C++语言异常处理机制的研究*)

裘宗燕
(北京大学数学学院信息科学系 北京100871)

On the Exception Handling of C++

QIU Zong-Yan
(Department of Informatics, Peking University, Beijing 100871)

Abstract Here we make a detailed investigation on the exception handling mechanism of C++, have much discussion on many design and implementation problems, and offer many suggestions on the use of it.

Keywords Exception handling, Object-oriented programming languages, C++

在程序开发中必须考虑和处理程序运行中可能遇到的非正常情况,这些情况的出现可能是由于环境因素(如硬件或用户交互),也可能源自程序不同部分间的相互作用。20世纪80年代末的一项研究表明,一个程序里处理错误的代码可能达到整个代码量的2/3。由此人们逐渐认识到需要在高级语言层上有效描述程序运行中非正常处理的控制流,并为此提供清晰易用的语言机制。这就导致了各种新高级语言中的异常处理机制,作为支持可靠软件实现的基础。C++语言也以一种实用方式提供了异常处理。然而,由于目前对异常处理机制的研究还不够深入,对许多问题的处理还是试探性的。C++的异常机制中也存在着许多不清晰的地方。本文对C++语言的这一机制做了比较细致的分析,以帮助实际的C++使用者了解其中的问题和陷阱,也可能为异常机制的设计提供一些线索。

1 C++语言的异常处理机制

C++通过try块及附着其后的catch子句(异常处理器)描述对异常的监测范围与处理方式,通过throw语句抛出异常。C++的运行系统不会抛出异常,所谓"标准异常"不过是C++标准库定义的一组异常。所有基本运行错误,例如内存的违规访问、算术运算错误等等,都不会转变为异常,不能通过语言的异常机制处理。

下面是一段包含异常处理部分的代码示例:

```
int f (… …)
try {
    … …
    try {
        … …
        throw E(… …);
        … …
    }
    catch (A& a){ … … }
    catch (const B& b){ … … }
    … …
}
catch (const C& c){ … … }
catch (D d){ … … }
```

这里函数f的体就是一个try块,其后附着了两个异常处理器,使f的整个函数体成为这两个处理器的监视范围。在函数体内部还有另一个try块。

C++程序的每个异常都属于某个特定类型。程序完全可以抛出int或double类型的异常,但语言的设计意图是希望用特别定义的类表示异常。这样可以提高程序的可读性,还使异常处理的设计中能利用类的继承性质。在C++里,一个以B作为参数类型的异常处理器不仅可以捕捉类型为B的异常,还可以捕捉B的派生类型的异常对象。此外,通过适当定义异常类,可以利用异常对象将抛出点任意数量的信息携带到相应处理器去。

如果一个程序在运行中抛出了异常e,其正常执行流立即中断,控制将转到某个能处理e的处理器。如果抛出点位于一个try块中,该try块后的各处理器就会被顺序检查。如果这里有一个与e类型匹配的处理器,程序就转去执行这个处理器体的代码。该处理器完成后,执行过程将在这一try块之后转回正常控制流。上例中的throw语句位于两个嵌套的try块中,如果异常由这里抛出,这两个try块之后的异常处理器都可能被检查。

如果所抛出异常不能在当前函数内部处理,该函数将立即结束,同一异常将从该函数的调用点再次抛出。这样就可能形成一个沿函数调用链查询异常处理器的动态过程。从抛出一个异常到它被捕捉期间可能退出一系列函数。异常处理中的这种非局部控制转移完全由抛出时的动态情况确定,是超越正常控制流之外的。下面将仔细分析这种过程的内涵。

C++还允许在函数定义中描述函数运行时可能抛出的异常,这种描述可能用于编译程序的检查和优化。下面函数g的原型声明带有异常描述部分:

```
int g(… …)throw(B,D);
```

这说明函数g在执行中可能抛出类型为B和D的异常。编译程序可以检查在g的调用环境里是否捕捉这些异常,以及调用它的函数的异常描述是否正确等等。

2 对C++异常处理机制的分析和研究

本节分析研究C++的异常处理机制设计所导致的一些情况及问题。

2.1 异常机制的结构设计

采用try块是一种设计选择,这使程序正文被划分为异常监测区和非监测区。有的程序语言允许将异常处理器附在任何作用域,甚至可以附着在普通语句或表达式之后,这就使程序中任何部分都需要监测,监测区可能形成复杂的嵌套结构。在C++里,try块清晰而有限(设计良好的程序中的函数

*)国家自然科学基金资助(No.60173003)。裘宗燕 教授,主要工作领域为程序理论、程序设计语言和程序技术等。

通常都很小，其中不会有很多 try 块），为处理异常而保存的信息不会太多，有利于提高异常处理的效率。

2.2 异常处理器查询中的局部与非局部控制转移

一个异常被抛出后，查寻处理器的过程分两个阶段。第一阶段在抛出异常的 throw 语句所在的函数里进行。此阶段能否找到对应异常处理器的问题常可以静态确定，通常就是静态的类型检查问题（此时可能出现动态类型问题）。在这种情况下，目标代码完全可以通过简单转跳实现从异常抛出位置到对应处理器的转移。然而，我们通过简单实例程序发现，所试验的编译程序都没有做这种优化。在所有试验中，由异常引起的局部控制转移比简单转跳慢得多，在某些实现不佳的 C++ 系统里，两者的效率差可达数千倍。这告诉我们，至少在目前系统里，不应该把抛出异常用作退出多层控制结构的一种机制。

如果函数内部不能处理抛出的异常，这一事件的发生就会导致非局部的控制转移，在这种转移过程中会出现许多特殊的情况，下面许多讨论都是分析在此期间产生的现象。

2.3 异常对象的生存期

抛出异常的最常见描述方式是（实际上 throw 之后可以写任意表达式）：

```
throw E();
```

假定 E 是表示异常的类，这一语句执行时创建一个 E 类型的临时对象。按照 C++ 的一般规定，临时对象的生存期在创建该对象的最外层表达式计算完成时结束。若按这种规定，上面语句创建的异常对象在语句结束时就销毁。对于临时异常对象，C++ 做了特别规定：throw 语句创建的临时异常对象将一直生存到捕捉它并完成处理的那个 catch 子句的结束处。如果在 catch 子句里用“throw;”再次抛出，抛出的仍是原来捕捉的那个临时对象（而不是另一对象，有关情况见下面分析），这将使该异常对象继续生存到下一次被捕捉和处理。另外，如果程序中抛出了异常却找不到对应的处理器，它就会转去调用特殊函数 terminate()，此时那个临时异常对象的生存期也将结束。

2.4 异常对象的复制和引用

在 catch 关键字后描述异常处理器所捕捉异常对象的类型，这里只有如下五种形式：

```
catch (B){ /* 处理语句 */ }
catch (B b){ /* 处理语句 */ }
catch (B& b){ /* 处理语句 */ }
catch (const B& b){ /* 处理语句 */ }
catch (…){ /* 处理语句 */ }
```

第一种形式表示捕捉 B 类型异常，但不关心有关的具体异常对象本身。第二种是“值参数”形式，捕捉时做异常对象的复制。第三种和第四种形式的形参 b 引用被捕获的异常对象。最后一种形式表示本处理器捕捉监视区域中抛出的一切异常。一个类型为 E 的异常对象能被以类型为 B 的处理器捕捉，当且仅当 B 是 E 的一个无歧义的 public 基类。C++ 不允许为一个异常处理器描述多个捕捉类型，它解决这类问题主要靠类型的层次结构。

第一和第五种情况的处理器中不访问捕捉的对象。但也应注意，在这两类处理器执行时，被捕获的异常对象尚未销毁。如果处理器执行语句“throw;”（不带参数），原来那个异常对象将被再次抛出。如果不再次抛出，处理器结束时将销毁捕捉到的那个异常对象。

在捕获异常时，“参数”传递采用函数的参数传递语义。对第二种到第四种参数形式，这一规定都将产生深刻影响。第二种形式下捕捉异常时有一次对象复制，用捕获的那个异常对象初始化处理器的形参 b。如果类型 B 是实际异常的基类型，复制中就可能出现对象切割，因为 b 可能不包含实际异常对象的某些数据域。如果异常类中存在虚函数，处理器内的调用同样按照一般的规则处理。如果写：

```
catch (B b){/* … */throw;}
```

最后抛出的并不是本处理器内处理的那个局部对象，而是原来捕捉的对象。这将与：

```
catch (B b){ /* … */throw b;}
```

不同。后一写法用处理器内的局部对象 b 初始化一个新创建的 B 类型临时异常对象并抛出。这个新异常对象的类型未必与处理器捕获的异常一致，而且是另一个对象。至此，被捕获的异常对象的生存期也结束了。

两种引用形式也值得注意。按照一般规则，只有 const 引用参数才能引用临时对象。C++ 特别规定在异常处理器的参数表里可以使用非 const 引用，这也意味着处理器能够修改捕捉到的那个异常对象，而后再通过“throw;”抛出那个对象。如果采用 const 引用，那么就意味着不准备修改捕捉的异常对象。

有时程序中发生的异常需要经过多个处理器逐步处理。完成这类处理的一种方式是采用非 const 引用参数，直接在原始异常对象里“积累”信息。另一种可能方式是在处理的每一步另行创建新的异常对象。采用后一种做法时还必须注意一个问题：如果需要在新建异常对象中保留所捕获的异常对象的信息，就必须复制有关的信息。因为，如果一个处理器没有（通过 throw;）再次抛出捕捉的对象，这个处理器结束时该对象就会销毁，直接引用那个对象就会导致悬空的引用（dangling reference）。

2.5 异常的存储问题

代表异常的临时对象在哪里创建？标准对此没有给出明确回答。由于异常对象的生存期情况，如在堆栈分配就可能要做多次复制（异常可能导致多个函数结束，处理器也可能调用其他函数并使用堆栈）。在堆中分配也有问题，因为导致异常的一个常见原因是存储耗尽，如果采用堆分配，而创建异常对象时空间耗尽（表示异常对象的类可以要求为创建对象分配动态空间），系统就无法创建这个异常，因此也就不可能处理了。按照有关存储管理的规定，存储耗尽时还是可以抛出 bad_alloc 异常，而且在创建 bad_alloc 异常时保证不会抛出新异常（这意味着 C++ 的运行系统为创建 bad_alloc 异常保留了空间）。

由上述分析可以看出，定义复杂的异常对象类，希望在抛出异常时将许多信息携带到处理点，是一种可能开销很大，本身也可能造成危险的方式。我们应该注意这种可能性。

2.6 异常类型的作用域

C++ 的异常匹配按类型进行，这是一种非常有趣也很有意义的设计，带来了许多应用可能性，主要是可以利用语言支持的类层次结构，使一个处理器可以自然地处理一组相关类型的异常。但也应看到问题的另一面：由于 C++ 的类型可以是局部的，而异常处理又是一种完全动态确定的机制，两者的交互作用会产生许多出人意料的结果。

首先看异常对象的类型识别，实际上，完成这一工作要靠

（下转第174页）

$$\mathrm{Min}[C_k(i,i+1)]=\begin{cases}R+P & L(i,i+1)>\Delta\\ L(i,i+1) & L(i,i+1)\leqslant\Delta\end{cases}$$

则当 $\Delta=R+P$ 时，VMIN 策略的效果可达最优。

VMIN 算法的前提是需预先知道进程的整个访问序列，这在实际操作中很难实现，此方法可以作为评价其他置换策略的标准。在置换策略的实施中，除用控制参数 Δ 的方法外，系统还可以通过缺页频率来调整工作集，缺页率高时则扩大工作集；缺页率低时则缩小工作集。

4 抖动与程序的行为特性

很多的存储管理方法考虑的出发点是基于程序的特性——局部性概念。程序局部化程度越高，程序执行时可经常集中在几个页面上访问，可减少缺页中断的次数，从而防止抖动现象的发生。程序的局部性包括时间局部性和空间局部性。

(1)时间局部性：是指一旦某个位置(数据或指令)被访问了，它常常很快又要被再次访问。这种现象体现在程序的循环结构以及变量和子程序等程序结构中。

(2)空间局部性：是指一旦某个位置被访问了，那么它附近的位置很快也要被访问到。这种现象体现在顺序的指令序列、线性的数据结构等程序结构中。

程序的行为特性与系统的缺页率，以至整个系统的效率有一定的关系。我们在进行程序设计的时候，要力求减少程序访问的离散性，提高程序访问的局部性。下面的例子很好地说明了程序的行为特性对缺页率的影响。

假定在页面大小为512字的分页系统中对矩阵 $A_{512\times512}$ 赋值，最基本的编码方式有以下两种：

程序编制方法1：

```
var A array [1..512] of array [1..512] of integer;
  For j:=1 to 512
    For i:=1 to 512
      A[i,j]:=0;
```

程序编制方法2：

```
var A array [1..512] of array [1..512] of integer;
  For i:=1 to 512
    For j:=1 to 512
      A[i,j]:=0;
```

方法1对矩阵赋值的执行顺序为：A[1,1]，A[2,1]，A[3,1]……A[512,1]，A[1,2]，A[2,2]，A[3,2]，……。方法2对矩阵赋值的执行顺序为：A[1,1]，A[1,2]，A[1,3]……A[1,512]，A[2,1]，A[2,2]，A[2,3]，……。矩阵在内存中是按行存放的，对512字的页面大小来说，每行刚好占一页。若系统分给这个进程只有二个物理实页，对方法1，因其使用矩阵的顺序是按列进行的，将引起512×(512/2)=131072次缺页中断。对方法2，因其使用矩阵的顺序是按行进行的，引起的缺页中断次数为512/2=256次。

结束语 抖动现象存在于虚拟存储管理的系统中，减少或消除抖动现象，对于充分发挥系统的性能，提高系统的效率具有非常重要的意义。引入工作集，采用适当的页面置换策略等方法，可防止系统抖动现象的发生，在满足各进程工作集的前提下，使内存的多道程序度接近最佳值。

参 考 文 献

1 冯耀霖，杜舜国. 操作系统. 西安电子科技大学出版社，2000. 92～100

2 Madnick S E, Donovan J J. OPERATING SYSTEM. Praha: Nakl. techn. Lit. ,1983. 564～575

3 汤子瀛，哲凤屏，汤小丹. 计算机操作系统. 西安电子科技大学出版社，2001. 165～186

4 张尧学，史美林. 计算机操作系统教程. 清华大学出版社，1993

5 邹鹏，王广芳. 操作系统原理. 国防科技大学出版社，1996. 110～120

(上接第156页)

类型声明描述中的继承关系。假定类型 E 是由类型 B 采用 public 方式派生的类型。如果程序运行中抛出了一个 E 类型的异常 e，这个异常可能传到一个在 E 类型定义的作用域之外的函数里，如果那里确实定义了要求捕捉 B 类型异常的处理器，那么这一处理器能否处理异常 e?这一问题无论怎样解释都有不合理的地方。按照实际的类型继承关系，e 应该在这里被捕捉和处理。但是这一实际类型关系在处理器的位置上又是不可见的。

此外，如果异常对象被传到其类型定义的作用域之外，还可能导致在需要复制或销毁该对象时，无法执行有关复制构造函数或析构函数。一种典型情况是异常在其类定义的作用域之外被通用处理器捕获，而这个类又定义了特殊析构函数，这时程序将无法调用该函数。注意，由于异常传播完全是动态确定的过程，编译处理时不可能对这种问题进行完全的检查。

这些情况告诉我们，表示异常的基类和派生类最好一起定义。这些类不宜具有复杂语义，例如定义具有特殊语义的复制语义和特定析构函数等等。

2.7 异常处理的实现和代价

可以将异常看成一种非局部控制转移机制。然而，一般来说，异常处理的代价远高于函数调用和退出。目前异常处理有两种基本实现方式：动态链方式和静态表方式。它们各有长短。动态链方式在出现异常时的效率略高，且能很好支持动态连接等。但采用这种方式，程序运行中即使没有抛出异常，也要付出额外开销。静态表方式采用静态的监视范围表，如果程序未实际抛出异常，就无需为异常处理机制的存在付出额外代价。但是，发生异常时就需要多次查询这个表，因此效率较低。这种方式的另一缺点是难以支持动态连接。C++异常机制的设计使之可以支持静态表方式，主要是为了它“不用就无需付出代价”的设计原则。

由于 C++异常的高代价特征，因此，人们建议只将异常处理机制用于真正需要处理异常的情况，而不要将它用作一种非局部转移的方便手段。当然，对于那些性能要求并不高，而可靠性要求很高的程序或者程序部分，这一问题的重要性就大大降低了。

总结 本文中对 C++语言的异常处理机制进行了深入细致的分析，包括其中的一些静态语法特征、动态性质、实现中的问题以及它们所可能造成的影响。通过这些分析，我们可以看到在程序语言，特别是面向对象语言中一般的异常处理机制所牵涉到的许多问题。这一分析也能帮助实际程序语言的使用者理解应该如何使用这方面的语言机制。

参 考 文 献

1 Koenig A, Stroustrup B. Exception Handling in C++. Jounal of Object Oriented Programming, 1990, 3(2): 16～33

2 Stroustroup B. The Design and Evolution of C++, Addison-Wesley, 1994

3 Buhr P A, Mok W Y R. Advanced exception handling mechanisms. IEEE Trans, on SE, 2000, 26(9): 820～836

4 Knudsen J L. Fault tolerance and exception handling in BETA, LNCS 2022, 1～17, Springer-Verlag, 2002

5 Dony C. A full object-orinted exception handling system: rationale and Smalltalk Implementation. Springer-Verlag, 2002. 18～38