异常

- 程序之所以在exceptions时出现良好行为，不是因为碰巧如此，而是因为它们加入了expections的考虑。
- exceptions的出现是一种必然，无法被忽略。如果一个函数利用"设定状态变量"的方式或是利用"返回错误码"的方式发出一个异常信号，无法保证此函数的调用者会检查那个错误码。于是程序会一直进行下去，远离错误的发生点。但是exceptions会发出异常信号，不处理则立刻终止。

Item 9 利用destructors避免泄露资源

考虑运行一个指针指向的函数，如果这个函数扔出一个异常，但是指针是裸露的，`delete`还在后面，那么会导致这个调用往后面走的东西全部都没有执行，进而产生了资源泄露。

解决这个问题的方法有两个：

- 使用`try...catch`语句，把异常收拾进`catch`里面。这样的代码可能不美观。
- 包装这个指针，把`delete`动作扔进这个包装类的析构函数里面。采用这样的做法的类很多，比如`smart_ptr<T>`，`auto_ptr<T>`等等。
    - 如果有可能的话，可以考虑自己设计一个类来包装，然后尽力避免编译器自行补充构造函数、`operator=`函数。

Item 10 在constructors内阻止资源泄露

通常来说，一个类中是有很大可能存在裸露的指针的。在这种情况下，如果初始化中的`new`失败而且没有处理，那么会直接造成资源泄露。甚至如果有多个指针的情况下，一个指针出问题，其他都要处理掉，而不是重新考虑赋值上去，因为`new`失败的后面基本上是内存不足。

C++只会析构已经构造完成的对象，所以用到这个指针的类也无法调用自己的析构函数。C++选择这么做不是没有理由的，虽然这个给程序员造成了比较大的麻烦。如果直接析构掉没有构造好的对象，那么显然是不安全的，里面是什么东西都不知道，构造到哪里也不知道；想要安全的析构则必须要给一个标记进行到哪里了，但是这样会直接影响程序的性能，把对象变得无比庞大。C++避免这样的额外开销，但你必须付出"部分构造完成"对象不会被自动销毁的代价。

意图解决这个问题，我们必须在函数本身把东西拦截下来。这通常来说有两个方法。

- 使用智能指针管理，如果出现分配失败，会自动`delete`掉，这是智能指针要做的事情。
- 在本身的构造函数中写入一个`try...catch`，只要分配失败就可以拦下来`delete`掉，然后怎么做随便你。

这么做的原因是比较充分的：

- `new`在分配失败的时候可以返回`nullptr`或者抛出一个异常，不会有`new`不出来就扔下这块内存不管的情况。这是因为`new`是一整块一整块的分配，一次`new`只会要到一整块。返回`nullptr`是早期的C++语言，现在的C++也可以通过`nothrow`来显式调用，默认的还是会扔一个异常出来。
- `delete`空指针是安全的，此时`delete`会什么都不做。

当然如果你有强迫症的话，可以写一个`private`函数包装起来，供构造函数和析构函数调用。

如果你希望指针在拿到一个值以后就不能再改变，那么可能`*const`是很好的选择，但是这也意味着在构造函数中`new`失败时，刚才的第二条机制就会完全失效。此时建议在类内`private`写函数把构造过程包装起来，返回值送给对象，异常交给函数内部处理。当然，RAII方法还是有效的。

Item 11 禁止异常流出destructors之外

析构函数有两种状况下会被调用。第一种是在正常状态下被销毁，或者主动调用析构函数（不知道是否是仅编译器可行）；第二种是被异常处理机制（stack-unwinding）销毁。如果在异常作用下调用到析构函数，而析构函数又抛出了一个异常，那么C++会直接把程序掐死。所以再写析构函数时，必须要假设有异常存在，不去处理原来的异常（这应该交给原本的调用者处理），然后自己不产生任何异常。

如果你无法保证自己写的过程不会产生异常，那么可以考虑`try...catch`大法。

Item 12 了解“抛出一个exception”与“传递一个参数”或“调用一个虚函数”之间的差异

传递一个参数和抛出一个exception的语法是非常相似的，但是他们的区别非常大，这是考虑到安全性的结果。

- 抛出异常时，内容必须复制一份出来，不论是by reference还是by value，甚至by value会被复制两次。这是因为`throw`一个东西以后，当前函数会停止执行，异常交给调用者处理，此时里面的所有临时变量都会析构。如果抛出的东西本身也是一个局部变量，且by reference时采取绑定而不是复制，那么在出去的时候，这个东西就认不出来，甚至发生内存错误。大框架又不好打破，那么只能修改exception的运行机制了。
    - 当然，抛出异常的速度自然要比传递一个参数要慢，这是拷贝带来的问题。
- 拷贝动作交给了copy constructor进行，这个东西参考的是对象的静态类型而不是动态类型。在这点上和传递一个参数非常相似。
- `throw`和`throw object`不是一个东西，后者是抛出了一个具体的对象的副本，而前者则是吧原来的exception重新抛出，传播给调用者。
- 函数调用把一个临时对象传递给non-`const` reference是不合法的，但是exception是个例外。
- 指针传递以就需要注意的是不要让`catch`拿到一个已被销毁的对象指针。
- 在类型转换方面，C++允许隐式转换，但是exception则不行，换句话说，如果`throw`一个`int`，那么`catch(double)`是抓不到这个异常的。对于异常而言，只有两种可以被接受。一种是继承中的转换，一种是有形指针转为无形指针。
- 在多个`catch`中，最先满足要求的会最先执行，剩下的就不管，这个和虚函数是不一样的。

Item 13 以by reference方式捕捉exceptions

在一个`catch`中，我们可以选的拿到值的方式一共就只有三类，by pointer，by value和by reference。

通过指针传递exceptions可以很好的避免复制对象带来的一大问题，但是也会遇到一个尴尬的情况：你无法确认你拿到的东西是否在函数内定义，如果在的话，那么在拿到的时候，指针指向的对象已经被析构了。

如果使用类似于`throw new exception();`的方法来传递指针，那么这个问题会更加麻烦，因为你不知道这个东西是来自于heap还是在其他地方，是不是要`delete`这个东西?

除此以外，exception传递指针有点违背已有的惯例。exception传递的东西在标准库内都是对象。

by value可以解决by pointer带来的上述问题，但是在除了多次复制以外，还有个问题就是如果在函数内扔了一个子类出来，外面以父类`catch`，那么在执行的时候会按照父类进行处理。

千万要注意，在exception里面，引用传递也会调用虚函数。

Item 14 明智运用exception specifications

这一个Item建议仔细看看。C++真的是把程序员当神看。

一个函数后面跟`throw(T)`标志了这个函数可以抛出什么样的异常，如果遇到了既定异常之外的异常，那么会执行`unexpected`函数，里面直接把程序终止掉。可以说这玩意不仅仅是一个文档式的辅助，也是一种实践机制。当然，这种说明仅仅是说能抛出什么异常，而不是一定只能接受某些异常，但是要注意的是如果没有处理对应异常，那么这种异常就会被传出去，同样也可以遇到`unexpected`。

由于大部分代码都没有这样的说明，所以有可能会遇到一个声明exception specifications的调用一个没有这样声明的，虽然编译器可以警告，但是编译器无法拒绝这样的东西。这个时候就要小心小心再小心。除此以外，还有一些比较好的建议可以避免前面提到的调用。

- 不要把expection放到`template`函数里面，因为你不知道里面的操作符、运算、传递、调用会不会出问题。
- 如果A调用了B，B没有exception specifications，那么A本身也不要这么写。
    - 特别是有指针包装则特别需要注意，包括但不限于函数指针。这时就需要尽力使用编译特性把throw作为强制标准。书上给出的回调函数就是一个例子。
- 处理系统可能抛出的exceptions。如果你使用的程序库没用异常，但是你需要使用的话，可以考虑自己写一个，然后把所有的都当作你自己写的异常来处理。具体的处理方法就是把`unexpected`换掉。
- 如果你真的害怕一个没处理的异常会把整个程序拖垮，建议`catch(...)`，再给程序一个机会。

Item 15 了解异常处理的成本

首先要了解到异常处理的成本到底用在了那里。

- exception specification的比对，在`throw(T)`中发生。
- 把exception交给调用者时发生的拷贝复制。

面对这些成本，我们别无选择，因为这是语言的一部分，我们必须要面对大部分人用不好但是少数牛人再用的事实，只要你的代码用了exception，那么整个程序就必须支持。当然，大部分支持这玩意的都有把exception完全放弃的优化，这可以提升程序运行效率，但是这需要保证整个程序的所有内容，包括你`include`的东西都没有exception。

抛出一个异常对于程序的性能影响可能是非常巨大的，在上述两点的加持下，还要考虑保留现场，跳转退出，提前析构等等操作，会拉低大约三个数量级的处理效率。相对于处理效率而言，发生异常带来的闪退等等问题可能会给你或者你的客户带来更多的损失，这才是更要命的问题。除非你是神仙，不需要异常也能设计好代码，写出一份好文档。

> 不论exception处理过程需要多少成本，你都不应该付出比你该付出的部分更多。

只有非用不可才需要`try...catch`。