**错误处理(Error-Handling)：为何、何时、如何**

By 刘未鹏(pongba)

C++的罗浮宫(<http://blog.csdn.net/pongba>)

TopLanguage(<http://groups.google.com/group/pongba>)

**引言**

错误处理（Error-Handling）这个重要议题从1997年（也许更早）到2004年左右一直是一个被广泛争论的话题，曾在新闻组上、博客上、论坛上引发口水无数（不亚于语言之争），Bjarne Stroustrup、James Gosling、Anders Hejlsberg、Bruce Eckel、Joel Spolsky、Herb Sutter、Andrei Alexandrescu、Brad Abrams、Raymond Chen、David Abrahams…，各路神仙纷纷出动，好不热闹:-)

如今争论虽然已经基本结束并结果；只不过结论散落在大量文献当中，且新旧文献陈杂，如果随便翻看其中的几篇乃至几十篇的话都难免管中窥豹。就连Gosling本人写的[《The Java Programming Language》](http://www.amazon.com/Java-TM-Programming-Language-4th/dp/0321349806/ref=pd_bbs_sr_1/103-4499860-5260624?ie=UTF8&s=books&qid=1192347011&sr=8-1)中也语焉不详。所以，写这篇文章的目的便是要对这个问题提供一个整体视图，相信我，这是个有趣的话题:-)

**为什么要错误处理？**

这是关于错误处理的问题里面最简单的一个。答案也很简单：现实世界是不完美的，意料之外的事情时有发生。

一个现实项目不像在学校里面完成大作业，只要考虑该完成的功能，走happy path（也叫One True Path）即可，忽略任何可能出错的因素（呃.. 你会说，怎么会出错呢？配置文件肯定在那，矩阵文件里面肯定含的是有效数字.. 因为所有的环境因素都在你的控制之下。就算出现什么不测，比如运行到一半被人拔了网线，那就让程序崩溃好了，再双击一下不就行了嘛）。

然而现实世界的软件就必须考虑错误处理了。如果一个错误是能够恢复的，要尽量恢复。如果是不能恢复的，要妥善的退出模块，保护用户数据，清理资源。如果有必要的话应该记录日志，或重启模块等等。

简而言之，错误处理的最主要目的是为了构造健壮系统。

**什么时候做错误处理？（或者：什么是“错误”？）**

错误，很简单，就是不正确的事情。也就是不该发生的事情。有一个很好的办法可以找出哪些情况是错误。首先就当自己是在一个完美环境下编程的，一切precondition都满足：文件存在那里，文件没被破坏，网络永远完好，数据包永远完整，程序员永远不会拿脑袋去打苍蝇，等等… 这种情况下编写的程序也被称为happy path（或One True Path）。

剩下的所有情况都可以看作是错误。即“不应该”发生的情况，不在算计之内的情况，或者预料之外的情况，whatever。

简而言之，**什么错误呢**？*调用方违反被调用函数的precondition、或一个函数无法维持其理应维持的invariants、或一个函数无法满足它所调用的其它函数的precondition、或一个函数无法保证其退出时的postcondition；以上所有情况都属于错误。*（详见[《C++ Coding Standards: 101 Rules, Guidelines, and Best Practices》](http://www.dearbook.com/book/91309)第70章，或[《Object Oriented Software Construction, 2nd edition》](http://www.amazon.com/Object-Oriented-Software-Construction-Prentice-Hall-International/dp/0136291554/ref=pd_bbs_sr_1/103-4499860-5260624?ie=UTF8&s=books&qid=1192346746&sr=8-1)第11、12章）

例如文件找不到（通常意味着一个错误）、配置文件语法错误、将一个值赋给一个总应该是正值的变量、文件存在但由于访问限制而不能打开，或打开不能写、网络传输错误、网络中断、数据库连接错误、参数无效等。

不过话说回来，现实世界中，错误与非错误之间的界定其实是很模糊的。例如文件缺失，可能大多数情况下都意味着一个错误（影响程序正常执行并得出正常结果），然而有的情况下也可能根本就不是错误（或者至少是可恢复的错误），比如你的街机模拟器的配置文件缺失了，一般程序只要创建一个缺省的即可。

因此，关于把哪些情况界定为错误，具体的答案几乎总是视具体情况而定的。但话虽如此，仍然还是有一些一般性的原则，如下：

**哪些情况不属于错误？**

1.      **控制程序流的返回值不是错误**：如果一个情况经常发生，往往意味着它是用来控制程序流程的，应该用status-code返回（注意，不同于error-code），比如经典的while(cin >> i)。读入整型失败是很常见的情况，而且，这里的“读入整型失败”其实真正的含义是“流的下一个字段不是整型”，后者很明确地不代表一个错误；再比如在一个字符串中寻找一个子串，如果找不到该子串，也不算错误。这类控制程序流的返回值都有一个共同的特点，即我们都一定会利用它们的返回值来编写if-else，循环等控制结构，如：  
  
  if(foo(…)) { … }  
  else { … }  
  
或  
  
  while(foo(…)) { … }  
  
这里再摘两个相应的具体例子，一个来自Gosling的《The Java Programming Language》，是关于stream的。  
  
使用status-code：  
  
  while ((token = stream.next()) != Stream.END)  
      process(token);  
  stream.close();  
  
使用exception：  
  
  try {  
    for(;;) {  
      process(stream.next());  
    }  
  } catch (StreamEndException e) {  
    stream.close();  
  }  
  
高下立判。  
  
另一个例子来自TC++PL（Well, not exactly）：  
  
  size\_t index;  
  try {  
    index = find(str, sub\_str);  
    … // case 1  
  } catch (ElementNotFoundException& e) {  
    … // case 2  
  }  
  
使用status-code：  
  
  int index = find(str, sub\_str)  
  if(index != -1) {  
    … // case 1  
  } else {  
    … // case 2  
  }  
  
以上这类情况的特点是，返回值本身也是程序主逻辑（happy path）的一部分，返回的两种或几种可能性，都是完全正常的、预料之中的。  
  
**1’. 另一方面，还有一种情况与此有微妙的区别**，即“可恢复错误”。可恢复错误与上面的情况的区别在于它虽说也是预料之中的，但它一旦发生程序往往就会转入一个错误恢复子过程，后者会尽可能恢复程序主干执行所需要的某些条件，恢复成功程序则再次转入主干执行，而一旦恢复失败的话就真的成了一个货真价实的让人只能干瞪眼的错误了。比如C++里面的operator new如果失败的话会尝试调用一个可自定义的错误恢复子过程，当然，后者并非总能成功将程序恢复过来。除了转入一个错误恢复子过程之外，另一种可能性是程序会degenerate入一条second-class的支流，后者也许能完成某些预期的功能，但却是“不完美”地完成的。  
  
这类错误如何处理后面会讨论。

2.      **编程bug不是错误**。属于同一个人维护的代码，或者同一个小组维护的代码，如果里面出现bug，使得一个函数的precondition得不到满足，那么不应该视为错误。而应该用assert来对付。因为编程bug发生时，你不会希望栈回滚，而是希望程序在assert失败点上直接中断，调用调试程序，查看中断点的程序状态，从而解决代码中的bug。  
  
关于这一点，需要尤其注意的是，它的前提是：必须要是同一个人或小组维护的代码。同一个小组内可以保证查看到源代码，进行debug。如果调用方和被调用方不属于同一负责人，则不能满足precondition的话就应该抛出异常。总之记住一个精神：assert是用来辅助debug的（assert的另一个作用是文档，描述程序在特定点上的状态，即便assert被关闭，这个描述功能也依然很重要）。  
  
注意，有时候，为了效率原因，也会在第三方库里面使用assert而不用异常来报告违反precondition。比如strcpy，std::vector的operator[]。

3.      **频繁出现的不是错误**。频繁出现的情况有两种可能，一是你的程序问题大了（不然怎么总是出错呢？）。二是出现的根本不是错误，而是属于程序的正常流程。后者应该改用status-code。

**插曲：异常（exception）vs错误代码（error-code）**

异常相较于错误代码的优势太多了，以下是一个（不完全）列表。

异常与错误代码的**本质区别之一**——异常会**自动**往上层栈传播：一旦异常被抛出，执行流就立即中止，取而代之的是**自动的**stack-unwinding操作，直到找到一个适当的catch子句。

相较之下，使用error-code的话，要将下层调用发生的错误传播到上层去，就必须手动检查每个调用边界，任何错误，都必须手动转发（返回）给上层，稍有遗漏，就会带着错误的状态继续往下执行，从而在下游不知道离了多远的地方最终引爆程序。这来了以下几个问题：

1.      **麻烦**。每一个可能返回错误代码的调用边界都需要检查，不管你实际上对不对返回的错误作响应，因为即便你自己不解决返回的错误，也要把它传播到上层去好让上层解决。

2.      **不优雅且不可伸缩(scalability)的代码**（错误处理代码跟One True Path（也叫happy path）搅和在一起）。关于这一条普遍的论述都不够明确，比如有人可能会反驳说，那错误反正是要检查的，用异常难道就不需要捕获异常了吗？当然是需要的，但关键是，有时候我们不一定会在异常发生的立即点上捕获并处理异常。这时候，异常的优势就显现出来了，比如：  
  
  void foo()  
  {  
    try {  
      op1;  
      op2;  
      …  
      opN;  
    } catch (…) {  
      … // log  
      … // clean up  
      throw;  
    }  
  }  
  
如果用error-code的话：  
  
  int foo()  
  {  
    if(!op1()) {   
      … // log? clean up?  
      return FAILED;  
    }  
    if(!op2()) {  
      … // log? clean up?  
      return FAILED;  
    }  
    …  
    return SUCCEEDED;  
  }  
  
好一点的是这样：  
  
  int foo()  
  {  
    if(!op1()) goto FAILED;  
    if(!op2()) goto FAILED;  
    …  
    if(!opN()) goto FAILED;  
    return SUCCEEDED;  
  FAILED:  
    … // log, clean up  
    return FAILED;  
  }  
  
就算是最后一种做法（所谓的“On Error Goto”），One True Path中仍然夹杂着大量的噪音（如果返回的错误值不只是FAILED/SUCCEEDED两种的话噪音会更大）。此外手动检查返回值的成功失败毕竟是很error-prone的做法。  
  
值得注意的是，这里我并没有用一个常被引用的例子，即：如果你是用C写代码（C不支持局部变量自动析构（RAII）），那么程序往往会被写成这样：  
  
  int f()  
  {  
    int returnCode = FAILED;  
    *acquire resource1*;  
    if(*resource1 is acquired*) {  
      *acquire resource2*;  
      if(*resource2 is acquired*) {  
        *acquire resource3*;  
        if(*resource3 is acquired*) {  
          if(doSomething1()) {  
            if(doSomething2()) {  
              if(doSomething3()) {  
                returnCode = SUCCEEDED;  
              }  
            }  
          }  
          *release resource3*;  
        }  
        *release resource2*;  
      }  
      *release resource1*;  
    }  
    return returnCode;  
  }  
  
或者像这样：  
  
  int f()  
  {  
    int returnCode = FAILED;  
    *acquire resource1*;  
    if(*resources1 is not acquired*)   
      return FAILED;  
    *acquire resource2*;  
    if(*resource2 is not acquired*) {  
      *release resource1*;  
      return FAILED;  
    }  
    *acquire resource3*;  
    if(*resource3 is not acquired*) {  
      *release resource2*;  
      *release resource1*;  
      return FAILED;  
    }  
    ... // do something  
    *release resource3*;  
    *release resource2*;  
    *release resource1*;  
    return SUCCEEDED;  
  }  
  
（一个更复杂的具体例子可以参考[[16]](http://java.sun.com/docs/books/tutorial/essential/exceptions/advantages.html)）  
  
以上两种方案在可伸缩性方面的问题是显而易见的：一旦需要获取的资源多了以后代码也会随着越来越难以卒读，要么是if嵌套层次随之线性增多，要么是重复代码增多。所以即便项目中因为某些现实原因只能使用error-code，也最好采用前面提到的“On Error Goto”方案。  
  
**另一方面，当整个函数需要保持异常中立的时候，异常的优势就更显现出来了**：使用error-code，你还是需要一次次的小心check每个返回的错误值，从而阻止执行流带着错误往下继续执行。用异常的话，可以直接书写One True Path，连try-catch都不要。  
  
当然，即便是使用异常作为错误汇报机制，错误安全（error-safety）还是需要保证的。值得注意的是，错误安全性属于错误处理的本质困难，跟使用异常还是error-code来汇报错误没有关系，一个常见的谬误就是许多人把在异常使用过程中遇到的错误安全性方面的困难归咎到异常身上。

3.      **脆弱（易错）**。只要忘了检查任意一个错误代码，执行流就必然会带着错误状态往下继续执行，后者几乎肯定不是你想要的。带着错误状态往下执行好一点的会立即崩溃，差一点的则在相差十万八千里的地方引发一个莫名其妙的错误。

4.      **难以（编写时）确保和（review时）检查代码的正确性**。需要检查所有可能的错误代码有没有都被妥善check了，其中也许大部分都是不能直接对付而需要传播给上级的错误。

5.      **耦合**。即便你的函数是一个异常中立的函数，不管底层传上来哪些错误一律抛给上层，你仍然需要在每个调用的边界检查，并妥善往上手动传播每一个错误代码。而一旦底层接口增加、减少或改动错误代码的话，你的函数就需要立即作出相应改动，检查并传播底层接口改动后相应的错误代码——**这是很不幸的**，因为你的函数只是想保持异常中立，不管底层出什么错一律抛给上层调用方，这种情况下**理想情况应该是不管底层的错误语意如何修改，当前层都应该不需要改动才对**。

6.      **没有异常，根本无法编写泛型组件**。泛型组件根本不知道底层会出哪些错，泛型组件的特点之一便是错误中立。但用error-code的话，怎么做到错误中立？泛型组件该如何检查，检查哪些底层错误？唯一的办法就是让所有的底层错误都用统一的SUCCEEDED和FAILED代码来表达，并且将其它错误信息用GetLastError来获取。姑且不说这个方案的丑陋，如何、由谁来统一制定SUCCEEDED和FAILED、GetLastError的标准？就算有这个统一标准，你也可以设想一下某个标准库泛型算法（如for\_each）编写起来该是如何丑陋。

7.      **错误代码不可以被忘掉（忽视）**。忘掉的后果见第3条。此外，有时候我们可能会故意不管某些错误，并用一个万能catch来捕获所有未被捕获的错误，log，向支持网站发送错误报告，并重启程序。用异常这就很容易做到——只要写一个unhandled exception handler（不同语言对此的支持机制不一样）即可。

异常与错误代码的**本质区别之二**——**异常的传播使用的是一个单独的信道**，而错误代码则占用了函数的返回值；函数的返回值本来的语意是用来返回“有用的”结果的，这个结果是属于程序的One True Path的，而不是用来返回错误的。

利用返回值来传播错误导致的问题如下：

8.      **所有函数都必须将返回值预留给错误**。如果你的函数最自然的语意是返回一个double，而每个double都是有效的。不行，你得把这个返回值通道预留着给错误处理用。你可能会说，我的函数很简单，不会出错。但如果以后你修改了之后，函数复杂了呢？到那个时候再把返回的double改为int并加上一个double&作为out参数的话，改动可就大了。

9.      **返回值所能承载的错误信息是有限的**。NULL？-1？什么意思？具体的错误信息只能用GetLastError来提供… 哦，对了，你看见有多少人用过GetLastError的？

10.  **不优雅的代码**。呃…这个问题前面不是说过了么？不，这里说的是另一个不优雅之处——占用了用来返回结果的返回值通道。本来很自然的“计算——返回结果”，变成了“计算——修改out参数——返回错误”。当然，你可以说这个问题不是很严重。的确，将double res = readInput();改为double res; readInput(&res);也没什么大不了的。如果是连调用呢？比如，process(readInput());呃… 或者readInput() + …？或者一般地，op1(op2(), op3(), …);？

11.  **错误汇报方案不一致性**。看看Win32下面的错误汇报机制吧：HRESULT、BOOL、GetLastError…本质上就是因为利用返回值通道是一个补丁方案，错误处理是程序的一个方面（aspect），理应有其单独的汇报通道。利用异常的话，错误汇报方案就立即统一了，因为这是一个first-class的语言级支持机制。

12.  **有些函数根本无法返回值**，如构造函数。**有些函数返回值是语言限制好了的**，如重载的操作符和隐式转换函数。

异常与错误代码的**本质区别之三**——**异常本身能够携带任意丰富的信息**。

13.  **有什么错误报告机制能比错误报告本身就包含尽量丰富的信息更好的呢？**使用异常的话，你可以往异常类里面添加数据成员，添加成员函数，携带任意的信息（比如Java的异常类就缺省携带了非常有用的调用栈信息）。而错误代码就只有一个单薄的数字或字符串，要携带其它信息只能另外存在其它地方，并期望你能通过GetLastError去查看。

异常与错误代码的**本质区别之四**——**异常是OO的**。

14.  **你可以设计自己的异常继承体系**。呃…那这又有什么用呢？当然有用，一个最大的好处就是你可以在任意抽象层次上catch一组异常（exception grouping），比如你可以用catch(IOException)来捕获所有的IO异常，用catch(SQLException)来捕获所有的SQL异常。用catch(FileException)来catch所有的文件异常。你也可以catch更明确一点的异常，如StreamEndException。总之，catch的粒度是粗是细，根据需要，随你调节。当然了，你可以设计自己的新异常。能够catch一组相关异常的好处就是你可以很方便的对他们做统一的处理。

异常与错误代码的**本质区别之五**——**异常是强类型的**。

15.  **异常是强类型的**。在catch异常的时候，一个特定类型的catch只能catch类型匹配的异常。而用error-code的话，就跟enum一样，类型不安全。-1 == foo()？FAILED == foo()？MB\_OK == foo()？大家反正都是整数。

异常与错误代码的**本质区别之六**——**异常是first-class的语言机制**。

16.  **代码分析工具可以识别出异常并进行各种监测或分析**。比如PerfMon就会对程序中的异常做统计。这个好处放在未来时态下或许怎么都不应该小觑。

**选择什么错误处理机制？**

看完上面的比较，答案相信应该已经很明显了：**异常**。

如果你仍然是error-code的思维习惯的话，可以假想将所有error-code的地方改为抛出exception。需要注意的是，error-code不是status-code。并非所有返回值都是用来报告真正的错误的，有些只不过是控制程序流的。就算返回的是bool值（比如查找子串，返回是否查找到），也并不代表false的情况就是一个错误。具体参加上一节：“哪些情况不属于错误”。

一个最为广泛的误解就是：异常引入了不菲的开销，而error-code没有开销，所以应该使用error-code。这个论点的漏洞在于，它认为只要是开销就是有问题的，而不关心到底是在什么情况下的开销。实际上，[现代的编译器早已能够做到异常在happy path上的零开销](http://www.codeproject.com/cpp/exceptionhandler.asp)。当然，空间开销还是有的，因为零开销方案用的是地址表方案；但相较于时间开销，这里的空间开销几乎从来都不是个问题。另一方面，一旦发生了异常，程序肯定就出了问题，这个时候的时间开销往往就不那么重要了。此外有人会说，那如果频繁抛出异常呢？如果频繁抛出异常，往往就意味着那个异常对应的并非一个错误情况。

《C++ Coding Standards: 101 Rules, Guidelines, and Best Practices》里面一再强调：不要在项目里面关闭异常支持。因为就算你的项目里面不抛出异常，标准库也依赖于异常。一旦关闭异常，不仅你的项目代码都要依赖于error-code（error-code的缺点见下一节），整个标准库便也都要依赖于非标准的途径来汇报错误，或者干脆就不汇报错误。如果你的项目是如此的硬实时，乃至于你在非常小心且深入的分析之后发觉某些操作真的负担不起异常些微的空间开销和unhappy path上的时间开销的话，也要尽量别在全局关闭异常支持，而是尽量将这些敏感的操作集中到一个模块中，按模块关闭异常。

**插曲：异常的例外情况**

凡事都有例外。《C++ Coding Standards: 101 Rules, Guidelines, and Best Practices》上面陈述了两个例外情况：一，用异常没有带来明显的好处的时候：比如所有的错误都会在立即调用端解决掉或者在非常接近立即调用端的地方解决掉。二，在实际作了测定之后发现异常的抛出和捕获导致了显著的时间开销：这通常只有两种情况，要么是在内层循环里面，要么是因为被抛出的异常根本不对应于一个错误。

**如何进行错误处理？**

这个问题同样极其重要。它分为三个子问题：

1.      **何时抛出异常。**

2.      **何时捕获异常。**

3.      **如何避开异常，保持异常中立（或“异常透明”）。**

其中最后一个问题最为重要，属于错误处理的本质性困难之一。

先说前两个问题。

从本质上，错误分为两种，一种是可恢复的，另一种是不可恢复的。

对于可恢复的错误。有两种方案：

1.      **在错误的发生点上立即就予以恢复**。比如配置文件不存在便创建一个缺省的，某个配置项缺失就使用缺省值等等。这一方案的好处是当前函数不返回任何错误，因为错误被当即搞定了，就像没发生一样。这种方案的前提是当前函数必须要有对付该错误的足够上下文，如果一个底层的函数对全局语意没有足够的视图，这时就可以抛出异常，由上层函数负责恢复。

2.      **在某个上层栈上恢复**。这种情况下，在负责恢复的那层栈以下的调用一般被看成一个整体事务，其中发生的任何错误都导致整个事务回滚，回滚到错误恢复栈层面时，由相应的catch子句进行恢复，并重新执行整个事务，或者将程序引向另一条备选路径（alternative）。

对于不可恢复的错误。也有两种方案：

1.      **Sudden Death**。在错误的发生点上退出模块（可能伴随着重启模块）。退出模块前往往需要先释放资源、保存关键数据、记录日志，等等。该方案的前提是在错误的发生点的上下文中必须要**能够**释放所有资源，要**能够**保存关键数据。要满足这个前提，可以用一个全局的沙盒来保存整个模块到当前为止申请的所有资源，从而在任何出错点上都可以将这个沙盒整个释放掉。也可以用智能垃圾收集，这样在出错点上只要记录日志和保存数据，把扫尾工作留给智能垃圾收集器完成。这个方案的弱点是如果释放资源是要按某种次序的就比较麻烦。

2.      **回滚**。如果你并没有用智能垃圾收集（要智能到能够回收文件句柄，网络端口等，不光是内存），或者你并没有在某个全局可访问的位置保存到当前为止模块申请的所有资源，或者你的资源互相之间有依赖关系，必须按照分配的逆序释放，等等，那么就必须按照调用栈的反方向回滚事务。回滚到一个所谓的Fault Barrier，用一个catch-all在那里等着，所谓Fault-Barrier的作用就是为了抓这些没法妥善恢复的错误的，它做的事情通常就是logging、发送错误报告、可能也会重启模块。Fault Barrier一般在一个内聚的单一职责的功能模块的边界出现。

严格来说，其实还有第三种情况，即编写当前代码的时候并不确定某个错误是否能被恢复。这种时候抛出一个异常往往是最灵活的选择，因为在没有想好恢复方案的时候，上层调用对该异常都是中立的。一旦后面想好恢复方案了，不管是在某个上层调用内捕获该异常，还是在最底层错误发生点上就立即解决错误从而根本取消掉该异常，都没有问题。

**异常转换**

在如何抛出和捕获异常的问题上，还有一个子问题就是异常的转换（translation）。以下情况下应该转换一个由底层传上来的异常：

1.      抛出一个对应于当前抽象层的异常。比如Document::open当接收到底层的文件异常（或数据库异常，网络异常，取决于这个Document来自何方）时，将其转换为“文档无效或被破坏”异常，增加高层语意，并避免暴露底层实现细节（对异常的一个批评就是会暴露内部实现，而实际上，通过适当转换异常，可以使得异常总位于当前的抽象层次上，成为接口的一部分）。

2.      在模块边界上。如果一个模块，在内部使用异常，但在边界上必须提供C API的话，就必须在边界上捕获异常并将其转换为error-code。

**没有时间机器——错误处理的本质困难**

刚才提到“回滚”。那么，在异常发生的时候，如何回滚既然已经发生的操作？这就是要说的第三个问题：如何在异常从发生点一路传播到捕获点的路径上保持异常中立，即回滚操作，释放资源。简而言之就是要做到错误安全（error-safe）。错误安全攸关强异常安全保证和事务语意。在异常编程里面，错误安全是最重要的一环。

理想情况下，我们要的是一个时间机器：打碎的杯子要能还原，释放的内存要能重新得到，销毁的对象就像没销毁前一模一样，发射的导弹就像从来也没有离开发射筒一样，数据库就像从来没被写入一样…

没有时间机器。

那么，如何回滚木已成舟的操作？

目前有两个主要方案：

1.      **Discard（丢弃）**：一个例子就能够说明这种做法，源自STL的“copy-swap手法”。比如一个vector，你要往里面插入一个元素，如果插入元素失败的话你想要vector维持原状，就好像从来没有动过一样。如何做到这一点呢？你可以先把这个vector拷贝一份，往拷贝里面插入元素，然后将两个vector调换（swap）一下即可，swap是不会失败的，因为它只是把两个指针互换了一下。而如果往那个拷贝里面插入元素失败的话，拷贝就会被Discard（丢弃掉），不会带来任何实际的副作用。当然，这种做法是有代价的，谁叫你要[强异常安全保证](http://www.boost.org/more/generic_exception_safety.html)呢？再比如一个拷贝赋值操作符可以这样写：MyClass(other).swap(\*this); 当然，前提还是swap()必须具有标准的no-throw语意。  
  
这种做法一般化的描述就是：“**在一个‘副本’里把所有的事情都做好了，然后用一个不会出错的函数提交（commit）**”。这样一来，中途出了任何错误只要丢弃那个副本即可（往往只要任其析构）。要做到这一点，一个原则就是：“在破坏一份信息之前要确保其新的版本一定能够无错的替换掉原信息”，例如在拷贝构造函数中，不能先delete再new，因为new可能失败，一旦new失败了，delete掉的信息可就找不回来了。

2.      **Undo（撤销）**：有时候，你一方面不想付出**Discard**方案的（通常不菲的）空间开销，另一方面你又想拥有强异常安全保证。这也是有办法的，比如：  
  
  void World::addPerson(Person const& person)  
  {  
    m\_persons.push\_back(person);  
    **scope**(**failure**) { m\_persons.pop\_back(); }  
      
    … // other operations  
  }  
  
scope(failure)是D语言的特性，其语意显而易见：如果当前的scope以失败（异常）退出的话，{}内的语句就被执行。这么一来，在上面的例子中，如果后续的操作失败，那么这个person就会被从m\_persons中pop\_back出来，一次事务撤销，使得这个函数对m\_persons的影响归零。  
  
该方案的前提有两个：一，回滚操作（比如这里的m\_persons.pop\_back()）必须存在且不会失败（no-throw）。比如missile.lunch()就不能回滚，操作系统API一般也无法回滚；I/O操作也无法回滚；另一方面，内存操作一般而言都是可以回滚的，只要回复原来内存的值即可。二，被回滚的操作（比如这里的m\_persons.push\_back(person)）也一定要是强异常保证的。比如在中间而不是尾部插入一个person（m\_persons.insert(iter, person)）就不是强保证的，这种时候就要诉诸前一个方案（Discard）。  
  
D的scope(failure)（还有scope(exit)、scope(success)）是非常强大的设施。利用它，一个具有事务语意的函数的一般模式如下：  
  
  void Transaction()  
  {  
    op1; // strong operation  
    scope(failure) { *undo* op1; }  
    op2; // strong operation  
    scope(failure) { *undo* op2; }  
    …  
    opN; // strong operation  
    scope(failure) { *undo* opN; }  
  }  
  
在C++里面也可以模拟D的scope(failure)，Andrei Alexandrescu曾实现了一个ScopeGuard类[11]，而旨在完全模拟D的scope特性的[boost.scope-exit](http://lists.boost.org/boost-announce/2007/08/0139.php)也在review中。只不过C++03里面的模拟方案有一些学习曲线和使用注意点，C++09之后会有更方便的方案。在其它不支持scope(failure)的语言中，也可以模拟这种做法，不过做法很笨拙。

3.      呃… 哪来的第三个方案？前面不是说了只有两个方案吗？是的。因为这第三个方案是“理想”方案，目前还没有进入主流语言，不过在haskell里面已经初见端倪了。强异常安全保证的核心思想其实就是事务语意，而事务语意的核心思想就是“不成功便成仁”（这个思想有许多有趣的说法，比如：“[武士道原则](http://c2.com/cgi/wiki?SamuraiPrinciple)”、“[要么直着回来要么横着回来](http://c2.com/cgi/wiki?WithYourShieldOrOnIt)”、“[干不了就去死](http://c2.com/cgi/wiki?DoOrDie)”），根据这个想法，其实最简单的方案是把一组属于同一事务的操作简单地圈起来（标记出来），把回滚操作留给语言实现去完成：  
  
  stm {  
    op1;  
    op2;  
    …  
    opN;  
  }  
  
只要op1至opN中任意一个失败，整个stm块对内存的写操作就全被自动抛弃（这要求编译器和运行时的支持，一般是用一个缓冲区或写日志来实现），然后异常被自动抛出这个stm块之外。这个方案的优点是它太优美了，我们几乎只要关注One True Path即可，唯一要做的事情就是用stm{…}圈出代码中需要强保证的代码段。这个方案的缺点有两个：一，它跟第一个方案一样，有空间开销，不过空间开销通常要小一点，因为只要缓存特定的写操作。二，当涉及到操作系统API，I/O等“外部”操作的时候，底层实现就未必能够回滚这些操作了。另一个理论上的可能性是，当回滚操作和被回滚操作并非严格物理对应（所谓物理对应就是说，回滚操作将内存回滚到目标操作发生之前的状态）的时候，底层实现也不知道如何回滚。  
  
（STM（软件事务内存）目前在haskell里面实现了，Intel也释出了C/C++的STM预览版编译器。只不过STM原本的意图是实现锁无关算法的。后者就是另一个话题了。）

**RAII**

实际上刚才还有一个问题没有说，那就是如何确保资源一定会被释放（即便发生异常），这在D里面对应的是scope(exit)，在Java里面对应的是finally，在C#   里面对应的是scoped using。

简而言之就是，不管当前作用域以何种方式退出，某某操作（通常是资源释放）都一定要被执行。

这个问题的答案其实C++程序员们应该耳熟能详了：[RAII](http://www.octopull.demon.co.uk/c++/dragons/)。RAII是C++最为强大的特性之一。在C++里面，局部变量的析构函数刚好满足这个语意：无论当前作用域以何种方式退出，所有局部变量的析构函数都必然会被倒着调用一遍。所以只要将有待释放的资源包装在析构函数里面，就能够保证它们即便在异常发生的情况下也会被释放掉了。为此C++提供了一系列的智能指针：auto\_ptr、scoped\_ptr、scoped\_array… 此外所有的STL容器也都是RAII的。在C++里面模拟D的scope(exit)也是利用的RAII。

RAII相较于java的finally的好处和C#的scoped using的好处是非常明显的。只要一段代码就高下立判：

  // in Java  
  String ReadFirstLineFromFile( String path )  
  {  
    StreamReader r = null;  
    String s = null;  
    try {  
      r = new StreamReader(path);  
      s = r.ReadLine();  
    } finally {  
      if ( r != null ) r.Dispose();  
    }  
    return s;  
  }

  // in C#  
  String ReadFirstLineFromFile( String path )   
  {  
    using ( StreamReader r = new StreamReader(path) ) {  
      return r.ReadLine();  
    }  
  }

显然，Java版本的（try-finally）最臃肿。C#版本（scoped using）稍微好一些，但using毕竟也不属于程序员关心的代码逻辑，仍然属于代码噪音；况且如果不连续地申请N个资源的话，使用using就会造成层层嵌套结构。

如果使用RAII手法来封装StreamReader类的话（std::fstream就是RAII类的一个范例），代码就简化为：

  // in C++, using RAII  
  String ReadFirstLineFromFile(String path)  
  {  
    StreamReader r(path);  
    return r.ReadLine();  
  }

好处是显而易见的。完全不用担心资源的释放问题，代码也变得“as simple as possible”。此外，值得注意的是，以上代码只是演示了最简单的情况，其中需要释放的资源只有一个。其实这个例子并不能最明显地展现出RAII强大的地方，当需要释放的资源有多个的时候，RAII的真正强大之处才被展现出来，一般地说，如果一个函数依次申请N个资源：

  void f()  
  {  
    *acquire resource1*;    …   
    *acquire resource2*;    …   
    *acquire resourceN*;    …   
  }

那么，使用RAII的话，代码就像上面这样简单。无论何时退出当前作用域，所有已经构造初始化了的资源都会被析构函数自动释放掉。然而如果使用try-finally的话，f()就变成了：

  void f()  
  {  
    try {  
      *acquire resource1*;  
      … // #1  
      *acquire resource2*;  
      … // #2  
      *acquire resourceN*;  
      … // #N  
    } finally {  
      if(*resource1 is acquired*) *release resource1*;  
      if(*resource2 is acquired*) *release resource2*;  
      …  
      if(*resourceN is acquired*) *release resourceN*;  
    }  
  }

为什么会这么麻烦呢，本质上就是因为当执行流因异常跳到finally块中时，你并不知道执行流是从#1处、#2处…还是#N处跳过来的，所以你不知道应该释放哪些资源，只能挨个检查各个资源是否已经被申请了，如果已申请了便将其释放；要能够检查每个资源是否已经被申请了，往往就要求你要在函数一开始将各个资源的句柄全都初始化为null，这样才可以通过if(hResN==null)来检查第N个资源是否已经申请。

最后，RAII其实是scope(exit)的特殊形式。但在资源释放方面，RAII有其特有的优势：如果使用scope(exit)的话，每个资源分配之后都需要用一个scope(exit)跟在后面保护起来；而如果用RAII的话，一个资源申请就对应于一个RAII对象的构造，释放工作则被隐藏在对象的析构函数中，从而使代码主干保持了清爽。

**总结**

本文讨论了错误处理的原因、时机和方法。错误处理是编程中极其重要的一环；也是最被忽视的一环，Gosling把这个归因于大多数程序员在学校的时候都是做着大作业当编程练习的，而大作业鲜有老师要求要妥善对待错误的，大家都只要“work on the one true path”即可；然而现实世界的软件可必须面对各种各样的意外情况，往往一个程序的错误处理机制在第一次面对错误的时候便崩溃了… 另一方面，错误处理又是一个极其困难的问题，其本质困难来源于两个方面：一，哪些情况算是错误。二，如何做到错误安全（error-safe）。相较之下在什么地点解决错误倒是容易一些了。本文对错误处理的问题作了详细的分析，总结了多年来这个领域争论的结果，提供了一个实践导引。

**下期预告**

关于在C++里面为什么不应该使用异常规格声明（exception specification），参考文章后面的延伸阅读[5]。

关于如何设计异常继承体系，checked exception/unchecked exception之争，关于异常处理的迷思、谬论、误解，等等同样有意思的问题，请听下回分解（如果有下回的话:-)）。

**延伸阅读**

**书**

[1] 《C++ Coding Standards: 101 Rules, Guidelines, and Best Practices》，“Error Handling and Exceptions”部分，第68至75章。

[2] 《The C++ Programming Language, 3rd special edition》，第14章，尤其是14.4.7、14.5、14.8。第21章的21.3.6。以及附录E。

[3] 《The Design and Evolution of C++》，整个16章。

[4] 《Exceptional C++》，整本书。

[5] 《Exceptional C++ Style》，第11、12、13章。

[6] 《The Java Programming Language, 4th edition》，第12章。

[7] 《Effective Java》，Item39至Item47。

[8] 《Object Oriented Software Construction, 2nd edition》，第11、12章。

**文章**

[1] B. Stroustrup: [Programming with Exceptions](http://www.research.att.com/~bs/eh_brief.pdf)(pdf)

[2] B. Stroustrup: [Exception Safety: Concepts and Techniques](http://www.research.att.com/~bs/except.pdf)

[3] D. Abrahams: [Error and Exception Handling](http://www.boost.org/more/error_handling.html)

[4] D. Abrahams: [Exception-Safety in Generic Components](http://www.boost.org/more/generic_exception_safety.html)

[5] Tom Cargill: [Exception Handling, A False Sense of Security](http://www.informit.com/content/images/020163371x/supplements/Exception_Handling_Article.html)

[6] Damien Katz: [Error codes or Exceptions? Why is Reliable Software so Hard?](http://damienkatz.net/2006/04/error_code_vs_e.html)

[7] Alan Griffiths: [Here be Dragons](http://www.octopull.demon.co.uk/c++/dragons/)

[8] Ned Batchelder: [Exceptions vs. status returns](http://www.nedbatchelder.com/text/exceptions-vs-status.html)

[9] Ned Batchelder: [Exceptions in the rainforest](http://www.nedbatchelder.com/text/exceptions-in-the-rainforest.html)

[10] Ned Batchelder: [Fix error handling first](http://www.nedbatchelder.com/text/fix-err-hand.html)

[11] A. Alexandrescu, P. Marginean: [Change the Way You Write Exception-Safe Code — Forever](http://www.ddj.com/cpp/184403758)

[12] Brad Abrams: [Exceptions and Error Codes](http://blogs.msdn.com/brada/archive/2003/09/30/50403.aspx)

[13] A. Alexandrescu: [Exception Safety Analysis](http://erdani.org/publications/cuj-2003-12.pdf)(pdf)

[14] Brian Goetz: [Exceptional practices](http://www.javaworld.com/javaworld/jw-08-2001/jw-0803-exceptions.html?)

[15] James Gosling: [Failure and Exceptions](http://www.artima.com/intv/solid.html)

[16] Sun: [Advantages of Exceptions](http://java.sun.com/docs/books/tutorial/essential/exceptions/advantages.html)

[17] Vishal Kochhar: [How a C++ compiler implements exception handling](http://www.codeproject.com/cpp/exceptionhandler.asp)