛出任何异常。

关于一个函数是否已经声明为**noexcept**是接口设计的事。函数的异常抛出行为是客户端代码最关心的。调用者可以查看函数是否声明为**noexcept**，这个可以影响到调用代码的异常安全性和效率。

就其本身而言，函数是否为**noexcept**和成员函数是否**const**一样重要。如果知道这个函数不会抛异常就加上**noexcept**是简单天真的接口说明。

不过这里还有给不抛异常的函数加上**noexcept**的动机：它允许编译器生成更好的目标代码。
要想知道为什么，了解C++98和C++11指明一个函数不抛异常的方式是很有用了。考虑一个函数**f**，它允许调用者永远不会受到一个异常。两种表达方式如下：

```
int f(int x) throw(); // C++98风格
int f(int x) noexcept; // C++11风格
```

如果在运行时，**f**出现一个异常，那么就和**f**的异常说明冲突了。在C++98的异常说明中，调用栈会展开至**f**的调用者，一些不合适的动作比如程序终止也会发生。C++11异常说明的运行时行为明显不同：调用栈只是*可能*在程序终止前展开。
展开调用栈和*可能*展开调用栈两者对于代码生成（code generation）有非常大的影响。在一个**noexcept**函数中，当异常传播到函数外，优化器不需要保证运行时栈的可展开状态，也不需要保证**noexcept**函数中的对象按照构造的反序析构。而"**throw()**"标注的异常声明缺少这样的优化灵活性，它和没加一样。可以总结一下：

```
RetType function(params) noexcept;   // 极尽所能优化
RetType function(params) throw();    // 较少优化
RetType function(params);            // 较少优化
```

这是一个充分的理由使得你当知道它不抛异常时加上**noexcept**。

还有一些函数让这个案例更充分。移动操作是绝佳的例子。假如你有一份C++98代码，里面用到了`std::vector<Widget>`。**Widget**通过**push_back**一次又一次的添加进`std::vector`：

```
std::vector<Widget> vw;
…
Widget w;
…                    // work with w
vw.push_back(w);     // add w to vw
```

假设这个代码能正常工作，你也无意修改为C++11风格。但是你确实想要C++11移动语义带来的性能优势，毕竟这里的类型是可以移动的(move-enabled types)。因此你需要确保**Widget**有移动操作，可以手写代码也可以让编译器自动生成，当然前提是自动生成的条件能满足（参见Item 17）。

当新元素添加到`std::vector`，`std::vector`可能没地方放它，换句话说，`std::vector`的大小(size)等于它的容量(capacity)。这时候，`std::vector`会分配一片的新的大块内存用于存放，然后将元素从已经存在的内存移动到新内存。在C++98中，移动是通过复制老内存区的每一个元素到新内存区完成的，然后老内存区的每个元素发生析构。
这种方法使得`push_back`可以提供很强的异常安全保证：如果在复制元素期间抛出异常，`std::vector`状态保持不变，因为老内存元素析构必须建立在它们已经成功复制到新内存的前提下。

在C++11中，一个很自然的优化就是将上述复制操作替换为移动操作。但是很不幸运，这回破坏`push_back`的异常安全。如果**n**个元素已经从老内存移动到了新内存区，但异常在移动第**n+1**个元素时抛出，那么`push_back`操作就不能完成。但是原始的`std::vector`已经被修改：有**n**个元素已经移动走了。恢复`std::vector`至原始状态也不太可能，因为从新内存移动到老内存本身又可能引发异常。

这是个很严重的问题，因为老代码可能依赖于`push_back`提供的强烈的异常安全保证。因此，C++11版本的实现不能简单的将`push_back`里面的复制操作替换为移动操作，除非知晓移动操作绝不抛异常，这时复制替换为移动就是安全的，唯一的副作用就是性能得到提升。

`std::vector::push_back`受益于"如果可以就移动，如果必要则复制"策略，并且它不是标准库中唯一采取该策略的函数。C++98中还有一些函数如`std::vector::reverse`,`std:;deque::insert`等也受益于这种强异常保证。对于这个函数只有在知晓移动不抛异常的情况下用C++11的move替换C++98的copy才是安全的。但是如何知道一个函数中的移动操作是否产生异常？答案很明显：它检查是否声明**noexcept**。

swap函数是**noexcept**的绝佳用地。**swap**是STL算法实现的一个关键组件，它也常用于拷贝运算符重载中。它的广泛使用意味着对其施加不抛异常的优化是非常有价值的。有趣的是，标准库的**swap**是否**noexcept**有时依赖于用户定义的**swap**是否**noexcept**。比如，数组和`std::pair`的**swap**声明如下：

```
template <class T, size_t N>
void swap(T (&a)[N], // see
          T (&b)[N]) noexcept(noexcept(swap(*a, *b))); // below

template <class T1, class T2>
struct pair {
    …
    void swap(pair& p) noexcept(noexcept(swap(first, p.first)) &&
    noexcept(swap(second, p.second)));
    …
};
```

这些函数视情况**noexcept**：它们是否**noexcept**依赖于`noexcept`声明中的表达式是否**noexcept**。假设有两个**Widget**数组，不抛异常的交换数组前提是数组中的元素交换不抛异常。对于**Widget**的交换是否**noexcept**决定了对于`Widget`数组的交换是否**noexcept**，反之亦然。类似的，交换两个存放**Widget**的`std::pair`是否**noexcept**依赖于**Widget**的交换是否**noexcept**。事实上交换高层次数据结构是否**noexcept**取决于它的构成部分的那些低层次数据结构是否异常，这激励你只要可以就提供**noexcept swap**函数（译注：因为如果你的函数不提供**noexcept**保证，其它依赖你的高层次**swap**就不能保证**noexcept**）。

现在，我希望你能为**noexcept**提供的优化机会感到高兴，同时我还得让你缓一缓别太高兴了。优化很重要，但是正确性更重要。我在这个条款的开头提到**noexcept**是函数接口的一部分，所以仅当你保证一个函数实现在长时间内不会抛出异常时才声明**noexcept**。如果你声明一个函数为**noexcept**，但随即又后悔了，你没有选择。你只能从函数声明中移除**noexcept**（即改变它的接口），这理所当然会影响客户端代码。你可以改变实现使得这个异常可以避免，再保留原版本（不正确的）异常说明。如果你这

么做，程序将会在异常离开这个函数时终止。或者你可以重新设计既有实现，改变实现后再考虑你希望它是什么样子。这些选择都不尽人意。

这个问题的本质是实际上大多数函数都是异常中立（**exception neutral**）的。这些函数自己不抛异常，但是它们内部的调用可能抛出。此时，异常中立函数允许那些抛出异常的函数在调用链上更进一步直到遇到异常处理程序，而不是就地终止。异常中立函数决不应该声明为**noexcept**，因为它们可能抛出那种"让它们过吧"的异常（译注：也就是说在当前这个函数内不处理异常，但是又不立即终止程序，而是让调用这个函数的函数处理）异常。因此大多数函数都不应该被指定为**noexcept**。

然而，一些函数很自然的不应该抛异常，更进一步值得注意的是移动操作和**swap**——使其不抛异常有重大意义，只要可能就应该将它们声明为**noexcept**。老实说，当你确保函数决不抛异常的时候，一定要将它们声明为**noexcept**。

请注意我说的那些很自然不应该抛异常的函数实现。为了**noexcept**而扭曲函数实现达成目的是本末倒置。是把马放到马车前，是一叶障目不见泰山。是...选择你喜欢的比喻吧。如果一个简单的函数实现可能引发异常（即调用它可能抛出异常），而你为了讨好调用者隐藏了这个（即捕获所有异常，然后替换为状态码或者特殊返回值），这不仅会使你的函数实现变得复杂，还会让所有调用点的代码变得复杂。调用者可能不得不检查状态码或特殊返回值。而这些**复杂**的运行时开销（额外的分支，大的函数放入指令缓存）可以超出**noexcept**带来的性能提升，再加上你会悲哀的发现这些代码又难读又难维护。那是糟糕的软件工程化。

对于一些函数，使其成为**noexcept**是很重要的，它们应当默认如是。在C++98构造函数和析构函数抛出异常是糟糕的代码设计——不管是用户定义的还是编译器生成的构造析构都是**noexcept**。因此它们不需要声明**noexcept**。（这么做也不会有问题，只是不合常规）。析构函数非隐式**noexcept**的情况仅当类的数据成员明确声明它的析构函数可能抛出异常（即，声明`noexcept(false)`）。这种析构函数不常见，标准库里面没有。如果一个对象的析构函数可能被标准库使用，析构函数又可能抛异常，那么程序的行为是未定义的。

值得注意的是一些库接口设计者会区分有宽泛契约(**wild contracts**)和严格契约(**narrow contracts**)的函数。有宽泛契约的函数没有前置条件。这种函数不管程序状态如何都能调用，它对调用者传来的实参不设约束。宽泛契约的函数决不表现出未定义行为。

反之，没有宽泛契约的函数就有严格契约。对于这些函数，如果违反前置条件，结果将会是未定义的。

如果你写了一个有宽泛契约的函数并且你知道它不会抛异常，那么遵循这个条款给它声明一个**noexcept** 是很容易的。
对于严格契约的函数，情况就有点微妙了。举个例子，假如你在写一个参数为**std::string**的函数**f**，并且这个函
数f很自然的决不引发异常。这就在建议我们**f**应该被声明为**noexcept** 。

现在假如**f**有一个前置条件：类型为**std::string**的参数的长度不能超过32个字符。如果现在调用f并传给它一个
大于32字符的参数，函数行为将是未定义的，因为违反了 *（口头/文档）定义的* 前置条件，导致了未定义行为。**f**没有
义务去检查前置条件，它假设这些前置条件都是满足的。（调用者有责任确保参数字符不超过32字符等这些假设有效。）。
即使有前置条件，将**f**声明为**noexcept**似乎也是合适的：

```
void f(const std::string& s) noexcept;  // 前置条件:
                                        // s.length() <= 32
```

f的实现者决定在函数里面检查前置条件冲突。虽然检查是没有必要的，但是也没禁止这么做。另外在系统测试时，检查
前置条件可能就是有用的了。debug一个抛出的异常一般都比跟踪未定义行为起因更容易。那么怎么报告前置条件冲突使得

测试工具或客户端错误处理程序能检测到它呢？简单直接的做法是抛出`"precondition was violated"`异常，但是如果
f声明了**noexcept**，这就行不通了；抛出一个异常会导致程序终止。因为这个原因，区分严格/宽泛契约库设计者一般
会将**noexcept**留给宽泛契约函数。

作为结束语，让我详细说明一下之前的观察，即编译器不会为函数实现和异常规范提供一致性保障。考虑下面的代码，它是
完全正确的：

```
void setup(); // 函数定义另在一处
void cleanup();
void doWork() noexcept
{
 setup();   // 前置设置
 …          // 真实工作
 cleanup(); // 执行后置清理
}
```

这里，**doWork**声明为**noexcept**，即使它调用了非**noexcept**函数`setup`和`cleanup`。看起来有点矛盾，
其实可以猜想`setup`和`cleanup`在文档上写明了它们决不抛出异常，即使它们没有写上**noexcept**。至于为什么
明明不抛异常却不写**noexcept**也是有合理原因的。比如，它们可能是用C写的库函数的一部分。（即使一些函数从
C标准库移动到了**std**命名空间，也可能缺少异常规范，**std::strlen**就是一个例子，它没有声明**noexcept**）。
或者它们可能是C++98库的一部分，它们不使用C++98异常规范的函数的一部分，到了C++11还没有修订。

因为有很多合理原因解释为什么**noexcept**依赖于缺少**noexcept**保证的函数，所以C++允许这些代码，编译器
一般也不会给出warnigns。

记住：

- **noexcept**是函数接口的一部分，这意味着调用者会依赖它、
- **noexcept**函数较之于非**noexcept**函数更容易优化
- **noexcept**对于移动语义,**swap**，内存释放函数和析构函数非常有用
- 大多数函数是异常中立的(译注：可能抛也可能不抛异常）而不是**noexcept**