C++异常处理的编程方法

主人公介绍

阿愚，曾经是一位小小程序员，现在仍是一位小小程序员，将来也还是一位小小程序员。读程序、写程序一直就是他的最爱，这过程给他带来许许多多的快乐，虽然期间也有过一些迷茫，但每次冲过迷雾，重见阳光的喜悦总是他对程序人生的更加执著追求。雨过天晴的空气才是最清新的。

异常处理的编程方法，程序员都很熟悉的一个东东，她和面向对象的方法是软件程序设计发展史上其中最重要的两项革新技术。现代程序设计语言拥有的一个重要的特性就是能较好地支持异常的处理（Exception Handling）。她就像一位美丽而优雅的公主，帮助程序员写出来的代码总是那样的整齐美观、层次清晰；同时它好像还是一位贤惠能干的贤内助，总能帮你料理好由于考虑不全所留下的多多少少的意外事件，她在背后默默的支持你的一切，使你写出来的作品是那样的高效、安全和完美。瞧！它深深地打动了我们我们的主人公阿愚，并续上了一段美丽的编程爱情故事。

内容的组织及编排

相遇篇

《第 1 集 初次与异常处理编程相邂逅》
《第 2 集 C++中异常处理的游戏规则》
《第 3 集 C++中 catch（…）如何使用》
《第 4 集 C++的异常处理和面向对象的紧密关系》
《第 5 集 C++的异常 rethrow》

相知篇

《第 6 集 对象的成员函数中抛出的异常》
《第 7 集 构造函数中抛出的异常》
《第 8 集 析构函数中抛出的异常》
《第 9 集 C++的异常对象如何传送》
《第 10 集 C++的异常对象按传值的方式被传递》
《第 11 集 C++的异常对象按引用方式被传递》
《第 12 集 C++的异常对象按指针方式被传递》
《第 13 集 C++异常对象三种方式传递的综合比较》
《第 14 集 再探 C++中异常的 rethrow》
《第 15 集 C 语言中的异常处理机制》
《第 16 集 C 语言中一种更优雅的异常处理机制》
《第 17 集 全面了解 setjmp 与 longjmp 的使用》
《第 18 集 玩转 setjmp 与 longjmp》
《第 19 集 setjmp 与 longjmp 机制，很难与 C++和睦相处》

相爱篇

爱的秘密

爱的结晶

第 1 集 初次与异常处理编程相邂逅

和其它很多程序员一样，本书的主人公阿愚也是在初学 C++时，在 C++的 sample 代码中与异常处理的编程方法初次邂逅的，如下：

```
// Normal program statements
...

try
{
// Execute some code that might throw an exception.
}
catch( CException* e )
```

```
{
// Handle the exception here.
// "e" contains information about the exception.
e->Delete();
}

// Other normal program statements
```

瞧瞧，代码看上去显得那么整齐、干净，try block 和 catch block 遥相呼应，多有对称美呀！因此主人公初次见面后就一见钟情了。

为什么要选用异常处理的编程方法？

当然更为重要的是，C++中引入的异常处理的编程机制提供给程序员一种全新的、更好的编程方法和思想。在 C++中明确提出 trycatch 异常处理编程方法的框架之前的年代，程序员是怎样编写程序的，如下：

```
void main(int argc, char* argv[])
{
if (Call_Func1(in, param out)
{
// 函数调用成功，我们正常的处理
if (Call_Func2(in, param out)
{
// 函数调用成功，我们正常的处理
while(condition)
{
//do other job
if (has error)
{
// 函数调用失败，表明程序执行过程中出现一些错误，
// 因此必须处理错误
process_error();
exit();
}
//do other job
}
}
else
{
// 函数调用失败，表明程序执行过程中出现一些错误，
// 因此必须处理错误
process_error();
exit();
}

}
```

```
else
{
// 函数调用失败，同样是错误处理
process_error();
exit();
}
}
```

因为程序的执行过程中总会遇到许多可预知或不可预知的错误事件，例如说，由于内存资源有限导致需要分配的内存失败了；或某个目录下本应存在的一个文件找不着了；或说不小心被零除了、内存越界了、数组越界了等等。这些错误事件存在非常大的隐患，因此程序员总需要在程序中不断加入 if 语句，来判断是否有异常出现，如果有，就必须要及时处理，否则可能带来意想不到的，甚至是灾难性的后果。这样一来，程序可读性差了很多，总是有许多与真正工作无关的代码，而且也给程序员增加了极大的工作负担，多数类似的处理错误的代码模块就像满山的牛屎一样遍地都是（程序员不大多是“牛”人吗？所以。。。哈哈）。

但 C++中的异常处理的机制彻底改变了这种面貌，它使真正的计算处理和错误处理分开来，让程序员不再被这些琐碎的事情所烦扰，能关注于真正的计算处理工作。同时代码的可读性也好了。因此我们有理由选择异常处理的编程方法。具体原因如下：

1、 把错误处理和真正的工作分开来；

2、 代码更易组织，更清晰，复杂的工作任务更容易实现；

3、 毫无疑问，更安全了，不至于由于一些小的疏忽而使程序意外崩溃了；

4、 由于 C++中的 try catch 可以分层嵌套，所以它提供了一种方法使得程序的控制流可以安全的跳转到上层（或者上上层）的错误处理模块中去。（不同于 return 语句，异常处理的控制流是可以安全地跨越一个或多个函数 ）。

5、 还有一个重要的原因就是，由于目前需要开发的软件产品总是变得越来越复杂、越来越庞大，如果系统中没有一个可靠的异常处理模型，那必定是一件十分糟糕的局面。

相信绝大多数程序员都知道 C++中的异常处理的编程方法，可还是有很多人已习惯原来单纯的面向过程的代码组织方式，不太习惯或较少使用 trycatch 异常处理。为了使您编写的代码更安全；为了使您编写的代码让他人更易阅读，主人公阿愚强烈建议在您书写的代码中尽可能多用异常处理机制，少一些不必要的 if 判断语句。

下一集详细介绍 C++中的异常处理的语法。

第 2 集 C++中异常处理的游戏规则

如果您喜欢玩一款游戏，您必须先要很好理解这款游戏的规则。同样主人公阿愚喜欢上 C++中异常处理后，当然也首先关注它的游戏规则，这就是 C++中异常处理的语法。

关键字

1、 try

2、 catch

3、 throw

其中关键字 try 表示定义一个受到监控、受到保护的程序代码块；关键字 catch 与 try 遥相呼应，定义当 try block（受监控的程序块）出现异常时，错误处理的程序模块，并且每个 catch block 都带一个参数（类似于函数定义时的数那样），这个参数的数据类型用于异常对象的数据类型进行匹配；而 throw 则是检测到一个异常错误发生后向外抛出一个异常事件，通知对应的 catch 程序块执行对应的错误处理。

语法

1、还是给一个例子吧！如下：

```
int main()
{
cout << "In main." << endl;

//定义一个 try block，它是用一对花括号{}所括起来的块作用域的代码块
try
{
cout << "在  try block  中, 准备抛出一个异常." << endl;

//这里抛出一个异常（其中异常对象的数据类型是 int，值为 1）
//由于在 try block 中的代码是受到监控保护的，所以抛出异常后，程序的
//控制流便转到随后的 catch block 中
throw 1;

cout << "在  try block  中, 由于前面抛出了一个异常，因此这里的代码是不会得以执行到的" << endl;
}
//这里必须相对应地，至少定义一个 catch block，同样它也是用花括号括起来的
catch( int& value )
{
cout << "在  catch block  中, 处理异常错误。异常对象 value 的值为："<< value << endl;
}

cout << "Back in main. Execution resumes here." << endl;
return 0;

}
```

2、语法很简单吧！的确如此。另外一个 try block 可以有多个对应的 catch block，可为什么要多个 catch block 呢？这是因为每个 catch block 匹配一种类型的异常错误对象的处理，多个 catch block 呢就可以针对不同的异常错误类型分别处理。毕竟异常错误也是分级别的呀！有致命的、有一般的、有警告的，甚至还有的只是事件通知。例子如下：

```
int main()
{
try
{
cout << "在  try block  中,  准备抛出一个 int 数据类型的异常." << endl;
throw 1;

cout << "在  try block  中,  准备抛出一个 double 数据类型的异常." << endl;
throw 0.5;
}
catch( int& value )
{
cout << "在  catch block  中, int 数据类型处理异常错误。”<< endl;
}
catch( double& d_value )
{
cout << "在  catch block  中, double 数据类型处理异常错误。”<< endl;
}

return 0;
}
```

3、一个函数中可以有多个 trycatch 结构块，例子如下：

```
int main()
{
try
{
cout << "在  try block  中,  准备抛出一个 int 数据类型的异常." << endl;
throw 1;
}
catch( int& value )
{
cout << "在  catch block  中, int 数据类型处理异常错误。”<< endl;
}

//这里是二个 trycatch 结构块，当然也可以有第三、第四个，甚至更多
try
{
cout << "在  try block  中,  准备抛出一个 double 数据类型的异常." << endl;
throw 0.5;
}
catch( double& d_value )
{
```

```
cout << "在  catch block  中, double 数据类型处理异常错误。”<< endl;
}

return 0;
}
```

4、上面提到一个 try block 可以有多个对应的 catch block，这样便于不同的异常错误分类处理，其实这只是异常错误分类处理的方法之一（暂且把它叫做横向展开的吧！）。另外还有一种就是纵向的，也即是分层的、trycatch 块是可以嵌套的，当在低层的 trycatch 结构块中不能匹配到相同类型的 catch block 时，它就会到上层的 trycatch 块中去寻找匹配到正确的 catch block 异常处理模块。例程如下：

```
int main()
{
try
{
//这里是嵌套的 trycatch 结构块
try
{
cout << "在  try block  中,  准备抛出一个 int 数据类型的异常." << endl;
throw 1;
}
catch( int& value )
{
cout << "在  catch block  中, int 数据类型处理异常错误。”<< endl;
}

cout << "在  try block  中,  准备抛出一个 double 数据类型的异常." << endl;
throw 0.5;
}
catch( double& d_value )
{
cout << "在  catch block  中, double 数据类型处理异常错误。”<< endl;
}

return 0;
}
```

5、讲到是 trycatch 块是可以嵌套分层的，并且通过异常对象的数据类型来进行匹配，以找到正确的 catch block 异常错误处理代码。这里就不得不详细叙述一下通过异常对象的数据类型来进行匹配找到正确的 catch block 的过程。

（1） 首先在抛出异常的 trycatch 块中查找 catch block，按顺序先是与第一个 catch block 块匹配，如果抛出的异常对象的数据类型与 catch block 中传入的异常对象的临时变量（就是 catch 语句后面参数）的数据类型完全相同，或是它的子类型对象，则匹配成功，进入到 catch block 中执行；否则到二步；

（2） 如果有二个或更多的 catch block，则继续查找匹配第二个、第三个，乃至最后一个 catch block，如匹配成功，则进入到对应的 catch block 中执行；否则到三步；

（3） 返回到上一级的 trycatch 块中，按规则继续查找对应的 catch block。如果找到，进入到对应的 catch block 中执行；否则到四步；

（4） 再到上上级的 trycatch 块中，如此不断递归，直到匹配到顶级的 trycatch 块中的最后一个 catch block，如果找到，进入到对应的 catch block 中执行；否则程序将会执行 terminate()退出。

另外分层嵌套的 trycatch 块是可以跨越函数作用域的，例程如下：

```
void Func() throw()
{
//这里实际上也是嵌套在里层的 trycatch 结构块
try
{
cout << "在  try block  中,  准备抛出一个 int 数据类型的异常." << endl;
//由于这个 trycatch 块中不能找到匹配的 catch block，所以
//它会继续查找到调用这个函数的上层函数的 trycatch 块。
throw 1;
}
catch( float& value )
{
cout << "在  catch block  中, int 数据类型处理异常错误。”<< endl;
}
}

int main()
{
try
{
Func();

cout << "在  try block  中,  准备抛出一个 double 数据类型的异常." << endl;
throw 0.5;
}
catch( double& d_value )
{
cout << "在  catch block  中, double 数据类型处理异常错误。”<< endl;
}
catch( int& value )
{
//这个例子中，Func()函数中抛出的异常会在此被处理
cout << "在  catch block  中, int 数据类型处理异常错误。”<< endl;
}
```

```
return 0;
}
```

6、刚才提到，嵌套的 trycatch 块是可以跨越函数作用域的，其实这里面还有另外一层涵义，就是抛出异常对象的函数中并不一定必须存在 trycatch 块，它可以是调用这个函数的上层函数中存在 trycatch 块，这样这个函数的代码也同样是受保护、受监控的代码；当然即便是上层调用函数不存在 trycatch 块，也只是不能找到处理这类异常对象错误处理的 catch block 而已，例程如下：

```
void Func() throw()
{
//这里实际上也是嵌套在里层的 trycatch 结构块
//由于这个函数中是没有 trycatch 块的，所以它会查找到调用这个函数的上
//层函数的 trycatch 块中。
throw 1;
}

int main()
{
try
{
//调用函数，注意这个函数里面抛出一个异常对象
Func();

cout << "在  try block  中, 准备抛出一个 double 数据类型的异常." << endl;
throw 0.5;
}
catch( double& d_value )
{
cout << "在  catch block  中, double 数据类型处理异常错误。”<< endl;
}
catch( int& value )
{
//这个例子中，Func()函数中抛出的异常会在此被处理
cout << "在  catch block  中, int 数据类型处理异常错误。”<< endl;
}

//如果这里调用这个函数，那么由于 main()已经是调用栈的顶层函数，因此不能找
//到对应的 catch block，所以程序会执行 terminate()退出。
Func();
// [特别提示]：在 C++标准中规定，可以在程序任何地方 throw 一个异常对象，
//  并不要求一定只能是在受到 try block 监控保护的作用域中才能抛出异常，但
//  如果在程序中出现了抛出的找不到对应 catch block 的异常对象时，C++标
//  准中规定要求系统必须执行 terminate()来终止程序。
//  因此这个例程是可以编译通过的，但运行时却会异常终止。这往往给软件
//  系统带来了不安全性。与此形成对比的是 java 中提供的异常处理模型却是不
```

```
// 永许出现这样的找不到对应 catch block 的异常对象，它在编译时就给出错误
// 提示，所以 java 中提供的异常处理模型往往比 C++要更完善，后面的章节
// 会进一步对这两种异常处理模型进行一个详细的分析比较。
return 0;
}
```

朋友们！C++中的异常处理模型的语法很简单吧！就是那么（one、two、three、…哈哈！数数呢！）简单的几条规则。怪不得主人公阿愚这么快就喜欢上她了，而且还居然像一个思想家一样总结出一条感想:好的东西往往都是简单的，简单就是美吗！哈哈！还挺臭美的。

下一集主人公阿愚愿和大家一起讨论一下 C++中的异常处理中的一种特殊的 catch 用法，那就是关于 catch(…)大探秘。

第 3 集 C++中 catch(...)如何使用

上一篇文章中详细讲了讲 C++异常处理模型的 trycatch 使用语法，其中 catch 关键字是用来定义 catch block 的，它后面带一个参数，用来与异常对象的数据类型进行匹配。注意 catch 关键字只能定义一个参数，因此每个 catch block 只能是一种数据类型的异常对象的错误处理模块。如果要想使一个 catch block 能抓获多种数据类型的异常对象的话，怎么办？C++标准中定义了一种特殊的 catch 用法，那就是” catch(…)”。

感性认识

1、catch(…)到底是一个什么样的东东，先来个感性认识吧！看例子先：

```
int main()
{
try
{
cout << "在 try block 中, 准备抛出一个异常." << endl;
//这里抛出一个异常（其中异常对象的数据类型是 int，值为 1）
throw 1;
}
//catch( int& value )
//注意这里 catch 语句
catch( …)
{
cout << "在 catch(…) block 中, 抛出的 int 类型的异常对象被处理" << endl;
}
}
```

2、哈哈！int 类型的异常被 catch(…)抓获了，再来另一个例子：

```
int main()
{
```

```
try
{
cout << "在 try block 中, 准备抛出一个异常." << endl;
//这里抛出一个异常（其中异常对象的数据类型是 double，值为 0.5）
throw 0.5;
}
//catch( double& value )
//注意这里 catch 语句
catch( …)
{
cout << "在 catch(…) block 中, double 类型的异常对象也被处理" << endl;
}
}
```

3、同样，double 类型的异常对象也被 catch(…)块抓获了。是的，catch(..)能匹配成功所有的数据类型的异常对象，包括 C++语言提供所有的原生数据类型的异常对象，如 int、double，还有 char*、int*这样的指针类型，另外还有数组类型的异常对象。同时也包括所有自定义的抽象数据类型。例程如下：

```
int main()
{
try
{
cout << "在 try block 中, 准备抛出一个异常." << endl;
//这里抛出一个异常（其中异常对象的数据类型是 char*）
char* p=0;
throw p;
}
//catch( char* value )
//注意这里 catch 语句
catch( …)
{
cout << "在 catch(…) block 中, char*类型的异常对象也被处理" << endl;
}
}


int main()
{
try
{
cout << "在 try block 中, 准备抛出一个异常." << endl;
//这里抛出一个异常（其中异常对象的数据类型是 int[]）
int a[4];
throw a;
}
```

```
//catch( int value[] )
//注意这里 catch 语句
catch( …)
{
cout << "在  catch(…) block  中, int[]类型的异常对象也被处理" << endl;
}
}
```

4、对于抽象数据类型的异常对象。catch(…)同样有效，例程如下：

```
class MyException
{
public:
protected:
int code;
};

int main()
{
try
{
cout << "在  try block  中,  准备抛出一个异常." << endl;
//这里抛出一个异常（其中异常对象的数据类型是 MyException）
throw MyException();
}
//catch(MyException& value )
//注意这里 catch 语句
catch( …)
{
cout << "在 catch(…) block 中, MyException 类型的异常对象被处理" << endl;
}
}
```

对 catch(…)有点迷糊？

1、究竟对 catch(…)有什么迷糊呢？还是看例子先吧！

```
void main()
{
int* p = 0;

try
{
//  注意：下面这条语句虽然不是 throw 语句，但它在执行时会导致系统
//  出现一个存储保护错误的异常（access violation exception）
*p = 13; // causes an access violation exception;
}
catch(...)
```

```
{
//catch(…)能抓获住上面的 access violation exception 异常吗？
cout << "在 catch(…) block 中" << endl;
}
}
```

请问上面的程序运行时会出现什么结果吗？catch(…)能抓获住系统中出现的 access violation exception 异常吗？朋友们！和我们的主人公阿愚一样，自己动手去测试一把！
结果又如何呢？实际上它有两种不同的运行结果，在 window2000 系统下用 VC 来测试运行这个小程序时，发现程序能输出"在 catch(…) block 中"的语句在屏幕上，也即 catch(…) 能成功抓获住系统中出现的 access violation exception 异常，很厉害吧！但如果这个同样的程序在 linux 下用 gcc 编译后运行时，程序将会出现崩溃，并在屏幕上输出”segment fault”的错误信息。

主人公阿愚有点急了，也开始有点迷糊了，为什么？为什么？为什么同样一个程序在两种不同的系统上有不同的表现呢？其原因就是：对于这种由于硬件或操作系统出现的系统异常（例如说被零除、内存存储控制异常、页错误等等）时，window2000 系统有一个叫做结构化异常处理（Structured Exception Handling，SEH）的机制，这个东东太厉害了，它能和 VC 中的 C++异常处理模型很好的结合上（实际上 VC 实现的 C++异常处理模型很大程度上建立在 SEH 机制之上的，或者说它是 SEH 的扩展，后面文章中会详细阐述并分析这个久富盛名的 SEH，看看 catch(…)是如何神奇接管住这种系统异常出现后的程序控制流的，不过这都是后话）。而在 linux 系统下，系统异常是由信号处理编程方法来控制的（信号处理编程，signal processing progamming。在介绍 unix 和 linux 下如何编程的书籍中，都会有对信号处理编程详细的介绍，当然执著的主人公阿愚肯定对它也不会放过，会深入到 unix 沿袭下来的信号处理编程内部的实现机制，并尝试完善改进它，使它也能够较好地和 C++异常处理模型结合上）。

那么 C++标准中对于这种同一个程序有不同的运行结果有何解释呢？这里需要注意的是，window2000 系统下 catch(…)能捕获住系统异常，这完全是它自己的扩展。在 C++标准中并没有要求到这一点，它只规定 catch(…)必须能捕获程序中所有通过 throw 语句抛出的异常。因此上面的这个程序在 linux 系统下的运行结果也完全是符合 C++标准的。虽然大家也必须承认 window2000 系统下对 C++异常处理模型的这种扩展确实是一个很不错的完善，极大得提高了程序的安全性。

为什么要用 catch(…)这个东东？

程序员朋友们也许会说，这还有问吗？这篇文章的一开始不就讲到了吗？catch(…)能够捕获多种数据类型的异常对象，所以它提供给程序员一种对异常对象更好的控制手段，使开发的软件系统有很好的可靠性。因此一个比较有经验的程序员通常会这样组织编写它的代码模块，如下：

```
void Func()
{
try
{
// 这里的程序代码完成真正复杂的计算工作，这些代码在执行过程中
// 有可能抛出 DataType1、DataType2 和 DataType3 类型的异常对象。
}
catch(DataType1& d1)
{
```

```
}
catch(DataType2& d2)
{
}
catch(DataType3& d3)
{
}
// 注意上面 try block 中可能抛出的 DataType1、DataType2 和 DataType3 三
// 种类型的异常对象在前面都已经有对应的 catch block 来处理。但为什么
// 还要在最后再定义一个 catch(…) block 呢？这就是为了有更好的安全性和
// 可靠性，避免上面的 try block 抛出了其它未考虑到的异常对象时导致的程
// 序出现意外崩溃的严重后果，而且这在用 VC 开发的系统上更特别有效，因
// 为 catch(…)能捕获系统出现的异常，而系统异常往往令程序员头痛了，现
// 在系统一般都比较复杂，而且由很多人共同开发，一不小心就会导致一个
// 指针变量指向了其它非法区域，结果意外灾难不幸发生了。catch(…)为这种
// 潜在的隐患提供了一种有效的补救措施。
catch(…)
{
}
}
```

还有，特别是 VC 程序员为了使开发的系统有更好的可靠性，往往在应用程序的入口函数中（如 MFC 框架的开发环境下 CXXXApp::InitInstance()）和工作线程的入口函数中加上一个顶层的 trycatch 块，并且使用 catch(…)来捕获一切所有的异常，如下：

```
BOOL CXXXApp::InitInstance()
{
if (!AfxSocketInit())
{
AfxMessageBox(IDP_SOCKETS_INIT_FAILED);
return FALSE;
}

AfxEnableControlContainer();

// Standard initialization
// If you are not using these features and wish to reduce the size
// of your final executable, you should remove from the following
// the specific initialization routines you do not need.

#ifdef _AFXDLL
Enable3dControls(); // Call this when using MFC in a shared DLL
#else
Enable3dControlsStatic(); // Call this when linking to MFC statically
#endif
```

```
// 注意这里有一个顶层的 trycatch 块，并且使用 catch(…)来捕获一切所有的异常
try
{
CXXXDlg dlg;
m_pMainWnd = &dlg;
int nResponse = dlg.DoModal();
if (nResponse == IDOK)
{
// TODO: Place code here to handle when the dialog is
// dismissed with OK
}
else if (nResponse == IDCANCEL)
{
// TODO: Place code here to handle when the dialog is
// dismissed with Cancel
}
}
catch(…)
{
// dump 出系统的一些重要信息，并通知管理员查找出现意外异常的原因。
// 同时想办法恢复系统，例如说重新启动应用程序等
}

// Since the dialog has been closed, return FALSE so that we exit the
// application, rather than start the application's message pump.
return FALSE;
}
```

通过上面的例程和分析可以得出，由于 catch(…)能够捕获所有数据类型的异常对象，所以在恰当的地方使用 catch(…)确实可以使软件系统有着更好的可靠性。这确实是大家使用 catch(…)这个东东最好的理由。但不要误会的是，在 C++异常处理模型中，不只有 catch(…)方法能够捕获几乎所有类型的异常对象（也许有其它更好的方法，在下一篇文章中主人公阿愚带大家一同去探讨一下），可 C++标准中为什么会想到定义这样一个 catch(…)呢？有过 java 或 C#编程开发经验的程序员会发现，在它们的异常处理模型中，并没有这样类似的一种语法，可这里不得不再次强调的是，java 中的异常处理模型是 C++中的异常处理模型的完善改进版，可它反而没有了 catch(…)，为何呢？还是先去看看下一章吧，“C++的异常处理和面向对象的紧密关系”。也许大家能找到一个似乎合理的原因。

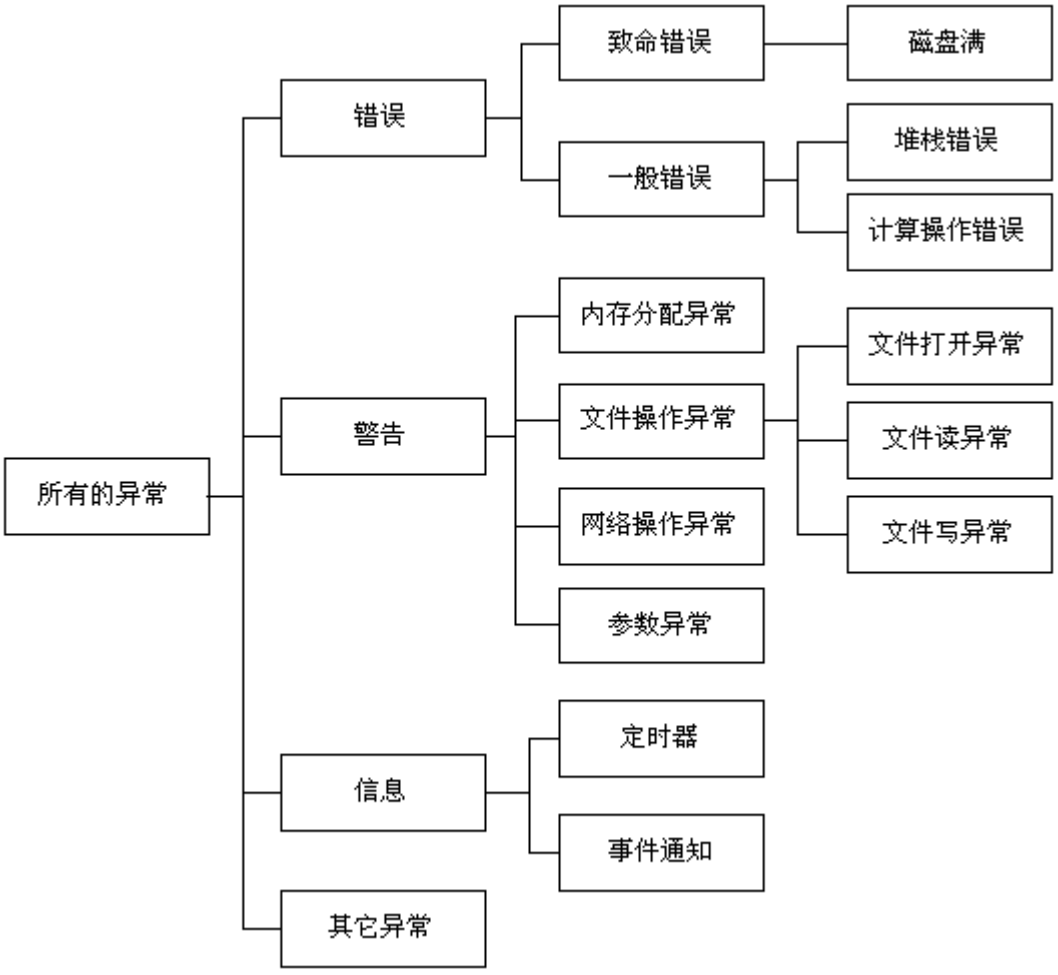
第 4 集 C++的异常处理和面向对象的紧密关系

如果有人问起 C++和 C 到底有那些本质上的不同点？主人公阿愚当然也会有自己的一份理解，他会毫不犹豫回答出：“与 C 相比，C++至少引入了两项重要技术，其一就是对面向对象的全面支持；还有一项就是 C++优良的异常处理模型”。是的，这两项技术对构建出一个优良的可靠复杂的软件系统都太重要了。可这两项技术之间又有何关系呢？非常客观公正的说，它们之间的关系实在是太紧密了，两者相互支持和依赖，是构建优良可靠复杂的软件系统最不可缺乏的两个东东。

用对象来描述程序中出现的异常

虽然前几篇文章的内容中列举的一些小例子程序大多都是 throw 一些如 int、double 类型的异常，但程序员朋友都很熟悉，实际开发环境中所抛出的异常都是一个个代表抽象数据类型的对象，如 C++标准库中的 std::exception()，MFC 开发库中 Cexception 等。用对象来描述的我们程序中的出现异常的类型和异常信息是 C++异常处理模型中最闪光之处，而且这一特点一直沿用到 java 语言的异常处理模型中。

为什么要用对象来描述程序中出现的异常呢？这样做的优势何在？主人公阿愚不喜欢穷摆出什么大道理，还是老办法，从具体的实例入手。由于异常有许许多多种类型，如有致命的错误、一般的错误、警告或其它事件通知等，而且不同类型的异常有不同的处理方法，有的异常是不可恢复的，而有的异常是可以恢复的（专业术语叫做“重入”吧！哈哈，主人公阿愚有时也会来点文绉绉的东西），所以程序员在开发系统时就必须考虑把各种可能出现的异常进行分类，以便能够分别处理。下面为一个应用系统设计出一个对异常进行分类的简单例子，如下：



从上面的异常分类来看，它有明显的层次性和继承性，这恰恰和面向对象的继承思想如出一辙，因此用对象来描述程序中出现的异常是再恰当不过的了。而且可以利用面向对象的特性很好的对异常进行分类处理，例如有这样一个例子：

```
void OpenFile(string f)
{
try
{
// 打开文件的操作，可能抛出 FileOpenException
}
catch(FileOpenException& fe)
{
// 处理这个异常，如果这个异常可以很好的得以恢复，那么处理完毕后函数
// 正常返回；否则必须重新抛出这个异常，以供上层的调用函数来能再次处
// 理这个异常对象
int result = ReOpenFile(f);
if (result == false) throw;
}
}

void ReadFile(File f)
{
try
{
// 从文件中读数据，可能抛出 FileReadException
}
catch(FileReadException& fe)
{
// 处理这个异常，如果这个异常可以很好的得以恢复，那么处理完毕后函数
// 正常返回；否则必须重新抛出这个异常，以供上层的调用函数来能再次处
// 理这个异常对象
int result = ReReadFile(f);
if (result == false) throw;
}
}

void WriteFile(File f)
{
try
{
// 往文件中写数据，可能抛出 FileWriteException
}
catch(FileWriteException& fe)
{
// 处理这个异常，如果这个异常可以很好的得以恢复，那么处理完毕后函数
```

```
// 正常返回；否则必须重新抛出这个异常，以供上层的调用函数来能再次处
// 理这个异常对象
int result = ReWriteFile(f);
if (result == false) throw;
}
}

void Func()
{
try
{
// 对文件进行操作，可能出现 FileWriteException、FileWriteException
// 和 FileWriteException 异常
OpenFile(…);

ReadFile(…);

WriteFile(…);
}
// 注意：FileException 是 FileOpenException、FileReadException 和 FileWriteException
// 的基类，因此这里定义的 catch(FileException& fe)能捕获所有与文件操作失败的异
// 常。
catch(FileException& fe)
{
ExceptionInfo* ef = fe.GetExceptionInfo();
cout << “操作文件时出现了不可恢复的错误，原因是：”<< fe << endl;
}
}
```

通过上面简单的例子可以看出，利用面向对象的方法，确实能很好地对异常进行分类处理，分层处理，如果异常能得以恢复的尽可能去实现恢复，否则向上层重新抛出异常表明当前的函数不能对这里异常进行有效恢复。同时特别值得一提的是，上层的 catch block 利用申明一个基类的异常对象作为 catch 关键字的参数，使得提供了对多种类型的异常对象的集中处理方法，这就是上一篇文章中所提到的除了 catch(…)以外，还有其它的来实现对多种类型的异常对象的集中处理方法，而且利用对象基类的方法显然要比 catch(…)优雅很多，方便很多，要知道在 catch(…)的异常处理模块中是没有多少办法获取一些关于异常出现时异常具体信息的，而对象基类的方法则完全不同，异常处理模块可以访问到真正的异常对象。

现在回想一下上一篇文章中提出的那个问题？就是既然有其它很好的方法（利用类的继承性）来可以代替 catch(…)提供的异常集中处理，那为什么 C++标准中还偏要提供 catch(…)这样一种奇怪的语法呢？其实这还是由于 C++本身一些特点所决定的，因为大家都知道，C++在业界有很多的版本，更重要的是没有一个统一的标准开发类库，或者说没有统一的标准开发环境，虽然存在标准 C 库和标准 C++库，但这远远不够，构成不了一个完整的开发支撑环境，因此在许多重要的开发库中都各自为政，它们在自己的开发库都各自定义了一套对异常进行分类支持的库。因此应用程序的开发环境往往都同时需要依赖于几个基础开发库之上（例如 MFC + XML4C + Standard C++），这样对开发人员而言，便没有一个共同的异常对象的基类，所以 C++标准中便提供了 catch(…)来捕获所有异常，这确实是一种不得已而为之的折衷方法（哈哈！

这个理解完全是主人公阿愚自己一相情愿，一个人胡思乱想而出来的，朋友们如有不同的意见可以和阿愚一起讨论！），另外 JAVA 中则不会出现这种情况，因为 JDK 是统一的，所有的异常对象都是从 java.lang.Throwable 接口继承而来的，因此只要在程序的入口函数中 catch(java.lang.Throwable all)，便可以捕获所有的异常。所以在 JAVA 的异常处理模型中没有类似 C++那样一个 catch(…)的东东，完全没必要。

异常处理中采用面向对象技术还有哪些好处呢？

上面讲到，用对象来描述程序中出现的异常除了能很好地分层处理异常外，还有那些好处呢？当然除了好处大大的外，好处也是多多的，例如：

（1） 面向对象的实现中，一般都很好的实现了对象的 RTTI 技术，如果异常用对象来表示，那么就可以很好完成异常对象的数据类型匹配，还有就是函数的多态，例如上面的那个例子中，即便是到了 catch(FileException& fe)异常处理模块中，也能知道到底是出现了那种具体的异常，是 FileOpenException 呢？还是其它的异常？

（2） 面向对象的实现中，一般都很好的实现了对象的构造、对象的销毁、对象的转存复制等等，所以这也为异常处理模型中，异常对象的转存复制和对象销毁提供了很好的支持，容易控制；

（3） 其它的吗？暂时没有，可能还有没想到的。

在异常处理的分层管理下，异常对象的重新抛出往往非常常见，本文中刚才的例子就有这样的情况，但当时仅一笔带过而已，为了表明对它的重视，下一篇文章重点讨论一下异常对象的 rethrow 处理。

第 5 集 C++的异常 rethrow

上一篇文章已经提到了 C++的异常 rethrow 现象，这篇文章将进一步深入讨论这个问题。当 catch 到一个异常后进入异常处理程序块中，此时需要从传入的异常对象中得到一些关于异常的详细信息，并判断这个异常是否能得以恢复，是否能在当前的异常处理程序块中得以处理。如果是，那么及时地处理掉这个异常，使程序恢复到正常的工作轨道上（也即从 catch block 后面的代码处继续执行）；否则就必须重新抛出异常(Excption Rethrow)，把这个异常交给上一层函数的异常处理模块去处理（反正我是处理不了了，而且我也通知了我的上层领导，所以责任吗，也就不由我担当了，哈哈 ^-^）。

语法

很简单，有两种用法，如下：

1、 throw ;

2、 throw exception_obj ;

第一种表示原来的异常对象再次被重新抛出；第二中呢，则表示原来的异常已处理或正在处理中，但此时又引发了另一个异常。示例如下：

```
void main()
{
try
{
try
```

```
{
throw 4;
}
catch(int value)
{
// 第一种用法，原来的异常被再次抛出
// 注意它不需要带参数。
throw;
}

try
{
throw 0.5;
}
catch(double value)
{
// 第二种用法，再次抛出另外的一个异常
// 它的语法和其它正常抛出异常的语法一样。
throw “another exception”;
}
}
catch(...)
{
cout << “unknow exception”<< endl;
}
}
```

在什么地方异常可以 rethrow？

当然，异常的 rethrow 只能在 catch block 中，或者说在 catch block 中抛出的异常才是异常的 rethrow，因此注意下面的示例程序中存在语法错误，如下：

```
void main()
{
try
{
try
{
throw 4;
}
catch(int value)
{
// 这里的语法是对的。
throw;
}
```

```
// 但这里的语法却是不对的。
// 不能在这里进行异常的 rethrow
throw;
}
catch(...)
{
cout << “unknow exception”<< endl;
}
}
```

异常 rethrow 需要注意的问题！

异常 rethrow 需要注意什么问题呢？看例子先！

```
void main()
{
try
{
try
{
throw 4;
}
catch(int value)
{
// 异常的 rethrow
throw;
}
catch(...)
{
cout << “能打印我这条消息吗?”<< endl;
}
}
catch(...)
{
cout << “unknow exception”<< endl;
}
}
```

上面的程序运行结果是：“unknow exception”

由此我们可以得出结论，异常的 rethrow 后，它会在它上一层的 trycatch 块开始查找匹配的 catch block 异常处理块，而在同一层中，如果当前的 catch block 后面还有其它的 catch block，它是不会去匹配的。所以程序中一般层次模型的 trycatch 要比线性结构的 trycatch 要好一些，如下(示例 2 要比示例 1 好)：

```
// 示例 1
void main()
{
```

```
try
{
}
catch(DataType1&)
{
}
catch(DataType2&)
{
}
catch(DataType3&)
{
}
catch(...)
{
}
}

// 示例 2
void main()
{
try
{
try
{
try
{
try
{
}
catch(DataType1&)
{
}
}
catch(DataType2&)
{
}
}
catch(DataType3&)
{
}
}
catch(...)
{
}
}
```

总结

相遇篇的文章到此结束。通过这几篇文章的介绍，目前已经对异常处理编程的思想，C++异常处理模型、语法，以及 C++异常处理与面向对象的关系等等，都有了一个大概性的了解。主人公阿愚根据自己的理解和经验，现在对相遇篇中的知识再做出如下一些总结：

（1） 异常处理编程和面向对象，是现在程序设计和编程中最不可缺少的两个好东东；

（2） C++异常处理模型是层次型的，能很好地支持嵌套；

（3） C++异常处理编程提供 try、catch 和 throw 三个关键字。其中 try 定义受监控的程序块；catch 定义异常处理模块；throw 让程序员可以在程序执行出错的地方抛出异常；

（4） C++异常处理模型的实现充分使用到了面向对象的思想和方法；

（5） C++异常处理模型中，异常是可以 rethrow 的。

从下篇文章开始，主人公阿愚对异常处理编程将进入到了一个相知的阶段。这一阶段中，阿愚将全面性地去深入了解异常处理编程中的各个细节和一些特点，并根据自己的理解阐述一些异常处理编程设计思想方面的东西。各位程序员朋友们，准备好了吗？Let's go!

第 6 集 对象的成员函数中抛出的异常

C++异常处理模型除了支持面向过程的 C 风格程序中的异常处理外（就是没有面向对象的概念，完全是 C 程序，整个程序实际就是函数的集合，但却用 C++编译器来编译这样的 C 程序，所以这样的程序中是可以 a 使用 C++的异常处理机制的，要不怎么说 C++是兼容 C 语言的呢？但是需要注意的是，单纯的 C 语言程序中是不能使用 C++异常处理模型进行编程的。是不是有点说拗口了？有点糊涂了呢？其实很简单，那就是如果程序中使用了 C++异常处理机制，也即代码中有 try、catch 和 throw 关键字，那么就必须使用 C++编译器来编译这个程序。许多程序员朋友们在这里有一个理解上的误区，认为只有程序中使用了面向对象的概念，即使用 class 关键字来定义一个类结构才算得上 C++程序，其实这种理解是片面的，如果程序中采用了 C++异常处理机制，那么也有理由认为这是一个 C++程序，哪怕程序的代码完全是 C 语言风格的，并且这样的程序用 C 编译器来编译肯定将会报错，提示未定义的 try 标示符等等错误信息），还支持面向对象程序中对象抛出的异常处理。

C++异常处理模型的确和面向对象是紧密结合的，除了在相遇篇中介绍到的用对象来描述程序中出现的异常之外，C++异常处理模型也对在面向对象程序中的对象实例所抛出的异常作了最完善的支持和处理。也许大家会觉得这很容易，没什么了不起的地方。但恰恰相反，实际上这才是 C++异常处理模型最成功、最不可思议和最闪光的地方。而且由于 C++异常处理模型对面向对象有了很好的支持和兼容，才使得 C++异常处理模型本身的实现变得特别复杂，因为它需要跟踪每一个对象的运行情况和状态（关于 C++异常处理模型的实现，会在爱的秘密篇中有详细剖析）。本文和接下来的几篇文章将讲述当对象实例抛出异常时将如何处理。

对象的生命周期一般有三种状态：构造、运行和析构销毁。因此对象抛出的异常也有这三种区别。是在对象构造时抛出的呢？还是对象运行时抛出的呢？或是析构对象时抛出的？这三种不同时候抛出的异常会将会产生不同的结果。本文首先讨论最常见的一种情况，在对象运行时抛出的异常，也即执行对象的成员函数时出现的异常。

对象的成员函数抛出的异常

1、老方法，看例子先，如下：

```
class MyTest_Base
{
public:
MyTest_Base (string name = “”) : m_name(name)
{
cout << “构造一个 MyTest_Base 类型的对象，对象名为：”<<m_name << endl;
}

virtual ~ MyTest_Base ()
{
cout << “销毁一个 MyTest_Base 类型的对象，对象名为：”<<m_name << endl;
}

void Func() throw()
{
throw std::exception(“故意抛出一个异常，测试！”);
}
void Other() {}

protected:
string m_name;
};

void main()
{
try
{
MyTest_Base obj1(“obj1”);

// 调用这个成员函数将抛出一个异常，注意 obj1 的析构函数会被执行吗？如果
// 会，又是在什么时候被执行呢？
obj1.Func();
obj1.Other();
}
catch(std::exception e)
{
cout << e.what() << endl;
}
catch(...)
{
cout << “unknow exception”<< endl;
```

```
}
}
```

C++程序员不难看出上面的程序的运行结果，如下：

构造一个 MyTest_Base 类型的对象，对象名为：obj1

销毁一个 MyTest_Base 类型的对象，对象名为：obj1

故意抛出一个异常，测试！

从运行结果可以得出如下结论：

（1） 对象的成员函数出现异常时，catch block 能捕获到异常，这一点就像 C 语言中的普通函数一样，没什么特别的地方；

（2） 对象的成员函数出现异常时，对象的析构函数将会得到执行（这一点很神奇吧！当然在这里不会做过多研究，在剖析 C++异常处理模型的实现时再做详细的阐述），这里与 C++标准中规定的面向对象的特性是相一致的，构造了的对象就必须保证在适当的地方要析构它，以释放可能的资源。因此前面说的“C++异常处理模型对面向对象提供了支持和兼容”是有根据的。而且注意它的析构函数是在异常处理模块之前执行的，这一点更与 C++标准中规定的面向对象的特性是一致的，当对象出了作用域时，它就必须要被析构。

2、把上面的程序小改一下，运行再看结果，如下：

```
void main()
{
// obj1 对象不在 trycatch 域中，注意它的析构函数在什么时候被执行？
MyTest_Base obj1(“obj1”);
try
{
// obj2 和 obj3 对象都在 trycatch 域中，其中 obj3.Func()函数被调用，因此
// obj3 会抛出异常，特别需要注意的是，obj2 的析构函数会被执行吗？如果
// 会，又是在什么时候被执行呢？
MyTest_Base obj2(“obj2”), obj3(“obj3”);

obj3.Other();

// 调用这个成员函数将抛出一个异常
obj3.Func();

// 注意：obj4 对象在构造之前，函数中就有异常抛出。所以 obj4 对象将不会
// 被构造，当然也不会被析构
MyTest_Base obj4(“obj4”);
obj3.Other();
}
catch(std::exception e)
{
cout << e.what() << endl;
}
```

```
catch(...)
{
cout << “unknow exception”<< endl;
}
}
```

上面的程序也难看出其运行结果，如下：

构造一个 MyTest_Base 类型的对象，对象名为：obj1
构造一个 MyTest_Base 类型的对象，对象名为：obj2
构造一个 MyTest_Base 类型的对象，对象名为：obj3
销毁一个 MyTest_Base 类型的对象，对象名为：obj3
销毁一个 MyTest_Base 类型的对象，对象名为：obj2
故意抛出一个异常，测试！
销毁一个 MyTest_Base 类型的对象，对象名为：obj1

结合程序中提出的问题和运行结果，可以又可得出如下结论：

（1） 在成员函数出现异常时，同一个作用域中异常出现点后面还未来得及构造的对象将不会被构造，当然也不会被析构；

（2） 在成员函数出现异常时，同一个作用域中异常出现点前面已经构造的对象也同样会被析构（这是不是更神奇了！）。因此这也显现出 C++异常处理不会破坏 C++标准中规定的面向对象的特性，当对象出了作用域时，它就必须要被析构，即便它自己本身没出现异常，总之不管是正常的执行过程导致对象退出了作用域，还是其它对象运行时发生了异常而导致自己退出了作用域；

（3） 在成员函数出现异常时，未被影响到的其它作用域中的对象将保持自己原来的执行流程。

对象的成员函数抛出的异常时概括性总结

哈哈^-^，其是就只有一句话，那就是“C++的异常处理不会破坏任何一条面向对象的特性！”，因此主人公阿愚建议大家其实无须要记住上面总结的 n 条结论，记住这一条足矣！

下篇文章讨论在构造函数中抛出异常时程序的执行情况，这有点复杂呀！朋友们，Let's go!

第 7 集 构造函数中抛出的异常

上一篇文章简单讨论了一下对象的成员函数抛出异常时的处理情况。本文中将继续讨论当在构造函数中抛出异常时，程序的执行情况又如何？这有点复杂呀！而且主人公阿愚还觉得这蛮有点意思！

构造函数中抛出的异常

1、标准 C++中定义构造函数是一个对象构建自己，分配所需资源的地方，一旦构造函数执行完毕，则表明这个对象已经诞生了，有自己的行为和内部的运行状态，之后还有对象的消亡过程（析构函数的执行）。可谁能保证对象的构造过程一定能成功呢？说不定系统当前的某个资源不够，导致对象不能完全构建好自己（人都有畸形儿，更何况别的呢？朋友们！是吧！），因此通过什么方法来表明对象的构造失败了呢？C++程序员朋友们知道，C++中的构造函数是没有返回值的，所以不少关于 C++编程方面的书上得出结论：“因为构造函数没有返回值，所以通知对象的构造失败的唯一方法那就是在构造函数中抛出异常”。主人公

阿愚非常不同意这种说法，谁说的，便不信邪！虽然 C++标准规定构造函数是没有返回值，可我们知道每个函数实际上都会有一个返回值的，这个值被保存在 eax 寄存器中，因此实际上是有办法通过编程来实现构造函数返回一个值给上层的对象创建者。当然即便是构造函数真的不能有返回值，我们也可以通过一个指针类型或引用类型的出参来获知对象的构造过程的状态。示例如下：

```
class MyTest_Base
{
public:
MyTest_Base (int& status)
{
//do other job

// 由于资源不够，对象构建失败
// 把 status 置 0，通知对象的构建者
status = 0;
}

protected:
};

void main()
{
int status;
MyTest_Base obj1(status);

// 检查对象的构建是否成功
if(status ==0) cout << “对象构建失败” << endl;
}
```

程序运行的结果是：

对象构建失败

是啊！上面我们不也得到了对象构造的成功与否的信息了吗？可大家有没有觉得这当中有点问题？主人公阿愚建议大家在此停留片刻，仔细想想它会有什么问题？OK！也许大家都知道了问题的所在，来验证一下吧！

```
class MyTest_Base
{
public:
MyTest_Base (int& status)
{
//do other job

// 由于资源不够，对象构建失败
// 把 status 置 0，通知对象的构建者
```

```
status = 0;
}

virtual ~ MyTest_Base ()
{
cout << "销毁一个 MyTest_Base 类型的对象" << endl;
}


protected:
};

void main()
{
int status;
MyTest_Base obj1(status);

// 检查对象的构建是否成功
if(status ==0) cout << "对象构建失败" << endl;
}
```

程序运行的结果是：
对象构建失败
销毁一个 MyTest_Base 类型的对象

没错，对象的析构函数被运行了，这与 C++标准中所规定的面向对象的一些特性是有冲突的。一个对象都没有完成自己的构造，又何来析构！好比一个夭折的畸形儿还没有出生，又何来死之言。因此这种方法是行不通的。那怎么办？那就是上面那个结论中的后一句话是对的，通知对象的构造失败的唯一方法那就是在构造函数中抛出异常，但原因却不是由于构造函数没有返回值而造成的。恰恰相反，C++标准中规定构造函数没有返回值正是由于担心很容易与面向对象的一些特性相冲突，因此干脆来个规定，构造函数不能有返回值（主人公阿愚的个人理解，有不同意见的朋友欢迎讨论）。

2、构造函数中抛出异常将导致对象的析构函数不被执行。哈哈^-^，阿愚很开心，瞧瞧！如果没有 C++的异常处理机制鼎立支持，C++中的面向对象特性都无法真正实现起来，C++标准总不能规定所有的对象都必须成功构造吧！这也太理想化了，也许只有等到共产主义社会实现的那一天（CPU 可以随便拿，内存可以随便拿，所有的资源都是你的！）才说不定有可能……,所以说 C++的异常处理和面向对象确实是谁也离不开谁。当然示例还是要看一下，如下：

```
class MyTest_Base
{
public:
MyTest_Base (string name = "") : m_name(name)
{
throw std::exception("在构造函数中抛出一个异常，测试！");
```

```
cout << “构造一个 MyTest_Base 类型的对象，对象名为：”<<m_name << endl;
}

virtual ~ MyTest_Base ()
{
cout << “销毁一个 MyTest_Base 类型的对象，对象名为：”<<m_name << endl;
}

void Func() throw()
{
throw std::exception(“故意抛出一个异常，测试！”);
}
void Other() {}

protected:
string m_name;
};

void main()
{
try
{
// 对象构造时将会抛出异常
MyTest_Base obj1(“obj1”);

obj1.Func();
obj1.Other();
}
catch(std::exception e)
{
cout << e.what() << endl;
}
catch(...)
{
cout << “unknow exception”<< endl;
}
}
```

程序的运行结果将会验证：“构造函数中抛出异常将导致对象的析构函数不被执行”

3、是不是到此，关于构造函数中抛出异常的处理的有关讨论就能结束了呢？非也！非也！主人公阿愚还有进一步的故事需要讲述！来看一个更复杂一点的例子吧！如下：

```
class MyTest_Base
{
```

```
public:
MyTest_Base (string name = "") : m_name(name)
{
cout << "构造一个 MyTest_Base 类型的对象，对象名为："<<m_name << endl;
}

virtual ~ MyTest_Base ()
{
cout << "销毁一个 MyTest_Base 类型的对象，对象名为："<<m_name << endl;
}

void Func() throw()
{
throw std::exception("故意抛出一个异常，测试！");
}
void Other() {}

protected:
string m_name;
};

class MyTest_Parts
{
public:
MyTest_Parts ()
{
cout << "构造一个 MyTest_Parts 类型的对象" << endl;
}

virtual ~ MyTest_Parts ()
{
cout << "销毁一个 MyTest_Parts 类型的对象"<< endl;
}
};

class MyTest_Derive : public MyTest_Base
{
public:
MyTest_Derive (string name = "") : m_component(), MyTest_Base(name)
{
throw std::exception("在 MyTest_Derive 对象的构造函数中抛出了一个异常！");

cout << "构造一个 MyTest_Derive 类型的对象，对象名为："<<m_name << endl;
}
```

```
virtual ~ MyTest_Derive ()
{
cout << "销毁一个 MyTest_Derive 类型的对象，对象名为："<<m_name << endl;
}

protected:
MyTest_Parts m_component;
};

void main()
{
try
{
// 对象构造时将会抛出异常
MyTest_Derive obj1("obj1");

obj1.Func();
obj1.Other();
}
catch(std::exception e)
{
cout << e.what() << endl;
}
catch(...)
{
cout << "unknow exception"<< endl;
}
}
```

程序运行的结果是：

构造一个 MyTest_Base 类型的对象，对象名为：obj1
构造一个 MyTest_Parts 类型的对象
销毁一个 MyTest_Parts 类型的对象
销毁一个 MyTest_Base 类型的对象，对象名为：obj1
在 MyTest_Derive 对象的构造函数中抛出了一个异常！

上面这个例子中，MyTest_Derive 从 MyTest_Base 继承，同时 MyTest_Derive 还有一个 MyTest_Parts 类型的成员变量。现在 MyTest_Derive 构造的时候，是在父类 MyTest_Base 已构造完毕和 MyTest_Parts 类型的成员变量 m_component 也已构造完毕之后，再抛出了一个异常，这种情况称为对象的部分构造。是的，这种情况很常见，对象总是由不断的继承或不断的聚合而来，对象的构造过程实际上是这些所有的子对象按规定顺序的构造过程，其中这些过程中的任何一个子对象在构造时发生异常，对象都不能说自己完成了全部的构造过程，因此这里就有一个棘手的问题，当发生对象的部分构造时，对象将析构吗？如果时，又将如何析构呢？

从运行结果可以得出如下结论：

（1） 对象的部分构造是很常见的，异常的发生点也完全是随机的，程序员要谨慎处理这种情况；

（2） 当对象发生部分构造时，已经构造完毕的子对象将会逆序地被析构（即异常发生点前面的对象）；而还没有开始构建的子对象将不会被构造了（即异常发生点后面的对象），当然它也就没有析构过程了；还有正在构建的子对象和对象自己本身将停止继续构建（即出现异常的对象），并且它的析构是不会被执行的。

构造函数中抛出异常时概括性总结

（1） C++中通知对象构造失败的唯一方法那就是在构造函数中抛出异常；

（2） 构造函数中抛出异常将导致对象的析构函数不被执行；

（3） 当对象发生部分构造时，已经构造完毕的子对象将会逆序地被析构；

（4） 哈哈^-^，其是还是那句话， “C++的异常处理不会破坏任何一条面向对象的特性！”，因此主人公阿愚再次建议朋友们，牢牢记住这一条！

下一篇文章讨论在对象的析构函数中抛出异常时程序的执行情况，这不仅有些复杂，而且很关键，它对我们的软件系统影响简直太大了，可许多人并未意识到这个问题的严重性！朋友们，不要错过下一篇文章，继续吧！

第 8 集 析构函数中抛出的异常

前两篇文章讨论了对象在构造过程中（构造函数）和运行过程中（成员函数）出现异常时的处理情况，本文将讨论最后一种情况，当异常发生在对象的析构销毁过程中时，又会有什么不同呢？主人公阿愚在此可以非常有把握地告诉大家，这将会有大大的不同，而且处理不善还将会毫不留情地影响到软件系统的可靠性和稳定性，后果非常严重。不危言耸听了，看正文吧！

析构函数在什么时候被调用执行？

对于 C++程序员来说，这个问题比较简单，但是比较爱唠叨的阿愚还是建议应该在此再提一提，也算回顾一下 C++的知识，而且这将对后面的讨论和理解由一定帮助。先看一个简单的示例吧！如下：

```
class MyTest_Base
{
public:
virtual ~ MyTest_Base ()
{
cout << "销毁一个 MyTest_Base 类型的对象"<< endl;
}
};


void main()
{
try
{
```

```
// 构造一个对象，当 obj 对象离开这个作用域时析构将会被执行
MyTest_Base obj;

}
catch(...)
{
cout << "unknow exception"<< endl;
}
}
```

编译运行上面的程序，从程序的运行结果将会表明对象的析构函数被执行了，但什么时候被执行的呢？按 C++标准中规定，对象应该在离开它的作用域时被调用运行。实际上各个厂商的 C++编译器也都满足这个要求，拿 VC 来做个测试验证吧！，下面列出的是刚刚上面的那个小示例程序在调试时拷贝出的相关程序片段。注意其中 obj 对象将会在离开 try block 时被编译器插入一段代码，隐式地来调用对象的析构函数。如下：

```
325: try
326: {
00401311 mov dword ptr [ebp-4],0
327: // 构造一个对象，当 obj 对象离开这个作用域时析构将会被执行
328: MyTest_Base obj;
00401318 lea ecx,[obj]
0040131B call @ILT+40(MyTest_Base::MyTest_Base) (0040102d)
329:
330: } // 瞧下面，编译器插入一段代码，隐式地来调用对象的析构函数
00401320 lea ecx,[obj]
00401323 call @ILT+15(MyTest_Base::~MyTest_Base) (00401014)
331: catch(...)
00401328 jmp __tryend$_main$1 (00401365)
332: {
333: cout << "unknow exception"<< endl;
0040132A mov esi,esp
0040132C mov
eax,[__imp_?endl@std@@YAAAV?$basic_ostream@DU?$char_traits@D@std@@@1@AAV21@@Z
(0041610c)
00401331 push eax
00401332 mov edi,esp
00401334 push offset string "unknow exception" (0041401c)
00401339 mov ecx,dword ptr [__imp_?cout@std@@3V?$basic_ostream@DU?$char_traits@D@std@@@1@A
(00416124)
0040133F push ecx
00401340 call dword ptr
[__imp_??6std@@YAAAV?$basic_ostream@DU?$char_traits@D@std@@@0@AAV10@PBD@Z (004
00401346 add esp,8
```

00401349 cmp edi,esp
0040134B call _chkesp (004016b2)
00401350 mov ecx,eax
00401352 call dword ptr
[__imp_??6?$basic_ostream@DU?$char_traits@D@std@@@std@@QAEAAV01@P6AAAV01@AAV01
00401358 cmp esi,esp
0040135A call _chkesp (004016b2)
334: }
0040135F mov eax,offset __tryend$_main$1 (00401365)
00401364 ret
335: }
析构函数中抛出的异常
1、仍然是先看示例，如下：

```
class MyTest_Base
{
public:
virtual ~ MyTest_Base ()
{
cout << "开始准备销毁一个 MyTest_Base 类型的对象"<< endl;

// 注意：在析构函数中抛出了异常
throw std::exception("在析构函数中故意抛出一个异常，测试！");
}

void Func() throw()
{
throw std::exception("故意抛出一个异常，测试！");
}

void Other() {}

};


void main()
{
try
{
// 构造一个对象，当 obj 对象离开这个作用域时析构将会被执行
MyTest_Base obj;


obj.Other();
}
```

```
catch(std::exception e)
{
cout << e.what() << endl;
}
catch(...)
{
cout << "unknow exception"<< endl;
}
}
```

程序运行的结果是：

开始准备销毁一个 MyTest_Base 类型的对象

在析构函数中故意抛出一个异常，测试！

从上面的程序运行结果来看，并没有什么特别的，在程序中首先是构造一个对象，当这个对象在离开它的作用域时，析构函数被调用，此时析构函数中抛出一个 std::exception 类型的异常，因此后面的 catch(std::exception e)块捕获住这个异常，并打印出异常错误信息。这个过程好像显现出，发生在析构函数中的异常与其它地方发生的异常（如对象的成员函数中）并没有什么太大的不同，除了析构函数是隐式调用的以外，但这也丝毫不会影响到异常处理的机制呀！那究竟区别何在？玄机何在呢？继续往下看吧！

2、在上面的程序基础上做点小的改动，程序代码如下：

```
void main()
{
try
{
// 构造一个对象，当 obj 对象离开这个作用域时析构将会被执行
MyTest_Base obj;


// 下面这条语句是新添加的
// 调用这个成员函数将抛出一个异常
obj.Func();

obj.Other();
}
catch(std::exception e)
{
cout << e.what() << endl;
}
catch(...)
{
cout << "unknow exception"<< endl;
}
}
```

注意，修改后的程序现在的运行结果：非常的不幸，程序在控制台上打印一条语句后就崩溃了（如果程序是 debug 版本，会显示一条程序将被终止的断言；如果是 release 版本，程序会被执行 terminate()函数后退出）。在主人公阿愚的机器上运行的 debug 版本的程序结果如下：

许多朋友对这种结果也许会觉得傻了眼，这简直是莫名奇妙吗？这是谁的错呀！难道是新添加的那条代码的问题，但这完全不会呀！（其实很多程序员朋友受到过太多这种类似的冤枉，例如一个程序原来运行挺好的，以后进行功能扩充后，程序却时常出现崩溃现象。其实有时程序扩充时也没添加多少代码，而且相关程序员也很认真仔细检查自己添加的代码，确认后来添加的代码确实没什么问题呀！可相关的负责人也许不这么认为，觉得程序以前一直运行挺好的，经过你这一番修改之后就出错了，能不是你添加的代码所导致的问题吗？真是程序开发领域的窦娥冤呀！其实这种推理完全是没有根据和理由的，客观公正一点地说，程序的崩溃与后来添加的模块代码肯定是会有一定的相关性！但真正的 bug 也许就在原来的系统中一直存在，只不过以前一直没诱发表现出来而已！瞧瞧！主人公阿愚又岔题了，有感而发！还是回归正题吧！）

那究竟是什么地方的问题呢？其实这实际上由于析构函数中抛出的异常所导致的，但这就更诧异了，析构函数中抛出的异常是没有问题的呀！刚才的一个例子不是已经测试过了吗？是的，但那只是一种假象。如果要想使你的系统可靠、安全、长时间运行无故障，你在进行程序的异常处理设计和编码过程中，至少要保证一点，那就是析构函数中是不永许抛出异常的，而且在 C++标准中也特别声明到了这一点，但它并没有阐述真正的原因。那么到底是为什么呢？为什么 C++标准就规定析构函数中不能抛出异常？这确实是一个非常棘手的问题，很难阐述得十分清楚。不过主人公阿愚还是愿意向大家论述一下它自己对这个问题的理解和想法，希望能够与程序员朋友们达成一些理解上的共识。

C++异常处理模型是为 C++语言量身设计的，更进一步的说，它实际上也是为 C++语言中面向对象而服务的，我们在前面的文章中多次不厌其烦的声明到，C++异常处理模型最大的特点和优势就是对 C++中的面向对象提供了最强大的无缝支持。好的，既然如此！那么如果对象在运行期间出现了异常，C++异常处理模型有责任清除那些由于出现异常所导致的已经失效了的对象（也即对象超出了它原来的作用域），并释放对象原来所分配的资源，这就是调用这些对象的析构函数来完成释放资源的任务，所以从这个意义上说，析构函数已经变成了异常处理的一部分。不知大家是否明白了这段话所蕴含的真正内在涵义没有，那就是上面的论述 C++异常处理模型它其实是有一个前提假设——析构函数中是不应该再有异常抛出的。试想！如果对象出了异常，现在异常处理模块为了维护系统对象数据的一致性，避免资源泄漏，有责任释放这个对象的资源，调用对象的析构函数，可现在假如析构过程又再出现异常，那么请问由谁来保证这个对象的资源释放呢？而且这新出现的异常又由谁来处理呢？不要忘记前面的一个异常目前都还没有处理结束，因此这就陷入了一个矛盾之中，或者说无限的递归嵌套之中。所以 C++标准就做出了这种假设，当然这种假设也是完全合理的，在对象的构造过程中，或许由于系统资源有限而致使对象需要的资源无法得到满足，从而导致异常的出现，但析构函数完全是可以做得到避免异常的发生，毕竟你是在释放资源呀！好比你在与公司续签合同的时候向公司申请加薪，也许公司由于种种其它原因而无法满足你的要求；但如果你主动申请不要薪水完全义务工作，公司能不乐意地答应你吗？

假如无法保证在析构函数中不发生异常，怎么办？

虽然 C++标准中假定了析构函数中不应该，也不永许抛出异常的。但有过的实际的软件开发的程序员朋友们中也许会体会到，C++标准中的这种假定完全是站着讲话不觉得腰痛，实际的软件系统开发中是很难保证到这一点的。所有的析构函数的执行过程完全不发生一点异常，这根本就是天方夜谭，或者说自己欺骗自己算了。而且大家是否还有过这种体会，有时候发现析构一个对象（释放资源）比构造一个对象还

更容易发生异常，例如一个表示引用记数的句柄不小心出错，结果导致资源重复释放而发生异常，当然这种错误大多时候是由于程序员所设计的算法在逻辑上有些小问题所导致的，但不要忘记现在的系统非常复杂，不可能保证所有的程序员写出的程序完全没有 bug。因此杜绝在析构函数中决不发生任何异常的这种保证确实是有点理想化了。那么当无法保证在析构函数中不发生异常时，该怎么办？我们不能眼睁睁地看着系统崩溃呀！

其实还是有很好办法来解决的。那就是把异常完全封装在析构函数内部，决不让异常抛出函数之外。这是一种非常简单，也非常有效的方法。按这种方法把上面的程序再做一点改动，那么程序将避免了崩溃的厄运。如下：

```
class MyTest_Base
{
public:
virtual ~ MyTest_Base ()
{
cout << "开始准备销毁一个 MyTest_Base 类型的对象"<< endl;

// 一点小的改动。把异常完全封装在析构函数内部
try
{
// 注意：在析构函数中抛出了异常
throw std::exception("在析构函数中故意抛出一个异常，测试！");
}
catch(…) {}

}

void Func() throw()
{
throw std::exception("故意抛出一个异常，测试！");
}

void Other() {}

};
```

程序运行的结果如下：
开始准备销毁一个 MyTest_Base 类型的对象
故意抛出一个异常，测试！

怎么样，现在是不是一切都已经风平浪静了。

析构函数中抛出异常时概括性总结

（1） C++中析构函数的执行不应该抛出异常；

（2） 假如析构函数中抛出了异常，那么你的系统将变得非常危险，也许很长时间什么错误也不会发生；但也许你的系统有时就会莫名奇妙地崩溃而退出了，而且什么迹象也没有，崩得你满地找牙也很难发现问题究竟出现在什么地方；

（3） 当在某一个析构函数中会有一些可能（哪怕是一点点可能）发生异常时，那么就必须要把这种可能发生的异常完全封装在析构函数内部，决不能让它抛出函数之外（这招简直是绝杀！呵呵！）；

（4） 主人公阿愚吐血地提醒朋友们，一定要切记上面这几条总结，析构函数中抛出异常导致程序不明原因的崩溃是许多系统的致命内伤！

至此在 C++程序中，各种可能的地方抛出的异常将如何地来处理这个主题已基本讨论完毕！从下一篇文章开始继续讨论有关 C++异常处理其它方面的一些问题。朋友们，CONTINUE！

第 9 集 C++的异常对象如何被传递

在相遇篇的第 4 集文章中，曾经讲到过在 C++的异常处理模型中，是用“对象”来描述程序中出现的异常，并且在那篇文章中详细讨论了这样做所带来的诸多好处，其中之一呢就是：对象一般都很好地实现了对象的构造、对象的销毁、对象的转存复制，这为异常处理模型中异常对象的转存复制和对象销毁提供了很好的支持。是的没错，但是所谓的异常对象到底是如何被复制和传递呢？从本篇文章开始，和接下来的几篇文章中，主人公阿愚将和大家一同比较深入地探讨这个问题，并力求弄清每一个重要的细节。

概述

呵呵！sorry，居然忘了阐述一下定义。那就是“C++的异常对象被传递”指的是什么？想当然大家也都知道，这指的就是异常出现时 throw 出的异常对象如何被传递到 catch block 块中，catch block 中的异常处理模块再根据异常对象提供的异常信息做出相应的处理。程序员朋友们也许认为这很简单，其实说简单也好像不太简单，因为这种对象的传递或复制可能发生在同一个函数的不同程序块作用域间，也有可能是从当前的函数传递到上一个函数中，更有可能是把异常对象传递复制到上上（甚至更多层）的函数中。

异常对象的传递有点类似于函数调用过程中的参数传递的过程。瞧！catch 关键字的语法不就跟函数的定义有点类似吗？作为入参的异常对象也是用括号被括起来的，只不过 catch 只能是拥有一个参数。另外连 catch(…)语法也是抄袭函数定义的方式，表示接受任意类型的数据对象。

C++程序中函数的调用是通过“栈”来实现的，其中参数的传递也是保存到栈中，以实现两个函数间的数据共享。那么异常对象的传递呢？当然也是通过栈，其实这是很明显的一件事情，因为异常对象本身肯定是局部变量，因此它也肯定是被保存在栈中的。不过异常对象的传递毕竟还是与函数参数的传递有很大的不同，函数参数的传递是严谨的、一级一级的对象数据的压栈过程和出栈过程；但异常对象的传递却远比这要复杂些，因为它这是逆序的，属于局部变量的异常对象可能要往上层（或更上层）函数传递，它的过程是一个跳跃式的或比较混乱的过程。关于异常对象的传递具体是如何实现的，在爱的秘密篇中分析 C++异常处理模型的实现时会再做详细阐述。而目前需要搞清楚的是，这个过程中所需要遵从的一些规律或标准。

函数的参数的传递一般有指针、传值和引用三种方式，同样，异常对象的传递也同样有这三种方式的区别。现在开始，主人公阿愚分别讲述每一种方式下异常对象是如何被传递的，不过在正式开始之前，还是先简要总结函数调用的过程，以及这过程栈的变化。因为这对随后的具体分析和理解也许大有帮助。

函数的调用过程与“栈”

C++程序员对这个过程肯定非常熟悉，因此这里不做细致的讲述，只做一个概要性的总结。

（1） 函数的调用过程实质上利用栈来实现的指令（eip）执行远程转移和返回的过程；它在 CPU 指令级别上就得到了支持（CALL 和 RET 指令）；

（2） 每个线程都有一个自己的栈，因此每个线程的函数调用执行是相互不受影响的；

（3） C 和 C++中的函数参数的入栈顺序一般是从右到左进行；

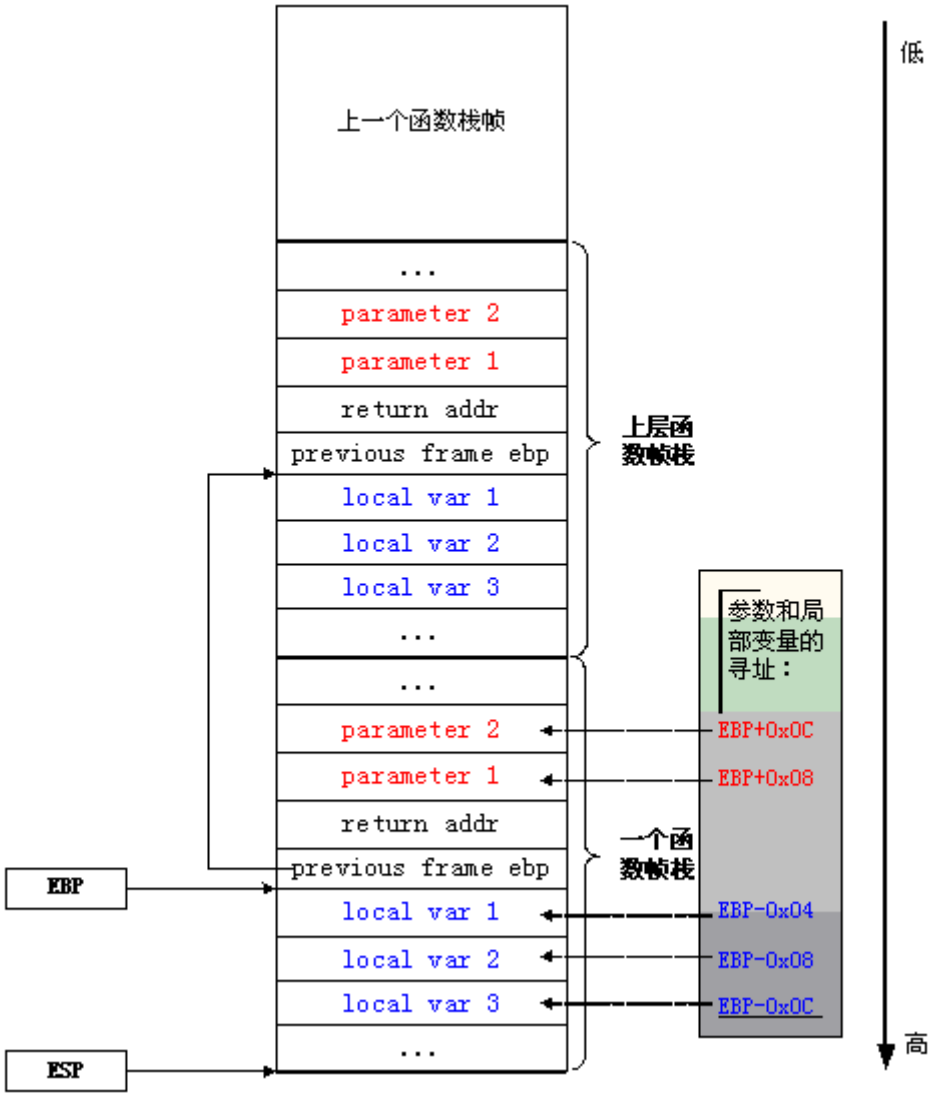
（4） C++中的函数的参数的传递一般有指针、传值和引用三种方式；

（5） C 和 C++中函数的返回值一般都是保存到 EAX 寄存器中返回的；

（6） C 和 C++中函数中寻址参数和局部变量，一般都是通过 EBP 寄存器加上偏移来进行的，如参数一般是：[EBP+XX]，而局部变量则一般是：[EBP-XX]；

（7） 在程序运行时，EBP 中的值一般是指向当前的函数调用帧，而 ESP 一般指向栈顶。

如果对上面论述有不太清楚或不太熟悉的朋友，建议先看看专门讲述 C++设计和编程方面的书籍。下面给出一个线程运行期间，它栈中所保存的数据的布局（部分片段），如下图：



总结

（1） 与函数的参数的传递类似， C++的异常对象的传递也分指针、传值和引用三种方式；
（2） 与函数的参数的传递不同的是，异常对象的传递是向上逆序的，而且是跳跃式的。

下一篇文章详细介绍 C++的异常对象按传值的方式被复制和传递。朋友们，不要错过，请继续吧！

第 10 集 C++的异常对象按传值的方式被复制和传递

上一篇文章中对 C++的异常对象如何被传递做了一个概要性的介绍，其中得知 C++的异常对象的传递方式有指针方式、传值方式和引用方式三种。现在开始讨论最简单的一种传递的方式：按值传递。

异常对象在什么时候构造？

1、按传值的方式传递异常对象时，被抛出的异常都是局部变量，而且是临时的局部变量。什么是临时的局部变量，这大家可能都知道，例如发生函数调用时，按值传递的参数就会被临时复制一份，这就是临时局部变量，一般临时局部变量转瞬即逝。

主人公阿愚对这开始有点不太相信。不会吧，谁说异常对象都是临时的局部变量，应该是普通的局部变量，甚至是全局性变量，而且还可以是堆中动态分配的异常变量。是的，这上面说的好象没错，但是实际真实发生的情况是，每当在 throw 语句抛出一个异常时，不管你原来构造的对象是什么性质的变量，此时它都会复制一份临时局部变量，还是具体看看例程吧！如下：

```
class MyException
{
public:
MyException (string name="none") : m_name(name)
{
cout << "构造一个 MyException 异常对象，名称为："<<m_name<< endl;
}

MyException (const MyException& old_e)
{
m_name = old_e.m_name;

cout << "拷贝一个 MyException 异常对象，名称为："<<m_name<< endl;
}

operator= (const MyException& old_e)
{
m_name = old_e.m_name;
```

```
cout << "赋值拷贝一个 MyException 异常对象，名称为："<<m_name<< endl;
}

virtual ~ MyException ()
{
cout << "销毁一个 MyException 异常对象，名称为：" <<m_name<< endl;
}

string GetName() {return m_name;}

protected:
string m_name;
};

void main()
{
try
{
{
// 构造一个异常对象，这是局部变量
MyException ex_obj1("ex_obj1");

// 这里抛出异常对象
// 注意这时 VC 编译器会复制一份新的异常对象，临时变量
throw ex_obj1;
}

}
catch(...)
{
cout<<"catch unknow exception"<<endl;
}
}
```

程序运行的结果是：

构造一个 MyException 异常对象，名称为：ex_obj1
拷贝一个 MyException 异常对象，名称为：ex_obj1
销毁一个 MyException 异常对象，名称为：ex_obj1
catch unknow exception
销毁一个 MyException 异常对象，名称为：ex_obj1

瞧见了吧，异常对象确实是被复制了一份，如果还不相信那份异常对象是在 throw ex_obj1 这条语句执行时被复制的，你可以在 VC 环境中调试这个程序，再把这条语句反汇编出来，你会发现这里确实插入了一段调用拷贝构造函数的代码。

2、而且其它几种抛出异常的方式也会有同样的结果，都会构造一份临时局部变量。执着的阿愚可是每种情况都测试了一下，代码如下：

```
// 这是全局变量的异常对象
// MyException ex_global_obj("ex_global_obj");
void main()
{
try
{
{
// 构造一个异常对象，这是局部变量
MyException ex_obj1("ex_obj1");

throw ex_obj1;

// 这种也是临时变量
// 这种方式是最常见抛出异常的方式
//throw MyException("ex_obj2");

// 这种异常对象原来是在堆中构造的
// 但这里也会复制一份新的异常对象
// 注意：这里有资源泄漏呦！
//throw *(new MyException("ex_obj2"));

// 全局变量
// 同样这里也会复制一份新的异常对象
//throw ex_global_obj;
}

}
catch(...)
{
cout<<"catch unknow exception"<<endl;
}
```

大家也可以对每种情况都试一试，注意是不是确实无论哪种情况都会复制一份本地的临时变量了呢！

另外请朋友们特别注意的是，这是 VC 编译器这样做的，其它的 C++编译器是不是也这样的呢？也许不一定，不过很大可能都是采取这样一种方式（阿愚没有在其它每一种 C++编译器都做过测试，所以不敢妄下结论）。

为什么要再复制一份临时变量呢？是不是觉得有点多此一举，不！朋友们，请仔细再想想，因为假如不这样做，不把异常对象复制一份临时的局部变量出来，那么是不是会导致一些问题，或产生一些矛盾呢？的确如此！试想在抛出异常后，如果异常对象是局部变量，那么 C++标准规定了无论在何种情况下，只要局部变量离开其生存作用域，局部变量就必须要被销毁，可现在如果作为局部变量的异常对象在控制进入

catch block 之前，它就已经被析构销毁了，那么问题不就严重了吗？因此它这里就复制了一份临时变量，它可以在 catch block 内的异常处理完毕以后再销毁这个临时的变量。

主人公阿愚现在好像是逐渐得明白了，原来如此，但仔细一想，不对呀！上面描述的不准确呀！难道不可以在离开抛出异常的那个函数的作用域时，先把异常对象拷贝复制到上层的 catch block 中，然后再析构局部变量，最后才进入到 catch block 里面执行吗！分析的非常的棒！阿愚终于有些系统分析员的头脑了。是的，现在的 VC 编译器就是按这种顺序工作的。

可那到底为什么要复制临时变量呢？呵呵！要请教阿愚一个问题，如果 catch 后面的是采用引用传递异常对象的方式，也即没有拷贝复制这一过程，那么怎办？那个引用指向谁呀，指向一个已经析构了的异常对象！（总不至于说，等执行完 catch block 之后，再来析构原来属于局部变量的异常对象，这也太荒唐了）。所以吗？才如此。

可阿愚还是觉得不对劲呀！现在谈论的是异常对象按传值的方式被复制和传递的情况，你又怎么牵扯讨论到引用的方式了呢！OK！OK！OK！即便是引用传递异常对象的方式下，需要一份临时的异常对象（能保证不被析构，而局部变量则…），那么也可以在引用传递异常方式下采用这样的一种复制一份临时异常对象的做法；而在按值传递的方式就没有必要这样做（毕竟对象复制需要时间，会降低效率）。呵呵！想得倒是挺好，挺美！可不要忘记的是，程序员在抛出异常的时候怎么知道上面的 catch block 是采用哪种方式（是引用还是传值）？万一哪位大仙写出的程序，在上层的 catch block 有的是引用传递方式，而有的是按值传递方式，那怎么办！所以没辙了吧！采用复制一个临时的变量的方式是最安全、最可靠的方式，虽然说这样做会影响效率。

异常对象按传值复制

现在开始涉及到关键的地方，当 catch block 捕获到一个异常后，控制流准备转移到 catch block 之前，异常对象必须要通过一定的方式传递过来，假如是按传值传递（根据 catch 关键字后面定义的异常对象的数据类型），那么此时就会发生一次异常对象的拷贝构造过程。示例如下：

```
void main()
{
try
{
{
// 构造一个对象，当 obj 对象离开这个作用域时析构将会被执行
MyException ex_obj1("ex_obj1");

throw ex_obj1;

}

}
// 由于这里定义的是“按值传递”，所以这里会发生一次拷贝构造过程
catch(MyException e)
{
cout<<"捕获到一个 MyException 类型的异常，名称为："<<e.GetName()<<endl;
```

```
}
}
```

程序运行的结果是：

构造一个 MyException 异常对象，名称为：ex_obj1
拷贝一个 MyException 异常对象，名称为：ex_obj1
拷贝一个 MyException 异常对象，名称为：ex_obj1
销毁一个 MyException 异常对象，名称为：ex_obj1
捕获到一个 MyException 类型的异常，名称为：ex_obj1
销毁一个 MyException 异常对象，名称为：ex_obj1
销毁一个 MyException 异常对象，名称为：ex_obj1

通过结果可以看出确实又多发生了一次异常对象的拷贝复制过程，因此在 catch block 中进行错误处理时，我们可以放心存储异常对象，因为不管 C++异常处理模型到底是采用什么方法，总之当前这个异常对象已经被复制到了当前 catch block 的作用域中。

异常对象什么时候被销毁

通过上面的那个程序运行结果还可以获知，每个被拷贝复制出来的异常对象都会得到被销毁的机会。而且销毁都是在 catch block 执行之后，包括那个被抛出的属于临时局部变量的异常对象也是在执行完 catch block 之后，这很神奇吧！不过暂时先不管它。先搞清 catch block 中的那个按值拷贝传入的异常对象到底确切的是在什么时候被析构。示例如下：

```
void main()
{
try
{
{
// 构造一个对象，当 obj 对象离开这个作用域时析构将会被执行
MyException ex_obj1("ex_obj1");

throw ex_obj1;

}

}
// 由于这里定义的是“按值传递”，所以这里会发生一次拷贝构造过程
catch(MyException e)
{
cout<<"捕获到一个 MyException 类型的异常，名称为："<<e.GetName()<<endl;
}

// 加入一条语句，判断 e 什么时候销毁
cout<<"在这之前还是之后呢？"<<endl;
}
```

程序运行的结果是：
构造一个 MyException 异常对象，名称为：ex_obj1
拷贝一个 MyException 异常对象，名称为：ex_obj1
拷贝一个 MyException 异常对象，名称为：ex_obj1
销毁一个 MyException 异常对象，名称为：ex_obj1
捕获到一个 MyException 类型的异常，名称为：ex_obj1
销毁一个 MyException 异常对象，名称为：ex_obj1
销毁一个 MyException 异常对象，名称为：ex_obj1
在这之前还是之后呢？

看到了吗！发生那条语句之前，因此基本可以判断那个异常对象是在离开 catch block 时发生的析构，这样也算是情理之中，毕竟 catch block 中的异常处理模块对异常对象的存取使用已经完毕，过河拆桥有何不对！。为了进一步验证一下。从 VC 中 copy 出相关的反汇编代码。如下：

```
368: catch(MyException e)
00401CDA mov byte ptr [ebp-4],3
369: {
370: cout<<"捕获到一个 MyException 类型的异常，名称为："<<e.GetName()<<endl;
00401CDE lea eax,[ebp-64h]
00401CE1 push eax
00401CE2 lea ecx,[e]
00401CE5 call @ILT+60(MyException::GetName) (00401041)
00401CEA mov dword ptr [ebp-70h],eax
00401CED mov ecx,dword ptr [ebp-70h]
00401CF0 mov dword ptr [ebp-74h],ecx
00401CF3 mov byte ptr [ebp-4],4
00401CF7 mov esi,esp
00401CF9 mov edx,dword ptr
[__imp_?endl@std@@YAAAV?$basic_ostream@DU?$char_traits@D@std@@@1@AAV21@@Z
00401CFF push edx
00401D00 mov eax,dword ptr [ebp-74h]
00401D03 push eax
00401D04 mov edi,esp
00401D06 push offset string "\xb2\xb6\xbb\xf1\xb5\xbd\xd2\xbb\xb8\xf6MyException\xc0\xe0\xd0\xcd\xb5\x
00401D0B mov ecx,dword ptr [__imp_?cout@std@@3V?$basic_ostream@DU?$char_traits@D@std@@@1@A
(0041614c)
00401D11 push ecx
00401D12 call dword ptr
[__imp_??6std@@YAAAV?$basic_ostream@DU?$char_traits@D@std@@@0@AAV10@PBD@Z (004
00401D18 add esp,8
00401D1B cmp edi,esp
00401D1D call _chkesp (00401982)
00401D22 push eax
00401D23 call std::operator<< (0040194e)
00401D28 add esp,8
```

```
00401D2B mov ecx,eax
00401D2D call dword ptr
[__imp_??6?$basic_ostream@DU?$char_traits@D@std@@@std@@QAEAAV01@P6AAAV01@AAV01
00401D33 cmp esi,esp
00401D35 call _chkesp (00401982)
00401D3A mov byte ptr [ebp-4],3
00401D3E mov esi,esp
00401D40 lea ecx,[ebp-64h]
00401D43 call dword ptr
[__imp_??1?$basic_string@DU?$char_traits@D@std@@V?$allocator@D@2@@std@@QAE@XZ
00401D49 cmp esi,esp
00401D4B call _chkesp (00401982)
371: } // 瞧瞧下面，catch block 后不是调用析构函数了吗？
00401D50 mov byte ptr [ebp-4],2
00401D54 lea ecx,[e]
00401D57 call @ILT+45(MyException::~MyException) (00401032)
00401D5C mov eax,offset __tryend$_main$1 (00401d62)
00401D61 ret
372: cout<<"在这之前还是之后呢？"<<endl;
00401D62 mov dword ptr [ebp-4],0FFFFFFFFh
```

异常对象标示符的有效作用域

到目前为止大家已经可以知道，按值传递的异常对象的作用域是在 catch block 内，它在进入 catch block 块之前，完成一次异常对象的拷贝构造复制过程（从那个属于临时局部变量的异常对象进行复制），当离开 catch block 时再析构销毁异常对象。据此可以推理出，异常对象标示符（也就是变量的名字）也应该是在 catch block 内有效的，实际 catch block 有点像函数内部的子函数，而 catch 后面跟的异常对象就类似于函数的参数，它的标示符的有效域也和函数参数的很类似。看下面的示例程序，它是可以编译通过的。

```
void main()
{
// 这里定义一个局部变量，变量名为 e;
MyException e;
try
{

}
// 这里有一个 catch block，其中变量名也是 e;
// 实际可以理解函数内部的子函数
catch(MyException e)
{
cout<<"捕获到一个 MyException 类型的异常，名称为："<<e.GetName()<<endl;
}
// 这里又一个 catch block，其中变量名还是 e；而且数据类型也不同了。
catch(std::exception e)
```

```
{
e.what();
}
}
```

小心异常对象发生对象切片

C++程序员知道，当函数的参数按值传递时，可能会发生对象的切片现象。同样，如果异常对象按传值方式复制异常对象时，也可能会发生异常对象的切片。示例如下：

```
class MyException
{
public:
MyException (string name="none") : m_name(name)
{
cout << "构造一个 MyException 异常对象，名称为："<<m_name<< endl;
}

MyException (const MyException& old_e)
{
m_name = old_e.m_name;

cout << "拷贝一个 MyException 异常对象，名称为："<<m_name<< endl;
}

operator= (const MyException& old_e)
{
m_name = old_e.m_name;

cout << "赋值拷贝一个 MyException 异常对象，名称为："<<m_name<< endl;
}

virtual ~ MyException ()
{
cout << "销毁一个 MyException 异常对象，名称为：" <<m_name<< endl;
}

string GetName() {return m_name;}

virtual string Test_Virtual_Func() { return "这是 MyException 类型的异常对象";}

protected:
string m_name;
};
```

```
class MyMemoryException : public MyException
{
public:
MyMemoryException (string name="none") : MyException(name)
{
cout << "构造一个 MyMemoryException 异常对象，名称为："<<m_name<< endl;
}

MyMemoryException (const MyMemoryException& old_e)
{
m_name = old_e.m_name;

cout << "拷贝一个 MyMemoryException 异常对象，名称为："<<m_name<< endl;
}

virtual string Test_Virtual_Func() { return "这是 MyMemoryException 类型的异常对象";}

virtual ~ MyMemoryException ()
{
cout << "销毁一个 MyMemoryException 异常对象，名称为：" <<m_name<< endl;
}
};

void main()
{
try
{
{
MyMemoryException ex_obj1("ex_obj1");

cout <<endl<< "抛出一个 MyMemoryException 类型的异常" <<endl<<endl;
throw ex_obj1;

}

}
// 注意这里发生了对象切片，异常对象 e 已经不是原原本本的那个被 throw 出
// 的那个对象了
catch(MyException e)
{
// 调用虚函数，验证一下这个异常对象是否真的发生了对象切片
cout<<endl<<e.Test_Virtual_Func()<<endl<<endl;
}
}
```

程序运行的结果是：

总结

（1） 被抛出的异常对象都是临时的局部变量；
（2） 异常对象至少要被构造三次；
（3） catch 后面带的异常对象的作用域仅限于 catch bock 中；
（4） 按值传递方式很容易发生异常对象的切片。

下一篇文章讨论 C++的异常对象按引用的方式被复制和传递。继续吧！

第 11 集 C++的异常对象按引用方式被传递

上一篇文章详细讨论了 C++的异常对象按值传递的方式，本文继续讨论另外的一种的方式：引用传递。

异常对象在什么时候构造？

其实在上一篇文章中就已经讨论到了，假如异常对象按引用方式被传递，异常对象更应该被构造出一个临时的变量。因此这里不再重复讨论了。

异常对象按引用方式传递

引用是 C++语言中引入的一种数据类型形式。它本质上是一个指针，通过这个特殊的隐性指针来引用其它地方的一个变量。因此引用与指针有很多相似之处，但是引用用起来较指针更为安全，更为直观和方便，所以 C++语言建议 C++程序员在编写代码中尽可能地多使用引用的方式来代替原来在 C 语言中使用指针的地方。这些地方主要是函数参数的定义上，另外还有就是 catch 到的异常对象的定义。

所以异常对象按引用方式传递，是不会发生对象的拷贝复制过程。这就导致引用方式要比传值方式效率高，此时从抛出异常、捕获异常再到异常错误处理结束过程中，总共只会发生两次对象的构造过程（一次是异常对象的初始化构造过程，另一次就是当执行 throw 语句时所发生的临时异常对象的拷贝复制的构造过程）。而按值传递的方式总共是发生三次。看看示例程序吧！如下：

```
void main()
{
try
{
{
throw MyException();

}

}
// 注意：这里是定义了引用的方式
catch(MyException& e)
```

```
{
cout<<"捕获到一个 MyException 类型的异常，名称为："<<e.GetName()<<endl;
}
}
```

程序运行的结果是：

构造一个 MyException 异常对象，名称为：none
拷贝一个 MyException 异常对象，名称为：none
销毁一个 MyException 异常对象，名称为：none
捕获到一个 MyException 类型的异常，名称为：none
销毁一个 MyException 异常对象，名称为：none

程序的运行结果是不是显示出：异常对象确实是只发生两次构造过程。并且在执行 catch block 之前，局部变量的异常对象已经被析构销毁了，而属于临时变量的异常对象则是在 catch block 执行错误处理完毕后才销毁的。

那个被引用的临时异常对象究竟身在何处？

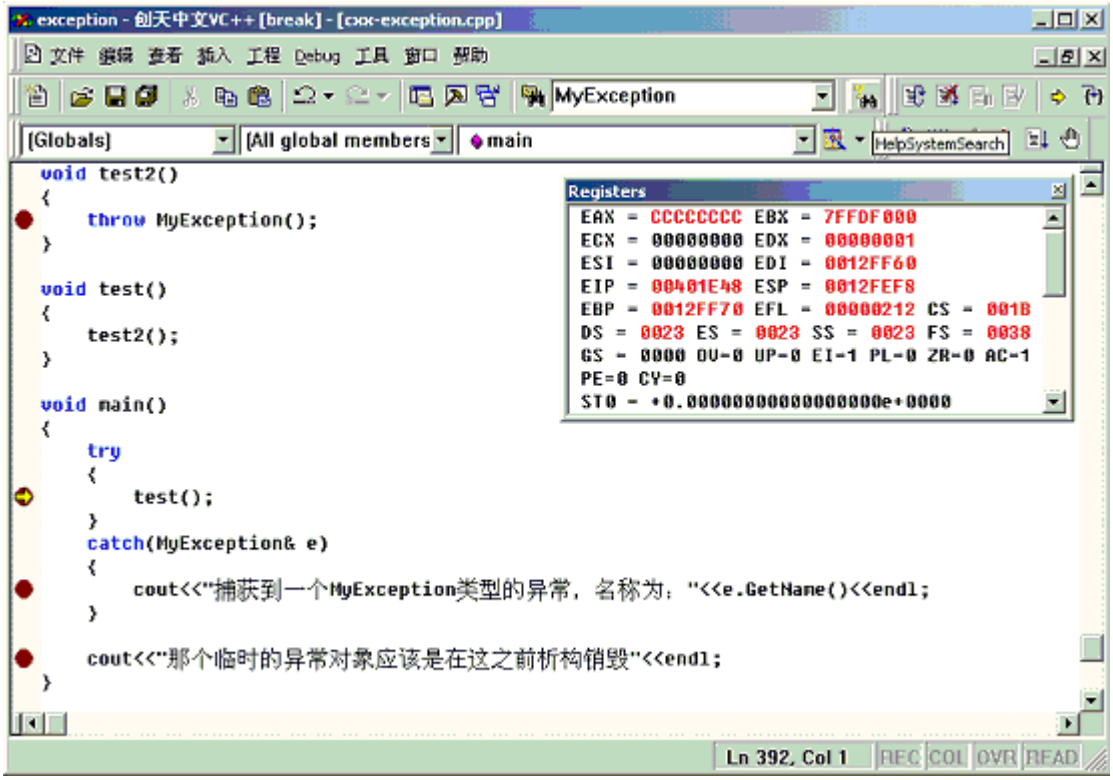
呵呵！这还用问吗，临时异常对象当然是在栈中。是的没错，就像发生函数调用时，与引用类型的参数传递一样，它也是引用栈中的某块区域的一个变量。但请大家提高警惕的是，这两处有着非常大的不同，其实在一开始讨论异常对象如何传递时就提到过，函数调用的过程是有序的的压栈过程，请回顾一下《第 9 集 C++的异常对象如何传送》中函数的调用过程与“栈”那一节的内容。栈是从高往低的不断延伸扩展，每发生一次函数调用时，栈中便添加了一块格式非常整齐的函数帧区域（包含参数、返回地址和局部变量），当前的函数通过 ebp 寄存器来寻址函数传入的参数和函数内部的局部变量。因此这样对栈中的数据存储是非常安全的，依照函数的调用次序（call stack），在栈中都有唯一的一个对应的函数帧一层层地从上往下整齐排列，当一个函数执行完毕，那么最低层的函数帧清除（该函数作用域内的局部变量都析构销毁了），返回到上一层，如此不断有序地进行函数的调用与返回。

但发生异常时的情况呢？它的异常对象传递却并没有这么简单，它需要在栈中把异常对象往上传送，而且可能还要跳跃多个函数帧块完成传送，所以这就复杂了很多，当然即便如此，只要我们找到了源对象数据块和目标对象数据块，也能很方便地完成异常对象的数据的复制。但现在最棘手的问题是，如果采用引用传递的方式将会有很大的麻烦，为什么？试想！前面多次提到的临时异常对象是在那里构造的？对象数据又保存在什么地方？毫无疑问，对象数据肯定是在当前（throw 异常的函数）的那个函数帧区域，这是处于栈的最低部，现在假使匹配到的 catch block 是在上层（或更上层）的函数中，那么将会导致出现一种现象：就是在 catch block 的那个函数（执行异常处理的模块代码中）会引用下面抛出异常的那个函数帧中的临时异常对象。主人公阿愚现在终于恍然大悟了（不知阅读到此处的 C++程序员朋友们现在领会了作者所说的意思没有！如果还没有，自己动手画画栈图看看），是啊！确是如此，这太不安全了，按理说当执行到 catch block 中的代码时，它下面的所有的函数帧（包括抛出异常的哪个函数帧）都将会无效，但此时却引用到了下面的已经失效了的函数帧中的临时异常对象，虽说这个异常对象还没有被析构，但完全有可能会发生覆盖呀（栈是往下扩展的）！

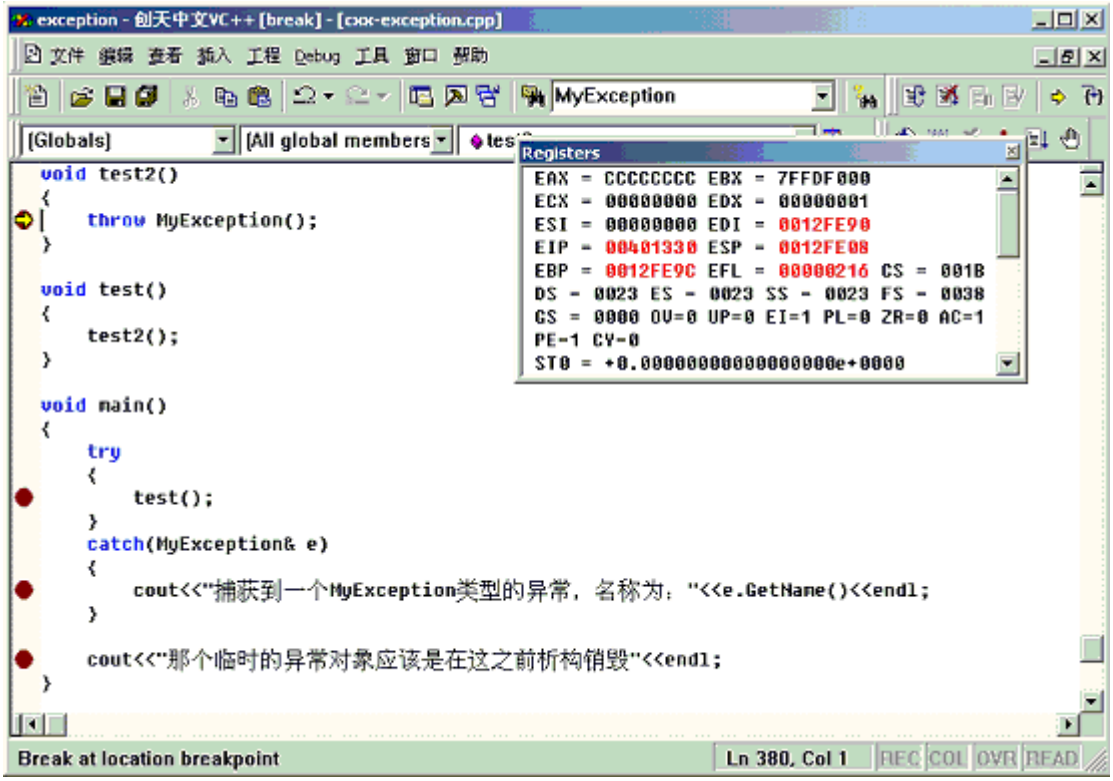
怎么办！难道真的有可能会发生覆盖吗？那就太危险了。朋友们！放心吧！实际情况是绝对不会发生覆盖的。为什么？哈哈！编译器真是很聪明，它这里采用了一点点技巧，巧妙的避免的这个问题。下面用一个跨越了多个函数的异常的例子程序来详细阐述之，如下：

```
void test2()
{
throw MyException();
}

void test()
{
test2();
}

void main()
{
try
{
test();
}
catch(MyException& e)
{
cout<<"捕获到一个 MyException 类型的异常，名称为："<<e.GetName()<<endl;
}

cout<<"那个临时的异常对象应该是在这之前析构销毁"<<endl;
}
```

怎样来分析呢？当然最简单的方法是调试一下，跟踪它的 ebp 和 esp 的变化。首先在函数调用的地方和抛出异常的地方设置好断点，F5 开始调试，截图如下：

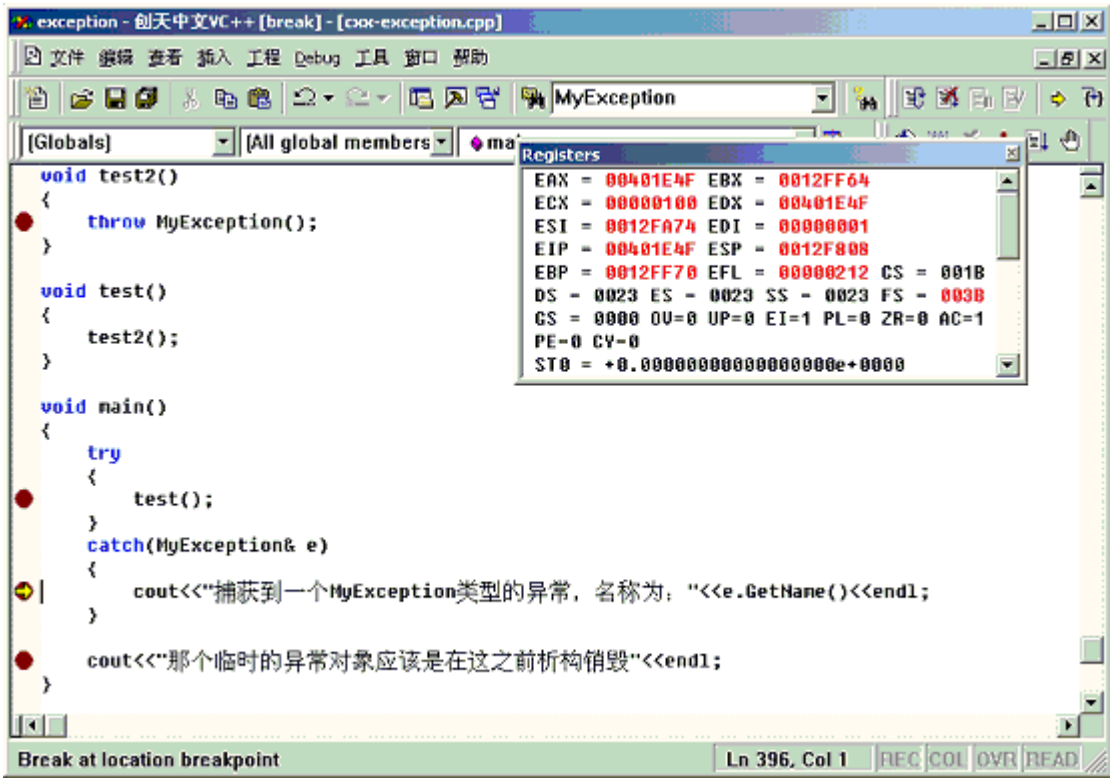


纪录一下 ebp 和 esp 的值（ebp 0012FF70；esp 0012FEF8），通过 ebp 和 esp 可以确定 main 函数的函数帧在栈中位置，F5 继续，截图如下：

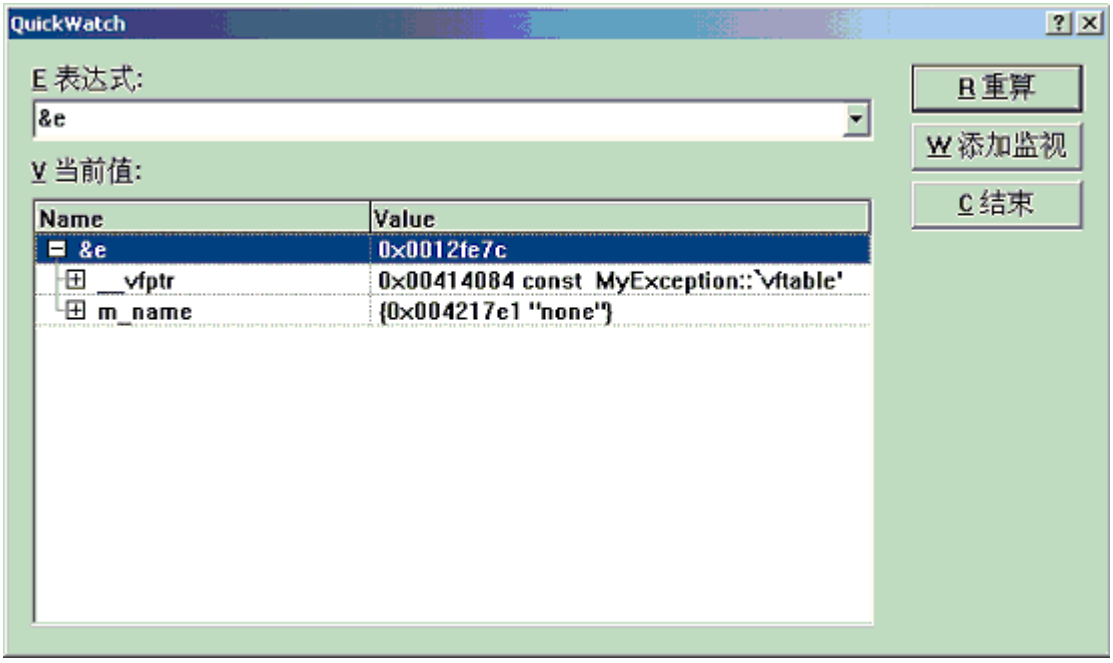


同样也纪录一下 ebp 和 esp 的值（ebp 0012FE9C；esp 0012FE08），通过 ebp 和 esp 可以看出栈是往下

扩展，此时的 ebp 和 esp 指向抛出异常的 test2 函数的函数帧在栈中位置，F5 继续，此时抛出异常，控制进入 main 函数中的 catch(MyException& e)中，截图如下：

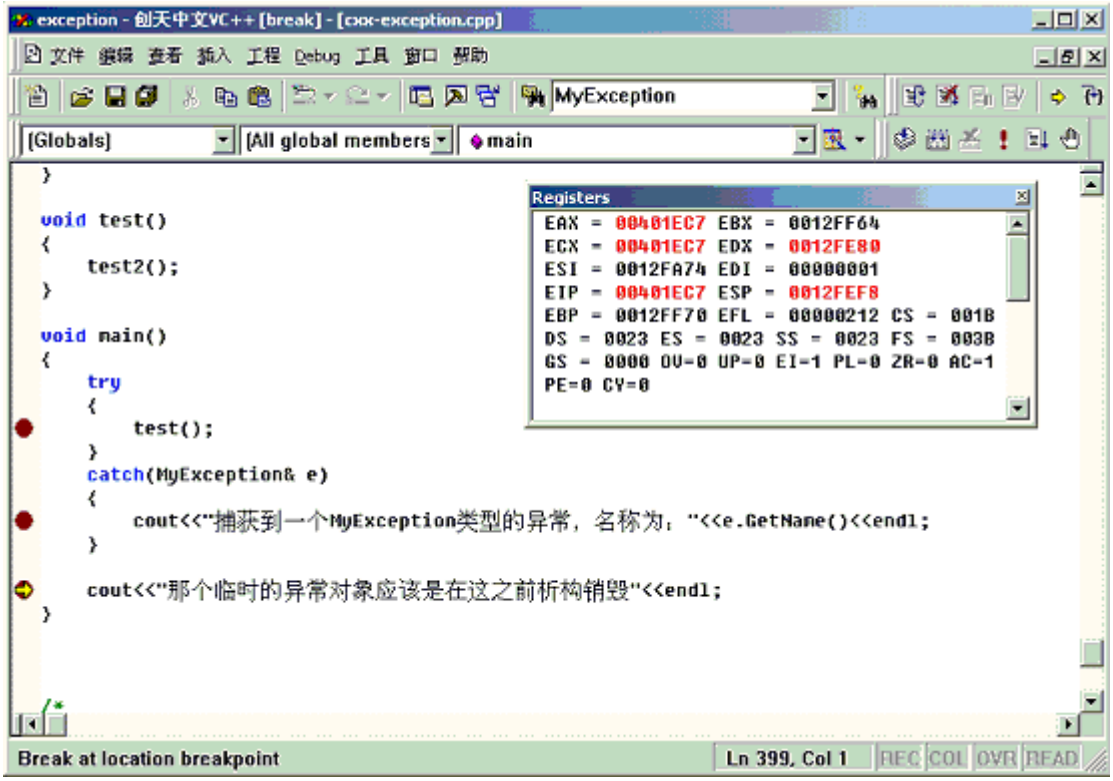


请注意了，现在 ebp 恢复了 main 函数在先前时的函数帧在栈中位置，但 esp 却并没有，它甚至比刚刚抛出异常的那个 test2 函数中的 esp 还要往下，这就是编译器编译程序时耍的小技巧，当前 ebp 和 esp 指向的函数帧实际上并不是真正的 main 函数原来的哪个函数帧，它实际上包含了多个函数的函数帧，因此 catch block 执行程序时当然不会发生覆盖。我们还是看看异常对象所引用指向的临时的变量究竟身在何处。截图如下：



哈哈！e 指向了 0x0012fe7c 内存区域，再看看上面的抛出异常的 test2 函数的函数帧的 ebp 和 esp 的值。结果 0x0012fe7c 恰好是 ebp 0012FE9C 和 esp 0012FE08 之间。

不过阿愚又开始有点疑惑了，哦！这样做岂不是破坏了函数的帧栈吗，结果还不导致程序崩溃呀！呵呵！不用担心，F5 继续，截图如下：



当离开了 catch block 作用域之后，再看看 ebp 和 esp 的值，是不是和最开始的那个 main 函数进入时的 ebp 和 esp 一模一样，哈哈！恢复了，厉害吧！先暂时不管它是如何恢复的，总之 ebp 和 esp 都是得以恢复

了，而且同时 catch block 执行时也不会发生异常对象的覆盖。这就解决了异常对象按引用传递时可能存在的不安全隐患。

引用方式下，异常对象会发生对象切片吗？

当然不会，要不测试一下，把上一篇文章中的对应的那个例子改为按引用的方式接受异常对象。示例如下：

```
void main()
{
try
{
{
MyMemoryException ex_obj1("ex_obj1");

cout <<endl<< "抛出一个 MyMemoryException 类型的异常" <<endl<<endl;
throw ex_obj1;

}

}
// 注意这里引用的方式了
// 还会发生了对象切片吗？
catch(MyException& e)
{
// 调用虚函数，验证一下这个异常对象是否发生了对象切片
cout<<endl<<e.Test_Virtual_Func()<<endl<<endl;
}
}
```

程序运行的结果是：

```
构造一个 MyException 异常对象，名称为：ex_obj1
构造一个 MyMemoryException 异常对象，名称为：ex_obj1

抛出一个 MyMemoryException 类型的异常

构造一个 MyException 异常对象，名称为：none
拷贝一个 MyMemoryException 异常对象，名称为：ex_obj1
销毁一个 MyMemoryException 异常对象，名称为：ex_obj1
销毁一个 MyException 异常对象，名称为：ex_obj1

这是 MyMemoryException 类型的异常对象

销毁一个 MyMemoryException 异常对象，名称为：ex_obj1
销毁一个 MyException 异常对象，名称为：ex_obj1
```

总结

（1） 被抛出的异常对象都是临时的局部变量；
（2） 异常对象至少要被构造二次；
（3） catch 后面带的异常对象的作用域仅限于 catch bock 中；
（4） 按引用方式传递不会发生异常对象的切片。

下一篇文章讨论 C++的异常对象被按指针的方式传递。继续吧！

第 12 集 C++的异常对象按指针方式被传递

上两篇文章分别详细讨论了 C++的异常对象按值传递和按引用传递的两种方式，本文继续讨论最后的一种的方式：按指针传递。

异常对象在什么时候构造？

1、与按值和按引用传递异常的方式相比，在按指针传递异常的方式下，异常对象的构造方式有很大的不同。它必须是在堆中动态构造的异常对象，或者是全局性 static 的变量。示例程序如下：

```
void main()
{
try
{
// 动态在堆中构造的异常对象
throw new MyMemoryException("ex_obj1");
}
catch(MyException* e)
{
cout<<endl<<"捕获到一个 MyException*类型的异常，名称为："<<e->GetName()<<endl;

delete e;
}
}
```

2、注意，通过指针方式传递的异常对象不能是局部变量，否则后果很严重，示例如下：

```
void main()
{
try
{
// 局部变量，异常对象
MyMemoryException ex_obj1("ex_obj1");
```

```
// 抛出一个指针类型的异常
// 注意：这样做很危险，因为 ex_obj1 这个对象离开了这个作用域即
// 析构销毁
throw &ex_obj1;
}
catch(MyException* e)
{
// 下面语句虽然不会导致程序崩溃，但是 e->GetName()取得的结果
// 也是不对的。
cout<<endl<<"捕获到一个 MyException*类型的异常，名称为："<<e->GetName()<<endl;

// 这条语句会导致程序崩溃
// delete e;
}
}
```

程序运行的结果是：

构造一个 MyException 异常对象，名称为：ex_obj1

构造一个 MyMemoryException 异常对象，名称为：ex_obj1

销毁一个 MyMemoryException 异常对象，名称为：ex_obj1

销毁一个 MyException 异常对象，名称为：ex_obj1

捕获到一个 MyException*类型的异常，名称为：

异常对象按指针方式被传递

指针是历史悠久的一种数据类型形式，这可以追溯到 C 语言和 PASCAL 语言中。异常对象按指针方式传递，当然更是不会发生对象的拷贝复制过程。所以这种方式传递异常对象是效率最高的，它从抛出异常、捕获异常再到异常错误处理结束过程中，总共只会发生唯一的一次对象的构造过程，那就是异常对象最初在堆中的动态创建时的初始化的构造过程。也看看示例程序吧！如下：

```
void main()
{
try
{
// 动态在堆中构造的异常对象
throw new MyMemoryException("ex_obj1");
}
// 注意：这里是定义了按指针方式传递异常对象
catch(MyException* e)
{
cout<<endl<<"捕获到一个 MyException*类型的异常，名称为："<<e->GetName()<<endl;

delete e;
}
}
```

程序运行的结果是：
构造一个 MyException 异常对象，名称为：ex_obj1
构造一个 MyMemoryException 异常对象，名称为：ex_obj1

捕获到一个 MyException*类型的异常，名称为：ex_obj1
销毁一个 MyMemoryException 异常对象，名称为：ex_obj1
销毁一个 MyException 异常对象，名称为：ex_obj1

呵呵！程序的运行结果是不是显示出异常对象只有一次的构造过程。挺好挺好！

异常对象什么时候被销毁

异常对象动态地在堆上被创建，同时它也要动态的被销毁，否则就必然会发生内存泄漏。那么异常对象应该在什么时候被销毁比较合适呢？当然应该是在 catch block 块中处理完毕后再销毁它才比较合理。示例如下：

```
void main()
{
try
{
// 动态在堆中构造的异常对象
throw new MyMemoryException("ex_obj1");
}
catch(MyException* e)
{
cout<<endl<<"捕获到一个 MyException*类型的异常，名称为："<<e->GetName()<<endl;

// 这里需要显示的删除异常对象
delete e;
}
}
```

指针方式下，异常对象会发生对象切片吗？
当然不会，试都不用试，阿愚非常有把握确信这一点。

总结

（1） 被抛出的异常对象不能是局部变量或临时变量，必须是在堆中动态构造的异常对象，或者是全局性 static 的变量；
（2） 异常对象只会被构造一次；
（3） catch 后面带的异常对象的作用域仅限于 catch bock 中；
（4） 异常对象动态地在堆上被创建，同时它也要动态的被销毁。

至此为止，C++的异常对象的传递的三种方式（指针、传值和引用）都已经讨论过了，主人公阿愚到此算是松了一大口气，终于把这块比较难肯一点的骨头给拿下了（呵呵！^_^）。为了更进一步巩固这些知识。下一篇文章准备对这三种方式来一个综合性的大比较！去看看吧！

第 13 集 C++异常对象三种方式传递的综合比较

上几篇文章已经分别对 C++的异常对象的几种不同的传递方式进行了详细地讨论。它们可以被分为按值传递，按引用传递，以及按指针传递等三种方式，现在该是对它们进行全面盘点总结的时候了。希望这种对比、总结及分析对朋友们理解这三种方式的各种区别有所帮助。

	按值传递	引用传递	指针传递
语法	catch(std::exception e)	catch(std::exception& e)	catch(std::exception* e)
如何抛出异常？	①throw exception() ②exception ex;throw ex; ③throw ex_global;	①throw exception() ②exception ex;throw ex; ③throw ex_global;	①throw new exception();
异常对象的构造次数	三次	二次	一次
效率	低	中	高
异常对象什么时候被销毁	①局部变量离开作用域时销毁 ②临时变量在 catch block 执行完毕后销毁 ③catch 后面的那个类似参数的异常对象也是在 catch block 执行完毕后销毁	①局部变量离开作用域时销毁 ②临时变量在 catch block 执行完毕后销毁	异常对象动态地在堆上被创建，同时它也要动态的被销毁，销毁的时机是在 catch block 块中处理完毕后进行
发生对象切片	可能会	不会	不会
安全性	较低，可能会发生对象切片	很好	低，依赖于程序员的能力，可能会发生内存泄漏；或导致程序崩溃
综合性能	差	好	一般
易使用性	好	好	一般

至此，对 C++中的异常处理机制与模型已经进行了非常全面的阐述和分析，包括 C++异常的语法，C++异常的使用技巧，C++异常与面向对象的相互关系，以及异常对象的构造、传递和最后析构销毁的过程。

主人公阿愚现在已经开始有点小有成就感了，他知道自己对她（C++中的异常处理机制）已有了相当深入的了解，并且把她完全当成了一个知己，在自己的编程生涯中再也开始离不开她了。而且他与她的配合已经变得十分的默契，得心应手。但是有时好像还是有点迷糊，例如，对于在 C++异常重新被抛出时（rethrow），异常对象的构造、传递和析构销毁的过程又将如何？有哪些不同之处？，要想了解更多的细节，程序员朋友们！请跟主人公阿愚进入到下一篇文章中，GO！

第 14 集 再探 C++中异常的 rethrow

在相遇篇中的《第 5 集 C++的异常 rethrow》文章中，已经比较详细讨论了异常重新被抛出的处理过程。但是有一点却并没有叙述到，那就是 C++异常重新被抛出时（rethrow），异常对象的构造、传递和析构销毁的过程会有哪些变化和不同之处。为了精益求精，力求对每一个细节都深入了解和掌握，下面再全面阐述一下各种不同组合情况下的异常构造和析构的过程。

大家现在知道，异常的重新被抛出有两种方式。其一，由于当前的 catch block 块处理不了这个异常，所以这个异常对象再次原封不动地被重新抛出；其二，就是在当前的 catch block 块处理异常时，又激发了另外一个异常的抛出。另外，由于异常对象的传递方式有三种：传值、传引用和传指针。所以实际上这就导致了有 6 种不同的组合情况。下面分别阐述之。

异常对象再次原封不动地被重新抛出

1、首先讨论异常对象“按值传递”的方式下，异常对象的构造、传递和析构销毁的过程有何不同之处？毫无疑问，在异常被重新被抛出时，前面的一个异常对象的构造和传递过程肯定不会被影响，也即“按值传递”的方式下，异常被构造了 3 次，异常对象被“按值传递”到这个 catch block 中。实际上，需要研究的是，当异常被重新被抛出时，这个异常对象是否在离开当前的这个 catch block 域时会析构销毁掉，并且这个异常对象是否还会再次被复制构造？以及重新被抛出的异常对象按什么方式被传递？看如下例程：

```
class MyException
{
public:
MyException (string name="none") : m_name(name)
{
number = ++count;
cout << "构造一个 MyException 异常对象，名称为："<<m_name<<":"<<number<< endl;
}

MyException (const MyException& old_e)
{
m_name = old_e.m_name;
number = ++count;

cout << "拷贝一个 MyException 异常对象，名称为："<<m_name<<":"<<number<< endl;
}

operator= (const MyException& old_e)
{
m_name = old_e.m_name;
number = ++count;

cout << "赋值拷贝一个 MyException 异常对象，名称为："<<m_name<<":"<<number<< endl;
}
```

```
virtual ~ MyException ()
{
cout << "销毁一个 MyException 异常对象，名称为：" <<m_name<<":"<<number<< endl;
}

string GetName()
{
char tmp[20];
memset(tmp, 0, sizeof(tmp));
sprintf(tmp, "%s:%d", m_name.c_str(), number);
return tmp;
}

virtual string Test_Virtual_Func() { return "这是 MyException 类型的异常对象";}

protected:
string m_name;
int number;

static int count;
};
int MyException::count = 0;

void main()
{
try
{
try
{
// 抛出一个异常对象
throw MyException("ex_obj1");
}
// 异常对象按值传递
catch(MyException e)
{
cout<<endl<<"捕获到一个 MyException*类型的异常，名称为："<<e.GetName()<<endl;
cout<<"下面重新抛出异常"<<endl<<endl;

// 异常对象重新被抛出
throw;
}
}
// 异常对象再次按值传递
catch(MyException e)
{
```

```
cout<<endl<<"捕获到一个 MyException*类型的异常，名称为："<<e.GetName()<<endl;
}
}
```

程序运行的结果是：

```
构造一个 MyException 异常对象，名称为：ex_obj1:1
拷贝一个 MyException 异常对象，名称为：ex_obj1:2
拷贝一个 MyException 异常对象，名称为：ex_obj1:3
销毁一个 MyException 异常对象，名称为：ex_obj1:1

捕获到一个 MyException*类型的异常，名称为：ex_obj1:3
下面重新抛出异常

拷贝一个 MyException 异常对象，名称为：ex_obj1:4
销毁一个 MyException 异常对象，名称为：ex_obj1:3

捕获到一个 MyException*类型的异常，名称为：ex_obj1:4
销毁一个 MyException 异常对象，名称为：ex_obj1:4
销毁一个 MyException 异常对象，名称为：ex_obj1:2
```

通过上面的程序运行结果，可以很明显地看出，异常对象在被重新抛出时，又有了一次拷贝复制的过程，瞧瞧！正常情况下，按值传递异常的方式应该是有 3 次构造对象的过程，可现在有了 4 次。那么这个异常对象在什么时候又再次被复制构造的呢？仔细分析一下，其实也不难明白，“异常对象 ex_obj1:1”是局部变量；“异常对象 ex_obj1:2”是临时变量；“异常对象 ex_obj1:3”是第一个（内层的）catch block 中的参数变量。当在 catch block 中再次 throw 异常对象时，它会即刻准备离开当前的 catch block 域，继续往上搜索对应的 catch block 模块，找到后，即完成异常对象的又一次复制构造过程，也即把异常对象传递给上一层的 catch block 域中。之后，正式离开内层的 catch block 域，并析构销毁这个 catch block 域中的异常对象 ex_obj1:3，注意此时，属于临时变量形式的异常对象 ex_obj1:2 并没有被析构，而是直至到后一个 catch block 处理完后，先析构销毁异常对象 ex_obj1:4，再才销毁异常对象 ex_obj1:2。整个程序的执行流程如图 14-1 所示。

```
void main()
{
   try
   {
      try
      {
         // 动态在堆中构造的异常对象
         throw MyException("ex_obj1"); ①
      } ③
      catch(MyException e) ②
      {
         cout<<endl<<"捕获到一个MyException*类型的异常，名称为："<<e.GetName()<<endl; ④
         cout<<"下面重新抛出异常"<<endl<<endl; ⑤

         throw; ⑥
      } ⑧
   }
   // 注意：这里是定义了按值传递的方式传递异常对象
   catch(MyException e) ⑦
   {
      cout<<endl<<"捕获到一个MyException*类型的异常，名称为："<<e.GetName()<<endl; ⑨
   } ⑩
}
```

图 14-1 异常对象构造和销毁的过程

下面来一步一步看它的流程，第①步执行的操作，及执行完它之后的状态，如图 14-2 所示。它构造了一个局部变量形式的异常对象和拷贝复制了一个临时变量形式的异常对象。

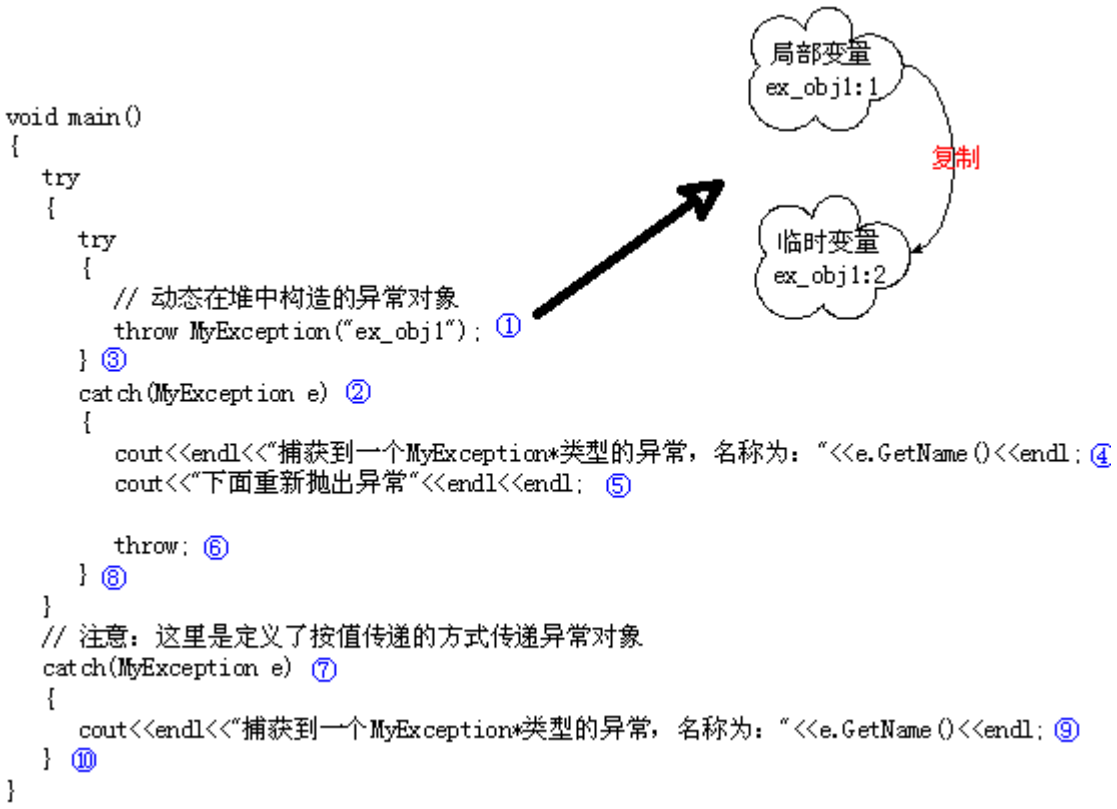


图 14-2 第一步，抛出异常

由于第①步执行的抛出异常的操作，因此找到了相应的 catch block 后，便执行第②步复制异常对象的过程，如图 14-3 所示。

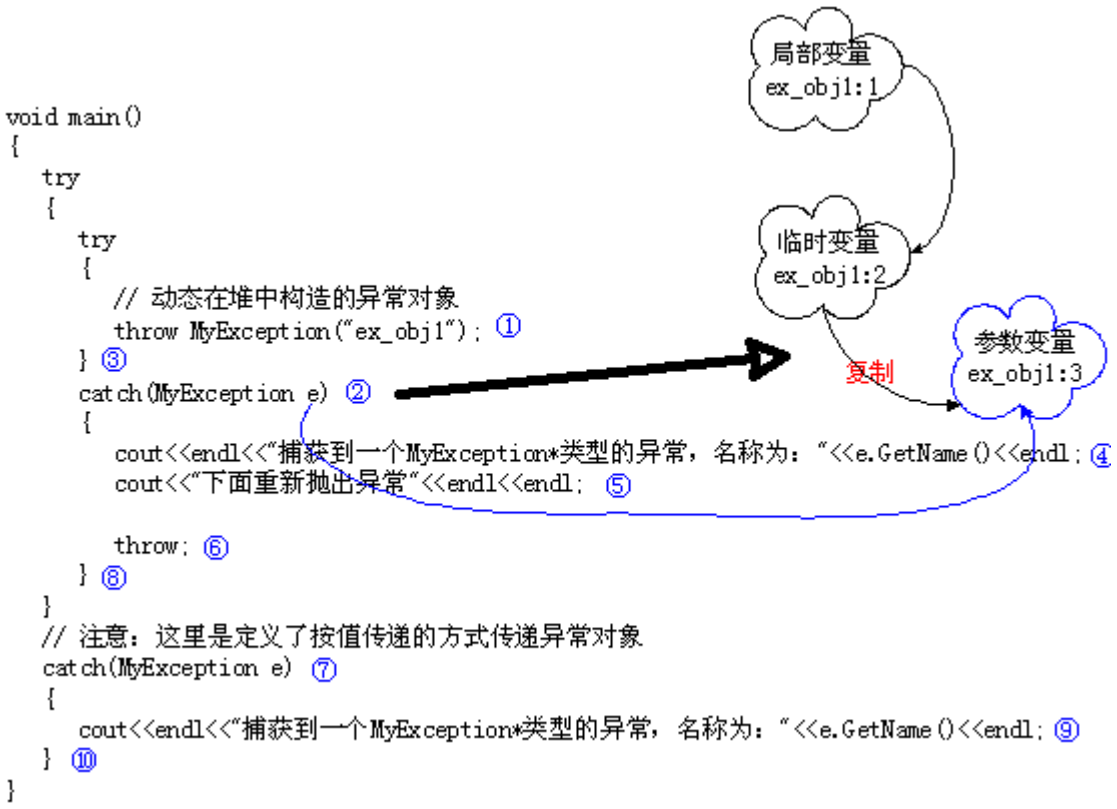


图 14-3 第二步，复制异常到 catch block 域

第②步复制异常对象完毕后，便进入到第③步，离开原来抛出异常的作用域，如图 14-4 所示。这一步的操作是由系统所完成的，主要析构销毁这个当前作用域已经构造过的对象，其中也包括属于局部变量的异常对象。

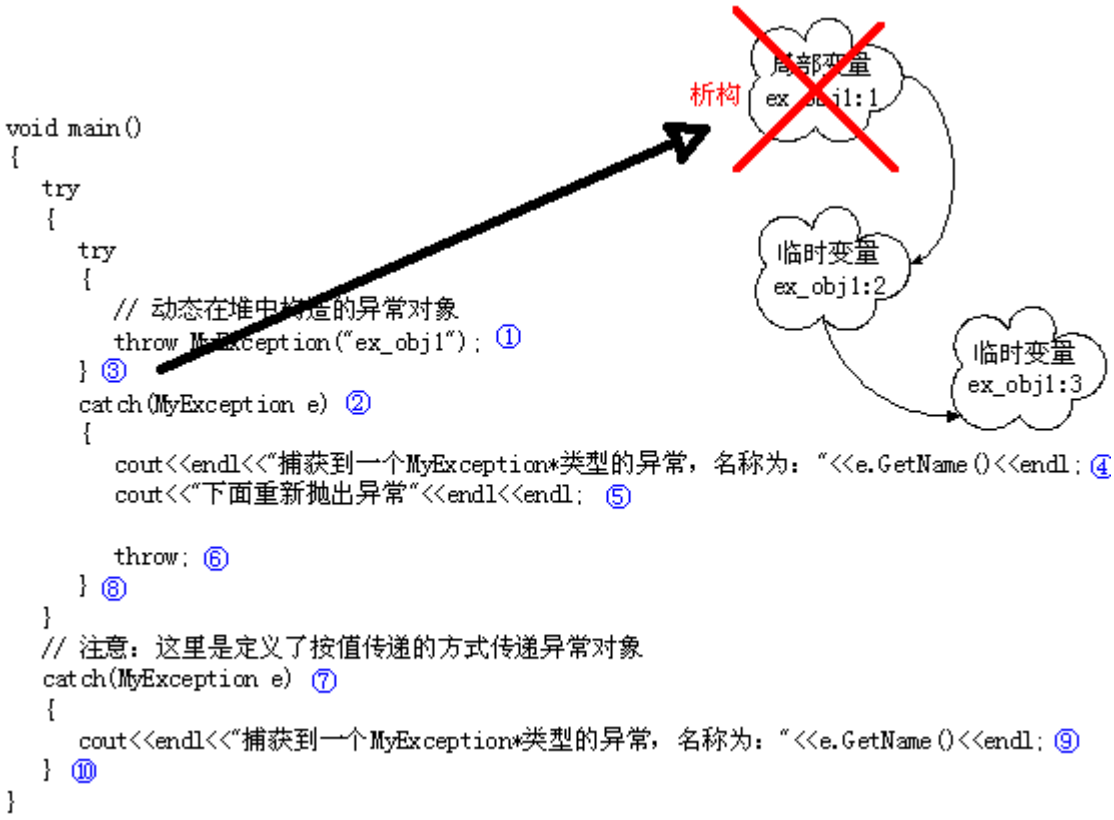


图 14-4 第三步，析构局部变量

接下来正式进入到 catch block 的异常处理模块中，如图 14-5 所示。

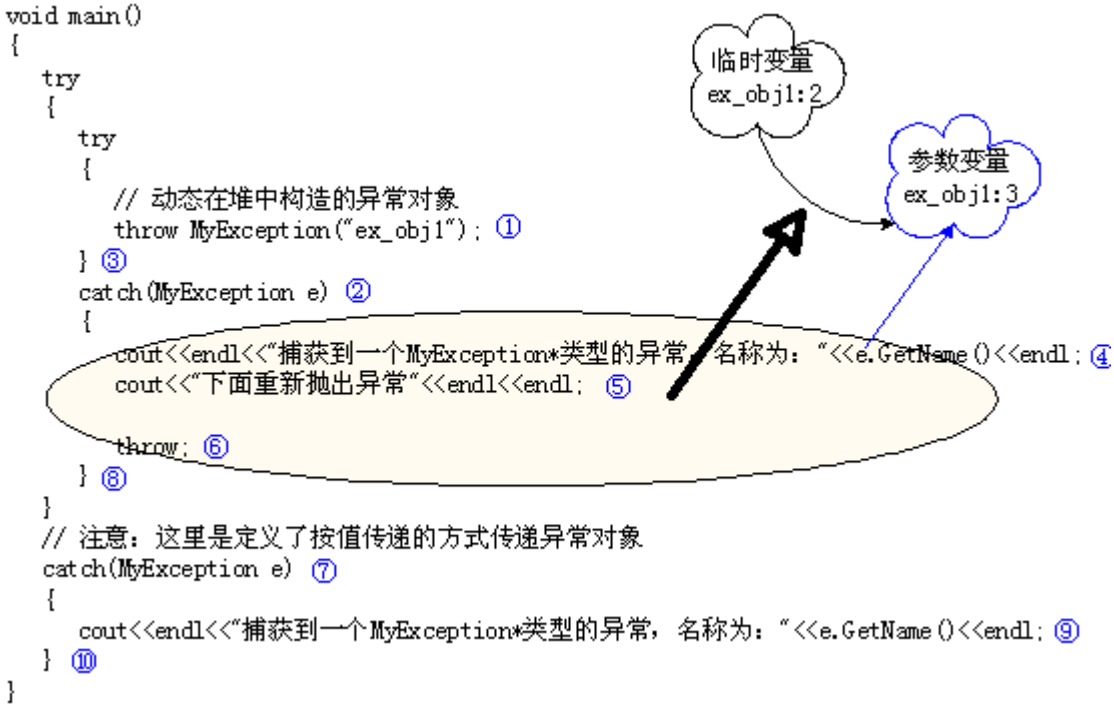


图 14-5 第四步，异常处理模块中

当在异常模块中再次原封不动地把原来的异常对象重新抛出，那么系统将会继续下一次的查找 catch block 模块的过程，如图 14-6 所示。注意，它这里并不会再次复制一个另外的临时异常对象，而只是在新的 catch block 模块中完成一次异常对象的复制过程。

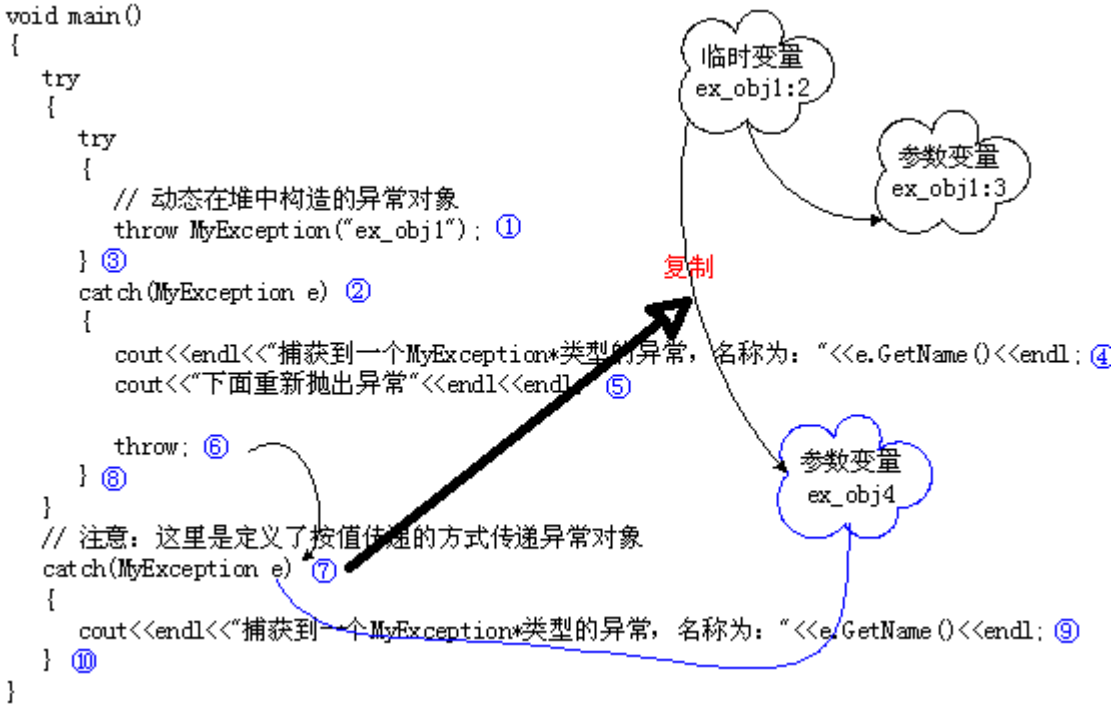


图 14-6 第五步，又一次复制异常到 catch block 域

同样，在复制完毕异常对象以后。程序控制流又会回到原来的作用域去销毁局部变量。如图 14-7 所示。注意，这里并不会析构销毁临时变量的异常对象，而只是销毁当前作用域内部的局部变量，如“异常对象 ex_obj1:3”。

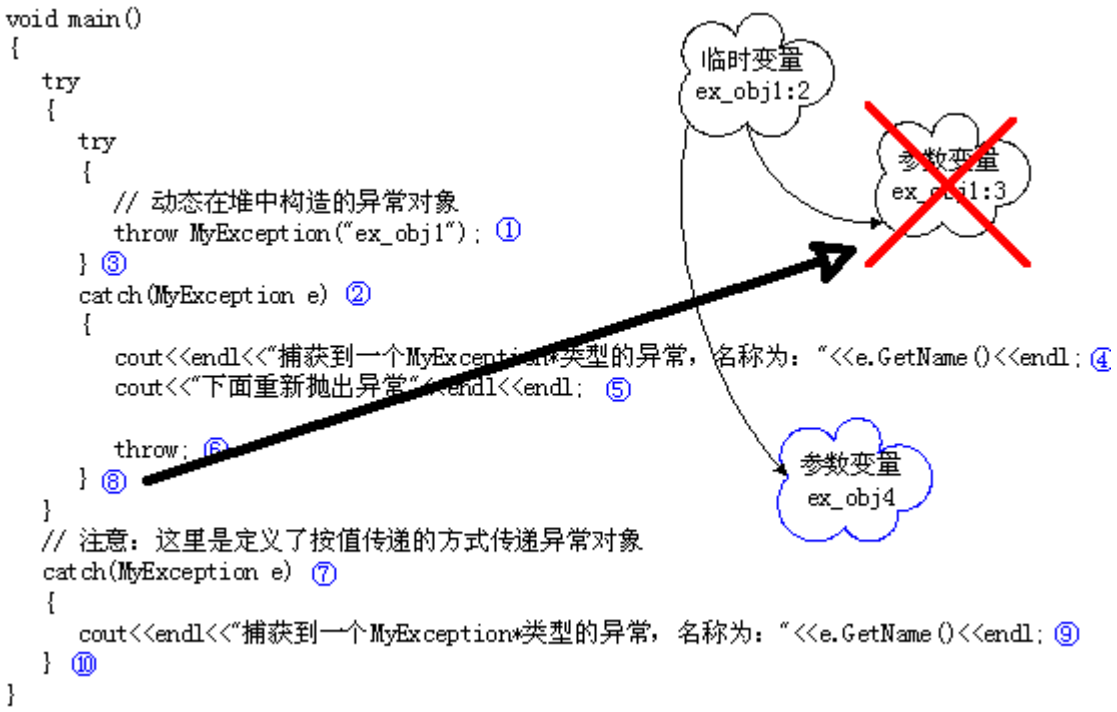


图 14-7 第六步，离开前面的 catch block 作用域，并析构该作用域范围内的局部变量

再接下来就是进入到又一次的异常处理模块中，如图 14-8 所示。注意，此时系统中存在“异常对象 ex_obj1:2”和“异常对象 ex_obj1:4”。

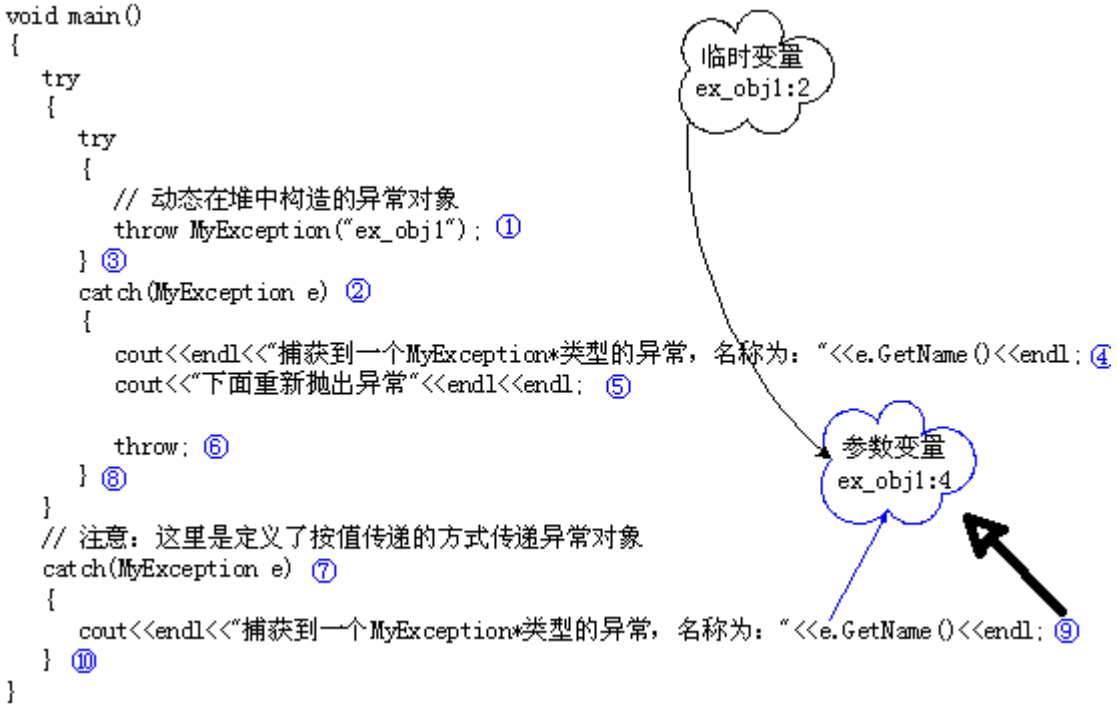


图 14-8 第七步，又一次的异常处理模块中

最后就是全部处理完毕，如图 14-9 所示。注意，它先析构销毁“异常对象 ex_obj1:4”，再才销毁“异常对象 ex_obj1:2”。

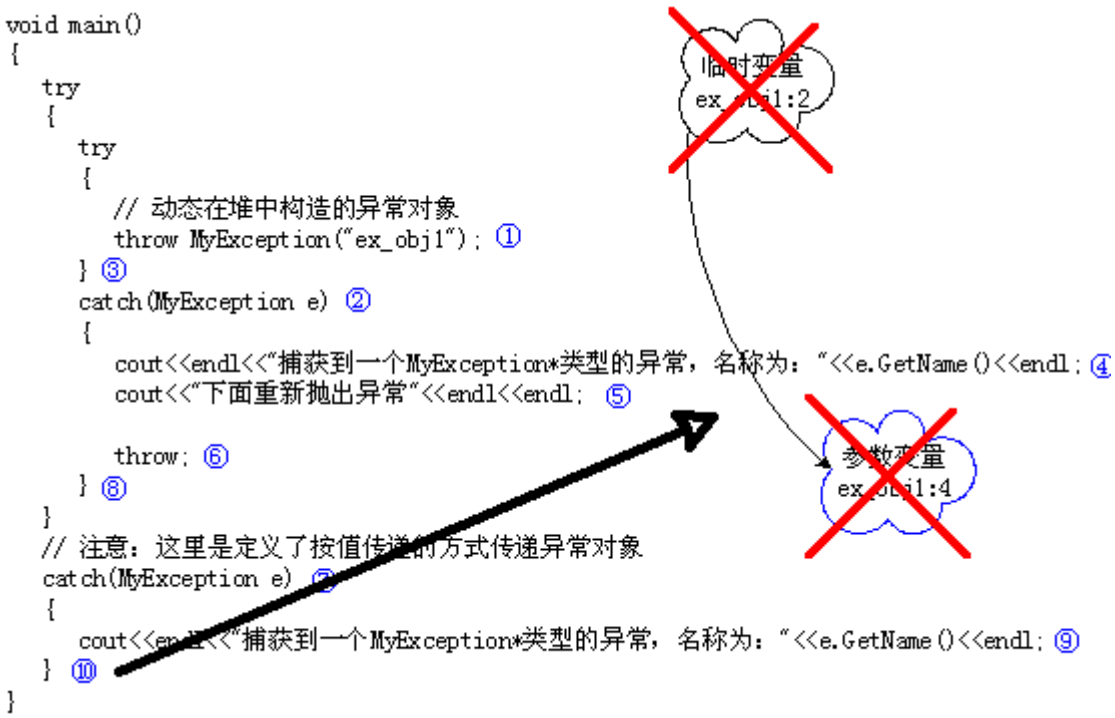


图 14-9 第八步，销毁另外的两个异常对象

通过以上可以清晰地看出，在“按值传递”的方式下，异常对象的被重新 rethrow 后，它的执行过程虽然与正常抛出异常的情况虽然有所差异，但是在原理上，它们完全是相一致的，而且异常的 rethrow 是可以不断地向上抛出，就好像是接力赛一样，同时，每再抛出一次后，异常对象将会被复制构造一次。所以说，这里更进一步说明了异常对象的“按值传递”的方式是效率很低的。但是亲爱的程序员朋友们，大家是否也象那个有点傻气，但好像又有些灵气的主人公阿愚一样，联想到了另外一种有些奇怪的组合方式，那就是如果异常对象第一次是“按值传递”的方式，但第二、第三次，甚至后来的更多次，是否可以按其它方式（如“按引用传递”的方式）呢？如果可以，那么又会出现什么结果呢？还是先看看示例吧！上面的程序只作了一点改动，如下：

```
void main()
{
try
{
try
{
// 抛出一个异常对象
throw MyException("ex_obj1");
```

```
}
// 异常对象按值传递
catch(MyException e)
{
cout<<endl<<"捕获到一个 MyException*类型的异常，名称为："<<e.GetName()<<endl;
cout<<"下面重新抛出异常"<<endl<<endl;

// 异常对象重新被抛出
throw;
}
}
// 注意，这里改为按引用传递的方式
catch(MyException& e)
{
cout<<endl<<"捕获到一个 MyException*类型的异常，名称为："<<e.GetName()<<endl;
}
}
```

程序运行的结果是：

```
构造一个 MyException 异常对象，名称为：ex_obj1:1
拷贝一个 MyException 异常对象，名称为：ex_obj1:2
拷贝一个 MyException 异常对象，名称为：ex_obj1:3
销毁一个 MyException 异常对象，名称为：ex_obj1:1

捕获到一个 MyException*类型的异常，名称为：ex_obj1:3
下面重新抛出异常

销毁一个 MyException 异常对象，名称为：ex_obj1:3

捕获到一个 MyException*类型的异常，名称为：ex_obj1:2
销毁一个 MyException 异常对象，名称为：ex_obj1:2
```

哈哈！主人公阿愚非常开心。因为它果然不出所料，是完全可以的，而且结果也合乎情理。异常对象还是只构造了三次，并未因为异常的再次抛出，而多复制构造一次异常对象。实际上，大家已经知道，控制异常对象的传递方式是由 catch block 后面的参数所决定，所以对于无论是最初抛出的异常，还是异常在 catch block 块被再次抛出，它们无须来关心，也控制不了。在最初抛出的异常时，完成两次异常对象的构造过程，其中最重要的是临时的异常对象，它是提供向其它参数形式的异常对象复制构造的原型，也即异常在不断接力地被抛出之后，如果上层的某个 catch block 定义“按值传递”的方式，那么系统就会从这个临时变量的异常对象复制一份；如果上层的某个 catch block 定义“按引用传递”的方式，那么系统会把引用指向这个临时变量的异常对象。而这个临时变量的异常对象，只有在最后一个 catch block 块（也即没有再次抛出）执行处理完毕之后，才会把这个异常对象予以析构销毁（实际上，在这里销毁是最恰当的，因为异常的重新被抛出，表明这个异常还没有被处理完毕，所以只有到最后一个 catch block 之后，这个临时变量的异常对象才真正不需要了）。

另外，还有一点需要进一步阐述，那就是上层的某个 catch block 定义“按值传递”的方式下，系统从临时变量的异常对象所复制一份参数形式的异常对象，它一定会在它这个作用域无效时，把它给析构销毁掉。

2、接下来，讨论异常对象“按引用传递”的方式下，异常对象的构造、传递和析构销毁的过程有何不同之处？其实这在刚才已经详细讨论过了，不过，还是看看例程来验证一下，如下：

```
void main()
{
try
{
try
{
// 抛出一个异常对象
throw MyException("ex_obj1");
}
// 这里改为按引用传递的方式
catch(MyException& e)
{
cout<<endl<<"捕获到一个 MyException*类型的异常，名称为："<<e.GetName()<<endl;
cout<<"下面重新抛出异常"<<endl<<endl;

// 异常对象重新被抛出
throw;
}
}
// 这里改为按引用传递的方式
catch(MyException& e)
{
cout<<endl<<"捕获到一个 MyException*类型的异常，名称为："<<e.GetName()<<endl;
}
}
```

程序运行的结果是：

构造一个 MyException 异常对象，名称为：ex_obj1:1
拷贝一个 MyException 异常对象，名称为：ex_obj1:2
销毁一个 MyException 异常对象，名称为：ex_obj1:1

捕获到一个 MyException*类型的异常，名称为：ex_obj1:2
下面重新抛出异常

捕获到一个 MyException*类型的异常，名称为：ex_obj1:2
销毁一个 MyException 异常对象，名称为：ex_obj1:2

结果不出所料，异常对象永远也只会被构造两次。所以异常对象“按引用传递”的方式，是综合性能最好的一种方式，效率既非常高（仅比“按指针传递”的方式多一次），同时也很安全和友善直观（这一点比“按

指针传递”的方式好很多）。另外，这里也同样可以把“按引用传递”的方式和“按值传递”的方式相混合，代码示例如下：

```
void main()
{
try
{
try
{
// 抛出一个异常对象
throw MyException("ex_obj1");
}
// 这里按引用传递的方式
catch(MyException& e)
{
cout<<endl<<"捕获到一个 MyException*类型的异常，名称为："<<e.GetName()<<endl;
cout<<"下面重新抛出异常"<<endl<<endl;

// 异常对象重新被抛出
throw;
}
}
// 这里按值传递的方式
catch(MyException e)
{
cout<<endl<<"捕获到一个 MyException*类型的异常，名称为："<<e.GetName()<<endl;
}
}
```

3、最后，讨论异常对象“按指针传递”的方式下，异常对象的构造、传递和析构销毁的过程有何不同之处？其实这种方式不需要过多讨论，因为异常对象“按指针传递”的方式下，异常对象永远也只会需要被构造一次，实际上，它被传递只是一个 32bit 的指针值而已，不会涉及到异常对象的拷贝复制过程。但是有一点是需要注意的，那就是对异常对象的析构销毁必须要放在最后一个 catch block 处理完之后，中间层的 catch block 是决不应该 delete 掉这个一般在堆中分配的异常对象。

catch block 块处理异常时，又激发了另外一个异常的抛出

呵呵！表面上看起来，这种情况下会很复杂，因为好像前面一个异常错误还没有被处理完，又引发了另外的一个异常错误，岂不是很麻烦呀！其实不然，系统对这种接力方式的异常重新抛出的处理策略往往很简单，那就是系统认为，当在 catch block 的代码执行过程中，如果抛出另一个异常，而导致控制流离开此 catch block 域，那么前一个异常会被认为处理完毕，并释放临时的异常对象，同时产生下一个异常的搜索 catch block 过程和异常处理的过程等。也即就是说，系统会把这种异常的重新抛出情况，认为是两次分离的异常。虽然它们是连在一起，并能够形成异常的接力抛出，但是处理上，它们完全是被分开进行的。所以说，这种情况下，往往会产生后一次异常对前一次异常的覆盖。

另一种特殊的形式的异常被重新抛出

前面我们所讨论的异常被重新抛出，它们都会导致控制流离开 catch block 模块，也即整个异常的接力处理过程是分层进行的。但实际上，异常的重新抛出后，是可以把它局限于当前的 catch block 域内，让它逃离不开这个作用域。也即通过在 catch block 再潜套一个 try catch 块，示例代码如下：

```
void main()
{
try
{
try
{
// 动态在堆中构造的异常对象
throw MyException("ex_obj1");
}
catch(MyException& e)
{
// 这里再潜套一个 try catch 块
try
{
cout<<endl<<"捕获到一个 MyException*类型的异常，名称为："<<e.GetName()<<endl;
cout<<"下面重新抛出异常"<<endl<<endl;

//重新抛出异常
throw;

// 或者这样重新抛出异常
throw e;
}
catch(MyException& ex)
{
}
// 永远也逃不出我的魔掌
catch(...)
{
}
}
}
catch(MyException& e)
{
cout<<endl<<"捕获到一个 MyException*类型的异常，名称为："<<e.GetName()<<endl;
}
}
```

呵呵，通过上面的方式，可以让当前的 catch block 的处理得以安全保障，防止可能潜在的异常再次出现，而导致上层可能处理不了其它的意外异常，而引发程序崩溃。所以说，C++异常模型的确是非常灵活，功能也非常强大。

总结

（1） 异常重新的被抛出，它的处理过程虽然稍微略有些复杂，但是总体上还是比较易于理解的。它更不会影响并破坏前面章节中所讲述过的许多规则，它们的处理策略和思想是相一致的；

（2） 再次强调，异常对象“按引用传递”的方式是综合性能最佳的方式。

对 C++中的异常处理机制的阐述，到此暂告一个段落，从下篇文章开始，主人公阿愚将继续开阔自身的视野，以对异常处理编程进入一个更加广泛的探讨之中。各位程序员朋友们，继续吧！

第 15 集 C 语言中的异常处理机制

在这之前的所有文章中，都是阐述关于 C++的异常处理机制。的确，在 C++语言中，它提供的异常处理的模型是非常完善的，主人公阿愚因此才和“异常处理”结下了不解之缘，才有了这一系列文章的基本素材，同时主人公阿愚在自己的编程开发过程中，也才更离不开她，喜欢并依赖于她。

另外，C++语言中完善的异常处理的模型，也更激发了主人公阿愚更多其它的思考。难道异常处理机制只有在 C++语言中才有吗？不是的，绝对不是这样的。实际上，异常处理的机制是无处不在的，它与软件的编程思想的发展，与编程语言的发展是同步的。异常处理机制自身的发展和完善过程，也是并记录了我们在编程思想上和编程方法上的改变、进步和发展的过程和重要的足迹。

在前面的文章中，早就讲到过，异常处理的核心思想是，把功能模块代码与系统中可能出现错误的处理代码分离开来，以此来达到使我们的代码组织起来更美观、逻辑上更清晰，并且同时从根本上来提高我们软件系统长时间稳定运行的可靠性。那么，现在回过头来看，实际上在计算机系统的硬件设计中，操作系统的总体设计中，早期的许多面向结构化程序设计语言中（例如 C 语言），都有异常处理的机制和方法的广泛运用。只不过是到了像 C++这样面向对象的程序设计语言中，才把异常处理的模型设计到了一个相当理想和完善的程度。下面来看看主人公阿愚对在 C 语言中，异常处理机制的如何被运用？

goto 语句，实现异常处理编程，最初也最原始的支持手段

1、goto 语句，程序员朋友们对它太熟悉了，它是 C 语言中使用最为灵活的一条语句，由它也充分体现出了 C 语言的许多特点或者说是优点。它虽然是一条高级语言中提供的语句，但是它一般却直接对应一条“无条件直接跳转的机器指令”，所以说它非常地特别，它引起过许多争议，但是这条语句仍然一直被保留了下来，即便是今天的 C++语言中，也有对它的支持（虽然不建议使用它）。goto 语句有非常多的用途或优点，例如，它特别适合于在编写系统程序中被使用，它能使编写出来的代码非常简练。另外，goto 语句另外一个最重要的作用就是，它实际上是一种对异常处理编程，最初也最原始的支持手段或方法。它能把错误处理模块的代码有效与其它代码分离开来。例程如下（请与第一集文章中的示例代码相比较）：

```
void main(int argc, char* argv[])
{
```

```
if (Call_Func1(in, param out)
{
// 函数调用成功，我们正常的处理
if (Call_Func2(in, param out)
{
// 函数调用成功，我们正常的处理
while(condition)
{
//do other job

// 如果错误直接跳转
if (has error) goto Error;

//do other job
}
}
// 如果错误直接跳转
else goto Error;

}
// 如果错误直接跳转
else goto Error;

// 错误处理模块
Error:
process_error();
exit();

}
```

呵呵！上面经过改善后的代码是不是更加清晰了一些，也更简练了一些。因此说，goto 语句确是是能够很好地完成一些简易的异常处理编程的实现。虽然它较 C++语言中提供的异常处理编程模型相差甚远。

为什么不建议使用 goto 语句来实现异常处理编程

虽然 goto 语句能有效地支持异常处理编程的实现。但是没有人却建议使用它，即便是在 C 语言中。因为：

（1） goto 语句能破坏程序的结构化设计，使代码难于测试，且包含大量 goto 的代码模块不易理解和阅读。它一直遭结构化程序设计思想所抛弃，强烈建议程序员不易使用它；

（2） 与 C++语言中提供的异常处理编程模型相比，它的确是太弱了一些。例如，它一般只能是在某个函数的局部作用域内跳转，也即它不能有效和方便地实现程序控制流的跨函数远程的跳转。

（3） 如果在 C++语言中，用 goto 语句来实现异常处理，那么它将给面向对象构成极大破坏，并影响到效率。这一点，以后会继续深入阐述。

总结

虽然 goto 语句缺点多多，但不管如何，goto 语句的确为程序员朋友们，在 C 语言中，有效运用异常处理思想来进行编程处理，提供了一种途径或简易的手段。当然，运用 goto 语句来进行异常处理编程已经成为历史。因为，在 C 语言中，早就已经提供了一种更加优雅的异常处理机制。去看看吧！继续！

第 16 集 C 语言中一种更优雅的异常处理机制

上一篇文章对 C 语言中的 goto 语句进行了较深入的阐述，实际上 goto 语句是面向过程与面向结构化程序语言中，进行异常处理编程的最原始的支持形式。后来为了更好地、更方便地支持异常处理编程机制，使得程序员在 C 语言开发的程序中，能写出更高效、更友善的带有异常处理机制的代码模块来。于是，C 语言中出现了一种更优雅的异常处理机制，那就是 setjmp()函数与 longjmp()函数。

实际上，这种异常处理的机制不是 C 语言中自身的一部分，而是在 C 标准库中实现的两个非常有技巧的库函数，也许大多数 C 程序员朋友们对它都很熟悉，而且，通过使用 setjmp()函数与 longjmp()函数组合后，而提供的对程序的异常处理机制，以被广泛运用到许多 C 语言开发的库系统中，如 jpg 解析库，加密解密库等等。

也许 C 语言中的这种异常处理机制，较 goto 语句相比较，它才是真正意义上的、概念上比较彻底的，一种异常处理机制。作风一向比较严谨、喜欢刨根问底的主人公阿愚当然不会放
弃对这种异常处理机制进行全面而深入的研究。下面一起来看看。

setjmp 函数有何作用？

前面刚说了，setjmp 是 C 标准库中提供的一个函数，它的作用是保存程序当前运行的一些状态。它的函数原型如下：

```
int setjmp( jmp_buf env );
```

这是 MSDN 中对它的评论，如下：

setjmp 函数用于保存程序的运行时的堆栈环境，接下来的其它地方，你可以通过调用 longjmp 函数来恢复先前被保存的程序堆栈环境。当 setjmp 和 longjmp 组合一起使用时，它们能提供一种在程序中实现“非本地局部跳转”（"non-local goto"）的机制。并且这种机制常常被用于来实现，把程序的控制流传递到错误处理模块之中；或者程序中不采用正常的返回（return）语句，或函数的正常调用等方法，而使程序能被恢复到先前的一个调用例程（也即函数）中。

对 setjmp 函数的调用时，会保存程序当前的堆栈环境到 env 参数中；接下来调用 longjmp 时，会根据这个曾经保存的变量来恢复先前的环境，并且当前的程序控制流，会因此而返回到先前调用 setjmp 时的程序执行点。此时，在接下来的控制流的例程中，所能访问的所有的变量（除寄存器类型的变量以外），包含了 longjmp 函数调用时，所拥有的变量。

setjmp 和 longjmp 并不能很好地支持 C++中面向对象的语义。因此在 C++程序中，请使用 C++提供的异常处理机制。

好了，现在已经对 setjmp 有了很感性的了解，暂且不做过多评论，接着往下看 longjmp 函数。

longjmp 函数有何作用？

同样，longjmp 也是 C 标准库中提供的一个函数，它的作用是用于恢复程序执行的堆栈环境，它的函数原型如下：

void longjmp(jmp_buf env, int value);

这是 MSDN 中对它的评论，如下：

longjmp 函数用于恢复先前程序中调用的 setjmp 函数时所保存的堆栈环境。setjmp 和 longjmp 组合一起使用时，它们能提供一种在程序中实现“非本地局部跳转”（"non-local goto"）的机制。并且这种机制常常被用于来实现，把程序的控制流传递到错误处理模块，或者不采用正常的返回（return）语句，或函数的正常调用等方法，使程序能被恢复到先前的一个调用例程（也即函数）中。

对 setjmp 函数的调用时，会保存程序当前的堆栈环境到 env 参数中；接下来调用 longjmp 时，会根据这个曾经保存的变量来恢复先前的环境，并且因此当前的程序控制流，会返回到先前调用 setjmp 时的执行点。此时，value 参数值会被 setjmp 函数所返回，程序继续得以执行。并且，在接下来的控制流的例程中，它所能够访问到的所有的变量（除寄存器类型的变量以外），包含了 longjmp 函数调用时，所拥有的变量；而寄存器类型的变量将不可预料。setjmp 函数返回的值必须是非零值，如果 longjmp 传送的 value 参数值为 0，那么实际上被 setjmp 返回的值是 1。

在调用 setjmp 的函数返回之前，调用 longjmp，否则结果不可预料。

在使用 longjmp 时，请遵守以下规则或限制：

· 不要假象寄存器类型的变量将总会保持不变。在调用 longjmp 之后，通过 setjmp 所返回的控制流中，例程中寄存器类型的变量将不会被恢复。

· 不要使用 longjmp 函数，来实现把控制流，从一个中断处理例程中传出，除非被捕获的异常是一个浮点数异常。在后一种情况下，如果程序通过调用_fpreset 函数，来首先初始化浮点数包后，它是可以通过 longjmp 来实现从中断处理例程中返回。

· 在 C++程序中，小心对 setjmp 和 longjmp 的使用，应为 setjmp 和 longjmp 并不能很好地支持 C++中面向对象的语义。因此在 C++程序中，使用 C++提供的异常处理机制将会更加安全。

把 setjmp 和 longjmp 组合起来，原来它这么厉害！

现在已经对 setjmp 和 longjmp 都有了很感性的了解，接下来，看一个示例，并从这个示例展开分析，示例代码如下（来源于 MSDN）：

```
/* FPRESET.C: This program uses signal to set up a
* routine for handling floating-point errors.
*/

#include <stdio.h>
#include <signal.h>
#include <setjmp.h>
#include <stdlib.h>
#include <float.h>
#include <math.h>
#include <string.h>

jmp_buf mark; /* Address for long jump to jump to */
int fperr; /* Global error number */

void __cdecl fphandler( int sig, int num ); /* Prototypes */
void fpcheck( void );

void main( void )
{
double n1, n2, r;
int jmpret;
/* Unmask all floating-point exceptions. */
_control87( 0, _MCW_EM );
/* Set up floating-point error handler. The compiler
* will generate a warning because it expects
* signal-handling functions to take only one argument.
*/
if( signal( SIGFPE, fphandler ) == SIG_ERR )

{
fprintf( stderr, "Couldn't set SIGFPE\n" );
abort(); }

/* Save stack environment for return in case of error. First
* time through, jmpret is 0, so true conditional is executed.
* If an error occurs, jmpret will be set to -1 and false
* conditional will be executed.
*/

// 注意，下面这条语句的作用是，保存程序当前运行的状态
jmpret = setjmp( mark );
if( jmpret == 0 )
{
printf( "Test for invalid operation - " );
```

```
printf( "enter two numbers: " );
scanf( "%lf %lf", &n1, &n2 );

// 注意，下面这条语句可能出现异常，
// 如果从终端输入的第 2 个变量是 0 值的话
r = n1 / n2;
/* This won't be reached if error occurs. */
printf( "\n\n%4.3g / %4.3g = %4.3g\n", n1, n2, r );

r = n1 * n2;
/* This won't be reached if error occurs. */
printf( "\n\n%4.3g * %4.3g = %4.3g\n", n1, n2, r );
}
else
fpcheck();
}
/* fphandler handles SIGFPE (floating-point error) interrupt. Note
* that this prototype accepts two arguments and that the
* prototype for signal in the run-time library expects a signal
* handler to have only one argument.
*
* The second argument in this signal handler allows processing of
* _FPE_INVALID, _FPE_OVERFLOW, _FPE_UNDERFLOW, and
* _FPE_ZERODIVIDE, all of which are Microsoft-specific symbols
* that augment the information provided by SIGFPE. The compiler
* will generate a warning, which is harmless and expected.

*/
void fphandler( int sig, int num )
{
/* Set global for outside check since we don't want
* to do I/O in the handler.
*/
fperr = num;
/* Initialize floating-point package. */
_fpreset();
/* Restore calling environment and jump back to setjmp. Return
* -1 so that setjmp will return false for conditional test.
*/
// 注意，下面这条语句的作用是，恢复先前 setjmp 所保存的程序状态
longjmp( mark, -1 );
}
void fpcheck( void )
{
char fpstr[30];
```

```
switch( fperr )
{
case _FPE_INVALID:
strcpy( fpstr, "Invalid number" );
break;
case _FPE_OVERFLOW:
strcpy( fpstr, "Overflow" );

break;
case _FPE_UNDERFLOW:
strcpy( fpstr, "Underflow" );
break;
case _FPE_ZERODIVIDE:
strcpy( fpstr, "Divide by zero" );
break;
default:
strcpy( fpstr, "Other floating point error" );
break;
}
printf( "Error %d: %s\n", fperr, fpstr );
}
```

程序的运行结果如下：

```
Test for invalid operation - enter two numbers: 1 2


1 / 2 = 0.5


1 * 2 = 2
```

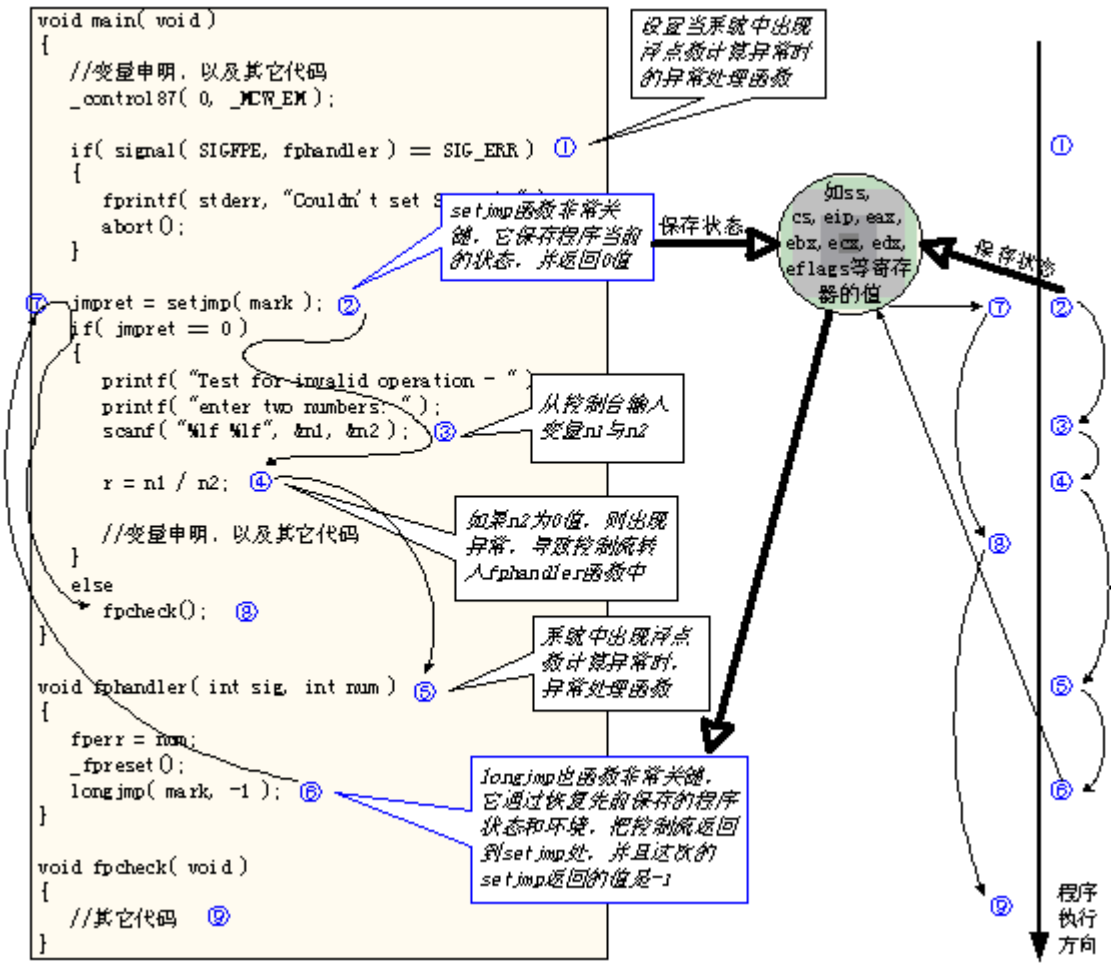
上面的程序运行结果正常。另外程序的运行结果还有一种情况，如下：

```
Test for invalid operation - enter two numbers: 1 0
Error 131: Divide by zero
```

呵呵！程序运行过程中出现了异常（被 0 除），并且这种异常被程序预先定义的异常处理模块所捕获了。厉害吧！可千万别轻视，这可以 C 语言编写的程序。

分析 setjmp 和 longjmp

现在，来分析上面的程序的执行过程。当然，这里主要分析在异常出现的情况下，程序运行的控制转移流程。由于文章篇幅有限，分析时，我们简化不相关的代码，这样更也易理解控制流的执行过程。如下图所示。



呵呵！现在是否对程序的执行流程一目了然，其中最关键的就是 setjjmp 和 longjmp 函数的调用处理。我们分别来分析之。

当程序运行到第②步时，调用 setjmp 函数，这个函数会保存程序当前运行的一些状态信息，主要是一些系统寄存器的值，如 ss，cs，eip，eax，ebx，ecx，edx，eflags 等寄存器，其中尤其重要的是 eip 的值，因为它相当于保存了一个程序运行的执行点。这些信息被保存到 mark 变量中，这是一个 C 标准库中所定义的特殊结构体类型的变量。

调用 setjmp 函数保存程序状态之后，该函数返回 0 值，于是接下来程序执行到第③步和第④步中。在第④步中语句执行时，如果变量 n2 为 0 值，于是便引发了一个浮点数计算异常，，导致控制流转入 fphandler 函数中，也即进入到第⑤步。

然后运行到第⑥步，调用 longjmp 函数，这个函数内部会从先前的 setjmp 所保存的程序状态，也即 mark 变量中，来恢复到以前的系统寄存器的值。于是便进入到了第⑦步，注意，这非常有点意思，实际上，通过 longjmp 函数的调用后，程序控制流（尤其是 eip 的值）再次戏剧性地进入到了 setjmp 函数的处理内部中，但是这一次 setjmp 返回的值是 longjmp 函数调用时，所传入的第 2 个参数，也即-1，因此程序接下来进入到了第⑧步的执行之中。

总结

与 goto 语句不同，在 C 语言中，setjmp()与 longjmp()的组合调用，为程序员提供了一种更优雅的异常处理机制。它具有如下特点：

（1） goto 只能实现本地跳转，而 setjmp()与 longjmp()的组合运用，能有效的实现程序控制流的非本地（远程）跳转；

（2） 与 goto 语句不同，setjmp()与 longjmp()的组合运用，提供了真正意义上的异常处理机制。例如，它能有效定义受监控保护的模块区域（类似于 C++中 try 关键字所定义的区域）；同时它也能有效地定义异常处理模块（类似于 C++中 catch 关键字所定义的区域）；还有，它能在程序执行过程中，通过 longjmp 函数的调用，方便地抛出异常（类似于 C++中 throw 关键字）。

现在，相信大家已经对在 C 语言中提供的这种异常处理机制有了很全面地了解。但是我们还没有深入它研究它，下一篇文章中继续探讨吧！go！

第 17 集 全面了解 setjmp 与 longjmp 的使用

上一篇文章对 setjmp 函数与 longjmp 函数有了较全面的了解，尤其是这两个函数的作用，函数所完成的功能，以及将 setjmp 函数与 longjmp 函数组合起来，实现异常处理机制时，程序模块控制流的执行过程等。这里更深入一步，将对 setjmp 与 longjmp 的具体使用方法和适用的场合，进行一个非常全面的阐述。

另外请特别注意，setjmp 函数与 longjmp 函数总是组合起来使用，它们是紧密相关的一对操作，只有将它们结合起来使用，才能达到程序控制流有效转移的目的，才能按照程序员的预先设计的意图，去实现对程序中可能出现的异常进行集中处理。

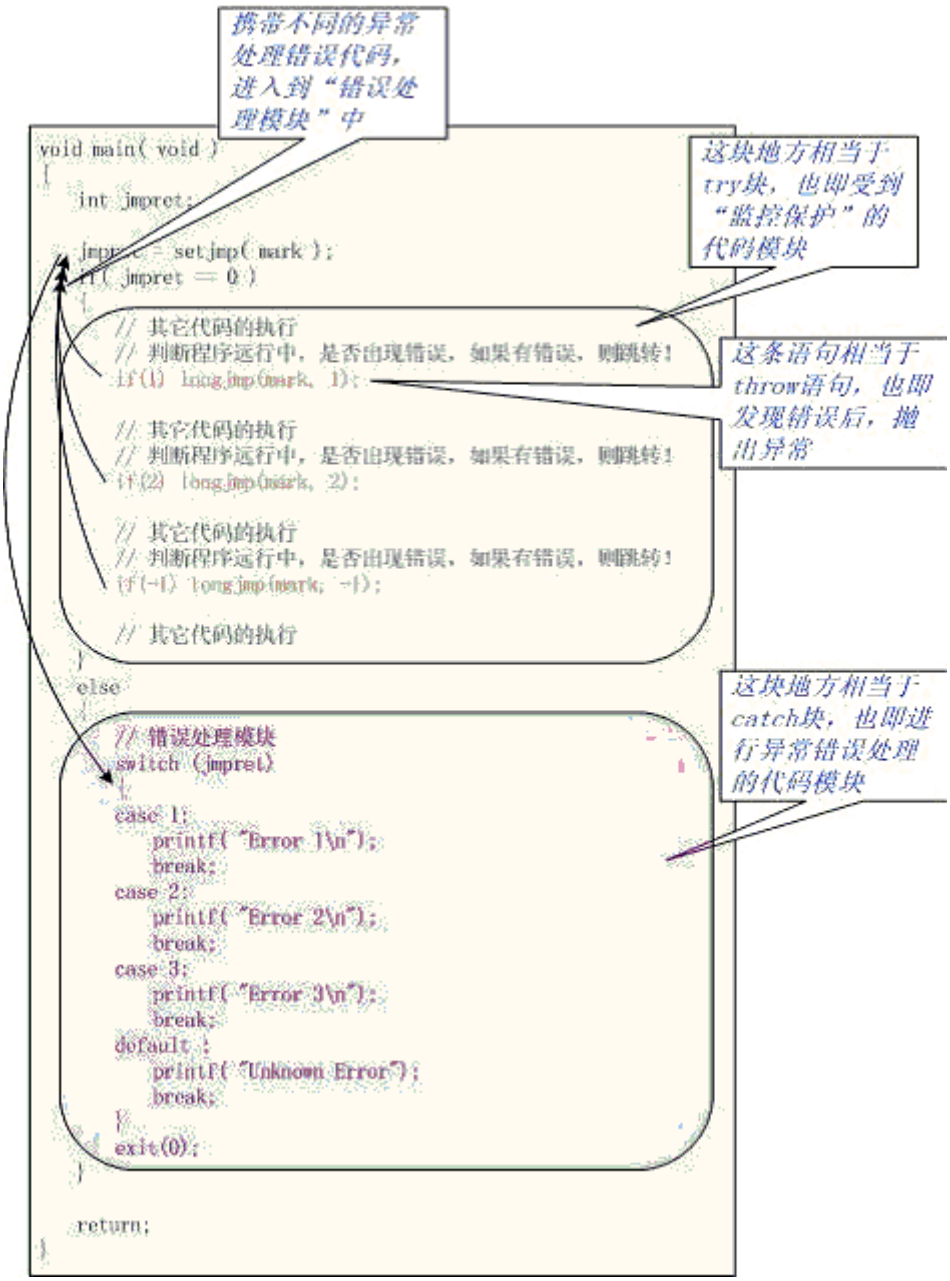
与 goto 语句的作用类似，它能实现本地的跳转

这种情况容易理解，不过还是列举出一个示例程序吧！如下：

```
void main( void )
{
int jmpret;

jmpret = setjmp( mark );
if( jmpret == 0 )
{
// 其它代码的执行
// 判断程序远行中，是否出现错误，如果有错误，则跳转！
if(1) longjmp(mark, 1);
```

```
// 其它代码的执行
// 判断程序远行中，是否出现错误，如果有错误，则跳转！
if(2) longjmp(mark, 2);

// 其它代码的执行
// 判断程序远行中，是否出现错误，如果有错误，则跳转！
if(-1) longjmp(mark, -1);

// 其它代码的执行
}
else
{
// 错误处理模块
switch (jmpret)
{
case 1:
printf( "Error 1\n");
break;
case 2:
printf( "Error 2\n");
break;
case 3:
printf( "Error 3\n");
break;
default :
printf( "Unknown Error");
break;
}
exit(0);
}

return;
}
```

上面的例程非常地简单，其中程序中使用到了异常处理的机制，这使得程序的代码非常紧凑、清晰，易于理解。在程序运行过程中，当异常情况出现后，控制流是进行了一个本地跳转（进入到异常处理的代码模块，是在同一个函数的内部），这种情况其实也可以用 goto 语句来予以很好的实现，但是，显然 setjmp 与 longjmp 的方式，更为严谨一些，也更为友善。程序的执行流如图 17-1 所示。



setjmp 与 longjmp 相结合，实现程序的非本地的跳转

呵呵！这就是 goto 语句所不能实现的。也正因为如此，所以才说在 C 语言中，setjmp 与 longjmp 相结合的方式，它提供了真正意义上的异常处理机制。其实上一篇文章中的那个例程，已经演示了 longjmp 函数的非本地跳转的场景。这里为了更清晰演示本地跳转与非本地跳转，这两者之间的区别，我们在上面刚才的那个例程基础上，进行很小的一点改动，代码如下：

```
void Func1()
{
// 其它代码的执行
// 判断程序运行中，是否出现错误，如果有错误，则跳转！
```

```
if(1) longjmp(mark, 1);
}

void Func2()
{
// 其它代码的执行
// 判断程序远行中，是否出现错误，如果有错误，则跳转！
if(2) longjmp(mark, 2);
}

void Func3()
{
// 其它代码的执行
// 判断程序远行中，是否出现错误，如果有错误，则跳转！
if(-1) longjmp(mark, -1);
}

void main( void )
{
int jmpret;

jmpret = setjmp( mark );
if( jmpret == 0 )
{
// 其它代码的执行

// 下面的这些函数执行过程中，有可能出现异常
Func1();

Func2();

Func3();

// 其它代码的执行
}
else
{
// 错误处理模块
switch (jmpret)
{
case 1:
printf( "Error 1\n");
break;
case 2:
printf( "Error 2\n");
```

```
break;
case 3:
printf( "Error 3\n");
break;
default :
printf( "Unknown Error");
break;
}
exit(0);
}

return;
}
```

回顾一下，这与 C++中提供的异常处理模型是不是很相近。异常的传递是可以跨越一个或多个函数。这的确为 C 程序员提供了一种较完善的异常处理编程的机制或手段。

setjmp 和 longjmp 使用时，需要特别注意的事情

1、setjmp 与 longjmp 结合使用时，它们必须有严格的先后执行顺序，也即先调用 setjmp 函数，之后再调用 longjmp 函数，以恢复到先前被保存的“程序执行点”。否则，如果在 setjmp 调用之前，执行 longjmp 函数，将导致程序的执行流变的不可预测，很容易导致程序崩溃而退出。请看示例程序，代码如下：

```
class Test
{
public:
Test() {printf("构造对象\n");}
~Test() {printf("析构对象\n");}
}obj;

//注意，上面声明了一个全局变量 obj

void main( void )
{
int jmpret;

// 注意，这里将会导致程序崩溃，无条件退出
Func1();
while(1);

jmpret = setjmp( mark );
if( jmpret == 0 )
{
// 其它代码的执行
```

```
// 下面的这些函数执行过程中，有可能出现异常
Func1();

Func2();

Func3();

// 其它代码的执行
}
else
{
// 错误处理模块
switch (jmpret)
{
case 1:
printf( "Error 1\n");
break;
case 2:
printf( "Error 2\n");
break;
case 3:
printf( "Error 3\n");
break;
default :
printf( "Unknown Error");
break;
}
exit(0);
}

return;
}
```

上面的程序运行结果，如下：

```
构造对象
Press any key to continue
```

的确，上面程序崩溃了，由于在 Func1()函数内，调用了 longjmp，但此时程序还没有调用 setjmp 来保存一个程序执行点。因此，程序的执行流变的不可预测。这样导致的程序后果是非常严重的，例如说，上面的程序中，有一个对象被构造了，但程序崩溃退出时，它的析构函数并没有被系统来调用，得以清除一些必要的资源。所以这样的程序是非常危险的。（另外请注意，上面的程序是一个 C++程序，所以大家演示并测试这个例程时，把源文件的扩展名改为 xxx.cpp）。

2、除了要求先调用 setjmp 函数，之后再调用 longjmp 函数（也即 longjmp 必须有对应的 setjmp 函数）之外。另外，还有一个很重要的规则，那就是 longjmp 的调用是有一定域范围要求的。这未免太抽象了，还是先看一个示例，如下：

```
int Sub_Func()
{
int jmpret, be_modify;

be_modify = 0;

jmpret = setjmp( mark );
if( jmpret == 0 )
{
// 其它代码的执行
}
else
{
// 错误处理模块
switch (jmpret)
{
case 1:
printf( "Error 1\n");
break;
case 2:
printf( "Error 2\n");
break;
case 3:
printf( "Error 3\n");
break;
default :
printf( "Unknown Error");
break;
}

//注意这一语句，程序有条件地退出
if (be_modify==0) exit(0);
}

return jmpret;
}

void main( void )
{
Sub_Func();
```

```
// 注意，虽然 longjmp 的调用是在 setjmp 之后，但是它超出了 setjmp 的作用范围。
longjmp(mark, 1);
}
```

如果你运行或调试（单步跟踪）一下上面程序，发现它真是挺神奇的，居然 longjmp 执行时，程序还能够返回到 setjmp 的执行点，程序正常退出。但是这就说明了上面的这个例程的没有问题吗？我们对这个程序小改一下，如下：

```
int Sub_Func()
{
// 注意，这里改动了一点
int be_modify, jmpret;

be_modify = 0;

jmpret = setjmp( mark );
if( jmpret == 0 )
{
// 其它代码的执行
}
else
{
// 错误处理模块
switch (jmpret)
{
case 1:
printf( "Error 1\n");
break;
case 2:
printf( "Error 2\n");
break;
case 3:
printf( "Error 3\n");
break;
default :
printf( "Unknown Error");
break;
}

//注意这一语句，程序有条件地退出
if (be_modify==0) exit(0);
}

return jmpret;
}
```

```
void main( void )
{
Sub_Func();

// 注意，虽然 longjmp 的调用是在 setjmp 之后，但是它超出了 setjmp 的作用范围。
longjmp(mark, 1);
}
```

运行或调试（单步跟踪）上面的程序，发现它崩溃了，为什么？这就是因为，“在调用 setjmp 的函数返回之前，调用 longjmp，否则结果不可预料”（这在上一篇文章中已经提到过，MSDN 中做了特别的说明）。为什么这样做会导致不可预料？其实仔细想想，原因也很简单，那就是因为，当 setjmp 函数调用时，它保存的程序执行点环境，只应该在当前的函数作用域以内（或以后）才会有效。如果函数返回到了上层（或更上层）的函数环境中，那么 setjmp 保存的程序的环境也将会无效，因为堆栈中的数据此时将可能发生覆盖，所以当然会导致不可预料的执行后果。

3、不要假象寄存器类型的变量将总会保持不变。在调用 longjmp 之后，通过 setjmp 所返回的控制流中，例程中寄存器类型的变量将不会被恢复。（MSDN 中做了特别的说明，上一篇文章中，这也已经提到过）。寄存器类型的变量，是指为了提高程序的运行效率，变量不被保存在内存中，而是直接被保存在寄存器中。寄存器类型的变量一般都是临时变量，在 C 语言中，通过 register 定义，或直接嵌入汇编代码的程序。这种类型的变量一般很少采用，所以在使用 setjmp 和 longjmp 时，基本上不用考虑到这一点。

4、MSDN 中还做了特别的说明，“在 C++程序中，小心对 setjmp 和 longjmp 的使用，因为 setjmp 和 longjmp 并不能很好地支持 C++中面向对象的语义。因此在 C++程序中，使用 C++提供的异常处理机制将会更加安全。”虽然说 C++能非常好的兼容 C，但是这并非是 100%的完全兼容。例如，这里就是一个很好的例子，在 C++程序中，它不能很好地与 setjmp 和 longjmp 和平共处。在后面的一些文章中，有关专门讨论 C++如何兼容支持 C 语言中的异常处理机制时，会做详细深入的研究，这里暂且跳过。

总结

主人公阿愚现在对 setjmp 与 longjmp 已经是非常钦佩了，虽然它没有 C++中提供的异常处理模型那么好用，但是毕竟在 C 语言中，有这么好用的东东，已经是非常不错了。为了更上一层楼，使 setjmp 与 longjmp 更接近 C++中提供的异常处理模型（也即 try()catch()语法）。阿愚找到了不少非常有价值的资料。不要错过，继续到下一篇文章中去吧！让程序员朋友们“玩转 setjmp 与 longjmp”，Let’s go！

第 18 集 玩转 setjmp 与 longjmp

通过上两篇文章中，对 setjmp 与 longjmp 两个函数的深入研究与分析，相信大家已经和主人公阿愚一样，对 C 语言中提供的这种异常处理机制的使用方法了如指掌了。请不要骄傲和自满，让我们更上一层楼，彻底玩转 setjmp 与 longjmp 这两个函数。

不要忘记，前面我们得出过结论，C 语言中提供的这种异常处理机制，与 C++中的异常处理模型很相似。例如，可以定义出类似的 try block（受到监控的代码）；catch block（异常错误的处理模块）；以及可以随时抛出的异常（throw 语句）。所以说，我们可以通过一种非常有技巧的封装，来达到对 setjmp 和 longjmp

的使用方法（或者说语法规则），基本与 C++中的语法一致。很有诱惑吧！

首先展示阿愚封装的在 C 语言环境中异常处理框架

1、首先是接口的头文件，主要采用“宏”技术！代码如下：

```
/*********************************************
* author:  王胜祥  *
* email: <mantx@21cn.com> *
* date: 2005-03-07 *
* version: *
* filename: ceh.h *
*********************************************/

/*********************************************************************

This file is part of CEH(Exception Handling in C Language).

CEH is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or
(at your option) any later version.

CEH is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.

    注意：这个异常处理框架不支持线程安全，不能在多线程的程序环境下使用。
如果您想在多线程的程序中使用它，您可以自己试着来继续完善这个
框架模型。
*********************************************************************/

#include <stdio.h>
#include <signal.h>
#include <setjmp.h>
#include <stdlib.h>
#include <float.h>
#include <math.h>
#include <string.h>


//////////////////////////////////////////////////
/*  与异常有关的结构体定义  */
```

```
typedef struct _CEH_EXCEPTION {
int err_type; /* 异常类型 */
int err_code; /* 错误代码 */
char err_msg[80]; /* 错误信息 */
}CEH_EXCEPTION; /* 异常对象 */

typedef struct _CEH_ELEMENT {
jmp_buf exec_status;
CEH_EXCEPTION ex_info;

struct _CEH_ELEMENT* next;
} CEH_ELEMENT; /* 存储异常对象的链表元素 */
//////////////////////////////////////////////////////


//////////////////////////////////////////////////////
/* 内部接口定义，操纵维护链表数据结构 */
extern void CEH_push(CEH_ELEMENT* ceh_element);
extern CEH_ELEMENT* CEH_pop();
extern CEH_ELEMENT* CEH_top();
extern int CEH_isEmpty();
//////////////////////////////////////////////////////


/* 以下是外部接口的定义 */
//////////////////////////////////////////////////////
/* 抛出异常 */
extern void thrower(CEH_EXCEPTION* e);

/* 抛出异常 (throw)
a 表示 err_type
b 表示 err_code
c 表示 err_msg
*/
#define throw(a, b, c) \
{ \
CEH_EXCEPTION ex; \
memset(&ex, 0, sizeof(ex)); \
ex.err_type = a; \
ex.err_code = b; \
strncpy(ex.err_msg, c, sizeof(c)); \
thrower(&ex); \
}
```

```
/* 重新抛出原来的异常 (rethrow)*/
#define rethrow thrower(ceh_ex_info)
//////////////////////////////////////////////////////


//////////////////////////////////////////////////////
/* 定义 try block（受到监控的代码）*/
#define try \
{ \
int ___ceh_b_catch_found, ___ceh_b_occur_exception; \
CEH_ELEMENT ___ceh_element; \
CEH_EXCEPTION* ceh_ex_info; \
memset(&___ceh_element, 0, sizeof(___ceh_element)); \
CEH_push(&___ceh_element); \
ceh_ex_info = &___ceh_element.ex_info; \
___ceh_b_catch_found = 0; \
if (!(___ceh_b_occur_exception=setjmp(___ceh_element.exec_status))) \
{


/* 定义 catch block（异常错误的处理模块）
catch 表示捕获所有类型的异常
*/
#define catch \
} \
else \
{ \
CEH_pop(); \
___ceh_b_catch_found = 1;


/* end_try 表示前面定义的 try block 和 catch block 结束 */
#define end_try \
} \
{ \
/* 没有执行到任何的 catch 块中 */ \
if(!___ceh_b_catch_found) \
{ \
CEH_pop(); \
/* 出现了异常，但没有捕获到任何异常 */ \
if(___ceh_b_occur_exception) thrower(ceh_ex_info); \
} \
} \
}
```

```
/* 定义 catch block（异常错误的处理模块）
catch_part 表示捕获一定范围内的异常
*/
#define catch_part(i, j) \
} \
else if(ceh_ex_info->err_type>=i && ceh_ex_info->err_type<=j) \
{ \
CEH_pop(); \
___ceh_b_catch_found = 1;


/* 定义 catch block（异常错误的处理模块）
catch_one 表示只捕获一种类型的异常
*/
#define catch_one(i) \
} \
else if(ceh_ex_info->err_type==i) \
{ \
CEH_pop(); \
___ceh_b_catch_found = 1;
//////////////////////////////////////////////////////


//////////////////////////////////////////////////////
/* 其它可选的接口定义 */
extern void CEH_init();
//////////////////////////////////////////////////////
```

2、另外还有一个简单的实现文件，主要实现功能封装。代码如下：

```
/************************************************
* author: 王胜祥 *
* email: <mantx@21cn.com> *
* date: 2005-03-07 *
* version: *
* filename: ceh.c *
************************************************/


/*********************************************************************

This file is part of CEH(Exception Handling in C Language).
```

CEH is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or
(at your option) any later version.

CEH is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.

注意：这个异常处理框架不支持线程安全，不能在多线程的程序环境下使用。
如果您想在多线程的程序中使用它，您可以自己试着来继续完善这个
框架模型。
**********/

```
#include "ceh.h"

//////////////////////////////////////////////////
static CEH_ELEMENT* head = 0;

/*  把一个异常插入到链表头中  */
void CEH_push(CEH_ELEMENT* ceh_element)
{
if(head) ceh_element->next = head;
head = ceh_element;
}


/*  从链表头中，删除并返回一个异常  */
CEH_ELEMENT* CEH_pop()
{
CEH_ELEMENT* ret = 0;

ret = head;
head = head->next;

return ret;
}


/*  从链表头中，返回一个异常  */
CEH_ELEMENT* CEH_top()
{
return head;
}
```

```
/* 链表中是否有任何异常 */
int CEH_isEmpty()
{
return head==0;
}
////////////////////////////////////////////////////


////////////////////////////////////////////////////
/* 缺省的异常处理模块 */
static void CEH_uncaught_exception_handler(CEH_EXCEPTION *ceh_ex_info)
{
printf("捕获到一个未处理的异常，错误原因是：%s! err_type:%d err_code:%d\n",
ceh_ex_info->err_msg, ceh_ex_info->err_type, ceh_ex_info->err_code);
fprintf(stderr, "程序终止!\n");
fflush(stderr);
exit(EXIT_FAILURE);
}
////////////////////////////////////////////////////


////////////////////////////////////////////////////
/* 抛出异常 */
void thrower(CEH_EXCEPTION* e)
{
CEH_ELEMENT *se;

if (CEH_isEmpty()) CEH_uncaught_exception_handler(e);

se = CEH_top();
se->ex_info.err_type = e->err_type;
se->ex_info.err_code = e->err_code;
strncpy(se->ex_info.err_msg, e->err_msg, sizeof(se->ex_info.err_msg));

longjmp(se->exec_status, 1);
}
////////////////////////////////////////////////////


////////////////////////////////////////////////////
static void fphandler( int sig, int num )
{
_fpreset();
```

```
switch( num )
{
case _FPE_INVALID:
throw(-1, num, "Invalid number" );
case _FPE_OVERFLOW:
throw(-1, num, "Overflow" );
case _FPE_UNDERFLOW:
throw(-1, num, "Underflow" );
case _FPE_ZERODIVIDE:
throw(-1, num, "Divide by zero" );
default:
throw(-1, num, "Other floating point error" );
}
}

void CEH_init()
{
_control87( 0, _MCW_EM );

if( signal( SIGFPE, fphandler ) == SIG_ERR )
{
fprintf( stderr, "Couldn't set SIGFPE\n" );
abort();
}
}
//////////////////////////////////////////////////////
```

体验上面设计出的异常处理框架

请花点时间仔细揣摩一下上面设计出的异常处理框架。呵呵！程序员朋友们，大家是不是发现它与 C++提供的异常处理模型非常相似。例如，它提供的基本接口有 try、catch、以及 throw 等三条语句。还是先看个具体例子吧！以便验证一下这个 C 语言环境中异常处理框架是否真的比较好用。代码如下：

```
#include "ceh.h"

int main(void)
{
//定义 try block 块
try
{
int i,j;
printf("异常出现前\n\n");

// 抛出一个异常
// 其中第一个参数，表示异常类型；第二个参数表示错误代码
// 第三个参数表示错误信息
throw(9, 15, "出现某某异常");
```

```
printf("异常出现后\n\n");
}
//定义 catch block 块
catch
{
printf("catch 块，被执行到\n");
printf("捕获到一个异常，错误原因是：%s! err_type:%d err_code:%d\n",
ceh_ex_info->err_msg, ceh_ex_info->err_type, ceh_ex_info->err_code);
}
// 这里稍有不同，需要定义一个表示当前的 try block 结束语句
// 它主要是清除相应的资源
end_try
}
```

注意，上面的测试程序可是 C 语言环境下的程序（文件的扩展名请使用.c 结尾），虽然它看上去很像 C++程序。请编译运行一下，发现它是不是运行结果如下：

```
异常出现前

catch 块，被执行到

捕获到一个异常，错误原因是：出现某某异常! err_type:9 err_code:15
```

呵呵！程序的确是在按照我们预想的流程在执行。再次提醒，这可是 C 程序，但是它的异常处理却非常类似于 C++中的风格，要知道，做到这一点其实非常地不容易。当然，上面异常对象的传递只是在一个函数的内部，同样，它也适用于多个嵌套函数间的异常传递，还是用代码验证一下吧！在上面的代码基础下，小小修改一点，代码如下：

```
#include "ceh.h"

void test1()
{
throw(0, 20, "hahaha");
}

void test()
{
test1();
}

int main(void)
{
try
{
int i,j;
printf("异常出现前\n\n");
```

```
// 注意，这个函数的内部会抛出一个异常。
test();

throw(9, 15, "出现某某异常");

printf("异常出现后\n\n");
}
catch
{
printf("catch 块，被执行到\n");
printf("捕获到一个异常，错误原因是：%s! err_type:%d err_code:%d\n",
ceh_ex_info->err_msg, ceh_ex_info->err_type, ceh_ex_info->err_code);
}
end_try
}
```

同样，在上面程序中，test1()函数内抛出的异常，可以被上层 main()函数中的 catch block 中捕获到。运行结果就不再给出了，大家可以自己编译运行一把，看看运行结果。
另外这个异常处理框架，与 C++中的异常处理模型类似，它也支持 try catch 块的多层嵌套。很厉害吧！还是看演示代码吧！，如下：

```
#include "ceh.h"

int main(void)
{
// 外层的 try catch 块
try
{
// 内层的 try catch 块
try
{
throw(1, 15, "嵌套在 try 块中");
}
catch
{
printf("内层的 catch 块被执行\n");
printf("捕获到一个异常，错误原因是：%s! err_type:%d err_code:%d\n",
ceh_ex_info->err_msg, ceh_ex_info->err_type, ceh_ex_info->err_code);

printf("外层的 catch 块被执行\n");
}
end_try
```

```
throw(2, 30, "再抛一个异常");
}
catch
{
printf("外层的 catch 块被执行\n");
printf("捕获到一个异常，错误原因是：%s! err_type:%d err_code:%d\n",
ceh_ex_info->err_msg, ceh_ex_info->err_type, ceh_ex_info->err_code);
}
end_try
}
```

请编译运行一下，程序的运行结果如下：

内层的 catch 块被执行

捕获到一个异常，错误原因是：嵌套在 try 块中! err_type:1 err_code:15

外层的 catch 块被执行

捕获到一个异常，错误原因是：再抛一个异常! err_type:2 err_code:30

还有，这个异常处理框架也支持对异常的分类处理。这一点，也完全是模仿 C++中的异常处理模型。不过，由于 C 语言中，不支持函数名重载，所以语法上略有不同，还是看演示代码吧！，如下：

```
#include "ceh.h"

int main(void)
{
try
{
int i,j;
printf("异常出现前\n\n");

throw(9, 15, "出现某某异常");

printf("异常出现后\n\n");
}
// 这里表示捕获异常类型从 4 到 6 的异常
catch_part(4, 6)
{
printf("catch_part(4, 6)块，被执行到\n");
printf("捕获到一个异常，错误原因是：%s! err_type:%d err_code:%d\n",
ceh_ex_info->err_msg, ceh_ex_info->err_type, ceh_ex_info->err_code);
}
// 这里表示捕获异常类型从 9 到 10 的异常
catch_part(9, 10)
{
printf("catch_part(9, 10)块，被执行到\n");
printf("捕获到一个异常，错误原因是：%s! err_type:%d err_code:%d\n",
```

```
ceh_ex_info->err_msg, ceh_ex_info->err_type, ceh_ex_info->err_code);
}
// 这里表示只捕获异常类型为 1 的异常
catch_one(1)
{
printf("catch_one(1)块，被执行到\n");
printf("捕获到一个异常，错误原因是：%s! err_type:%d err_code:%d\n",
ceh_ex_info->err_msg, ceh_ex_info->err_type, ceh_ex_info->err_code);
}
// 这里表示捕获所有类型的异常
catch
{
printf("catch 块，被执行到\n");
printf("捕获到一个异常，错误原因是：%s! err_type:%d err_code:%d\n",
ceh_ex_info->err_msg, ceh_ex_info->err_type, ceh_ex_info->err_code);
}
end_try
}
```

请编译运行一下，程序的运行结果如下：

```
异常出现前

catch_part(9, 10)块，被执行到
```

捕获到一个异常，错误原因是：出现某某异常! err_type:9 err_code:15

与 C++中的异常处理模型相似，它这里的对异常的分类处理不仅支持一维线性的；同样，它也支持分层的，也即在当前的 try catch 块中找不到相应的 catch block，那么它将会到上一层的 try catch 块中继续寻找。演示代码如下：

```
#include "ceh.h"

int main(void)
{
try
{
try
{
throw(1, 15, "嵌套在 try 块中");
}
catch_part(4, 6)
{
printf("catch_part(4, 6)块，被执行到\n");
printf("捕获到一个异常，错误原因是：%s! err_type:%d err_code:%d\n",
ceh_ex_info->err_msg, ceh_ex_info->err_type, ceh_ex_info->err_code);
```

```
}
end_try

printf("这里将不会被执行到\n");
}
catch_part(2, 3)
{
printf("catch_part(2, 3)块，被执行到\n");
printf("捕获到一个异常，错误原因是：%s! err_type:%d err_code:%d\n",
ceh_ex_info->err_msg, ceh_ex_info->err_type, ceh_ex_info->err_code);
}
// 找到了对应的 catch block
catch_one(1)
{
printf("catch_one(1)块，被执行到\n");
printf("捕获到一个异常，错误原因是：%s! err_type:%d err_code:%d\n",
ceh_ex_info->err_msg, ceh_ex_info->err_type, ceh_ex_info->err_code);
}
catch
{
printf("catch 块，被执行到\n");
printf("捕获到一个异常，错误原因是：%s! err_type:%d err_code:%d\n",
ceh_ex_info->err_msg, ceh_ex_info->err_type, ceh_ex_info->err_code);
}
end_try

}
```

到目前为止，大家是不是已经觉得，这个主人公阿愚封装的在 C 语言环境中异常处理框架，已经与 C++中的异常处理模型 95%相似。无论是它的语法结构；还是所完成的功能；以及它使用上的灵活性等。下面我们来看一个各种情况综合的例子吧！代码如下：

```
#include "ceh.h"

void test1()
{
throw(0, 20, "hahaha");
}

void test()
{
test1();
}
```

```
int main(void)
{
try
{
test();
}
catch
{
printf("捕获到一个异常，错误原因是：%s! err_type:%d err_code:%d\n",
ceh_ex_info->err_msg, ceh_ex_info->err_type, ceh_ex_info->err_code);
}
end_try

try
{
try
{
throw(1, 15, "嵌套在 try 块中");
}
catch
{
printf("捕获到一个异常，错误原因是：%s! err_type:%d err_code:%d\n",
ceh_ex_info->err_msg, ceh_ex_info->err_type, ceh_ex_info->err_code);
}
end_try

throw(2, 30, "再抛一个异常");
}
catch
{
printf("捕获到一个异常，错误原因是：%s! err_type:%d err_code:%d\n",
ceh_ex_info->err_msg, ceh_ex_info->err_type, ceh_ex_info->err_code);

try
{
throw(0, 20, "嵌套在 catch 块中");
}
catch
{
printf("捕获到一个异常，错误原因是：%s! err_type:%d err_code:%d\n",
ceh_ex_info->err_msg, ceh_ex_info->err_type, ceh_ex_info->err_code);
}
end_try
}
```

```
end_try
}
```

请编译运行一下，程序的运行结果如下：

```
捕获到一个异常，错误原因是：hahaha! err_type:0 err_code:20
捕获到一个异常，错误原因是：嵌套在 try 块中! err_type:1 err_code:15
捕获到一个异常，错误原因是：再抛一个异常! err_type:2 err_code:30
捕获到一个异常，错误原因是：嵌套在 catch 块中! err_type:0 err_code:20
```

最后，为了体会到这个异常处理框架，更进一步与 C++中的异常处理模型相似。那就是它还支持异常的重新抛出，以及系统中能捕获并处理程序中没有 catch 到的异常。看代码吧！如下：

```
#include "ceh.h"

void test1()
{
throw(0, 20, "hahaha");
}

void test()
{
test1();
}

int main(void)
{
// 这里表示程序中将捕获浮点数计算异常
CEH_init();

try
{
try
{
try
{
try
{
double i,j;
j = 0;
// 这里出现浮点数计算异常
i = 1/j ;

test();

throw(9, 15, "出现某某异常");
}
end_try
```

```
}
catch_part(4, 6)
{
printf("catch_part(4, 6)块，被执行到\n");
printf("捕获到一个异常，错误原因是：%s! err_type:%d err_code:%d\n",
ceh_ex_info->err_msg, ceh_ex_info->err_type, ceh_ex_info->err_code);
}
catch_part(2, 3)
{
printf("catch_part(2, 3)块，被执行到\n");
printf("捕获到一个异常，错误原因是：%s! err_type:%d err_code:%d\n",
ceh_ex_info->err_msg, ceh_ex_info->err_type, ceh_ex_info->err_code);
}
// 捕获到上面的异常
catch
{
printf("内层的 catch 块，被执行到\n");
printf("捕获到一个异常，错误原因是：%s! err_type:%d err_code:%d\n",
ceh_ex_info->err_msg, ceh_ex_info->err_type, ceh_ex_info->err_code);

// 这里再次把上面的异常重新抛出
rethrow;

printf("这里将不会被执行到\n");
}
end_try
}
catch_part(7, 9)
{
printf("catch_part(7, 9)块，被执行到\n");
printf("捕获到一个异常，错误原因是：%s! err_type:%d err_code:%d\n",
ceh_ex_info->err_msg, ceh_ex_info->err_type, ceh_ex_info->err_code);

throw(2, 15, "出现某某异常");
}
// 再次捕获到上面的异常
catch
{
printf("外层的 catch 块，被执行到\n");
printf("捕获到一个异常，错误原因是：%s! err_type:%d err_code:%d\n",
ceh_ex_info->err_msg, ceh_ex_info->err_type, ceh_ex_info->err_code);

// 最后又抛出了一个异常，
// 但是这个异常没有对应的 catch block 处理，所以系统中处理了
throw(2, 15, "出现某某异常");
```

```
}
end_try
}
```

请编译运行一下，程序的运行结果如下：

```
内层的 catch 块，被执行到
捕获到一个异常，错误原因是：Divide by zero! err_type:-1 err_code:131
外层的 catch 块，被执行到
捕获到一个异常，错误原因是：Divide by zero! err_type:-1 err_code:131
捕获到一个未处理的异常，错误原因是：出现某某异常! err_type:2 err_code:15
程序终止!
```

总结

主人公阿愚封装的这个在 C 语言环境中的异常处理框架，是完全建立在 setjmp 与 longjmp 的机制之上的。它的目的使我们对 setjmp 和 longjmp 的使用起来更方便和友善；同时，也更加深我们对 setjmp 和 longjmp 的了解；另外，还有一个重要的目的，就是给我们大家自己一个机会，来思考 C++中异常处理模型的实现，当然后者会更复杂，但是这里将是一个预习课，它对后面真正阐述 C++中异常处理模型的实现将大有帮助。

另外，这个异常处理框架不支持线程安全，不能在多线程的程序环境下使用。如果您想在多线程的程序中使用它，您可以自己试着来继续完善这个框架模型。当然，您也可以给主人公阿愚发 email（mantx@21cn.com），来共同探讨合理的解决方案。

下一节将继续讨论 setjmp 与 longjmp 在 C++中的应用情况，这很关键呦！是许多程序员朋友对 setjmp 与 longjmp 认识上的盲区。不要错过，继续到下一篇文章中去吧！

第 19 集 setjmp 与 longjmp 机制，很难与 C++和睦相处

在《第 16 集 C 语言中一种更优雅的异常处理机制》中，就已经提到过，“setjmp 和 longjmp 并不能很好地支持 C++中面向对象的语义。因此在 C++程序中，请使用 C++提供的异常处理机制”。它在 MSDN 中的原文如下：

setjmp and longjmp do not support C++ object semantics. In C++ programs, use the C++ exception-handling mechanism.

这究竟是为什么？大家知道，C++语言中是基本兼容 C 语言中的语义的。但是为什么，在 C++程序中，唯独却不能使用 C 语言中的异常处理机制？虽然大家都知道，在 C++程序中，实际上是没有必要这么做，因为 C++语言中提供了更完善的异常处理模型。但是，在许多种特殊情况下，C++语言来开发的应用程序系统中，可能采用了 C 语言中的异常处理机制。例如说，一个应用程序由于采用 C++开发，它里面使用了 C++提供的异常处理机制；但是它可能需要调用其它已经由 C 语言实现的程序库，而恰恰在这个被复用的

程序库中，它也采用了异常处理机制。因此对于整个应用程序系统而言，它不可避免地出现了这种矛盾的局面。并且这种情况是非常多见的，也可能是非常危险的。因为毕竟，“setjmp and longjmp do not support C++ object semantics”。所以，我们非常有必要来了解它究竟为什么不会兼容。

在本篇文章中，主人公阿愚将和程序员朋友们一起，深入探讨 setjmp 与 longjmp 机制，为什么它很难与 C++和睦相处？另外还有，如果 C++语言来开发的应用程序系统中，不得不同时使用这两种异常处理模型时，又如何来尽可能保证程序系统的安全？

C++语言中使用 setjmp 与 longjmp

闲话少说，还是看例程先吧！代码如下：

```
// 注意，这是一个 C++程序。文件扩展名应该为.cpp 或其它等。例如，c++setjmp.cpp
#include <stdio.h>
#include <setjmp.h>
#include <stdlib.h>

//定义一个测试类
class MyTest
{
public:
MyTest ()
{
printf("构造一个 MyTest 类型的对象\n");
}

virtual ~ MyTest ()
{
printf("析构销毁一个 MyTest 类型的对象\n");
}
};

jmp_buf mark;

void test1()
{
// 注意，这里抛出异常
longjmp(mark, 1);
}

void test()
{
test1();
}
```

```
void main( void )
{
int jmpret;

// 设置好异常出现时，程序的回溯点
jmpret = setjmp( mark );
if( jmpret == 0 )
{
// 建立一个对像
MyTest myobj;

test();
}
else
{
printf("捕获到一个异常\n");
}
}
```

请编译运行一下，程序的运行结果如下：

构造一个 MyTest 类型的对象

析构销毁一个 MyTest 类型的对象

捕获到一个异常

上面的程序运行结果，那么到底是不是合理的呢？阿愚感到有些纳闷，这结果肯定是合乎情理的呀！从这个例程来看，setjmp 和 longjmp 并不能破坏 C++中面向对象的语义，它们之间融洽得很好呀！那么为什么会说，“setjmp and longjmp do not support C++ object semantics”。请不要着急，沉住气！继续看看其它的情况，代码如下：

```
#include <stdio.h>
#include <setjmp.h>
#include <stdlib.h>

class MyTest
{
public:
MyTest ()
{
printf("构造一个 MyTest 类型的对象\n");
}

virtual ~ MyTest ()
{
printf("析构销毁一个 MyTest 类型的对象\n");
```

```
}
};

jmp_buf mark;

void test1()
{
// 注意，这里在上面程序的基础上，进行了一点小小的改动
// 把对像的构造挪到这里来
MyTest myobj;

longjmp(mark, 1);
}

void test()
{
test1();
}

void main( void )
{
int jmpret;

jmpret = setjmp( mark );
if( jmpret == 0 )
{
test();
}
else
{
printf("捕获到一个异常\n");
}
}
```

同样也编译运行一下，看程序的运行结果，如下：

```
构造一个 MyTest 类型的对象
捕获到一个异常
```

呵呵！那个对像的构造建立过程只不过是被稍稍挪了一下位置，而且先后顺序还没有改变，都是在if(jmpret == 0)语句之后，longjmp(mark, 1)之前。可为什么程序运行的结果却不同了呢？显然，从这个例程的运行结果来看，setjmp 和 longjmp 已经破坏 C++中面向对象的语义，因为那个 MyTest 类型的对像只被构造了，但是它却没有被析构销毁！这与大家所知道的面向对象的理论是相违背的。程序员朋友们，不要小看这个问题，有时这种错误将给应用程序系统带来极大的灾难（不仅仅是内存资源得不到释放，更糟糕的是可能引发系统死锁或程序崩溃）。

由此可以看出，setjmp 与 longjmp 机制，有时的确是不能够与 C++和睦相处。那么，为什么第 1 个例子中会安然无恙呢？它有什么规律吗？请继续看另外的一个例子，代码如下：

```
#include <stdio.h>
#include <setjmp.h>
#include <stdlib.h>

class MyTest
{
public:
MyTest ()
{
printf("构造一个 MyTest 类型的对象\n");
}

virtual ~ MyTest ()
{
printf("析构销毁一个 MyTest 类型的对象\n");
}
};

jmp_buf mark;

void test1()
{
longjmp(mark, 1);
}

void test()
{
/ // 注意，现在又把它挪到了这里
MyTest myobj;

test1();
}

void main( void )
{
int jmpret;

jmpret = setjmp( mark );
if( jmpret == 0 )
{
test();
}
```

```
else
{
printf("捕获到一个异常\n");
}
}
```

请编译运行一下，程序的运行结果如下：

构造一个 MyTest 类型的对象

析构销毁一个 MyTest 类型的对象

捕获到一个异常

呵呵！这里的运行结果也是对的。所以主人公阿愚总结出了一条结论，那就是，“在 longjmp 被调用执行的那个函数作用域中，绝对不能够存在局部变量形式的对象（也即在堆栈中的对象，longjmp 执行时还没有被析构销毁），否则这些对象将得不到析构的机会”。切忌切忌！又例如下面的例子同样也会有问题，代码如下：

```
#include <stdio.h>
#include <setjmp.h>
#include <stdlib.h>

class MyTest
{
public:
MyTest ()
{
printf("构造一个 MyTest 类型的对象\n");
}

virtual ~ MyTest ()
{
printf("析构销毁一个 MyTest 类型的对象\n");
}
};

jmp_buf mark;

void main( void )
{
int jmpret;

jmpret = setjmp( mark );
if( jmpret == 0 )
{
MyTest myobj;
longjmp(mark, 1);
```

```
}
else
{
printf("捕获到一个异常\n");
}
}
```

总结

虽然说，setjmp 与 longjmp 机制，很难与 C++和睦相处。但是它并非那么可怕，只要掌握了它的规律，整个应用程序系统的安全性仍掌握在你手心。而且，本文开头提到的例子（采用 C++开发的应用程序，它里面调用了 C 语言实现的其它程序库，并且库代码中使用了 setjmp 和 longjmp 机制），它并没有任何的问题。因为这个 C 库中，其中调用 longjmp 的函数域内，决不会有对象的构造定义。

现在为止，对 setjmp 和 longjmp 的讨论暂告一个段落。在“爱的秘密”篇中，会进一步阐述它的实现。下一篇文章将讨论在 C++中，如何兼容并支持 C 语言中提供的其它方式的异常处理机制（例如，C++中对 goto 语句的支持）。哈哈！ goto next！

第 20 集 C++中如何兼容并支持 C 语言中提供的异常处理机制

C 语言中提供的异常处理机制并不是十分严谨，而且比较杂，功能也非常有限。最常见的除了 setjmp 与 longjmp 之外，goto 语句在实际编程中也使用很广泛（虽然不建议使用它）。大家现在也都知道，在 C++语言中，它并不完全兼容并支持 setjmp 与 longjmp 函数的使用。但是 C++语言对待 goto 语句又将如何呢？

C++语言中如何处理 goto 语句

大家知道，在 C 语言程序中，goto 语句被编译成机器指令后，它只对应一条 jmp 指令。但是在 C++语言程序中，goto 语句也会这么简单吗？no！为什么这么说呢？因为 C++语言是面向对象的语言，如果 goto 语句只会简单地对应一条 jmp 指令，那么在许多情况下，这会破坏面向对象的一些特性。例如下面的示例程序，代码如下：

```
#include <stdio.h>
#include <setjmp.h>
#include <stdlib.h>

class MyTest
{
public:
MyTest ()
{
```

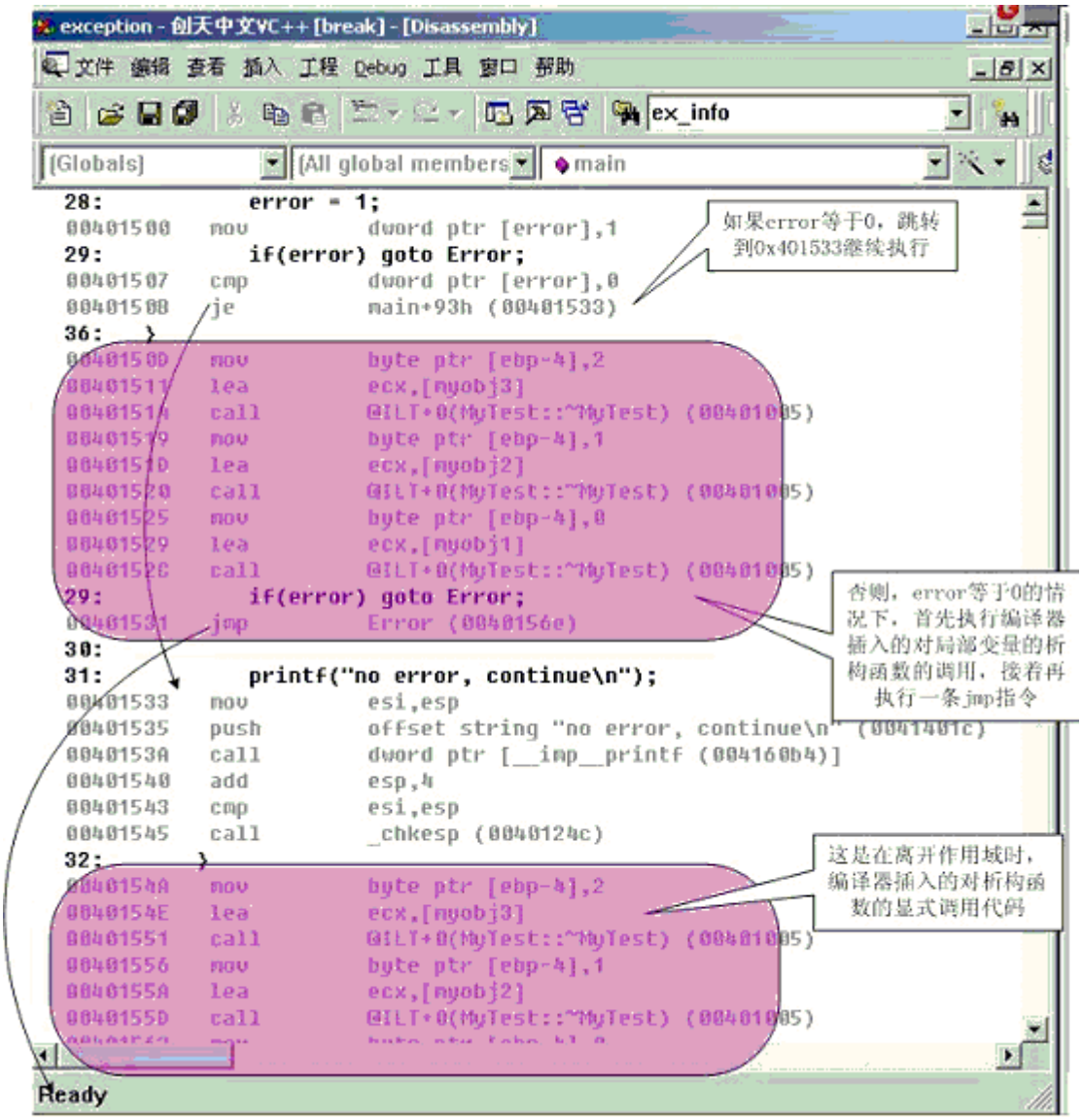
```
printf("构造一个 MyTest 类型的对象\n");
}

virtual ~ MyTest ()
{
printf("析构销毁一个 MyTest 类型的对象\n");
}
};

void main( void )
{
MyTest myobj0;
{
int error;
MyTest myobj1;
MyTest myobj2;
MyTest myobj3;

error = 1;

// 注意下面这条 goto 语句，如果它只是一条简单的 jmp 指令，
// 那么 myobj1,myobj2,myobj3 对象将如何被析构销毁呢？
if(error) goto Error;

printf("no error, continue\n");
}

Error:
return;
}
```

请编译运行一下，程序的运行结果如下：

构造一个 MyTest 类型的对象
构造一个 MyTest 类型的对象
构造一个 MyTest 类型的对象
构造一个 MyTest 类型的对象
析构销毁一个 MyTest 类型的对象
析构销毁一个 MyTest 类型的对象
析构销毁一个 MyTest 类型的对象
析构销毁一个 MyTest 类型的对象

呵呵！从程序的运行结果来看，显然，它符合面向对象的规则定义，“一个对象被构造了，就必然会有析构的过程”。所以说，在 C++语言中，它是能够很好兼容并支持 goto 语句的语义，这也与 C++是 C 语言的继承、扩充、完善的版本等承诺是相一致的。但是同时我们也知道，C++中对象的析构，是由编译器来予以支持的，那就是当编译器在编译程序时，如果局部对象在离开它的作用域时，编译器会显式地插入一

些调用对象析构函数的代码，来销毁这些即将无效掉的局部对象。但是程序中如果遭遇到 goto 语句时，显然，编译器也需要插入对局部对象的析构函数的显式调用。请在上面的程序中 goto 语句那一行，按 F9 设置一个断点；然后 F5，调试程序；接着 Alt+8 切换到汇编代码的显示状态下，注意查看 if(error) goto Error 语句对应的汇编程序。截图如下：

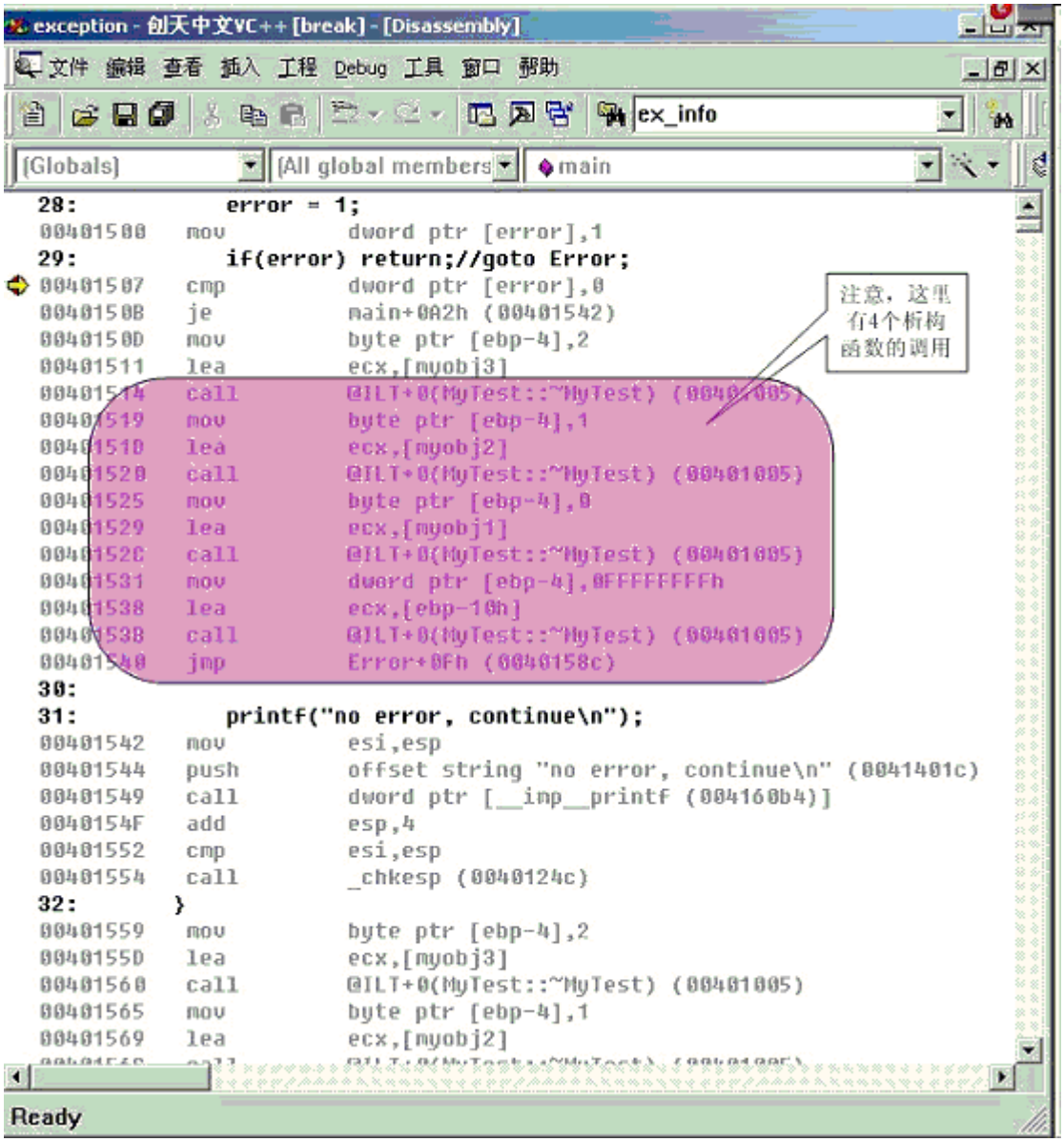


呵呵！从上图可以很明显的看出，编译器在处理 goto 语句时，需要进行更多的工作，它必须要插入所有当前局部对象的析构函数的显式调用代码，然后才能真正执行 jmp 指令。其它对于许多其它类似的语句，编译器的处理也是类似，例如对于 return 语句的处理也是如此，把上面的那个程序小小改动一点，代码如下：

```
#include <stdio.h>
#include <setjmp.h>
#include <stdlib.h>
```

```
class MyTest
{
public:
MyTest ()
{
printf("构造一个 MyTest 类型的对象\n");
}

virtual ~ MyTest ()
{
printf("析构销毁一个 MyTest 类型的对象\n");
}
};

void main( void )
{
MyTest myobj0;
{
int error;
MyTest myobj1;
MyTest myobj2;
MyTest myobj3;

error = 1;

// 用 return 语句直接返回
if(error) return; //goto Error;

printf("no error, continue\n");
}

Error:
return;
}
```

同样也调试程序，接着 Alt+8 切换到汇编代码的显示状态下，注意查看 if(error) return 语句对应的汇编程序。截图如下：



总结

虽然说，在 C++语言中，它能够很好兼容并支持 goto 语句的语义（也包括其它一些与异常处理相关的语句）。但是，主人公阿愚强烈建议程序员朋友在编写 C++程序代码时，不要轻易使用 goto 语句，因为与 C 程序中的 goto 语句相比，它不仅破坏了结构化的程序设计，破坏了程序代码的整体美感，而且它更导致了 C++程序模块的臃肿（编译器因此而导致需要插入了太多重复性代码）。

到目前为止，主人公阿愚引领大家，对 C++和 C 语言中的异常处理机制，进行了广泛而深入的探讨，阿愚深感收获甚多，当然也有可能认识上的不少错误，欢迎朋友们指出并共同讨论。

从下一篇文章中，开始对操作系统提供的异常处理机制进行一个全面而系统的介绍和较深入的研究。尤其是 Windows 平台提供的结构化异常处理机制，也即大名鼎鼎的 SEH（Structured Exception Handling）。

熟悉 SHE 的朋友们，请 GO！因为阿愚期待与大家一同学习探讨；当然，哪些不太熟悉 SHE 的朋友们，也请 GO！因为阿愚在这里，一定把最深切的学习体会和经验总结奉献给大家！继续吧！

第 21 集 Windows 系列操作系统平台中所提供的异常处理机制

大家现在知道，在 C++中有完善的异常处理机制，同样在 C 语言中也有很不错的异常处理机制来支持，另外在其它许多现代编程语言中，也都有各自的异常处理编程机制，如 Ada 语言等。那么为什么现在此处还在讨论，操作系统平台中所提供的异常处理机制呢？这是因为在许多系统中，编程语言所提供的异常处理机制的实现，都是建立在操作系统中所提供的异常处理机制之上，如 Windows 平台上的 VC 编译器所实现的 C++异常处理模型，它就是建立在 SEH 机制之上的，在“爱的秘密”篇中，探讨 VC 中异常处理模型实现的时候，会进行更深入的研究和分析。另外，当然也有其它的系统，如 Linux 操作系统上的 gcc 就没有采用到操作系统中所提供的异常处理机制。但是这样会有一个很大的缺点，那就是对于应用程序的开发者而言，它不能够很好在自己的应用程序中，来有效控制操作系统中所出现的一些意外的系统异常，例如程序执行过程中可能出现的段错误，被 0 除等计算异常，以及其它许多不同类型的系统异常等。所以 Linux 操作系统上的 gcc 编译的程序中，它只能捕获程序中，曾经被自己显式地 throw 出来的异常，而对于系统异常，catch block 是毫无办法的。

因此，操作系统平台中所提供的异常处理机制是非常有必要的。而且，异常处理机制的实现也是操作系统设计时的一个重要课题。通常，类 Unix 操作系统所提供的异常处理机制是大家非常熟悉的，那就是操作系统中的信号量处理（Signal Handling），好像这也应该是 Posix 标准所定义异常处理接口，因此 Window 系列操作系统平台也支持这种机制，“信号量处理”编程机制也会在后面的章节中进一步深入讨论。而现在（以及在接下来的几篇文章中），将全面阐述 Window 系列操作系统平台提供的另外一种更完善的异常处理机制，那就是大名鼎鼎的结构化异常处理（Structured Exception Handling，SEH）的编程方法。

SEH 设计的主要动机

下面是出自《Window 核心编程》中一小段内容的引用：

“微软在 Windows 中引入 SEH 的主要动机是为了便于操作系统本身的开发。操作系统的开发人员使用 SEH，使得系统更加强壮。我们也可以使用 SEH，使我们的自己的程序更加强壮。使用 SEH 所造成的负担主要由编译程序来承担，而不是由操作系统承担。当异常块（exception block）出现时，编译程序要生成特殊的代码。编译程序必须产生一些表（ t a b l e）来支持处理 SEH 的数据结构。编译程序还必须提供回调（ c a l l b a c k）函数，操作系统可以调用这些函数，保证异常块被处理。编译程序还要负责准备栈结构和其他内部信息，供操作系统使用和参考。在编译程序中增加 SEH 支持不是一件容易的事。不同的编译程序厂商会以不同的方式实现 SEH，这一点并不让人感到奇怪。幸亏我们可以不必考虑编译程序的实现细节，而只使用编译程序的 SEH 功能。”

的确，SEH 设计的主要动机就是为了便于操作系统本身的开发。为什么这么说呢？这是因为操作系统是一个非常庞大的系统，而且它还是处于计算机整个系统中，非常底层的系统软件，所以要求“操作系统”这个关键的系统软件必须要非常强壮，可靠性非常高。当然提升操作系统软件的可靠性有许多有效的方法，例如严谨的设计，全面而科学的测试。但是俗话说得好，“智者千虑，必有一失”。因此在编程代码中，有

一个非常有效的异常处理机制，将大大提高软件系统的可靠性。说道这里，也许很多朋友们说：“阿愚呀！你有没有搞错，无论是 C++还是 C 语言中，不都有很好的异常处理机制吗？为什么还需要另外再设计一种呢？”。的确没错，但是请注意，操作系统由于效率的考虑，它往往不会考虑采用 C++语言来编写，它大部分模块都是采用 C 语言来编码的，另外还包括一小部分的汇编代码。所以，这样操作系统中最多只能用 goto 语句，以及 setjmp 和 longjmp 等机制。但 Window 的设计者认为这远远不够他们的需要，离他们的要求相差甚远。而且，尤其是在操作系统中，最大的软件模块就是设备的驱动程序了，而且设备的驱动程序模块中，其中的又有绝大部分是由第 3 方（硬件提供商）来开发的。这些第 3 方开发的驱动程序模块，与操作系统核心关联紧密，它将严重影响到整个操作系统的可靠性和稳定性。所以，如何来促使第 3 方开发的驱动程序模块变得更加可靠呢？它至少不会影响到操作系统内核的正常工作，或者说甚至导致操作系统内核的崩溃。客观地说，这的确很难做得到，因为自己的命运掌握在别人手里了。但是 Window 的设计者想出了一个很不错的方法，那就是为第 3 方的驱动程序开发人员提供 SEH 机制，来最大程度上提升第 3 方开发出的驱动程序的可靠性和稳定性。相信有过在 Windows 平台上开发过驱动程序的朋友们对这一点深有感触，而且有关驱动程序开发的大多数书籍中，都强烈建议你在编程中使用 SEH 机制。

现在来谈谈为什么说，C 语言中的 setjmp 和 longjmp 机制没有 SEH 机制好呢？呵呵！当然在不深入了解 SEH 机制之前，来讨论并很好地解释清楚这个问题也许是比较困难的。但是主人公阿愚却认为，这里讨论这个问题是最恰当不过的了。因为这样可以很好地沿着当年的 SEH 的设计者们的足迹，来透析他们的设计思想。至少我们现在可以充分地分析 setjmp 和 longjmp 机制的不足之处。例如说，在实际的编程中（尤其是系统方面的编程），是不是经常遇到这种状况，示例代码如下：

```
#include <stdio.h>
#include <setjmp.h>
#include <stdlib.h>


jmp_buf mark;

void test1()
{
char* p1, *p2, *p3, *p4;
p1 = malloc(10);
if (!p1) longjmp(mark, 1);

p2 = malloc(10);
if (!p2)
{
// 这里虽然可以释放资源，
// 但是程序员很容易忘记，也容易出错
free(p1);
longjmp(mark, 1);
}

p3 = malloc(10);
if (!p3)
{
```

```
// 这里虽然可以释放资源
// 但是程序员很容易忘记，也容易出错
free(p1);
free(p2);
longjmp(mark, 1);
}

p4 = malloc(10);
if (!p4)
{
// 这里虽然可以释放资源
// 但是程序员很容易忘记，也容易出错
free(p1);
free(p2);
free(p3);
longjmp(mark, 1);
}

// do other job

free(p1);
free(p2);
free(p3);
free(p4);
}

void test()
{
char* p;
p = malloc(10);

// do other job

test1();

// do other job

// 这里的资源可能得不到释放
free(p);
}

void main( void )
{
int jmpret;
```

```
jmpret = setjmp( mark );
if( jmpret == 0 )
{
char* p;
p = malloc(10);

// do other job

test1();

// do other job
//  这里的资源可能得不到释放
free(p);
}
else
{
printf("捕获到一个异常\n");
}
}
```

上面的程序很容易导致内存资源泄漏，而且程序的代码看上去也比较别扭。程序员编写代码时，总要非常地小心翼翼地释放相关的资源。有可能一个稍微的处理不当，就可能导致资源泄漏，甚至更严重的是引发系统的死锁，因为在编写驱动程序时，毕竟与一般的应用程序有很大的不同，会经常使用到进程间的一些同步和互斥控制等。

怎么办？如果有一种机制，在程序中出现异常时，能够给程序员一个恰当的机会，来释放这些系统资源，那该多好呀！其实这种需求有点相当于 C++中的析构函数，因为它不管何种情况下，只要对象离开它生存的作用域时，它总会被得以执行。但是在 C 语言中的 setjmp 和 longjmp 机制，却显然做不到这一点。SEH 的设计者们考虑到了这一点，因此在 SEH 机制中，它提供了此项功能给程序开发人员。这其实也是 SEH 中最强大，也最有特色的地方。另外，SEH 还提供了程序出现异常后，有效地被得以恢复，程序继续执行的机制，这也是其它异常处理模型没有的（虽然一般情况下，没有必要这样做；但是对于系统模块，有时这是非常有必要的）。

SEH 究竟何物？

首先还是引用 MSDN 中关于 SEH 的有关论述，如下：

Windows 95 and Windows NT support a robust approach to handling exceptions, called structured exception handling, which involves cooperation of the operating system but also has direct support in the programming language.

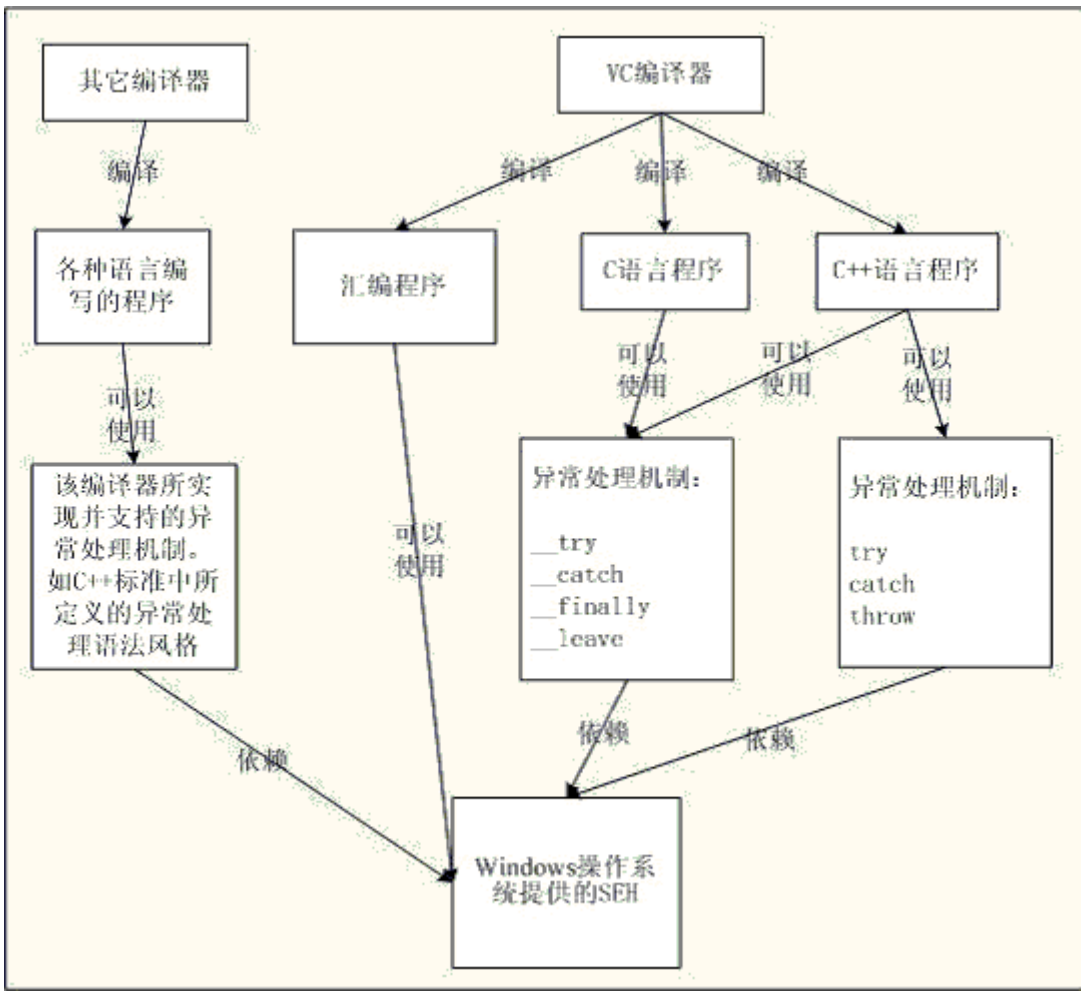
An “exception” is an event that is unexpected or disrupts the ability of the process to proceed normally. Exceptions can be detected by both hardware and software. Hardware exceptions include dividing by zero and overflow of a numeric type. Software exceptions include those you detect and signal to the system by calling the RaiseException function, and special situations detected by Windows 95 and Windows NT.

You can write more reliable code with structured exception handling. You can ensure that resources, such as

memory blocks and files, are properly closed in the event of unexpected termination. You can also handle specific problems, such as insufficient memory, with concise, structured code that doesn't rely on goto statements or elaborate testing of return codes.
Note These articles describe structured exception handling for the C programming language. Although structured exception handling can also be used with C++, the new C++ exception handling method should be used for C++ programs. See Using Structured Exception Handling with C++ for information on special considerations.

我们虽然都知道，SEH 是 Window 系列操作系统平台提供的一种非常完善的异常处理机制。但这毕竟有些过于抽象了点，对于程序员而言，它应该有一套类似于像 C++中那样的 try，catch，throw 等几个关键字组成的完整的异常处理模型。是的，这的确没错，SEH 也有类似的语法，那就是它由如下几个关键字组成：
__try
__except
__finally
__leave

呵呵！这是不是很多在 Windows 平台上做开发的程序员朋友们都用过。很熟悉吧！但是这里其实有一个认识上的误区，大多数程序员，包括很多计算机书籍中，都把 SEH 机制与__try，__except，__finally，__leave 异常模型等同起来。这种提法是不对的，至少是不准确的。因为 SHE 狭义上讲，只能算是 Window 系列操作系统所提供的一种异常处理机制；而无论是__try，__except，__finally，__leave 异常模型，还是 try，catch，throw 异常模型，它们都是 VC 所实现并提供给开发者的。其中__try，__except，__finally，__leave 异常模型主要是提供给 C 程序员使用；而 try，catch，throw 异常模型，它则是提供给 C++程序员使用，而且它也遵循 C++标准中异常模型的定义。这多种异常机制之间的关系如下图所示：



总结

为了更进一步认识 SEH 机制，认识 SEH 与__try，__except，__finally，__leave 异常模型机制的区别。下一篇文章中，主人公阿愚还是拿出一个具体的例子程序来与大家一起分享。注意，它那个例子中，它既没有使用到__try，__except，__finally，__leave 异常模型；也没有利用到 try，catch，throw 异常模型。它直接使用到 Windows 操作系统所提供的 SEH 机制，并完成一个简单的封装。继续吧！GO!

第 22 集 更进一步认识 SHE

上一篇文章阿愚对结构化异常处理（Structured Exception Handling，SEH）有了初步的认识，而且也知道了 SEH 是__try，__except，__finally，__leave 异常模型机制和 try，catch，throw 方式的 C++异常模型的奠基石。

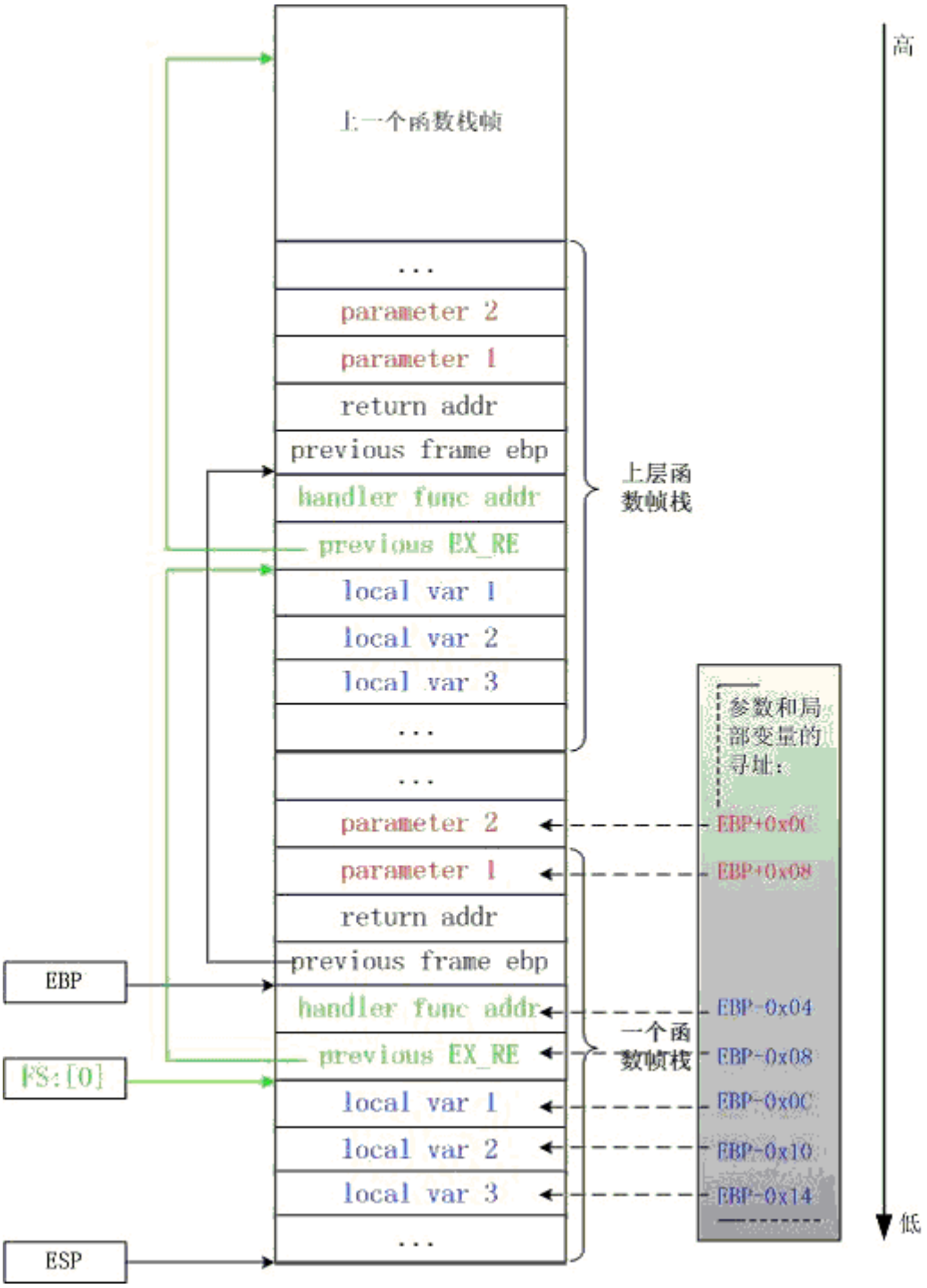
为了更进一步认识 SEH 机制，更深刻的理解 SEH 与__try，__except，__finally，__leave 异常模型机制的区别。本篇文章特别对狭义上的 SEH 进行一些极为细致的讲解。

SEH 设计思路

SEH 机制大致被设计成这样一种工作流程：用户应用程序对一些可能出现异常错误的代码模块，创建

一个相对应的监控函数（也即回调函数），并向操作系统注册；这样应用程序在执行过程中，如果什么异常也没出现的话，那么程序将按正常的执行流顺序来完成相对应的工作任务；否则，如果受监控的代码模块中，在运行时出现一个预知或未预知的异常错误，那么操作系统将捕获住这个异常，并暂停程序的执行过程（实际上是出现异常的工作线程被暂停了），然后，操作系统也收集一些有关异常的信息（例如，异常出现的地点，线程的工作环境，异常的错误种类，以及其它一些特殊的字段等），接着，操作系统根据先前应用程序注册的监控性质的回调函数，来查询当前模块所对应的监控函数，找到之后，便立即来回调它，并且传递一些必要的异常的信息作为监控函数的参数。此时，用户注册的监控函数便可以根据异常错误的种类和严重程度来进行分别处理，例如终止程序，或者试图恢复错误后，再使程序正常运行。

细心的朋友们现在可能想到，用户应用程序如何来向操作系统注册一系列的监控函数呢？其实 SEH 设计的巧妙之处就在与此，它这里有两个关键之处。其一，就是每个线程为一个完全独立的注册主体，线程间互不干扰，也即每个线程所注册的所有监控回调函数会连成一个链表，并且链表头被保存在与线程本地存储数据相关的区域（也即 FS 数据段区域，这是 Windows 操作系统的设计范畴，FS 段中的数据一般都是一些线程相关的本地数据信息，例如 FS:[0]就是保存监控回调函数数据结构体的链表头。有关线程相关的本地数据，这里不再详细赘述，感兴趣的朋友可以参考其它更为详细地资料）；其二，那就是每个存储监控回调函数指针的数据结构体，实际上它们一般并不是被存储在堆（Heap）中，而是被存储在栈（Stack）中。大家还记得在《第 9 集 C++的异常对象如何传送》中，有关“函数调用栈”的布局，呵呵！那只是比较理想化的栈布局，实际上，无论是 C++还是 C 程序中，如果函数模块中，存在异常处理机制的情况下，那么栈布局都会略有些变化，会变得更为复杂一些，因为在栈中，需要插入一些“存储监控回调函数指针的数据结构体”数据信息。例如典型的带 SEH 机制的栈布局如下图所示。



上图中，注意其中绿线部分所连成链表数据结构，这就是用户应用程序向操作系统注册的一系列的监控函数。如果某个函数中声明了异常处理机制，那么在函数帧栈中将分配一个数据结构体（EXCEPTION_REGISTRATION），这个数据结构体有点类似与局部变量的性质，它包含两个字段，其中一个是指向监控函数的指针（handler function address）；另一个就是链表指针（previous EXCEPTION_REGISTRATION）。特别需要注意的是，并不是每个函数帧栈中都有EXCEPTION_REGISTRATION 数据结构。另外链表头指针被保存到 FS:[0]中，这样无论是操作系统，还是应用程序都能够很好操纵这个链表数据体变量。

EXCEPTION_REGISTRATION 的定义如下：

```
typedef struct _EXCEPTION_REGISTRATION
{
struct _EXCEPTION_REGISTRATION *prev;
DWORD handler;
}EXCEPTION_REGISTRATION, *PEXCEPTION_REGISTRATION;
```

通过一个简单例子，来理解 SEH 机制

也许上面的论述过于抽象化和理论化了，还是看一个简单的例子吧！这样也很容易来理解 SEH 的工作机制原来是那么的简单。示例代码如下：

```
//seh.c
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>


typedef struct _EXCEPTION_REGISTRATION
{
struct _EXCEPTION_REGISTRATION *prev;
DWORD handler;
}EXCEPTION_REGISTRATION, *PEXCEPTION_REGISTRATION;


// 异常监控函数
EXCEPTION_DISPOSITION myHandler(
EXCEPTION_RECORD *ExcRecord,
void * EstablisherFrame,
CONTEXT *ContextRecord,
void * DispatcherContext)
{
printf("进入到异常处理模块中\n");
printf("不进一步处理异常，程序直接终止退出\n");

abort();
return ExceptionContinueExecution;
}


int main()
{
DWORD prev;
EXCEPTION_REGISTRATION reg, *preg;
```

```
// 建立异常结构帧（EXCEPTION_REGISTRATION）
reg.handler = (DWORD)myHandler;

// 把异常结构帧插入到链表中
__asm
{
mov eax, fs:[0]
mov prev, eax
}
reg.prev = (EXCEPTION_REGISTRATION*) prev;

// 注册监控函数
preg = &reg;
__asm
{
mov eax, preg
mov fs:[0], eax
}

{
int* p;
p = 0;

// 下面的语句被执行，将导致一个异常
*p = 45;
}

printf("这里将不会被执行到.\n");

return 0;
}
```

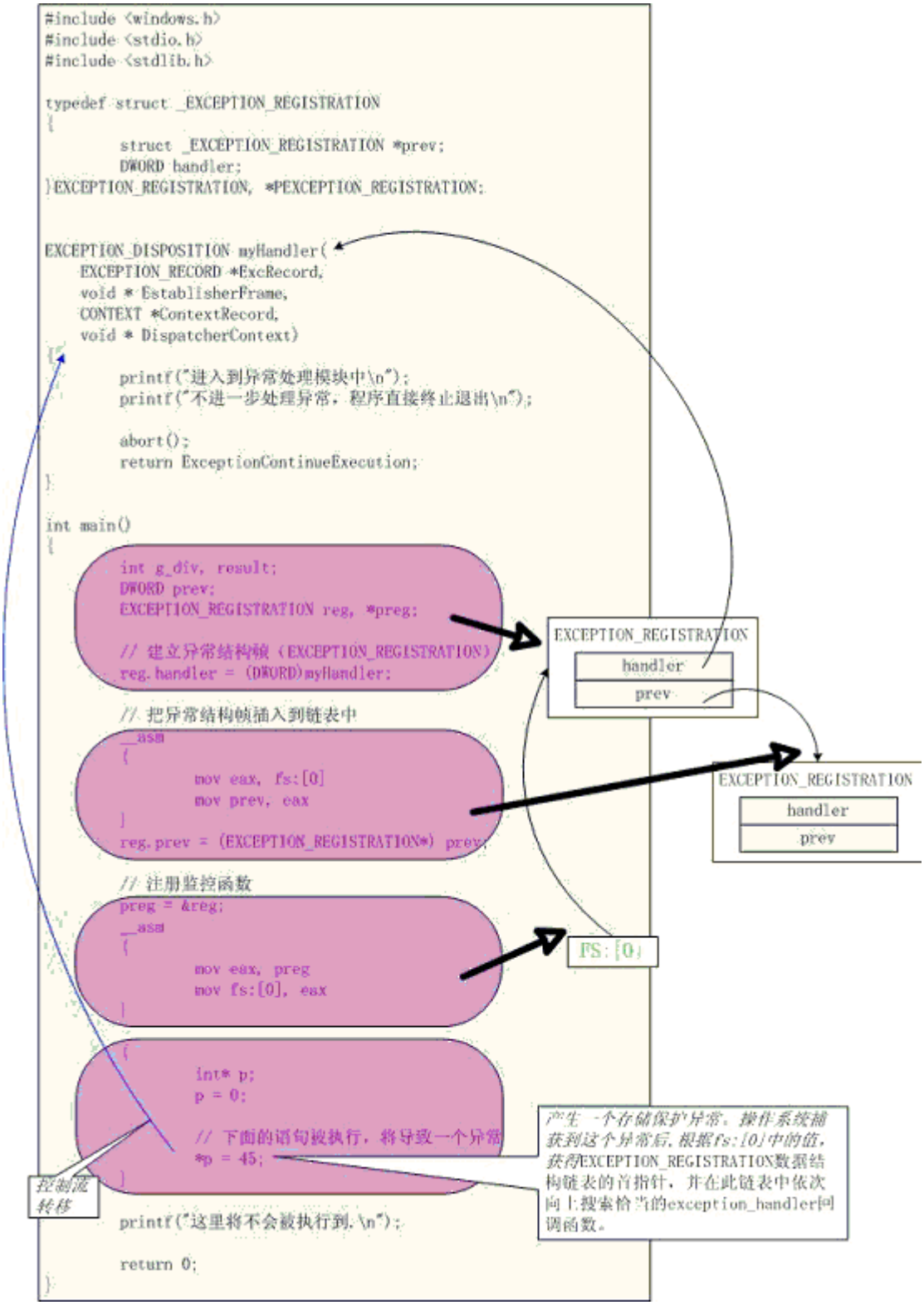
上面的程序运行结果如下：

通过上面的演示的简单例程，现在应该非常清楚了 Windows 操作系统提供的 SEH 机制的基本原理和控制流转移的具体过程。另外，这里分别详细介绍一下 exception_handler 回调函数的各个参数的涵义。其中第一个参数为 EXCEPTION_RECORD 类型，它记录了一些与异常相关的信息。它的定义如下：

```
typedef struct _EXCEPTION_RECORD {
DWORD ExceptionCode;
DWORD ExceptionFlags;
struct _EXCEPTION_RECORD *ExceptionRecord;
PVOID ExceptionAddress;
DWORD NumberParameters;
UINT_PTR ExceptionInformation[EXCEPTION_MAXIMUM_PARAMETERS];
} EXCEPTION_RECORD;
```

第二个参数为 PEXCEPTION_REGISTRATION 类型，既当前的异常帧指针。第三个参数为指向 CONTEXT 数据结构的指针，CONTEXT 数据结构体中记录了异常发生时，线程当时的上下文环境，主要包括寄存器的值，这一点有点类似于 setjmp 函数的作用。第四个参数 DispatcherContext，它也是一个指针，表示调度的上下文环境，这个参数一般不被用到。

最后再来看一看 exception_handler 回调函数的返回值有何意义？它基本上有两种返回值，一种就是返回 ExceptionContinueExecution，表示异常已经被恢复了，程序可以正常继续执行。另一种就是 ExceptionContinueSearch，它表示当前的异常回调处理函数不能有效处理这个异常错误，系统将会根据 EXCEPTION_REGISTRATION 数据链表，继续查找下一个异常处理的回调函数。上面的例程的详细分析如下图所示：



来一个稍微复杂一点例子，来更深入理解 SEH 机制

现在，相信大家已经对 SEH 机制，既有了非常理性的理解，也有非常感性的认识。实际上，从用户角

度上来分析，SEH 机制确是比较简单。它首先是用户注册一系列的异常回调函数（也即监控函数），操作系统为每个线程维护一个这样的链表，每当程序中出现异常的时候，操作系统便获得控制权，并纪录一些与异常相关的信息，接着系统便依次搜索上面的链表，来查找并调用相应的异常回调函数。

说到这里，也许朋友们有点疑惑了？上一篇文章中讲述到，“无论是__try，__except，__finally，__leave 异常模型机制，或是 try，catch，throw 方式的 C++异常模型，它们都是在 SEH 基础上来实现的”。但是从这里看来，好像上面描述的 SEH 机制与 try，catch，throw 方式的 C++异常模型不太相关。是的，也许表面上看起来区别是比较大的，但是 SEH 机制，它的的确确是上面讲到的其它两种异常处理模型的基础。这一点，在深入分析 C++异常模型的实现时，会再做详细的叙述。这里为了更深入理解 SEH 机制，主人公阿愚设计了一个稍微复杂一点例子。它仍然只有 SEH 机制，没有__try，__except，__finally，__leave 异常模型的任何影子，但是它与真实的__try，__except，__finally，__leave 异常模型的实现却有几分相似之处。

```
// seh.c
#include <windows.h>
#include <stdio.h>


typedef struct _EXCEPTION_REGISTRATION
{
struct _EXCEPTION_REGISTRATION *prev;
DWORD handler;
}EXCEPTION_REGISTRATION, *PEXCEPTION_REGISTRATION;

#define SEH_PROLOGUE(pFunc_exception) \
{ \
DWORD pFunc = (DWORD)pFunc_exception; \
_asm mov eax, FS:[0] \
_asm push pFunc \
_asm push eax \
_asm mov FS:[0], esp \
}

#define SEH_EPILOGUE() \
{ \
_asm pop FS:[0] \
_asm pop eax \
}

void printfErrorMsg(int ex_code)
{
char msg[20];

memset(msg, 0, sizeof(msg));
switch (ex_code)
{
```

```
case EXCEPTION_ACCESS_VIOLATION :
strcpy(msg, "存储保护异常");
break;
case EXCEPTION_ARRAY_BOUNDS_EXCEEDED :
strcpy(msg, "数组越界异常");
break;
case EXCEPTION_BREAKPOINT :
strcpy(msg, "断点异常");
break;
case EXCEPTION_FLT_DIVIDE_BY_ZERO :
case EXCEPTION_INT_DIVIDE_BY_ZERO :
strcpy(msg, "被 0 除异常");
break;
default :
strcpy(msg, "其它异常");
}

printf("\n");
printf("%s，错误代码为：0x%x\n", msg, ex_code);
}

EXCEPTION_DISPOSITION my_exception_Handler(
EXCEPTION_RECORD *ExcRecord,
void * EstablisherFrame,
CONTEXT *ContextRecord,
void * DispatcherContext)
{
int _ebp;

printfErrorMsg(ExcRecord->ExceptionCode);

printf("跳过出现异常函数，返回到上层函数中继续执行\n");
printf("\n");

_ebp = ContextRecord->Ebp;
_asm
{
// 恢复上一个异常帧
mov eax, EstablisherFrame
mov eax, [eax]
mov fs:[0], eax

// 返回到上一层的调用函数
mov esp, _ebp
pop ebp
```

```
mov eax, -1
ret
}

// 下面将绝对不会被执行到
exit(0);
return ExceptionContinueExecution;
}

EXCEPTION_DISPOSITION my_RaiseException_Handler(
EXCEPTION_RECORD *ExcRecord,
void * EstablisherFrame,
CONTEXT *ContextRecord,
void * DispatcherContext)
{
int _ebp;

printfErrorMsg(ExcRecord->ExceptionCode);

printf("跳过出现异常函数，返回到上层函数中继续执行\n");
printf("\n");

_ebp = ContextRecord->Ebp;
_asm
{
// 恢复上一个异常帧
mov eax, EstablisherFrame
mov eax, [eax]
mov fs:[0], eax

// 返回到上一层的调用函数
mov esp, _ebp
pop ebp
mov esp, ebp
pop ebp
mov eax, -1
ret
}

// 下面将绝对不会被执行到
exit(0);
return ExceptionContinueExecution;
}
```

```
void test1()
{
SEH_PROLOGUE(my_exception_Handler);

{
int zero;
int j;
zero = 0;

// 下面的语句被执行，将导致一个异常
j = 10 / zero;

printf("在 test1()函数中，这里将不会被执行到.j=%d\n", j);
}

SEH_EPILOGUE();
}

void test2()
{
SEH_PROLOGUE(my_exception_Handler);

{
int* p;
p = 0;

printf("在 test2()函数中，调用 test1()函数之前\n");
test1();
printf("在 test2()函数中，调用 test1()函数之后\n");
printf("\n");


// 下面的语句被执行，将导致一个异常
*p = 45;

printf("在 test2()函数中，这里将不会被执行到\n");
}

SEH_EPILOGUE();
}

void test3()
{
SEH_PROLOGUE(my_RaiseException_Handler);
```

```
{
// 下面的语句被执行，将导致一个异常
RaiseException(0x999, 0x888, 0, 0);

printf("在 test3()函数中，这里将不会被执行到\n");
}

SEH_EPILOGUE();
}

int main()
{
printf("在 main()函数中，调用 test1()函数之前\n");
test1();
printf("在 main()函数中，调用 test1()函数之后\n");

printf("\n");

printf("在 main()函数中，调用 test2()函数之前\n");
test2();
printf("在 main()函数中，调用 test2()函数之后\n");

printf("\n");

printf("在 main()函数中，调用 test3()函数之前\n");
test3();
printf("在 main()函数中，调用 test3()函数之后\n");

return 0;
}
```

上面的程序运行结果如下：

```
在 main()函数中，调用 test1()函数之前

被 0 除异常，错误代码为：0xc0000094
跳过出现异常函数，返回到上层函数中继续执行

在 main()函数中，调用 test1()函数之后

在 main()函数中，调用 test2()函数之前
在 test2()函数中，调用 test1()函数之前

被 0 除异常，错误代码为：0xc0000094
跳过出现异常函数，返回到上层函数中继续执行

在 test2()函数中，调用 test1()函数之后
```

存储保护异常，错误代码为：0xc0000005
跳过出现异常函数，返回到上层函数中继续执行

在 main()函数中，调用 test2()函数之后

在 main()函数中，调用 test3()函数之前

其它异常，错误代码为：0x999
跳过出现异常函数，返回到上层函数中继续执行

在 main()函数中，调用 test3()函数之后
Press any key to continue

总结

本文所讲到的异常处理机制，它就是狭义上的 SEH，虽然它很简单，但是它是 Windows 系列操作系统平台上其它所有异常处理模型实现的奠基石。有了它就有了基本的物质保障，
另外，通常一般所说的 SEH，它都是指在本篇文章中所阐述的狭义上的 SEH 机制基础之上，实现的__try，__except，__finally，__leave 异常模型，因此从下一篇文章中，开始全面介绍__try，__except，__finally，__leave 异常模型，实际上，它也即广义上的 SEH。此后所有的文章内容中，如没有特别注明，SEH 机制都表示__try，__except，__finally，__leave 异常模型，这也是为了与 try，catch，throw 方式的 C++异常模型相区分开。

朋友们！有点疲劳了吧！可千万不要放弃，继续到下一篇的文章中，可要知道，__try，__except，__finally，__leave 异常模型，它可以说是最优先的异常处理模型之一，甚至比 C++的异常模型还好，功能还强大！即便是 JAVA 的异常处理模型也都从它这里继承了许多优点，所以不要错过呦，Let’s go!

第 23 集 SEH 的强大功能之一

从本篇文章开始，将全面阐述__try，__except，__finally，__leave 异常模型机制，它也即是 Windows 系列操作系统平台上提供的 SEH 模型。主人公阿愚将在这里与大家分享 SEH 的学习过程和经验总结。

SEH 有两项非常强大的功能。当然，首先是异常处理模型了，因此，这篇文章首先深入阐述 SEH 提供的异常处理模型。另外，SEH 还有一个特别强大的功能，这将在下一篇文章中进行详细介绍。

try-except 入门

SEH 的异常处理模型主要由 try-except 语句来完成，它与标准 C++所定义的异常处理模型非常类似，也都是可以定义出受监控的代码模块，以及定义异常处理模块等。还是老办法，看一个例子先，代码如下：

```
//seh-test.c
#include <stdio.h>
```

```
void main()
{
puts("hello");
// 定义受监控的代码模块
__try
{
puts("in try");
}
//定义异常处理模块
__except(1)
{
puts("in except");
}
puts("world");
}
```

呵呵！是不是很简单，而且与 C++异常处理模型很相似。当然，为了与 C++异常处理模型相区别，VC 编译器对关键字做了少许变动。首先是在每个关键字加上两个下划线作为前缀，这样既保持了语义上的一致性，另外也尽最大可能来避免了关键字的有可能造成名字冲突而引起的麻烦等；其次，C++异常处理模型是使用 catch 关键字来定义异常处理模块，而 SEH 是采用__except 关键字来定义。并且，catch 关键字后面往往好像接受一个函数参数一样，可以是各种类型的异常数据对象；但是__except 关键字则不同，它后面跟的却是一个表达式（可以是各种类型的表达式，后面会进一步分析）。

try-except 进阶

与 C++异常处理模型很相似，在一个函数中，可以有多个 try-except 语句。它们可以是一个平面的线性结构，也可以是分层的嵌套结构。例程代码如下：

```
// 例程 1
// 平面的线性结构
#include <stdio.h>

void main()
{
puts("hello");
__try
{
puts("in try");
}
__except(1)
{
puts("in except");
}
```

```
// 又一个 try-except 语句
__try
{
puts("in try");
}
__except(1)
{
puts("in except");
}

puts("world");
}


// 例程 2
// 分层的嵌套结构
#include <stdio.h>

void main()
{
puts("hello");
__try
{
puts("in try");
// 又一个 try-except 语句
__try
{
puts("in try");
}
__except(1)
{
puts("in except");
}
}
__except(1)
{
puts("in except");
}

puts("world");
}

// 例程 3
// 分层的嵌套在__except 模块中
#include <stdio.h>
```

```
void main()
{
puts("hello");
__try
{
puts("in try");
}
__except(1)
{
// 又一个 try-except 语句
__try
{
puts("in try");
}
__except(1)
{
puts("in except");
}

puts("in except");
}

puts("world");
}
```

try-except 异常处理规则

try-except 异常处理规则与 C++异常处理模型有相似之处，例如，它们都是向上逐级搜索恰当的异常处理模块，包括跨函数的多层嵌套 try-except 语句。但是，它们的处理规则也有另外一些很大的不同之处，例如查找匹配恰当的异常处理模块的过程，在 C++异常处理模型中，它是通过异常对象的类型来匹配；但是在 try-except 语句的异常处理规则中，则是通过__except 关键字后面括号中的表达式的值来匹配查找正确的异常处理模块。还是看看 MSDN 中怎么说的吧！摘略如下：

The compound statement after the __try clause is the body or guarded section. The compound statement after the __except clause is the exception handler. The handler specifies a set of actions to be taken if an exception is raised during execution of the body of the guarded section. Execution proceeds as follows:

1. The guarded section is executed.
2. If no exception occurs during execution of the guarded section, execution continues at the statement after the __except clause.
3. If an exception occurs during execution of the guarded section or in any routine the guarded section calls, the __except expression is evaluated and the value determines how the exception is handled. There are three values:

EXCEPTION_CONTINUE_EXECUTION (–1) Exception is dismissed. Continue execution at the point where the exception occurred.

EXCEPTION_CONTINUE_SEARCH (0) Exception is not recognized. Continue to search up the stack for a handler, first for containing try-except statements, then for handlers with the next highest precedence.

EXCEPTION_EXECUTE_HANDLER (1) Exception is recognized. Transfer control to the exception handler by executing the __except compound statement, then continue execution at the assembly instruction that was executing when the exception was raised.
Because the __except expression is evaluated as a C expression, it is limited to a single value, the conditional-expression operator, or the comma operator. If more extensive processing is required, the expression can call a routine that returns one of the three values listed above.

对查找匹配恰当的异常处理模块的过程等几条规则翻译如下：

1. 受监控的代码模块被执行（也即__try 定义的模块代码）；

2. 如果上面的代码执行过程中，没有出现异常的话，那么控制流将转入到__except 子句之后的代码模块中；

3. 否则，如果出现异常的话，那么控制流将进入到__except 后面的表达式中，也即首先计算这个表达式的值，之后再根据这个值，来决定做出相应的处理。这个值有三种情况，如下：

EXCEPTION_CONTINUE_EXECUTION (–1) 异常被忽略，控制流将在异常出现的点之后，继续恢复运行。

EXCEPTION_CONTINUE_SEARCH (0) 异常不被识别，也即当前的这个__except 模块不是这个异常错误所对应的正确的异常处理模块。系统将继续到上一层的 try-except 域中继续查找一个恰当的__except 模块。

EXCEPTION_EXECUTE_HANDLER (1) 异常已经被识别，也即当前的这个异常错误，系统已经找到了并能够确认，这个__except 模块就是正确的异常处理模块。控制流将进入到__except 模块中。

上面的规则其实挺简单的，很好理解。当然，这个规则也非常的严谨，它能很好的满足开发人员的各种需求，满足程序员对异常处理的分类处理的要求，它能够给程序员提供一个灵活的控制手段。

其中比较特殊的就是__except 关键字后面跟的表达式，它可以是各种类型的表达式，例如，它可以是一个函数调用，或是一个条件表达式，或是一个逗号表达式，或干脆就是一个整型常量等等。例如代码如下：

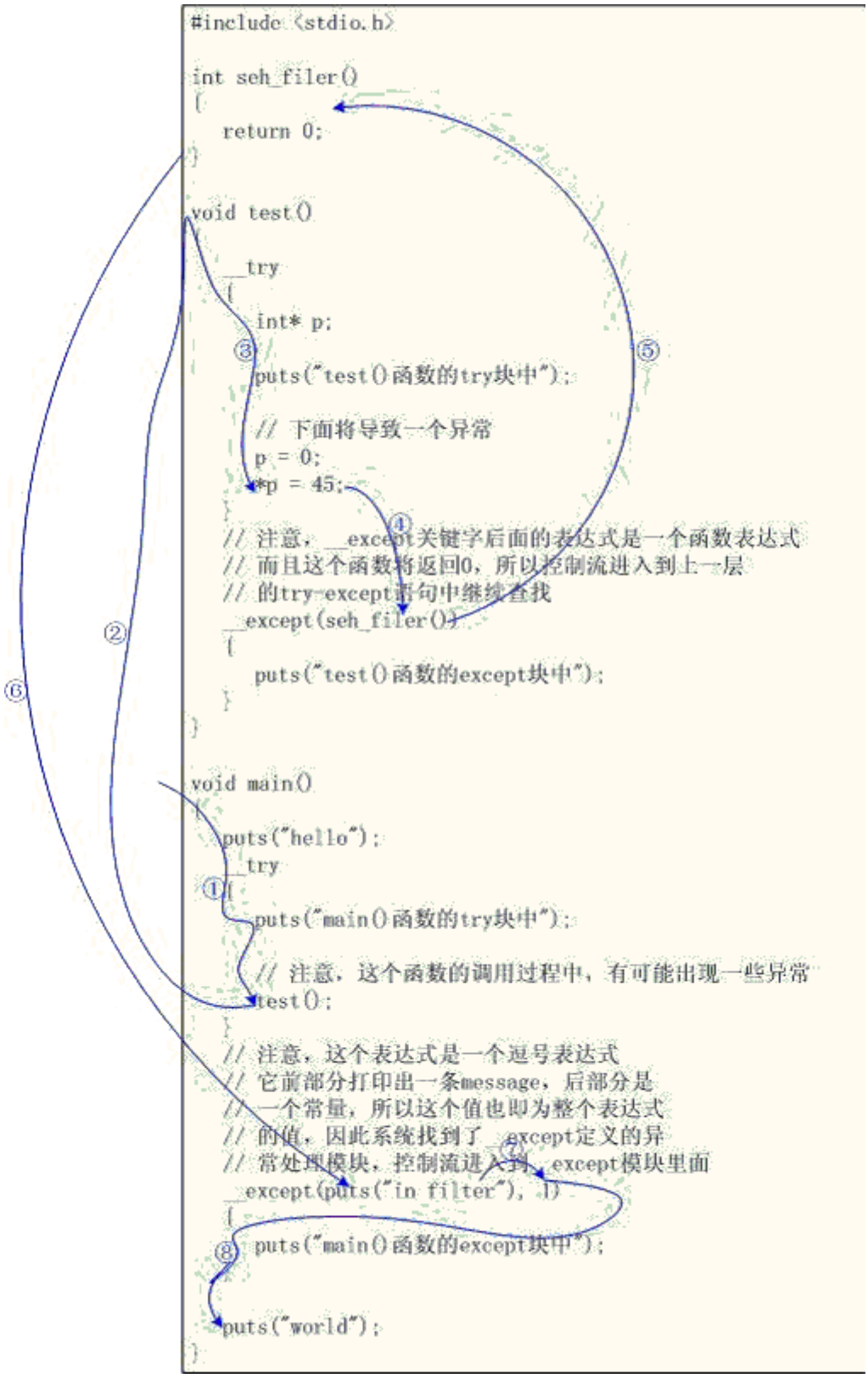
```
// seh-test.c
//  异常处理模块的查找过程演示
#include <stdio.h>

int seh_filer()
{
return 0;
}

void test()
{
__try
{
int* p;

puts("test()函数的 try 块中");
```

```
// 下面将导致一个异常
p = 0;
*p = 45;
}
// 注意，__except 关键字后面的表达式是一个函数表达式
// 而且这个函数将返回 0，所以控制流进入到上一层
// 的 try-except 语句中继续查找
__except(seh_filer())
{
puts("test()函数的 except 块中");
}
}

void main()
{
puts("hello");
__try
{
puts("main()函数的 try 块中");

// 注意，这个函数的调用过程中，有可能出现一些异常
test();
}
// 注意，这个表达式是一个逗号表达式
// 它前部分打印出一条 message，后部分是
// 一个常量，所以这个值也即为整个表达式
// 的值，因此系统找到了__except 定义的异
// 常处理模块，控制流进入到__except 模块里面
__except(puts("in filter"), 1)
{
puts("main()函数的 except 块中");
}

puts("world");
}
```

上面的程序运行结果如下：

```
hello
main()函数的 try 块中
test()函数的 try 块中
in filter
main()函数的 except 块中
world
Press any key to continue
```

这种运行结果应该是在意料之中吧！为了对它的流程进行更清楚的分析，下图描述出了程序的运行控制流转移过程，如下。



另外，对于__except 关键字后面表达式的值，上面的规则中已经做了详细规定。它们有三种值，其中如果为 0，那么系统继续查找；如果为 1，表示系统已经找到正确的异常处理模块。其实这两个值都很好理

解，可是如果值为-1 的话，那么处理将比较特殊，上面也提到了，此种情况下，“异常被忽略，控制流将在异常出现的点之后，继续恢复运行。”实际上，这就等同于说，程序的执行过程将不受干扰，好像异常从来没有发生一样。看一个例程吧！代码如下：

```
#include <stdio.h>

void main()
{
int j, zero;

puts("hello");
__try
{
puts("main()函数的 try 块中");

zero = 0;
j = 10;
// 下面将导致一个异常
j = 45 / zero;

// 注意，异常出现后，程序控制流又恢复到了这里
printf("这里会执行到吗？值有如何呢？j=%d \n", j);
}
// 注意，这里把 zero 变量赋值为 1，试图恢复错误，
// 当控制流恢复到原来异常点时，避免了异常的再次发生
__except(puts("in filter"), zero = 1, -1)
{
puts("main()函数的 except 块中");
}

puts("world");
}
```

上面的程序运行结果如下：

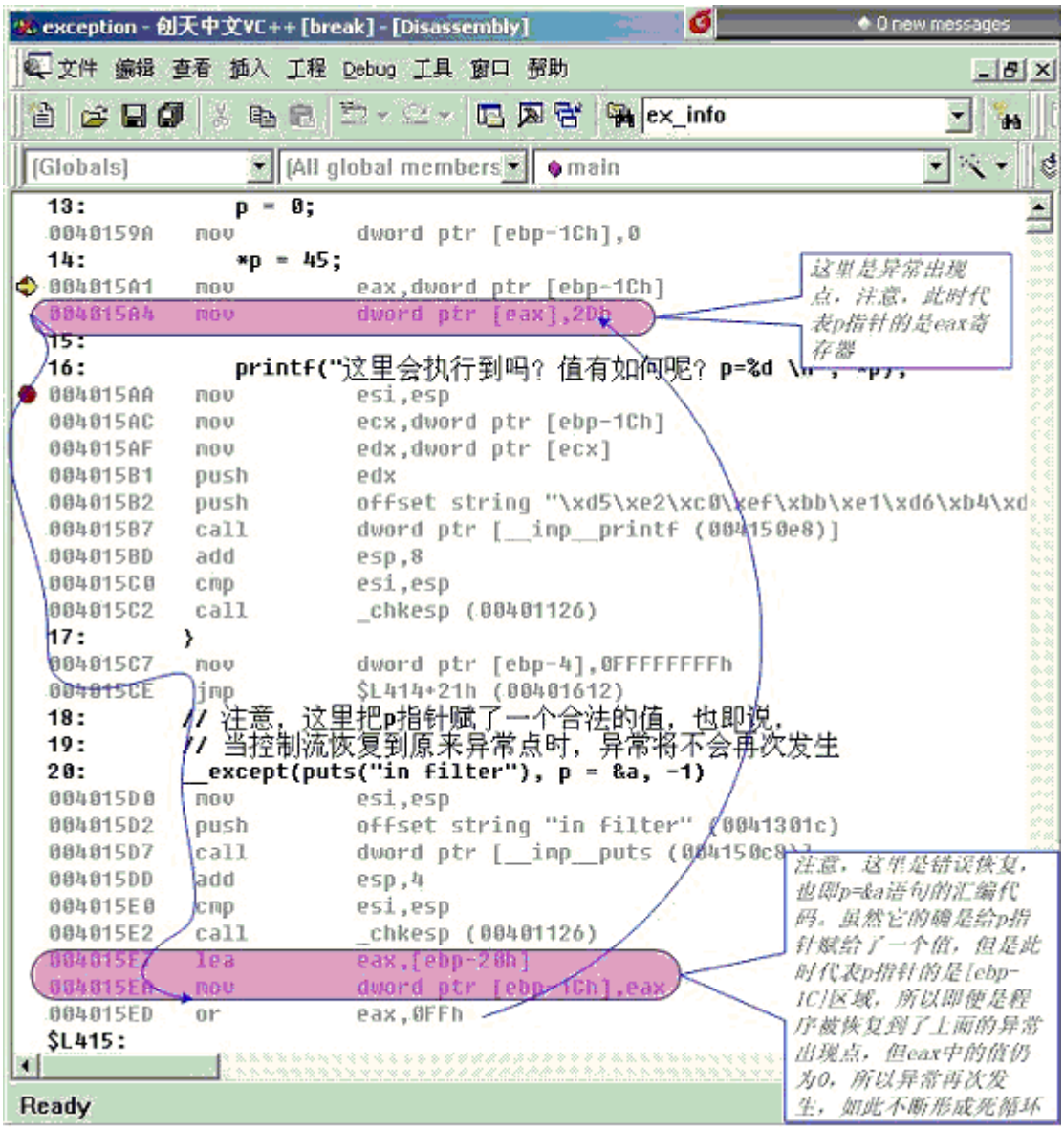
```
hello
main()函数的 try 块中
in filter
这里会执行到吗？值有如何呢？j=45
world
Press any key to continue
```

呵呵！厉害吧！要知道 C++异常处理模型可没有这样的能力。但是请注意，一般这项功能不能轻易采用，为什么呢？因为它会导致不稳定，再看下面一个示例，代码如下：

```
#include <stdio.h>
```

```
void main()
{
int* p, a;

puts("hello");
__try
{
puts("main()函数的 try 块中");

// 下面将导致一个异常
p = 0;
*p = 45;

printf("这里会执行到吗？值有如何呢？p=%d \n", *p);
}
// 注意，这里把 p 指针赋了一个合法的值，也即说，
// 当控制流恢复到原来异常点时，异常将不会再次发生
__except(puts("in filter"), p = &a, -1)
{
puts("main()函数的 except 块中");
}

puts("world");
}
```

呵呵！大家猜猜上面的程序的运行结果如何呢？是不是和刚才的那个例子一样，异常也得以被恢复了。朋友们！还是亲自运行测试一把。哈哈！程序运行结果是死了，进行一个无限循环当中，并且控制终端内不断输出“in filter”信息。为什么会出现这种情况，难道 MSDN 中有关的阐述的有问题吗？或这个异常处理模型实现上存在 BUG？NO！不是这样的，实际上这就是由于表达式返回-1 值时，给程序所带来的不稳定性。当然，MSDN 中有关的阐述也没有错，那么究竟具体原因是为何呢？这是因为，表达式返回-1 值时，系统将把控制流恢复到异常出现点之后继续运行。这意味着什么呢？也许大家都明白了，它这里的异常恢复点是基于一条机器指令级别上的。这样就有很大的风险，因为上面的例程中，所谓的异常恢复处理，也即 p = &a 语句，它实际上的确改变了 p 指针值，但是这个指针值是栈上的某个内存区域，而真正出现异常时，代表 p 指针值的很有可能是某个寄存器。呵呵！是不是挺费解的，没关系！还是看看调试界图吧！如下：



try-except 深入

上面的内容中已经对 try-except 进行了全面的了解，但是有一点还没有阐述到。那就是如何在__except 模块中获得异常错误的相关信息，这非常关键，它实际上是进行异常错误处理的前提，也是对异常进行分层分级别处理的前提。可想而知，如果没有这些起码的信息，异常处理如何进行？因此获取异常信息非常的关键。Windows 提供了两个 API 函数，如下：

LPEXCEPTION_POINTERS GetExceptionInformation(VOID);
DWORD GetExceptionCode(VOID);

其中 GetExceptionCode()返回错误代码，而 GetExceptionInformation()返回更全面的信息，看它函数的声明，返回了一个 LPEXCEPTION_POINTERS 类型的指针变量。那么 EXCEPTION_POINTERS 结构如何呢？如下，

```
typedef struct _EXCEPTION_POINTERS { // exp
PEXCEPTION_RECORD ExceptionRecord;
PCONTEXT ContextRecord;
} EXCEPTION_POINTERS;
```

呵呵！仔细瞅瞅，这是不是和上一篇文章中，用户程序所注册的异常处理的回调函数的两个参数类型一样。是的，的确没错！其中 EXCEPTION_RECORD 类型，它记录了一些与异常相关的信息；而 CONTEXT 数据结构体中记录了异常发生时，线程当时的上下文环境，主要包括寄存器的值。因此有了这些信息，__except 模块便可以对异常错误进行很好的分类和恢复处理。不过特别需要注意的是，这两个函数只能是在__except 后面的括号中的表达式作用域内有效，否则结果可能没有保证（至于为什么，在后面深入分析异常模型的实现时候，再做详细阐述）。看一个例程吧！代码如下：

```
#include <windows.h>
#include <stdio.h>

int exception_access_violation_filter(LPEXCEPTION_POINTERS p_exinfo)
{
if(p_exinfo->ExceptionRecord->ExceptionCode == EXCEPTION_ACCESS_VIOLATION)
{
printf("存储保护异常\n");
return 1;
}
else return 0;
}

int exception_int_divide_by_zero_filter(LPEXCEPTION_POINTERS p_exinfo)
{
if(p_exinfo->ExceptionRecord->ExceptionCode == EXCEPTION_INT_DIVIDE_BY_ZERO)
{
printf("被 0 除异常\n");
return 1;
}
else return 0;
}

void main()
{
puts("hello");
__try
{
__try
{
int* p;
```

```
// 下面将导致一个异常
p = 0;
*p = 45;
}
// 注意，__except 模块捕获一个存储保护异常
__except(exception_access_violation_filter(GetExceptionInformation()))
{
puts("内层的 except 块中");
}
}
// 注意，__except 模块捕获一个被 0 除异常
__except(exception_int_divide_by_zero_filter(GetExceptionInformation()))
{
puts("外层的 except 块中");
}

puts("world");
}
```

上面的程序运行结果如下：

```
hello
存储保护异常
内层的 except 块中
world
Press any key to continue
```

呵呵！感觉不错，大家可以在上面的程序基础之上改动一下，让它抛出一个被 0 除异常，看程序的运行结果是不是如预期那样。

最后还有一点需要阐述，在 C++的异常处理模型中，有一个 throw 关键字，也即在受监控的代码中抛出一个异常，那么在 SEH 异常处理模型中，是不是也应该有这样一个类似的关键字或函数呢？是的，没错！SEH 异常处理模型中，对异常划分为两大类，第一种就是上面一些例程中所见到的，这类异常是系统异常，也被称为硬件异常；还有一类，就是程序中自己抛出异常，被称为软件异常。怎么抛出呢？还是 Windows 提供了的 API 函数，它的声明如下：

```
VOID RaiseException(
DWORD dwExceptionCode, // exception code
DWORD dwExceptionFlags, // continuable exception flag
DWORD nNumberOfArguments, // number of arguments in array
CONST DWORD *lpArguments // address of array of arguments
);
```

很简单吧！实际上，在 C++的异常处理模型中的 throw 关键字，最终也是对 RaiseException()函数的调用，也即是说，throw 是 RaiseException 的上层封装的更高级一类的函数，这以后再详细分析它的代码实现。这里还是看一个简单例子吧！代码如下：

```
#include <windows.h>
#include <stdio.h>

int seh_filer(int code)
{
switch(code)
{
case EXCEPTION_ACCESS_VIOLATION :
printf("存储保护异常，错误代码：%x\n", code);
break;
case EXCEPTION_DATATYPE_MISALIGNMENT :
printf("数据类型未对齐异常，错误代码：%x\n", code);
break;
case EXCEPTION_BREAKPOINT :
printf("中断异常，错误代码：%x\n", code);
break;
case EXCEPTION_SINGLE_STEP :
printf("单步中断异常，错误代码：%x\n", code);
break;
case EXCEPTION_ARRAY_BOUNDS_EXCEEDED :
printf("数组越界异常，错误代码：%x\n", code);
break;
case EXCEPTION_FLT_DENORMAL_OPERAND :
case EXCEPTION_FLT_DIVIDE_BY_ZERO :
case EXCEPTION_FLT_INEXACT_RESULT :
case EXCEPTION_FLT_INVALID_OPERATION :
case EXCEPTION_FLT_OVERFLOW :
case EXCEPTION_FLT_STACK_CHECK :
case EXCEPTION_FLT_UNDERFLOW :
printf("浮点数计算异常，错误代码：%x\n", code);
break;
case EXCEPTION_INT_DIVIDE_BY_ZERO :
printf("被 0 除异常，错误代码：%x\n", code);
break;
case EXCEPTION_INT_OVERFLOW :
printf("数据溢出异常，错误代码：%x\n", code);
break;
case EXCEPTION_IN_PAGE_ERROR :
printf("页错误异常，错误代码：%x\n", code);
break;
case EXCEPTION_ILLEGAL_INSTRUCTION :
printf("非法指令异常，错误代码：%x\n", code);
break;
case EXCEPTION_STACK_OVERFLOW :
```

```
printf("堆栈溢出异常，错误代码：%x\n", code);
break;
case EXCEPTION_INVALID_HANDLE :
printf("无效句病异常，错误代码：%x\n", code);
break;
default :
if(code & (1<<29))
printf("用户自定义的软件异常，错误代码：%x\n", code);
else
printf("其它异常，错误代码：%x\n", code);
break;
}

return 1;
}

void main()
{
puts("hello");
__try
{
puts("try 块中");

// 注意，主动抛出一个软异常
RaiseException(0xE0000001, 0, 0, 0);
}
__except(seh_filer(GetExceptionCode()))
{
puts("except 块中");
}

puts("world");
}
```

上面的程序运行结果如下：

```
hello
try 块中
用户自定义的软件异常，错误代码：e0000001
except 块中
world
Press any key to continue
```

上面的程序很简单，这里不做进一步的分析。我们需要重点讨论的是，在__except 模块中如何识别不同的异常，以便对异常进行很好的分类处理。毫无疑问，它当然是通过 GetExceptionCode()或 GetExceptionInformation ()函数来获取当前的异常错误代码，实际也即是 DwExceptionCode 字段。异常错误

代码在 winError.h 文件中定义，它遵循 Windows 系统下统一的错误代码的规则。每个 DWORD 被划分几个字段，如下表所示：

bit 位	31-30	29	28	27-16	15-0
字段涵义	严重性系数 0=成功 1=信息 2=警告 3=错误	微软/客户 0=微软定义的代码 1=客户定义的代码	保留 必须为 0	设备代码	异常代码

例如我们可以在 winbase.h 文件中找到 EXCEPTION_ACCESS_VIOLATION 的值为 0 xC0000005，将这个异常代码值拆开，来分析看看它的各个 bit 位字段的涵义。

C 0 0 0 0 0 0 5 （十六进制）

1100 0000 0000 0000 0000 0000 0000 0101 （二进制）

第 3 0 位和第 3 1 位都是 1，表示该异常是一个严重的错误，线程可能不能够继续往下运行，必须要及时处理恢复这个异常。第 2 9 位是 0，表示系统中已经定义了异常代码。第 2 8 位是 0，留待后用。第 1 6 位至 2 7 位是 0，表示是 FACILITY_NULL 设备类型，它代表存取异常可发生在系统中任何地方，不是使用特定设备才发生的异常。第 0 位到第 1 5 位的值为 5，表示异常错误的代码。

如果程序员在程序代码中，计划抛出一些自定义类型的异常，必须要规划设计好自己的异常类型的划分，按照上面的规则来填充异常代码的各个字段值，如上面示例程序中抛出一个异常代码为 0xE0000001 软件异常。

总结

（1） C++异常模型用 try-catch 语法定义，而 SEH 异常模型则用 try-except 语法；

（2） 与 C++异常模型相似，try-except 也支持多层的 try-except 嵌套。

（3） 与 C++异常模型不同的是，try-except 模型中，一个 try 块只能是有一个 except 块；而 C++异常模型中，一个 try 块可以有多个 catch 块。

（4） 与 C++异常模型相似，try-except 模型中，查找搜索异常模块的规则也是逐级向上进行的。但是稍有区别的是，C++异常模型是按照异常对象的类型来进行匹配查找的；而 try-except 模型则不同，它通过一个表达式的值来进行判断。如果表达式的值为 1（EXCEPTION_EXECUTE_HANDLER），表示找到了异常处理模块；如果值为 0（EXCEPTION_CONTINUE_SEARCH），表示继续向上一层的 try-except 域中继续查找其它可能匹配的异常处理模块；如果值为-1（EXCEPTION_CONTINUE_EXECUTION），表示忽略这个异常，注意这个值一般很少用，因为它很容易导致程序难以预测的结果，例如，死循环，甚至导致程序的崩溃等。

（5） __except 关键字后面跟的表达式，它可以是各种类型的表达式，例如，它可以是一个函数调用，或是一个条件表达式，或是一个逗号表达式，或干脆就是一个整型常量等等。最常用的是一个函数表达式，并且通过利用 GetExceptionCode()或 GetExceptionInformation ()函数来获取当前的异常错误信息，便于程序

员有效控制异常错误的分类处理。

（6） SEH 异常处理模型中，异常被划分为两大类：系统异常和软件异常。其中软件异常通过 RaiseException()函数抛出。RaiseException()函数的作用类似于 C++异常模型中的 throw 语句。

本篇文章已经对 SEH 的异常处理进行了比较全面而深入的阐述，相信大家现在已经对 SEH 的异常处理机制胸有成竹了。但是 SEH 的精华仅只如此吗？非也，朋友们！继续到下一篇的文章中，主人公阿愚将和大家一起共同探讨 SEH 模型的另一项重要的机制，那就是“有效保证资源的清除”。这对于 C 程序可太重要了，因为在 C++程序中，至少还有对象的析构函数来保证资源的有效清除，避免资源泄漏，但 C 语言中则没有一个有效的机制，来完成此等艰巨的任务。呵呵！SEH 雪中送炭，它提供了完美的解决方案，所以千万不要错过，一起去看看吧！Let’s go!

第 24 集 SEH 的强大功能之二

上一篇文章讲述了 SEH 的异常处理机制，也即 try-except 模型的使用规则。本篇文章继续探讨 SEH 另外一项很重要的机制，那就是“有效保证资源的清除”，其实这才是 SEH 设计上最为精华的一个东东，对于 C 程序而言，它贡献简直是太大了。

SEH 的这项机制被称为结束处理（Termination Handling），它是通过 try-finally 语句来实现的，下面开始讨论吧！

try-finally 的作用

对于 try-finally 的作用，还是先看看 MSDN 中怎么说的吧！摘略如下：

The try-finally statement is a Microsoft extension to the C and C++ languages that enables 32-bit target applications to guarantee execution of cleanup code when execution of a block of code is interrupted. Cleanup consists of such tasks as deallocating memory, closing files, and releasing file handles. The try-finally statement is especially useful for routines that have several places where a check is made for an error that could cause premature return from the routine.

上面的这段话的内容翻译如下：

try-finally 语句是 Microsoft 对 C 和 C++语言的扩展，它能使 32 位的目标程序在异常出现时，有效保证一些资源能够被及时清除，这些资源的清除任务可以包括例如内存的释放，文件的关闭，文件句柄的释放等等。try-finally 语句特别适合这样的情况下使用，例如一个例程（函数）中，有几个地方需要检测一个错误，并且在错误出现时，函数可能提前返回。

try-finally 的语法规则

上面描述 try-finally 机制的有关作用时，也许一时我们还难以全面理解，不过没关系，这里还是先看一下 try-finally 的语法规则吧！其实它很简单，示例代码如下：

```
//seh-test.c
#include <windows.h>
#include <stdio.h>

void main()
{
puts("hello");
__try
{
puts("__try 块中");
}
// 注意，这里不是__except 块，而是__finally 取代
__finally
{
puts("__finally 块中");
}

puts("world");
}
```

上面的程序运行结果如下：

```
hello
__try 块中
__finally 块中
world
Press any key to continue
```

try-finally 语句的语法与 try-except 很类似，稍有不同的是，__finally 后面没有一个表达式，这是因为 try-finally 语句的作用不是用于异常处理，所以它不需要一个表达式来判断当前异常错误的种类。另外，与 try-except 语句类似，try-finally 也可以是多层嵌套的，并且一个函数内可以有多个 try-finally 语句，不管它是嵌套的，或是平行的。当然，try-finally 多层嵌套也可以是跨函数的。这里不一一列出示例，大家可以自己测试一番。

另外，对于上面示例程序的运行结果，是不是觉得有点意料之外呢？因为__finally 块中的 put("__finally 块中")语句也被执行了。是的，没错！这就是 try-finally 语句最具有魔幻能力的地方，即"不管在何种情况下，在离开当前的作用域时，finally 块区域内的代码都将会被执行到"。呵呵！这的确是很厉害吧！为了验证这条规则，下面来看一个更典型示例，代码如下：

```
#include <stdio.h>

void main()
{
puts("hello");
__try
{
puts("__try 块中");
```

```
// 注意，下面 return 语句直接让函数返回了
return;
}
__finally
{
puts("__finally 块中");
}

puts("world");
}
```

上面的程序运行结果如下：

```
hello
__try 块中
__finally 块中
Press any key to continue
```

上面的程序运行结果是不是有点意思。在__try 块区域中，有一条 return 语句让函数直接返回了，所以后面的 put(“world”)语句没有被执行到，这是很容易被理解的。但是请注意，__finally 块区域中的代码也将会被予以执行过了，这是不是进一步验证了上面了那条规则，呵呵！阿愚深有感触的想：“__finally 的特性真的很像对象的析构函数”，朋友们觉得如何呢？

另外，大家也许还特别关心的是，goto 语句是不是有可能破坏上面这条规则呢？因为在 C 语言中，goto 语句一般直接对应一条 jmp 跳转指令，所以如果真的如此的话，那么 goto 语句很容易破坏上面这条规则。还是看一个具体的例子吧！

```
#include <stdio.h>

void main()
{
puts("hello");
__try
{
puts("__try 块中");

// 跳转指令
goto RETURN;
}
__finally
{
puts("__finally 块中");
}
```

```
RETURN:
puts("world");
}
```

上面的程序运行结果如下：

```
hello
__try 块中
__finally 块中
world
Press any key to continue
```

呵呵！即便上面的示例程序中，goto 语句跳过了__finally 块，但是__finally 块区域中的代码还是被予以执行了。当然，大家也许很关心这到底是为什么？为什么 try-finally 语句具有如此神奇的功能？这里不打算深入阐述，在后面阐述 SEH 实现的时候会详细分析到。这里朋友们只牢记一点，“不管是顺序的线性执行，还是 return 语句或 goto 语句无条件跳转等情
况下，一旦执行流在离开当前的作用域时，finally 块区域内的代码必将会被执行”

try-finally 块中的异常

上面只列举了 return 语句和 goto 语句的情况下，但是如果程序中出现异常的话，那么 finally 块区域内的代码还会被执行吗？上面所讲到的那条规则仍然正确吗？还是看看示例，代码如下：

```
#include <stdio.h>

void test()
{
puts("hello");
__try
{
int* p;
puts("__try 块中");

// 下面抛出一个异常
p = 0;
*p = 25;
}
__finally
{
// 这里会被执行吗
puts("__finally 块中");
}

puts("world");
}
```

```
void main()
{
__try
{
test();
}
__except(1)
{
puts("__except 块中");
}
}
```

上面的程序运行结果如下：

```
hello
__try 块中
__finally 块中
__except 块中
Press any key to continue
```

从上面示例程序的运行结果来看，它是和“不管在何种情况下，在离开当前的作用域时，finally 块区域内的代码都将会被执行到”这条规则相一致的。

__leave 关键字的作用

其实，总结上面的__finally 块被执行的流程时，无外乎三种情况。第一种就是顺序执行到__finally 块区域内的代码，这种情况很简单，容易理解；第二种就是 goto 语句或 return 语句引发的程序控制流离开当前__try 块作用域时，系统自动完成对__finally 块代码的调用；第三种就是由于在__try 块中出现异常时，导致程序控制流离开当前__try 块作用域，这种情况下也是由系统自动完成对__finally 块的调用。无论是第 2 种，还是第 3 种情况，毫无疑问，它们都会引起很大的系统开销，编译器在编译此类程序代码时，它会为这两种情况准备很多的额外代码。一般第 2 种情况，被称为“局部展开（LocalUnwinding）”；第 3 种情况，被称为“全局展开（GlobalUnwinding）”。在后面阐述 SEH 实现的时候会详细分析到这一点。
第 3 种情况，也即由于出现异常而导致的“全局展开”，对于程序员而言，这也许是无法避免的，因为你在利用异常处理机制提高程序可靠健壮性的同时，不可避免的会引起性能上其它的一些开销。呵呵！这世界其实也算瞒公平的，有得必有失。

但是，对于第 2 种情况，程序员完全可以有效地避免它，避免“局部展开”引起的不必要的额外开销。实际这也是与结构化程序设计思想相一致的，也即一个程序模块应该只有一个入口和一个出口，程序模块内尽量避免使用 goto 语句等。但是，话虽如此，有时为了提高程序的可读性，程序员在编写代码时，有时可能不得不采用一些与结构化程序设计思想相悖的做法，例如，在一个函数中，可能有多处的 return 语句。针对这种情况，SEH 提供了一种非常有效的折衷方案，那就是__leave 关键字所起的作用，它既具有像 goto 语句和 return 语句那样类似的作用（由于检测到某个程序运行中的错误，需要马上离开当前的__try 块作用域），但是又避免了“局部展开” 的额外开销。还是看个例子吧！代码如下：

```
#include <stdio.h>
```

```
void test()
{
puts("hello");
__try
{
int* p;
puts("__try 块中");

// 直接跳出当前的__try 作用域
__leave;
p = 0;
*p = 25;
}
__finally
{
// 这里会被执行吗？当然
puts("__finally 块中");
}

puts("world");
}

void main()
{
__try
{
test();
}
__except(1)
{
puts("__except 块中");
}
}
```

上面的程序运行结果如下：

```
hello
__try 块中
__finally 块中
world
Press any key to continue
```

这就是__leave 关键字的作用，也许大家在编程时很少使用它。但是请注意，如果你的程序中，尤其在那些业务特别复杂的函数模块中，既采用了 SEH 机制来保证程序的可靠性，同时代码中又拥有大量的 goto 语句和 return 语句的话，那么你的源代码编译出来的二进制程序将是十分糟糕的，不仅十分庞大，而且效率也受很大影响。此时，建议不妨多用__leave 关键字来提高程序的性能。

try-finally 深入

现在，相信我们已经对 try-finally 机制有了非常全面的了解，为了更进一步认识 try-finally 机制的好处（当然，主人公阿愚认为，那些写过 Windows 平台下设备驱动程序的朋友一定深刻认识到 try-finally 机制的重要性），这里给出一个具体的例子。还记得，在《第 21 集 Windows 系列操作系统平台中所提供的异常处理机制》中，所讲述到的采用 setjmp 和 longjmp 异常处理机制实现的那个简单例程吗？现在如果有了 try-finally 机制，将能够很容易地来避免内存资源的泄漏，而且还极大地提高了程序模块的可读性，减少程序员由于不小心造成的程序 bug 等隐患。采用 SEH 重新实现的代码如下：

```
#include <stdio.h>
#include <stdlib.h>

void test1()
{
char* p1, *p2, *p3, *p4;

__try
{
p1 = malloc(10);
p2 = malloc(10);
p3 = malloc(10);
p4 = malloc(10);

// do other job
//  期间可能抛出异常
}
__finally
{
//  这里保证所有资源被及时释放
if(p1) free(p1);
if(p2) free(p2);
if(p3) free(p3);
if(p4) free(p4);
}
}

void test()
{
char* p;

__try
{
p = malloc(10);
```

```
// do other job
//  期间可能抛出异常

test1();

// do other job
}
__finally
{
//  这里保证资源被释放
if(p) free(p);
}
}

void main( void )
{
__try
{
char* p;

__try
{
p = malloc(10);

// do other job

//  期间可能抛出异常
test();

// do other job
}
__finally
{
//  这里保证资源被释放
if(p) free(p);
}
}
__except(1)
{
printf("捕获到一个异常\n");
}
}
```

呵呵！上面的代码与采用 setjmp 和 longjmp 机制实现的代码相比，是不是更简洁，更美观。这就是 try-finally 语句的贡献所在。

总结

（1） “不管在何种情况下，在离开当前的作用域时，finally 块区域内的代码都将会被执行到”，这是核心法则。

（2） try-finally 语句的作用相当于面向对象中的析构函数。

（3） goto 语句和 return 语句，在其它少数情况下，break 语句以及 continue 语句等，它们都可能会导致程序的控制流非正常顺序地离开__try 作用域，此时会发生 SEH 的“局部展开”。记住，“局部展开”会带来较大的开销，因此，程序员应该尽可能采用__leave 关键字来减少一些不必要的额外开销。

通过这几篇文章中对 SEH 异常处理机制的深入阐述，相信大家已经能够非常熟悉使用 SEH 来进行编程了。下一篇文章把 try-except 和 try-finally 机制结合起来，进行一个全面而综合的评述，继续吧！

第 25 集 SEH 的综合

SEH 模型主要包括 try-except 异常处理机制和 try-finally 结束处理机制，而且这两者能够很好地有机统一起来，它们结合使用时，能够提供给程序员非常强大、非常灵活的控制手段。其实这在上一篇文章中的几个例子中已经使用到，这里将继续进行系统的介绍，特别是 try-except 和 try-finally 结合使用时的一些细节问题。

try-except 和 try-finally 组合使用

try-except 和 try-finally 可以组合起来使用，它们可以是平行线性的关系，也可以是嵌套的关系。而且不仅是 try-except 语句中可以嵌套 try-finally 语句；同时 try-finally 语句中也可以嵌套 try-except 语句。所以它们的使用起来非常灵活，请看下面的代码：

```
// 例程 1，try-except 语句与 try-finally 语句平行关系
#include <stdio.h>
void main()
{
puts("hello");

__try
{
int* p;
puts("__try 块中");

// 下面抛出一个异常
p = 0;
*p = 25;
}
__except(1)
{
```

```
puts("__except 块中");
}

__try
{
}
__finally
{
puts("__finally 块中");
}

puts("world");
}


// 例程 2，try-except 语句中嵌套 try-finally
#include <stdio.h>

void main()
{
puts("hello");

__try
{
__try
{
int* p;
puts("__try 块中");

// 下面抛出一个异常
p = 0;
*p = 25;
}
__finally
{
// 这里会被执行吗
puts("__finally 块中");
}
}
__except(1)
{
puts("__except 块中");
}
```

```
puts("world");
}


// 例程 3，try-finally 语句中嵌套 try-except
#include <stdio.h>

void main()
{
puts("hello");

__try
{
__try
{
int* p;
puts("__try 块中");

// 下面抛出一个异常
p = 0;
*p = 25;
}
__except(1)
{
puts("__except 块中");
}
}
__finally
{
puts("__finally 块中");
}

puts("world");
}
```

try-except 和 try-finally 组合使用时，需注意的事情

在 C++异常模型中，一个 try block 块可以拥有多个 catch block 块相对应，但是在 SEH 模型中，一个__try 块只能是拥有一个__except 块或一个__finally 块相对应，例如下面的程序代码片断是存在语法错误的。

```
// 例程 1，一个__try 块，两个__except 块
#include <stdio.h>
void main()
{
puts("hello");
```

```
__try
{
int* p;
puts("__try 块中");

// 下面抛出一个异常
p = 0;
*p = 25;
}
__except(1)
{
puts("__except 块中");
}
// 这里有语法错误
__except(1)
{
puts("__except 块中");
}

puts("world");
}


// 例程 2，一个__try 块，两个__finally 块
#include <stdio.h>
void main()
{
puts("hello");

__try
{
puts("__try 块中");
}
__finally
{
puts("__finally 块中");
}
// 这里有语法错误
__finally
{
puts("__finally 块中");
}

puts("world");
}
```

```
// 例程 3，一个__try 块，对应一个__finally 块和一个__except 块
#include <stdio.h>
void main()
{
puts("hello");

__try
{
int* p;
puts("__try 块中");

// 下面抛出一个异常
p = 0;
*p = 25;
}
__finally
{
puts("__finally 块中");
}
// 这里有语法错误
__except(1)
{
puts("__except 块中");
}

puts("world");
}
```

温过而知新

这里给出最后一个简单的 try-except 和 try-finally 相结合的例子，让我们温过而知新。代码如下（这是 MSDN 中的例程）：

```
#include "stdio.h"

void test()
{
int* p = 0x00000000; // pointer to NULL

__try
{
puts("in try");
__try
```

```
{
puts("in try");

// causes an access violation exception;
//  导致一个存储异常
*p = 13;

//  呵呵，注意这条语句
puts("这里不会被执行到");
}
__finally
{
puts("in finally");
}

//  呵呵，注意这条语句
puts("这里也不会被执行到");
}
__except(puts("in filter 1"), 0)
{
puts("in except 1");
}
}

void main()
{
puts("hello");
__try
{
test();
}
__except(puts("in filter 2"), 1)
{
puts("in except 2");
}
puts("world");
}
```

上面的程序运行结果如下：

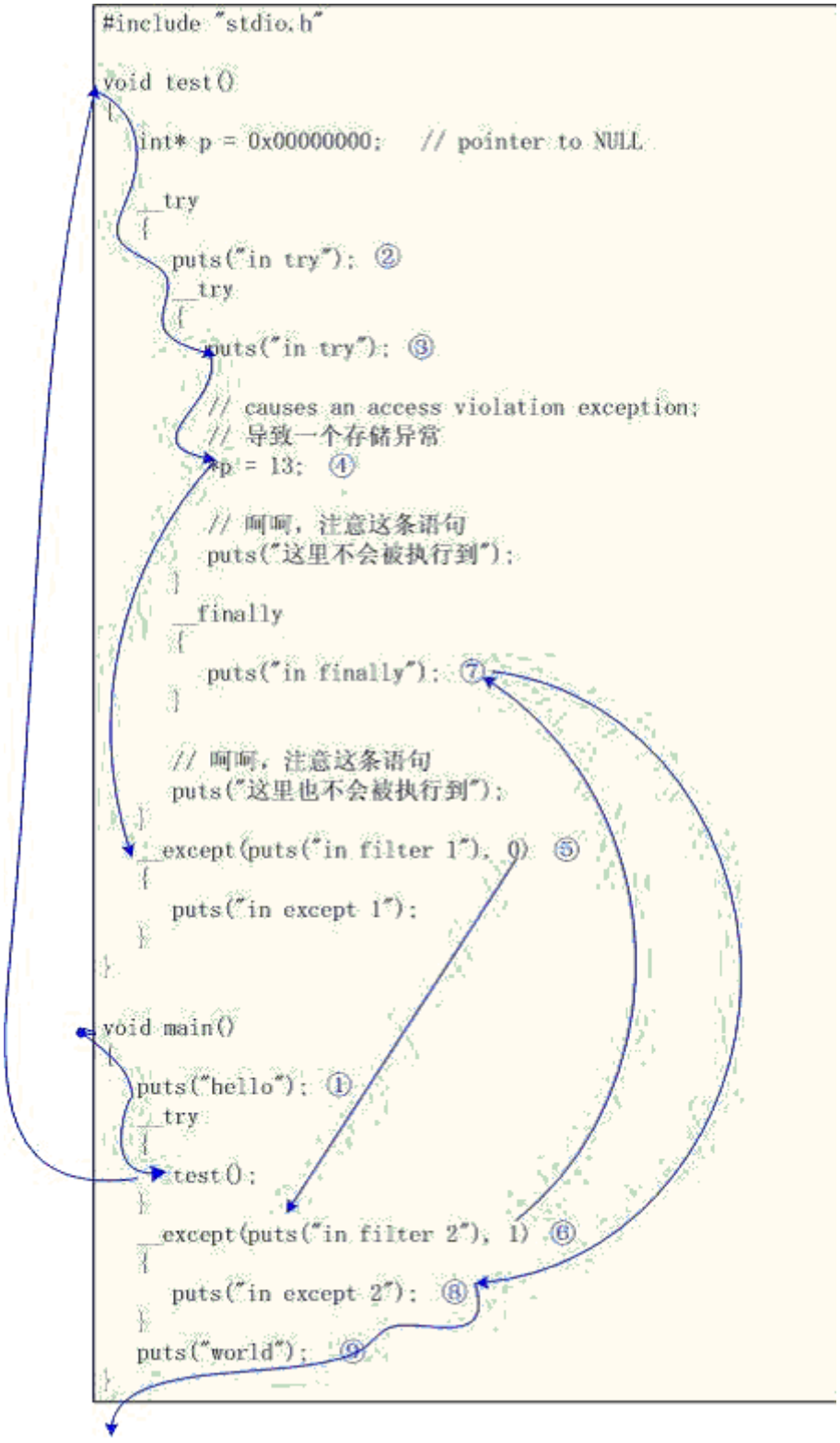
```
hello
in try
in try
in filter 1
in filter 2
in finally
```

in except 2

world

Press any key to continue

下面用图表描述一下上面例程运行时的执行流程，如下图所示。



总结

（1） try-except 和 try-finally 可以组合起来使用，它们可以是平行线性的关系，也可以是嵌套的关系。而且不仅是 try-except 语句中可以嵌套 try-finally 语句；同时 try-finally 语句中也可以嵌套 try-except 语句。

（2） 一个__try 块只能是拥有一个__except 块或一个__finally 块相对应。

至此，关于 SEH 的__try、__except、__finally、__leave 模型已经基本阐述完毕，但是主人公阿愚认为，有关 SEH 模型机制，还有一个非常关键的内容没有阐述到，那就是 SEH 与 C++异常处理模型可以结合使用吗？如果可以的话？它们组合使用时，有什么限制吗？或带来什么不良后果吗？

大家知道，Windows 平台提供的 SEH 机制虽然主要是应用于 C 语言的程序，以便第三厂商开发出高效的设备驱动程序来。但是__try、__except、__finally、__leave 模型同样也可以在 C++程序中使用，这在 MSDN 中已经提到，虽然微软还是建议，在 C++程序中尽量采用 C++的异常处理模型。

但是对于广大程序员而言，大家有必要知道，__try、__except、__finally、__leave 模型在 C++程序中使用时的一些限制。下一篇文章中，阿愚将把自己总结的一些经验和体会与大家一块分享。去看看吧！GO！

第 26 集 SEH 可以在 C++ 程序中使用

首先声明的是， C++ 中的异常处理机制是建立在 Windows 平台上的 SEH 机制之上，所以 SEH 当然可以在 C++ 程序中使用。不过“阿愚”多次强调过，我们平常一般狭义上的 SEH 都是指 try-except 和 try-finally 异常机制，而它们是给 C 语言（ VC 环境）编写 windows driver 而设计的，所以 SEH 主要应该在 C 程序中被使用，而 C++ 程序则应该使用 try-catch 机制的 C++ 异常处理模型（ micorsoft 的 MSDN 一直强烈建议程序员遵循此规则）。但是， SEH 到底能在 C++ 程序中使用吗？“当然可以，肯定可以”，其实在一开始阐述 Windows 平台多种异常机制之间的关系时，就已经清楚地表明了这一点。

这篇文章系统地来看看 SEH 在 C++ 程序中的各种使用情况。

先来一个简单的例子

其实简单的例子，就是把以前的使用 SEH 机制的 C 程序，改称 C++ 程序，看它能正常编译和运行否？什么意思，很简单，就是把原来的 .c 程序的扩展名改为 .cpp 文件，也即此时 VC 编译器会采用 C++ 语言来编译此程序（这也即为 C++ 程序了）。朋友们试试吧！代码如下：

```
// 注意，这是 C++ 程序，文件名为： SEH-test.cpp

#include "stdio.h"

void test()
```

```
{
int* p = 0x00000000; // pointer to NULL
__try
{
puts("in try");
__try
{
puts("in try");
// causes an access violation exception;
//  导致一个存储异常
*p = 13;
puts("  这里不会被执行到  ");
}
__finally
{
puts("in finally");
}
puts("  这里也不会被执行到  ");
}
__except(puts("in filter 1"), 0)
{
puts("in except 1");
}
}
```

```
void main()
{
puts("hello");
__try
{
test();
}
__except(puts("in filter 2"), 1)
{
puts("in except 2");
}
puts("world");
}
```

是不是能编译通过，而且运行结果也和以前 c 程序的运行结果完全一致，如下：

hello

in try

in try

in filter 1

in filter 2

in finally

in except 2

world

Press any key to continue

来一个真正意义上的 C++ 程序，且使用 SEH 机制

也许很多程序员朋友对上面的例子不以为然，觉得它说明不了什么问题。好的，现在我们来看一个 真正意义上的 C++ 程序，且使用 SEH 机制。什么是真正意义上的 C++ 程序，当然是采用了面向对象技术。看例子吧！代码如下（其实就是在上面程序的基础上加了一个 class ）：

```
// 注意，这是 C++ 程序，文件名为： SEH-test.cpp

#include "stdio.h"

class A

{

public:

void f1() {}

void f2() {}

};

void test1()

{

A a1;

A a2,a3;

a2.f1();

a3.f2();

}

void test()

{

int* p = 0x00000000; // pointer to NULL

__try

{
```

```
// 这里调用 test1 函数，它函数内部会创造 object
// 应该属于 100% 的 C++ 程序了吧！
test1();
puts("in try");
__try
{
puts("in try");
// causes an access violation exception;
// 导致一个存储异常
*p = 13;
puts(" 这里不会被执行到 ");
}
__finally
{
puts("in finally");
}
puts(" 这里也不会被执行到 ");
}
__except(puts("in filter 1"), 0)
{
puts("in except 1");
}
}
void main()
```

```
{
puts("hello");
__try
{
test();
}
__except(puts("in filter 2"), 1)
{
puts("in except 2");
}
puts("world");
}
```

总结

通过以上实践得知， SEH 的确可以在 C++ 程序中使用，而且 SEH 不仅可以在 C++ 程序中使用；更进一步， SEH 异常机制（ try-except 和 try-finally ）还可以与 C++ 异常处理模型（ try-catch ），两者在同一个 C++ 程序中混合使用，这在下一篇文章中接着讨论。

但问题是，微软 MSDN 的忠告（ C 程序中使用 try-except 和 try-finally ；而 C++ 程序则应该使用 try-catch ），这岂不是完全属于吓唬人吗？非也！非也！微软的建议一点也没有错， SEH 与 C++ 异常处理机制混合使用时，的确有一定的约束（虽然，平时我们很少关心这一点），这同样也在下一篇文章中详细接着讨论， 程序员朋友们，继续吧！

第 27 集 SEH 与 C++ 异常模型的混合使用

在上一篇文章中我们看到了，在 C++ 程序中可以能够很好地使用 SEH 的 try-except 和 try-finally 机制（虽然 MSDN 中不建议这样做），这一篇文章中我们继续讨论，在 C++ 程序中同时使用 try-except 异常机制（ SEH ）和 try-catch 异常机制（ C++ 异常模型 ）的情况。

朋友们，准备好了心情吗？这可是有点复杂呦！

如何混合使用呢？

同样，还是看例子先。仍然是在原来例程的代码基础上做修改，修改后的代码如下：

```
// 注意，这是 C++ 程序，文件名为： SEH-test.cpp
#include "stdio.h"
class A
{
public:
void f1() {}
// 抛出 C++ 异常
void f2() { throw 888;}
};
// 这个函数中使用了 try-catch 处理异常，也即 C++ 异常处理
void test1()
{
A a1;
A a2,a3;
try
{
a2.f1();
a3.f2();
}
catch(int errorcode)
{
printf("catch exception,error code:%d\n", errorcode);
}
```

```
}
// 这个函数没什么改变，仍然采用 try-except 异常机制，也即 SEH 机制
void test()
{
int* p = 0x00000000; // pointer to NULL
__try
{
// 这里调用 test1 函数
test1();
puts("in try");
__try
{
puts("in try");
// causes an access violation exception;
// 导致一个存储异常
*p = 13;
puts(" 这里不会被执行到 ");
}
__finally
{
puts("in finally");
}
puts(" 这里也不会被执行到 ");
}
```

```
__except(puts("in filter 1"), 0)
{
puts("in except 1");
}
}
void main()
{
puts("hello");
__try
{
test();
}
__except(puts("in filter 2"), 1)
{
puts("in except 2");
}
puts("world");
}
```

上面程序不仅能够被编译通过，而且运行结果也是正确的（和预期的一样，同样符合 C++ 异常处理模型的规则，和 SEH 异常模型的处理规则）。其结果如下：

hello

catch exception,error code:888

in try

in try

in filter 1

in filter 2

in finally

in except 2

world

Press any key to continue

继续深入刚才的例子

上面的例程中，虽然在同一个程序中既有 try-catch 机制，又有 try-except 机制，当然这也完全算得上 SEH 与 C++ 异常模型的混合使用 。但是，请注意，这两种机制其实是完全被分割开的，它们完全被分割在了两个函数的内部。也即这两种机制其实并没有完全交互起来，换句话说，它们还算不上两种异常处理机制真正的混合使用。这里继续给出一个更绝的例子，还是先来看看代码吧，如下：

```
// 注意，这是 C++ 程序，文件名为： SEH-test.cpp

#include "stdio.h"

class MyException

{

public:

MyException() {printf(" 构造一个 MyException 对象 \n");}

MyException(const MyException& e) {printf(" 复制一个 MyException 对象 \n");}

operator=(const MyException& e) {printf(" 复制一个 MyException 对象 \n");}

~MyException() {printf(" 析构一个 MyException 对象 \n");}

};

class A

{

public:

A() {printf(" 构造一个 A 对象 \n");}
```

```
~A() {printf(" 析构一个 A 对象 \n");}

void f1() {}

// 注意，这里抛出了一个 MyException 类型的异常对象

void f2() {MyException e; throw e;}

};

// 这个函数中使用了 try-catch 处理异常，也即 C++ 异常处理

void test1()

{

A a1;

A a2,a3;

try

{

a2.f1();

a3.f2();

}

// 这里虽然有 catch 块，但是它捕获不到上面抛出的 C++ 异常对象

catch(int errorcode)

{

printf("catch exception,error code:%d\n", errorcode);

}

}

// 这个函数没什么改变，仍然采用 try-except 异常机制，也即 SEH 机制

void test()

{
```

```
int* p = 0x00000000; // pointer to NULL

__try

{

// 这里调用 test1 函数

// 注意， test1 函数中会抛出一个 C++ 异常对象

test1();

puts("in try");

__try

{

puts("in try");

*p = 13;

puts(" 这里不会被执行到 ");

}

__finally

{

puts("in finally");

}

puts(" 这里也不会被执行到 ");

}

__except(puts("in filter 1"), 0)

{

puts("in except 1");

}

}
```

```
void main()

{

puts("hello");

__try

{

test();

}

// 这里能捕获到 C++ 异常对象吗？拭目以待吧！

__except(puts("in filter 2"), 1)

{

puts("in except 2");

}

puts("world");

}
```

仔细阅读上面的程序，不难看出， SEH 与 C++ 异常模型两种机制确实真正地交互起来了，上层的 main() 函数和 test() 函数采用 try-except 语句处理异常，而下层的 test1() 函数采用标准的 try-catch 语句处理异常，并且，下层的 test1() 函数所抛出的 C++ 异常会被上层的 try-except 所捕获到吗？还是看运行结果吧！如下：

```
hello

构造一个 A 对象

构造一个 A 对象

构造一个 A 对象

构造一个 MyException 对象

复制一个 MyException 对象

in filter 1
```

in filter 2

析构一个 MyException 对象

析构一个 A 对象

析构一个 A 对象

析构一个 A 对象

in except 2

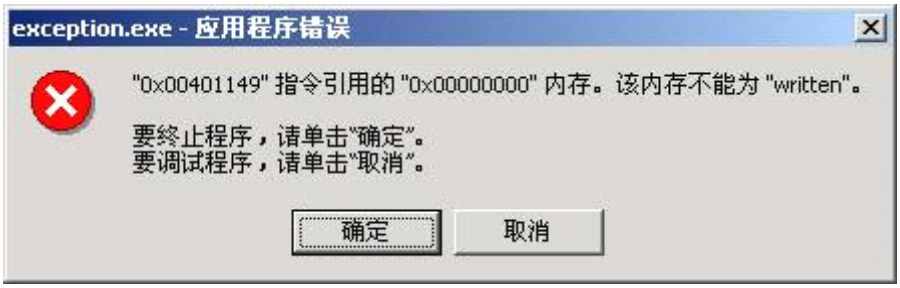
world

Press any key to continue

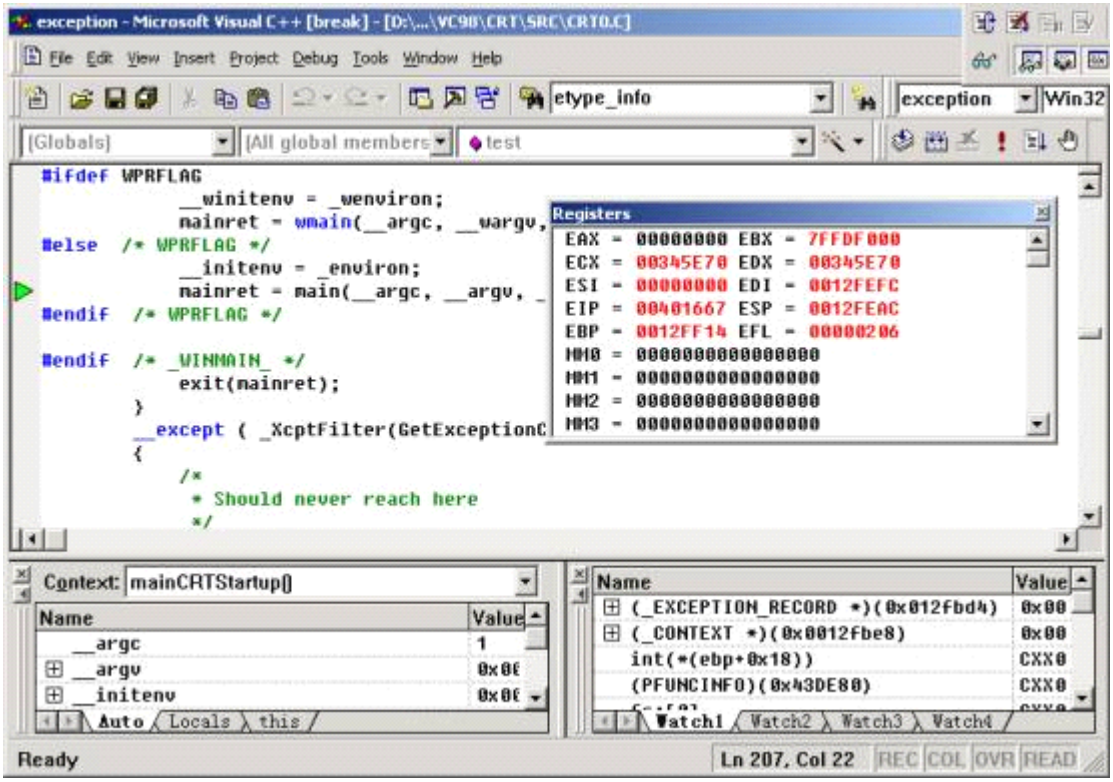
结果是否和朋友们的预期一致呢？它的的确确是上层的 try-except 块，能够捕获到下层的 test1() 函数所抛出的 C++ 异常，而且流程还是正确的，即符合了 SEH 异常模型的规则，又同时遵循了 C++ 异常模型的规则。同时，最难能可贵的是，在 test1() 函数中的三个局部变量，都被正确的析构了（这非常神奇吧！具体的机制这里暂且不详细论述了，在后面阐述“异常机制的实现”的文章中再做论述）。

细心的程序员朋友们，也许从上面程序的运行结果中发现了一些“问题”，什么呢？那就是“ *MyException 对象* ”构造了两次，但它只被析构了一次。呵呵！这也许就是 MSDN 中不建议混合使用这两种异常处理机制的背后原因之一吧！虽然说，这种问题不至于对整个程序造成很大的破坏性影响，但主人公阿愚却坚持认为，如果我们编程时滥用 try-except 和 try-catch 在一起混用，不仅使我们程序的整体结构和语义受到影响，而且也会造成一定的内存资源泄漏，甚至其它的不稳定因素。

总之，在 C++ 程序中运用 try-except 机制，只有在顶层的函数作用域中（例如，系统运行库中，或 plugin 的钩子中）才有必要这样做。如在 VC 编写的程序中，每当我们程序运行中出现意外异常导致的崩溃事件时，系统总能够弹出一个“应用程序错误”框，如下：



NT 操作系统是如何实现的呢？很简单，它就是在在 VC 运行库中的 顶层的函数内采用了 try-except 机制，不信，看看如下截图代码吧！



C++ 异常处理模型能捕获 SEH 异常吗?

呵呵！阿愚笑了，这还用问吗？当然了， VC 提供的 C++ 异常处理模型的强大之处就是，它不仅能捕获 C++ 类型的异常，而且它还能捕获属于系统级别的 SEH 异常。它就是利用了 catch(…) 语法，在前面专门阐述 catch(…) 语法时，我们也着重论述了这一点。不过，这里还是给出一个实际的例子吧，代码如下：

```
class MyException

{

public:

MyException() {printf(" 构造一个 MyException 对象 \n");}

MyException(const MyException& e) {printf(" 复制一个 MyException 对象 \n");}

operator=(const MyException& e) {printf(" 复制一个 MyException 对象 \n");}

~MyException() {printf(" 析构一个 MyException 对象 \n");}

};

class A

{
```

```
public:
A() {printf(" 构造一个 A 对象 \n");}
~A() {printf(" 析构一个 A 对象 \n");}
void f1() {}
// 抛出 C++ 异常
void f2() {MyException e; throw e;}
};
void test()
{
int* p = 0x00000000; // pointer to NULL
__try
{
puts("in try");
__try
{
puts("in try");
// causes an access violation exception;
// 导致一个存储异常
*p = 13;
// 呵呵，注意这条语句
puts(" 这里不会被执行到 ");
}
__finally
{
```

```
puts("in finally");

}

// 呵呵，注意这条语句

puts(" 这里也不会被执行到 ");

}

__except(puts("in filter 1"), 0)

{

puts("in except 1");

}

}

void test1()

{

A a1;

A a2,a3;

try

{

// 这里会产生一个 SEH 类型的系统异常

test();

a2.f1();

a3.f2();

}

// 捕获得到吗？

catch(...)

{
```

```
printf("catch unknown exception\n");

}

}

void main()

{

puts("hello");

__try

{

test1();

}

__except(puts("in filter 2"), 0)

{

puts("in except 2");

}

puts("world");

}
```

上面的程序很简单的，无须进一步讨论了。当然，其实我们还可以更进一步深入进去，因为 C++ 异常处理模型不仅能够正常捕获到 SEH 类型的系统异常，而且它还能够把 SEH 类型的系统异常转化为 C++ 类型的异常。我想，这应该放在单独的一篇文章中来阐述了，其实这在许多关于 Window 系统编程的书籍中也有详细讨论。

SEH 与 C++ 异常模型在混合使用时的“禁区”

刚才我们看到，利用 try-except 来捕获 C++ 异常有点小问题，但这毕竟算不上什么禁区。那么，何为 SEH 与 C++ 异常模型在混合使用时的“禁区”呢？看个例子吧，代码如下：

```
// 注意，这是 C++ 程序，文件名为： SEH-test.cpp

#include "stdio.h"
```

```
void main()

{

int* p = 0x00000000; // pointer to NULL

// 这里是 SEH 的异常处理语法

__try

{

puts("in try");

// 这里是 C++ 的异常处理语法

try

{

puts("in try");

// causes an access violation exception;

// 导致一个存储异常

*p = 13;

// 呵呵，注意这条语句

puts(" 这里不会被执行到 ");

}

catch(...)

{

puts("catch anything");

}

// 呵呵，注意这条语句

puts(" 这里也不会被执行到 ");

}
```

```
__except(puts("in filter 1"), 1)

{

puts("in except 1");

}

}
```

朋友们！不要急于编译并测试上面的小程序，先猜猜它会有什么结果呢？想到了吗？不妨实践一下，呵呵！实际结果是否令你吃惊呢？对了，没错， VC 就是会报出一个编译错误（“ error C2713: Only one form of exception handling permitted per function ”）。那么原因何在呢？主人公阿愚在此一定“知无不言，言无不尽”，这是因为： VC 实现的异常处理机制，不管是 try-except 模型，还是 try-catch 模型，它们都是以函数作为一个最基本“分析和控制”的目标，也即，如果一个函数内使用了异常处理机制， VC 编译器在编译该函数时，它会给此函数插入一些“代码和信息”（代码指的是当该函数中出现异常时的回调函数，而信息主要是指与异常出现相关的一些必要的链表），因此每份函数只能有一份这样的东东（“代码和信息”），故一个函数只能采用一种形式的异常处理规则。 朋友们！恍然大悟了吧！其实这倒不算最令人不可思议的。还有一种更为特殊的情况，看下面的例子，代码如下：

```
class A

{

public:

A() {printf(" 构造一个 A 对象 \n");}

~A() {printf(" 析构一个 A 对象 \n");}

void f1() {}

void f2() {}

};

void main()

{

__try

{

A a1, a2;
```

```
puts("in try");

}

__except(puts("in filter 1"), 1)

{

puts("in except 1");

}

}
```

其实上面的程序表面上看，好像是没有什么特别的吗？朋友们！仔细看看，真的没什么特别的吗？不妨编译一下该程序，又奇怪了吧！是的，它同样也编译报错了，这是我机器上编译时产生的信息，如下：

```
--------------------Configuration: exception - Win32 Debug--------------------

Compiling...

seh-test.cpp

f:\exception\seh-test.cpp(214) : warning C4509: nonstandard extension used: 'main' uses SEH and 'a2' has destructor

f:\exception\seh-test.cpp(211) : see declaration of 'a2'

f:\exception\seh-test.cpp(214) : warning C4509: nonstandard extension used: 'main' uses SEH and 'a1' has destructor

f:\exception\seh-test.cpp(211) : see declaration of 'a1'

f:\exception\seh-test.cpp(219) : error C2712: Cannot use __try in functions that require object unwinding

Error executing cl.exe.

Creating browse info file...

exception.exe - 1 error(s), 2 warning(s)
```

那么，上面的错误信息代表什么意思，我想是不是有不少朋友都遇到过这种莫名奇妙的编译问题。其实，这确实很令人费解，明明程序很简单的吗？而且程序中只用到了 SEH 异常模型的 try-except 语法，甚至 SEH 与 C++ 异常模型两者混合使用的情况都不存在。那么编译出错的原因究竟何在呢？实话告诉你吧！其实主人公阿愚在此问题上也是费透了脑筋，经过了一番深入而细致的钻研之后，才知道真正原因的。

那原因就是： 同样还是由于在一个函数不能采用两种形式的异常处理机制而导致的编译错误。 啊！这岂不是更迷惑了。其实不然，说穿了，朋友们就会明白了，这是因为： 在 C++ 异常处理模型中，为了能够在异常发生后，保证正确地释放相关的局部变量（也即调用析构函数），它必须要跟踪每一个“对象”的创建过程，这种由于异常产生而导致的对象析构的过程，被称为“ unwind ”（记得前面的内容中，也多次讲述到了这个名词），因此，如果一个函数中有局部对象的存在，那么它就一定会存在 C++ 的异常处理机制（也即会给此函数插入一些用于 C++ 异常处理“代码和信息”），这样，如果该函数中在再使用 try-except 机制，岂不是就冲突了吗？所以编译器也就报错了，因为它处理不了了！ 哈哈！朋友们，主人公阿愚把问题说清楚了吗？

总结

• SEH 与 C++ 异常模型，可以在一起被混合使用。但最好听从 MSDN 的建议：在 C 程序中使用 try-except 和 try-finally ；而 C++ 程序则应该使用 try-catch 。

• 混合使用时， C++ 异常模型可以捕获 SEH 异常；而 SEH 异常模型也可以捕获 C++ 类型的异常。而后者通常有点小问题，它一般主要运用在提高和保证产品的可靠性上（也即在顶层函数中使用 try-except 语句来 catch 任何异常）

• VC 实现的异常处理机制中，不管是 try-except 模型，还是 try-catch 模型，它们都是以函数作为一个最基本“分析和控制”的目标，也即一个函数中只能采用一种形式的异常处理规则。否则，编译这一关就会被“卡壳”。

下一篇文章中，主人公阿愚打算接着详细讨论一些关于 C++ 异常处理模型的高级使用技巧，也即 “ 如何把 SEH 类型的系统异常转化为 C++ 类型的异常？ ”， 程序员朋友们，继续吧！

第 28 集 如何把 SEH 类型的系统异常转化为 C++类型的异常？

在上一篇文章中，详细讨论了“ SEH 与 C++ 异常模型的混合使用”，这一篇文章中，主人公阿愚仍将这一主题继续深入，那就是“如何把 SEH 类型的系统异常转化为 C++ 类型的异常？”（其实，这本质上仍然属于 SEH 与 C++ 异常模型的混合使用，也即 C++ 异常模型来捕获 SEH 系统类型的异常）。

为什么要 把 SEH 类型的系统异常转化为 C++ 类型的异常？

做一件事情之前，我们最好要搞清为什么！“十万个为什么” 可曾造就了多少顶级奇才！呵呵！ WHY？？ WHY ？ WHY ？这对任何一个人来说，都绝对是个好习惯，阿愚同学就一直把这个当“宝贝”。那么，究竟 为什么要 把 SEH 类型的系统异常转化为 C++ 类型的异常？朋友们，大家都想想，整理整理自己的意见和想法。这里，阿愚给出它个人的理解，如下：

• 首先是由于我们在编程时，仍然最好遵循 MSDN 给出的建议和忠告（“ C 程序中使用 try-except 和 try-finally ；而 C++ 程序则应该使用 try-catch ”）。但是，为了提高程序的可靠性（防止意外系统异常的出现，而导致的程序无规则崩溃现象），我们还必须要求在 C++ 程序中，使用 try-catch 模型 能够捕获

并 处理系统异常， 这在第 3 集的文章中曾详细讨论过了它， 那就是它只能采用 catch(…) 语法来捕获 SEH 类型的系统异常。 catch(…) 的使用 虽然一定程度上提高了程序的可靠性，但是，“异常发生时的许多相关信息”它却什么也没有提供给程序员（包括何种类型的系统异常，出现的地点，以及其它有关异常的信息等等）。因此，我们需要一种有效途径，来把 SEH 类型的系统异常转化为 C++ 类型的异常？这无疑也就提供了一种在 C++ 异常处理模型中，不仅能够处理系统异常，而且还能够获取有关 SEH 类型系统异常中的许多详细信息的手段。

• 其次就是，阿愚多次阐述过， C++ 异常处理是和面向对象紧密联系的（它二位可是“哥俩好”），因此，如果把 SEH 类型的系统异常统一到面向对象体系结构设计的“异常分类”中去，那对程序员而言，岂不是妙哉！美哉！该解决方案真所谓是，即提高了可靠性；又不失优雅！

如何实现 把 SEH 类型的系统异常转化为 C++ 类型的异常？

虽然说把 SEH 类型的系统异常转化为 C++ 类型的异常，给 C++ 程序员带来的是好处多多，但是实现起来并不复杂，因为系统底层和 VC 运行库已经为我们铺路搭桥了，也即我们可以通过 VC 运行库中的“ _ set_se_translator ”函数来轻松搞定它。 MSDN 中对它解释如下：

Handles Win32 exceptions (C structured exceptions) as C++ typed exceptions.

typedef void (*_se_translator_function)(unsigned int, struct _EXCEPTION_POINTERS*);

_se_translator_function _set_se_translator(_se_translator_function *se_trans_func* **);**

*The **_set_se_translator** function provides a way to handle Win32 exceptions (C structured exceptions) as C++ typed exceptions. To allow each C exception to be handled by a C++ **catch** handler, first define a C exception "wrapper" class that can be used, or derived from, in order to attribute a specific class type to a C exception. To use this class, install a custom C exception translator function that is called by the internal exception-handling mechanism each time a C exception is raised. Within your translator function, you can throw any typed exception that can be caught by a matching C++ **catch** handler.*

*To specify a custom translation function, call **_set_se_translator** with the name of your translation function as its argument. The translator function that you write is called once for each function invocation on the stack that has **try** blocks. There is no default translator function.*

In a multithreaded environment, translator functions are maintained separately for each thread. Each new thread needs to install its own translator function. Thus, each thread is in charge of its own translation handling.

*The se_trans_func function that you write must take an unsigned integer and a pointer to a Win32 **_EXCEPTION_POINTERS** structure as arguments. The arguments are the return values of calls to the Win32 API **GetExceptionCode** and **GetExceptionInformation** functions, respectively.*

至于 _ set_se_translator 函数的具体机制，以及 把系统异常转化为 C++ 类型的异常的原理这里不再详细讨论，而仅仅是给出了大致的工作流程： 首先，通过 _ set_se_translator 函数设置一个对所有的 Windows 系统异常产生作用的回调处理函数（也是与 TLS 数据有关）；因此，每当程序运行时产生了系统异常之

后，前面我们设置的自定义回调函数于是便会接受程序的控制权；接着，我们在该函数的实现中，可以根据不同类型的系统异常（ EXCEPTION_POINTERS ），来分别抛出一个 C++ 类型的异常错误 。呵呵！简单吧！不再白话了，还是来瞧瞧阿愚所设计的一个简单演示例程吧！代码如下：

```
// FILENAME:SEH-test.cpp

#include <windows.h>

#include <cstdio>

#include <eh.h>

#include <string>

#include <exception>

using namespace std;

//////////////////////////////////////////////////////////////////////////////

class seh_exception_base : public std::exception

{

public:

seh_exception_base (const PEXCEPTION_POINTERS pExp, std::string what )

: m_ExceptionRecord(*pExp->ExceptionRecord),

m_ContextRecord(*pExp->ContextRecord),

m_what(what){};

~seh_exception_base() throw(){} ;

virtual const char* what() const throw()

{

return m_what.c_str();

}

virtual DWORD exception_code() const throw()

{
```

```
return m_ExceptionRecord.ExceptionCode;

}

virtual const EXCEPTION_RECORD& get_exception_record() const throw()

{

return m_ExceptionRecord;

}

virtual const CONTEXT& get_context() const throw()

{

return m_ContextRecord;

}

// 初始化函数

static void initialize_seh_trans_to_ce()

{

_set_se_translator( trans_func );

}

// 系统异常出现时的回调函数

static void trans_func( unsigned int u, EXCEPTION_POINTERS* pExp );

protected:

std::string m_what;

EXCEPTION_RECORD m_ExceptionRecord;

CONTEXT m_ContextRecord;

};

//////////////////////////////////////////////////////////////////////////////

// 下面是系统异常被转换后的 C++ 类型的异常
```

```
// 篇幅有限，因此只简单设计了对几种常见的系统异常的转换

//////////////////////////////////////////////////////////////////////////

class seh_exception_access_violation : public seh_exception_base

{

public:

seh_exception_access_violation (const PEXCEPTION_POINTERS pExp, std::string what)

: seh_exception_base(pExp, what) {};

~seh_exception_access_violation() throw(){};

};

//////////////////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////////////////

class seh_exception_divide_by_zero : public seh_exception_base

{

public:

seh_exception_divide_by_zero (const PEXCEPTION_POINTERS pExp, std::string what)

: seh_exception_base(pExp, what) {};

~seh_exception_divide_by_zero() throw(){};

};

//////////////////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////////////////

class seh_exception_invalid_handle : public seh_exception_base

{

public:

seh_exception_invalid_handle (const PEXCEPTION_POINTERS pExp, std::string what)
```

```
: seh_exception_base(pExp, what) {};

~seh_exception_invalid_handle() throw(){};

};

////////////////////////////////////////////////////////////////////////////

// 系统异常出现时的回调函数

// 这里是实现，很关键。针对不同的异常，抛出一个 C++ 类型的异常

void seh_exception_base::trans_func( unsigned int u, EXCEPTION_POINTERS* pExp )

{

switch(pExp->ExceptionRecord->ExceptionCode)

{

case EXCEPTION_ACCESS_VIOLATION :

throw seh_exception_access_violation(pExp, " 存储保护异常 ");

break;

case EXCEPTION_INT_DIVIDE_BY_ZERO :

throw seh_exception_divide_by_zero(pExp, " 被 0 除异常 ");

break;

case EXCEPTION_INVALID_HANDLE :

throw seh_exception_invalid_handle(pExp, " 无效句病异常 ");

break;

default :

throw seh_exception_base(pExp, " 其它 SEH 异常 ");

break;

}

}
```

```
// 来测试吧！

void main( void )

{

seh_exception_base::initialize_seh_trans_to_ce();

try

{

// 被 0 除

int x, y=0;

x = 5 / y;

// 存储保护

char* p =0;

*p = 0;

// 其它系统异常，例如中断异常

__asm int 3;

}

catch( seh_exception_access_violation& e )

{

printf( "Caught SEH_Exception. 错误原因： %s\n", e.what());

//other processing

}

catch( seh_exception_divide_by_zero& e )

{

printf( "Caught SEH_Exception. 错误原因： %s\n", e.what());

//other processing
```

```
}

catch( seh_exception_base& e )

{

printf( "Caught SEH_Exception. 错误原因： %s, 错误代码： %x\n", e.what(), e.exception_code());

}

}
```

总结

至此，关于 C++ 异常处理模型、 C 语言中的各种异常处理机制， Window 系统下的 SEH 原理，以及它们在使用上的各种技巧，和程序员在编程时使用这些异常处理机制时的注意事项等等，前面的相关文章中都做了较系统和深入的讨论和分析（也许错误多多！欢迎指出和讨论）。相信不仅是主人公阿愚，而且包括那些细细地读过、品味过，和阿愚一块走过来的人，现在都有了一种豁然开朗的感觉。感谢各种异常处理机制！感谢那些设计如此精妙技术和作品的天才！感谢天下所有有作为的程序员！

Java 语言中的异常处理模型是踩着 C++ 异常处理模型肩膀上过来的。它继承了 C++ 异常处理模型的风格和优点，同时，它也比 C++ 异常处理模型更安全，更可高，更强大和更丰富。下一篇文章中，主人公阿愚打算乘胜追击，来讨论一些关于 Java 语言中的异常处理模型。感兴趣的 朋友们，继续吧！

第 29 集 Java 语言中的异常处理模型

对于一个非常熟悉 C++ 异常处理模型的程序员来说，它几乎可以不经任何其它培训和学习，就可以完全接受和能够轻松地使用 Java 语言中的异常处理编程方法。这是因为 Java 语言中的异常处理模型几乎与 C++ 中异常处理模型有 99% 的相似度，无论是从语法规则，还是语义上来说，它们二者都几乎完全一致。

当然，如果你对 Java 语言中的异常处理模型有更多，或更深入的了解，你还是能够发现 Java 异常处理模型与 C++ 中异常处理模型还是存在不少差别的。是的， Java 语言本来就是 C++ 语言的完善改进版，所以， Java 语言中的异常处理模型也就必然会继承了 C++ 异常处理模型的风格和优点。但是，好的东西不仅仅是需要继承优点，更重要的是需要“去其糟粕，取其精华”，需要发展！！！毫无疑问， Java 语言中的异常处理模型完全达到了这一“发展”高度。它比 C++ 异常处理模型更安全，更可高，更强大和更丰富。

下面的内容中，阿愚不打算再向大家详细介绍一遍有关 Java 异常处理编程的具体语法和规则。因为这与 C++ 中的异常处理几乎完全一样，而且这些基础知识在太多太多的有关 java 编程的书籍中都有详细阐述。而阿愚认为： 这里更需要的是总结，需要的是比较，需要的是重点突出，需要的是关键之处 。所以，下面着重把 Java 语言中的异常处理模型与 C++ 异常处理模型展开比较，让我们透彻分析它到底有何发展？有何优势？与 C++ 异常处理模型到底有哪些细节上的不同？又为什么要这样做？

借鉴并引进了 SEH 异常模型中的 try-finally 语法

要说 Java 异常处理模型与 C++ 中异常处理模型的最大不同之处，那就是在 Java 异常处理模型中引入了 try-finally 语法，阿愚认为这是从微软的 SEH 借鉴而来。在前面的一些文章中，详细而深入阐述 SEH 异常处理模型的时候，我们从中获知， SEH 主要是为 C 语言而设计的，便于第三厂商开发的 Window 驱动程序有更好更高的安全保障。同时， SEH 异常处理模型中除了 try-except 来用于处理异常外，还有一个 try-finally 语法，它主要用来清除一些曾经分配的资源（由于异常出现，而导致这些资源不能够按正常的顺序被释放，还记得吗？这被称为“ UNWIND ”）， try-finally 本质上有点类似于面向对象编程中的析构函数的作用，由于这项机制的存在，才导致 SEH 的强大和风光无比。

现在， Java 异常处理模型也吸收了这项设计。可我们知道，无论是 JAVA 中，还是 C++ 中，它们都有“析构函数”呀！它们完全可以利用面向的析构函数来自动释放资源呀！是的，没错！理论上是这样的。可是在实践中，我们也许会发现或经常碰到，仅仅利用析构函数来释放资源，并不是那么好使，例如，我们经常需要动态得从堆上分配的对象，这时，释放对象必须要显式地调用 delete 函数来触发该对象的析构函数的执行。如果这期间发生了异常，不仅该对象在堆中的内存不能达到被释放的结果，而且，该对象的析构函数中释放更多资源的一些代码也不能得以执行。因此这种后果非常严重，这也算 C++ 异常处理模型中一种比较大的缺陷吧！（虽然， C++ 有其它补救措施，那就是利用“智能指针”来保证一些对象的及时而有效地被释放）。另外，其实很有许多类似的例子，例如关闭一个内核句柄等操作（ CloseHandle ）。

在 Java 语言中，由于它由于采用了垃圾回收技术，所以它能有效避免上述所讲的类似于 C++ 语言中的由于异常出现所导致的那种资源不能得以正确释放的尴尬局面。但是，它仍然还是在它的异常处理模型中引入了 try-finally 语法，因为，无论如何，它至少会给程序员带来了极大的方便，例如如下的程序片断，就可以充分反映出 try-finally 对提高我们代码的质量和美观是多么的重要。

```
import java.io.*;

public class Trans

{

public static void main(String[] args)

{

try

{

BufferedReader rd=null;

Writer wr=null;

try

{

File srcFile = new File((args[0]));
```

```
File dstFile = new File((args[1]));

rd = new BufferedReader(new InputStreamReader(new FileInputStream(srcFile), args[2]));

wr = new OutputStreamWriter(new FileOutputStream(dstFile), args[3]);

while(true)

{

String sLine = rd.readLine();

if(sLine == null) break;

wr.write(sLine);

wr.write("\r\n");

}

}

finally

{

// 这里能保证在何种情况下，文件流的句柄都得以被正确关闭

// 该方法主要用于清理非内存性质的资源（垃圾回收机制无法

// 处理的资源，如数据库连接、 Socket 关闭、文件关闭等等）。

wr.flush();

wr.close();

rd.close();

}

}

catch(Exception ex)

{

ex.printStackTrace();
```

```
}

}

}
```

所有的异常都必须从 Throwable 继承而来

在 C++ 异常处理模型中，它给予程序员最大的自由度和发挥空间（这与 C++ 中所主导的设计思想是一致的），例如它允许程序员可以抛出任何它想要的异常对象，它可以是语言系统中原本所提供的各种简单数据类型（如 int 、 float 、 double 等），也可以是用户自定义的抽象数据对象（如 class 的 object 实例）。虽然说，无论从何个角度上考量，我们都应该把异常处理与面向对象紧密结合起来（采用带有继承特点的层次化的类对象结构来描述系统中的各种异常），但是 C++ 语言规范中并无此约束；况且，即便大家都不约而同地采用面向对象的方法来描述“异常”，但也会由于各个子系统（基础运行库）不是一个厂商（某一个人）所统一设计，所以导致每个子系统所设计出的异常对象系统彼此相差甚远。这给最终使用（重用）这些库的程序员带来了很大的不一致性，甚至是很大的麻烦，我们不仅需要花费很多很宝贵的时间来学习和熟悉这些不同的异常对象子系统；而且更大的问题是，这些不同子系统之间语义上的不一致，而造成程序员在最终处理这些异常时，将很难把它们统一起来，例如， MFC 库系统中，采用 CMemoryException 来表示一个与内存操作相关的异常；而其它的库系统中很可能就会采用另外一个 class 来表示内存操作的异常错误。本质上说，这是由于缺乏规范和统一所造成的恶劣后果，所以说，如果在语言设计的时候，就充分考虑到这些问题，把它们纳入语言的统一规范之中，这对广大的程序员来说，无疑是个天大的好事情。

Java 语言毫无疑问很好地做到了这一点，它要求 java 程序中（无论是谁写的代码），所有抛出（ throw ）的异常都必须是从 Throwable 派生而来，例如，下面的代码编译时就无法通过。

```
import java.io.*;

public class Trans

{

public static void main(String[] args)

{

try

{

BufferedReader rd=null;

Writer wr=null;

try
```

```
{
File srcFile = new File((args[0]));
File dstFile = new File((args[1]));
rd = new BufferedReader(new InputStreamReader(new FileInputStream(srcFile), args[2]));
wr = new OutputStreamWriter(new FileOutputStream(dstFile), args[3]);
// 编译时，这里将被报错
// 而正确的做法可以是： throw new Exception("error! test!");
if (rd == null || wr == null) throw new String("error! test!");
while(true)
{
String sLine = rd.readLine();
if(sLine == null) break;
wr.write(sLine);
wr.write("\r\n");
}
}
finally
{
wr.flush();
wr.close();
rd.close();
}
}
catch(Exception ex)
```

```
{
ex.printStackTrace();
}
}
}
```

编译时，输出的错误信息如下：

E:\work\\Trans.java:20: incompatible types

found : java.lang.String

required: java.lang.Throwable

if (rd == null || wr == null) throw new String("error! test!");

1 error

当然，实际的 Java 编程中，由于 JDK 平台已经为我们设计好了非常丰富和完整的异常对象分类模型。因此， java 程序员一般是不需要再重新定义自己的异常对象。而且即便是需要扩展自定义的异常对象，也往往会从 Exception 派生而来。所以，对于 java 程序员而言，它一般只需要在它的顶级函数中 catch(Exception ex) 就可以捕获出所有的异常对象，而不必像 C++ 中采用 catch(…) 那样不伦不类，但又无可奈何的语法。因此，在 java 中也就不需要（也没有了） catch(…) 这样的语法。

至于 JDK 平台中的具体的异常对象分类模型，主人公阿愚打算放在另外单独的一篇文章中详细讨论，这里只简单的概括一下： 所有异常对象的根基类是 Throwable ， Throwable 从 Object 直接继承而来（这是 java 系统所强制要求的），并且它实现了 Serializable 接口（这为所有的异常对象都能够轻松跨越 Java 组件系统做好了最充分的物质准备）。从 Throwable 直接派生出的异常类有 Exception 和 Error 。 Exception 是 java 程序员所最熟悉的，它一般代表了真正实际意义上的异常对象的根基类。也即是说， Exception 和从它派生而来的所有异常都是应用程序能够 catch 到的，并且可以进行异常错误恢复处理的异常类型。而 Error 则表示 Java 系统中出现了一个非常严重的异常错误，并且这个错误可能是应用程序所不能恢复的，例如 LinkageError ，或 ThreadDeath 等。

对异常处理的管理更严格，也更严谨！

同样还是与 C++ 异常处理模型作比较，在 Java 系统中，它对异常处理的管理更严格，也更严谨！为什么这么说呢？下面请听阿愚娓娓道来！

首先还是看一个例子吧！代码如下：

```
import java.io.*;

public class Trans

{

public static void main(String[] args)

{

try

{

BufferedReader rd=null;

Writer wr=null;

try

{

File srcFile = new File((args[0]));

File dstFile = new File((args[1]));

rd = new BufferedReader(new InputStreamReader(new FileInputStream(srcFile), args[2]));

wr = new OutputStreamWriter(new FileOutputStream(dstFile), args[3]);

// 注意下面这条语句，它有什么问题吗？

if (rd == null || wr == null) throw new Exception("error! test!");

while(true)

{

String sLine = rd.readLine();

if(sLine == null) break;

wr.write(sLine);

wr.write("\r\n");

}
```

```
}
finally
{
wr.flush();
wr.close();
rd.close();
}
}
catch(IOException ex)
{
ex.printStackTrace();
}
}
}
```

熟悉 java 语言的程序员朋友们，你们认为上面的程序有什么问题吗？编译能通过吗？如果不能，那么原因又是为何呢？好了，有了自己的分析和预期之后，不妨亲自动手编译一下上面的小程序，呵呵！结果确实如您所料？是的，的确是编译时报错了，错误信息如下：

E:\Trans.java:20: unreported exception java.lang.Exception; must be caught or declared to be thrown

if (rd == null || wr == null) throw new Exception("error! test!");

1 error

上面这种编译错误信息，相信 Java 程序员肯定见过（可能还是屡见不鲜！），而且，这也是许多从 C++ 程序员刚刚转来写 Java 程序不久（还没有真正入行）时，最感到迷糊和不解的地方，“俺代码明明没什么问题吗？可为什么编译有问题呢”！

呵呵！阿愚相信老练一些的 Java 程序员一定非常清楚上述编译出错的原因。那就是如错误信息中（“ *must be caught* ”）描述的那样， 在 Java 的异常处理模型中，要求所有被抛出的异常都必须要有对应的“异常处理模块” 。也即是说，如果你在程序中 throw 出一个异常，那么在你的程序中（函数中）就必须要 catch 这个异常（处理这个异常）。例如上面的例子中，你在第 20 行代码处，抛出了一个 Exception 类型的异常，

但是在该函数中，却没有 catch 并处理掉此异常的地方。因此，这样的程序即便是能够编译通过，那么运行时也是致命的（可能导致程序的崩溃），所以， Java 语言干脆在编译时就尽可能地检查（并卡住）这种本不应该出现的错误，这无疑对提高程序的可靠性大有帮助。相反， C++ 中则不是这样，它把更多的事情和责任交给 C++ 程序员去完成，它总相信 C++ 程序员是最优秀的，会自己处理好这一切的所有事情。如果程序中真的出现了未被捕获的异常，系统它（实际上是 C++ 运行库中）就认为这是出现了一个致命的、不可恢复的错误，接着调用 terminate 函数终止该进程（ NT 系统中稍微负责任一些，还弹出个“系统错误”对华框，并且 report 出一些与异常相关的错误信息）。

从上面的分析可以看出， Java 异常处理模型的确比 C++ 异常处理模型更严谨和更安全。实际上，这还体现在更多的方面，例如，在 C++ 异常处理模型中，异常的声明已经可以成为声明函数接口的一部分，但这并非语言所强求的。虽然许多优秀的库系统在设计时，都把这当成了一种必须的约定，甚至是编程规范，例如 MFC 中，就有许多可能抛出异常的函数，都显式地做了声明，如 CFile::Read 函数，声明如下：

virtual UINT Read(void* lpBuf , UINT nCount);
throw(CFileException);

但是，在 Java 语言中，这就是必须的。 如果一个函数中，它运行时可能会向上层调用者函数抛出一个异常，那么，它就必须在该函数的声明中显式的注明（采用 throws 关键字，语法与 C++ 类似） 。还记得刚才那条编译错误信息吗？“ *must be caught or declared to be thrown* ”，其中“ *must be caught* ”上面已经解释了，而后半部分呢？“ *declared to be thrown* ”是指何意呢？其实指的就是“必须显式地声明某个函数可能会向外部抛出一个异常”，也即是说，如果一个函数内部，它可能抛出了一种类型的异常，但该函数内部并不想（或不宜） catch 并处理这种类型的异常，此时，它就必须（ 注意，这是必须的！强求的！而 C++ 中则不是必须的 ）使用 throws 关键字来声明该函数可能会向外部抛出一个异常，以便于该函数的调用者知晓并能够及时处理这种类型的异常。下面列出了这几种情况的比较，代码如下：

```
// 示例程序 1 ，这种写法能够编译通过

import java.io.*;

public class Trans

{

public static void main(String[] args)

{

try

{

test();

}

catch(Exception ex)
```

```
{
ex.printStackTrace();
}
}
static void test()
{
try
{
throw new Exception("test");
}
catch(Exception ex)
{
ex.printStackTrace();
}
}
}
// 示例程序 2 ，这种写法就不能够编译通过
import java.io.*;
public class Trans
{
public static void main(String[] args)
{
try
{
```

```
test();
}
// 虽然这里能够捕获到 Exception 类型的异常
catch(Exception ex)
{
ex.printStackTrace();
}
}
static void test()
{
throw new Exception("test");
}
}
// 示例程序 3 ，这种写法又能够被编译通过
import java.io.*;
public class Trans
{
public static void main(String[] args)
{
try
{
test();
}
catch(Exception ex)
```

```
{
ex.printStackTrace();
}
}
// 由于函数声明了可能抛出 Exception 类型的异常
static void test() throws Exception
{
throw new Exception("test");
}
}
// 示例程序 4 ，它又不能够被编译通过了
import java.io.*;
public class Trans
{
public static void main(String[] args)
{
try
{
// 虽然 test() 函数并没有真正抛出一个 Exception 类型的异常
// 但是由于它函数声明时，表示它可能抛出一个 Exception 类型的异常
// 所以，这里仍然不能被编译通过。
// 呵呵！体会到了 Java 异常处理模型的严谨吧！
test();
}
```

```
catch(IOException ex)

{

ex.printStackTrace();

}

}

static void test() throws Exception

{

}

}
```

不知上面几个有联系的示例是否能够给大家带来“豁然开朗”的感觉，坦率的说， Java 提供的异常处理模型并不复杂，相信太多太多 Java 程序员有着比阿愚更深刻的认识。也许，阿愚对于这些简单而琐碎的事情如此婆婆妈妈絮叨个没完，有点招人笑话了！但是，阿愚还是坚持认为，非常有必要把这些简单但非常管用的准则阐述透彻。最后，补充一种例外情况，请看如下代码：

```
import java.io.*;

public class Trans

{

public static void main(String[] args)

{

try

{

test();

}

catch(Exception ex)

{

ex.printStackTrace();
```

```
}

}

static void test() throws Error

{

throw new Error(" 故意抛出一个 Error");

}

}
```

朋友们！上面的程序能被编译通过吗？注意，按照刚才上面所总结出的规律：在 Java 的异常处理模型中，要求所有被抛出的异常都必须要有对应的 catch 块！那么上面的程序肯定不能被编译通过，因为 Error 和 Exception 都是从 Throwable 直接派生而来，而 test 函数声明了它可能抛出 Error 类型的异常，但在 main 函数中却并没有 catch(Error) 或 catch(Throwable) 块，所以它理当是会编译出错的！真的吗？不妨试试！呵呵！结果并非我们之预料，而它恰恰是正确编译通过了。为何？ WHY ？ WHY ？

其实，原因很简单，那就是因为 Error 异常的特殊性。 Java 异常处理模型中规定： Error 和从它派生而来的所有异常，都表示系统中出现了一个非常严重的异常错误，并且这个错误可能是应用程序所不能恢复的 （其实这在前面的内容中已提到过）。因此，如果系统中真的出现了一个 Error 类型的异常，那么则表明，系统已处于崩溃不可恢复的状态中，此时，作为编写 Java 应用程序的你，已经是没有必要（也没能力）来处理此等异常错误。所以， javac 编译器就没有必要来保证：“在编译时，所有的 Error 异常都有其对应的错误处理模块”。当然， Error 类型的异常一般都是由系统遇到致命的错误时所抛出的，它最后也由 Java 虚拟机所处理。而作为 Java 程序员的你，可能永远也不会考虑抛出一个 Error 类型的异常。因此 Error 是一个特例情况！

特别关注一下 RuntimeException

上面刚刚讨论了一下 Error 类型的异常处理情况， Java 程序员一般无须关注它（处理这种异常）。另外，其实在 Exception 类型的异常对象中，也存在一种比较特别的“异常”类型，那就是 RuntimeException ，虽然它是直接从 Exception 派生而来，但是 Java 编译器（ javac ）对 RuntimeException 却是特殊待遇，而且是照顾有加。不信，看看下面的两个示例吧！代码如下：

```
// 示例程序 1

// 它不能编译通过，我们可以理解

import java.io.*;

public class Trans

{
```

```
public static void main(String[] args)
{
test();
}
static void test()
{
// 注意这条语句
throw new Exception("  故意抛出一个  Exception");
}
}
// 示例程序  2
// 可它却为什么能够编译通过呢？
import java.io.*;
public class Trans
{
public static void main(String[] args)
{
test();
}
static void test()
{
// 注意这条语句
throw new RuntimeException("  故意抛出一个  RuntimeException");
}
```

}

对上面两个相当类似的程序， javac 编译时却遭遇了两种截然不同的处理，按理说，第 2 个示例程序也应该像第 1 个示例程序那样，编译时报错！但是 javac 编译它时，却例外地让它通过它，而且在运行时， java 虚拟机也捕获到了这个异常，并且会在 console 打印出详细的异常信息。运行结果如下：

java.lang.RuntimeException: 故意抛出一个 RuntimeException

at Trans.test(Trans.java:13)

at Trans.main(Trans.java:8)

Exception in thread "main"

为什么对于 RuntimeException 类型的异常（以及从它派生而出的异常类型）， javac 和 java 虚拟机都特殊处理呢？要知道，这可是与“ Java 异常处理模型更严谨和更安全”的设计原则相抵触的呀！究竟是为何呢？这简直让人不法理解呀！主人公阿愚劝大家稍安勿躁，我们不妨换个角度，还是从 C++ 异常处理模型展开对比和分析，也许这能使我们茅塞顿开！

在 VC 实现的 C++ 异常处理中（基于 SEH 机制之上），一般有两类异常。其一，就是通过 throw 语句，程序员在代码中人为抛出的异常（由于运行时动态地监测到了一个错误）；另外一个呢？就是系统异常，也即前面我们一直重点讨论的 SEH 异常，例如，“被 0 除”，“段存储保护异常”，“无效句柄” 等，这类异常实际上程序员完全可以做得到避免它（只要我们写代码时足够小心，足够严谨，使得写出的代码足够安全可靠），但实际上，这也许无法完全做到，太理想化了。因此，为了彻底解决这种隐患，提高程序整体可靠性（可以容忍程序员留下一些 BUG ，至少不至于因为编码时考虑不周，或一个小疏忽留下的一个小 BUG ，而导致整个应用系统在运行时崩溃！）， VC 提供的 C++ 异常处理中，它就能够捕获到 SEH 类型的系统异常。 这类异常在被处理时，实际上它首先会被操作系统的中断系统所接管（也即应用程序出现这类异常时，会导致触发了一个系统“中断”事件），接着，操作系统会根据应用程序中（实际上是 VC 运行库）所设置的一系列“异常”回调函数和其它信息，来把处理该“系统异常”的控制权转交给应用程序层的“ VC 异常处理模型”中。 还记得上一篇文章中，《第 28 集 如何把 SEH 类型的系统异常转化为 C++ 类型的异常》中的方法和技术原理吗？我们不妨回顾一下，“把 SEH 类型的系统异常转化为 C++ 类型的异常”之后，它就提供了一种很好的技术方法，使得 VC 提供的 C++ 异常处理中，完全把“系统异常”的处理容纳到了按 C++ 方式的异常处理之中。

毫无疑问，这种技术大大的好！大大的妙！同样，现在 Java 的异常处理模型中，它理当应该也具备此项技术功能（谁让它是 C++ 的发展呢？），朋友们，阿愚现在是否已经把该问题解释明白了呢？是的，实际上， RuntimeException 异常的作用，就相当于上一篇文章中，阿愚为 C++ 所设计的 seh_exception_base 异常那样，它们在本质上完成同样类似的功能。只不过， Java 语言中， RuntimeException 被统一纳入到了 Java 语言和 JDK 的规范之中。请看如下代码，来验证一下我们的理解！

```
import java.io.*;

public class Trans

{
```

```
public static void main(String[] args)

{

test();

}

static void test()

{

int i = 4;

int j = 0;

// 运行时，这里将触发了一个 ArithmeticException

// ArithmeticException 从 RuntimeException 派生而来

System.out.println("i / j = " + i / j);

}

}
```

运行结果如下：

java.lang.ArithmeticException: / by zero

at Trans.test(Trans.java:16)

at Trans.main(Trans.java:8)

Exception in thread "main"

又如下面的例子，也会产生一个 RuntimeException ，代码如下：

```
import java.io.*;

public class Trans

{

public static void main(String[] args)

{
```

```
test();

}

static void test()

{

String str = null;

// 运行时，这里将触发了一个 NullPointerException

// NullPointerException 从 RuntimeException 派生而来

str.compareTo("abc");

}

}
```

所以，针对 RuntimeException 类型的异常， javac 是无法通过编译时的静态语法检测来判断到底哪些函数（或哪些区域的代码）可能抛出这类异常（这完全取决于运行时状态，或者说运行态所决定的），也正因为如此， Java 异常处理模型中的“ *must be caught or declared to be thrown* ”规则也不适用于 RuntimeException （所以才有前面所提到过的奇怪编译现象，这也属于特殊规则吧）。但是， Java 虚拟机却需要有效地捕获并处理此类异常。当然， RuntimeException 也可以被程序员显式地抛出，而且为了程序的可靠性，对一些可能出现“运行时异常（ RuntimeException ）”的代码区域，程序员最好能够及时地处理这些意外的异常，也即通过 catch(RuntimeExcetion) 或 catch(Exception) 来捕获它们。如下面的示例程序，代码如下：

```
import java.io.*;

public class Trans

{

public static void main(String[] args)

{

try

{

test();

}
```

```
// 在上层的调用函数中，最好捕获所有的 Exception 异常！

catch(Exception e)

{

System.out.println("go here!");

e.printStackTrace();

}

}

// 这里最好显式地声明一下，表明该函数可能抛出 RuntimeException

static void test() throws RuntimeException

{

String str = null;

// 运行时，这里将触发了一个 NullPointerException

// NullPointerException 从 RuntimeException 派生而来

str.compareTo("abc");

}

}
```

总结

• Java 异常处理模型与 C++ 中异常处理模型的最大不同之处，就是在 Java 异常处理模型中引入了 try-finally 语法，它主要用于清理非内存性质的一些资源（垃圾回收机制无法处理的资源），例如，数据库连接、 Socket 关闭、文件流的关闭等。

• 所有的异常都必须从 Throwable 继承而来，不像 C++ 中那样，可以抛出任何类型的异常。因此，在 Java 的异常编程处理中，没有 C++ 中的 catch(…) 语法，而它的 catch(Throwable e) 完全可以替代 C++ 中的 catch(…) 的功能。

• 在 Java 的异常处理模型中，要求所有被抛出的异常都必须要有对应的“异常处理模块”。也即是说，如果你在程序中 throw 出一个异常，那么在你的程序中（函数中）就必须要 catch 这个异常（处理这个异常）。但是，对于 RuntimeException 和 Error 这两种类型的异常（以及它们的子类异常），却是例外的。其中，

Error 表示 Java 系统中出现了一个非常严重的异常错误；而 RuntimeException 虽然是 Exception 的子类，但是它却代表了运行时异常（这是 C++ 异常处理模型中最不足的，虽然 VC 实现的异常处理模型很好）

- 如果一个函数中，它运行时可能会向上层调用者函数抛出一个异常，那么，它就必须在该函数的声明中显式的注明（采用 throws 关键字，语法与 C++ 类似）。

本篇文章虽然有点长，但是如果你能够坚持完整地看下来，相信你对 Java 的异常处理模型一定有了更深一步的了解（当然，由于阿愚水平有限，难免会有不少认识上或理解上的错误，欢迎大家不吝赐教！呵呵！讨论和交流也可以）。在下一篇文章中，阿愚将继续分析和阐述一些有关 Java 异常处理模型中更细节的语法特点和使用上的技巧，以及注意事项。感兴趣的 朋友们，继续吧！ GO ！

第 30 集 Java 异常处理模型之细节分析

上一篇文章中，与其是说对 Java 语言中的异常处理模型展开讨论；倒不如是说，把 Java 异常处理模型与 C++ 异常处理模型展开了全面的比较和分析。俗话说得好，“不怕不识货，就怕货比货”，因此，通过与 C++ 异常处理模型的综合比较，使我们能够更清楚地认识 Java 中异常处理模型的特点、优势，以及设计时的一些背景及原因。

这篇文章，阿愚打算对 Java 异常处理模型中的一些重要的细节问题展开详细讨论，让我们更上一层楼吧！

finally 区域内的代码块在 return 之前被执行

由于 Java 程序中，所有的对象都是在堆上（ Heap ）分配存储空间的，这些空间完全由垃圾回收机制来对它们进行管理。因此，从这一点可以分析得出一个推论： Java 中的异常处理模型的实现，其实要比 C++ 异常处理模型简单得多。例如，它首先不需要像 C++ 异常处理模型中那样，必须要跟踪栈上的每一个“对象”的构造和析构过程（只有跟踪并掌握了这些信息，发生异常时， C++ 系统它才会知道当前应该析构销毁哪些对象呀！），这是因为 Java 程序中，栈上是绝对没有“对象”的（实际只是对堆上对象的引用）。另外，还有 Java 语言中的异常对象的传递也更为简单和容易了，它只需传递一个引用指针而已，而完全不用考虑异常对象的构造、复制和销毁过程。

当然， Java 异常处理模型较 C++ 异常处理模型复杂的地方是，它引入了 finally 机制（主要用于数据库连接的关闭、 Socket 关闭、文件流的关闭等）。其实，我们也知道 finally 语法最早是在微软的 SEH 所设计出的一种机制，虽然它功能很强大，但是实现起来却并不是很难，从表象上来理解：当代码在执行过程中，遭遇到 return 和 goto 等类似的语句所引发作用域（代码执行流）转移时，便会产生一个局部展开（ Local Unwinding ）；而由于异常而导致的 finally 块被执行的过程，往往被称为全局展开（ Global Unwinding ）。由于展开（ Unwinding ）而导致的 finally 块被执行的过程，非常类似于一个子函数（或子过程）被调用的过程。例如，当在 try 块中最后一条语句 return 被执行到的时候，一个展开操作便发生了，可以把展开操作想象成，是编译器在 return 语句之前插入了一些代码（这些代码完成对 finally 块的调用），因此可以得出结论： finally 区域内的代码块，肯定是在 return 之前被执行。

但是，请特别注意， finally 块区域中的代码虽然在 return 语句之前被执行，但是 finally 块区域中的代码是不能够通过重新赋值的方式来改变 return 语句的返回值。请看如下的示例代码：

```
import java.io.*;
```

```
public class Trans

{

public static void main(String[] args)

{

// 你认为 test 函数返回的值是多少呢？

System.out.println("test 的返回值为： " + test());

}

public static int test()

{

int ret = 1;

try

{

System.out.println("in try block");

// 是返回 1 ，还是返回 2 呢？

return (ret);

}

catch(Exception e)

{

System.out.println("in catch block");

e.printStackTrace();

}

finally

{

// 注意，这里重新改变了 ret 的值。
```

```
ret = 2;

System.out.println("in finally block!");

}

return 0;

}

}
```

上面的示例程序中，本来是想在 finally 区域中通过改变 ret 的值，来影响 test 函数最终 return 的值。但是真的影响了吗？否！否！否！不信，看看运行结果吧！

in try block

in finally block!

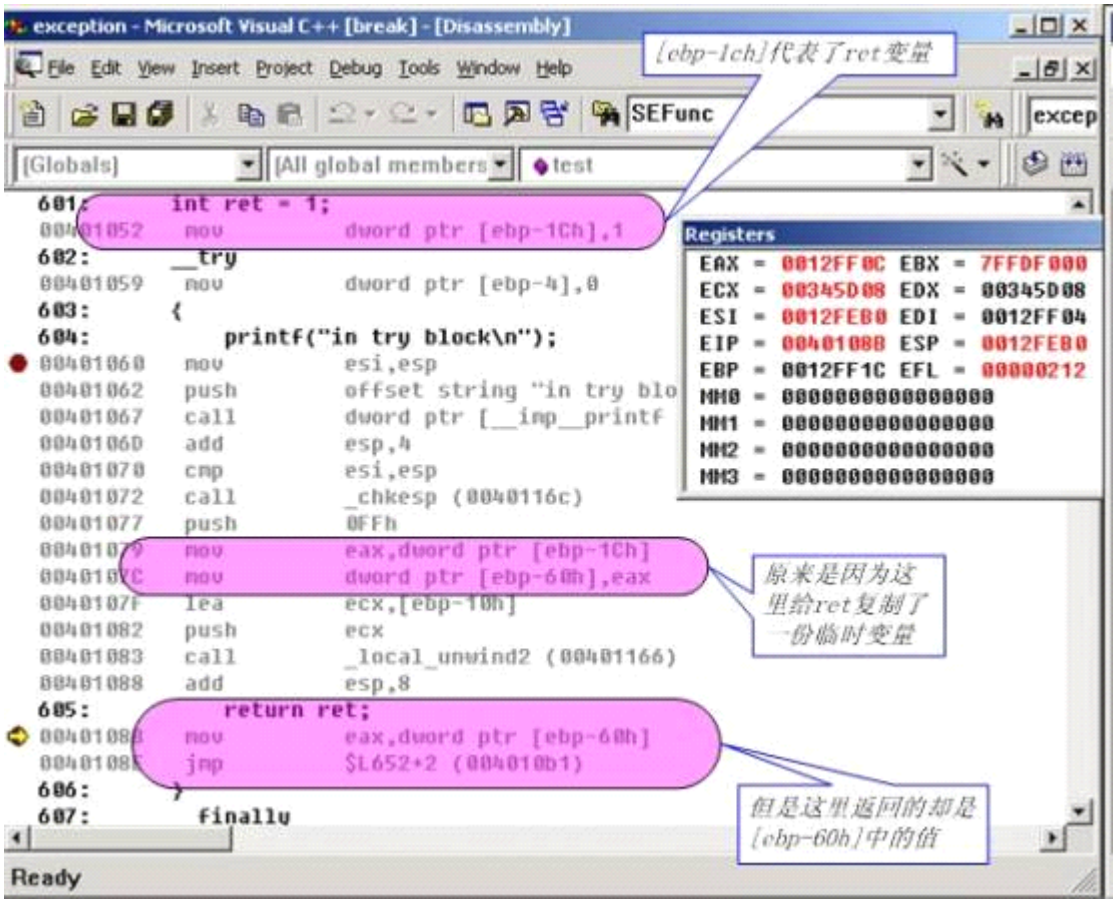
test 的返回值为： 1

其实，在 SEH 异常处理模型中， try-finally 语句对此情况也是有相同结果的处理结果，同样是上面的那个程序，把它改称 C 语言的形式，用 VC 编译运行一把，验证一下结果，代码如下：

```
#include "stdio.h"

int test()

{

int ret = 1;

__try

{

printf("in try block\n");

return ret;

}

__finally

{

ret = 2;
```

```
printf("in finally block!\n");

}

return 0;

}

void main()

{

printf("test 的返回值为： %d\n", test());

}
```

为了更清楚了认识一下这究竟是如何道理，还是看看对这个程序的 debug 调试情况，截图如下图所示：



通过调试可以很容易看出，在 return ret 之前，编译器实际上对 ret 赋值了一份临时变量，为的就是防止 finally 区域中的代码对这个 ret 值的改变。当然这只是 debug 版本的情况，实际上，在 release 版本，return ret 语句直接被编译器编译成 mov eax, 1 指令。

所以说，无论是在 SEH 异常处理模型中，还是 Java 的异常处理模型中， finally 块区域中的代码都是不能够通过重新赋值的方式来改变 try 区域中 return 语句的返回值。

强烈建议不要在 finally 内部使用 return 语句

上面刚刚说了， finally 块区域中的代码不会轻易影响 try 区域中 return 语句的返回值，但是有一种情况例外，那就是在 finally 内部使用 return 语句。示例程序如下：

```
// 示例程序 1 ， Java 程序

import java.io.*;

public class Trans

{

public static void main(String[] args)

{

System.out.println("test 的返回值为： " + test());

}

public static int test()

{

int ret = 1;

try

{

System.out.println("in try block");

return (ret);

}

catch(Exception e)

{

System.out.println("in catch block");
```

```
e.printStackTrace();
}
finally
{
ret = 2;
System.out.println("in finally block!");
// 这里添加了一条 return 语句
return ret;
}
}
}
// 示例程序 2 ， C 程序
#include "stdio.h"
int test()
{
int ret = 1;
__try
{
printf("in try block\n");
return ret;
}
__finally
{
ret = 2;
```

```
printf("in finally block!\n");

return ret;

}

printf(" 多余的 \n");

return 0;

}

void main()

{

printf("test 的返回值为： %d\n", test());

}
```

上面的程序运行结果如下：

in try block

in finally block!

test 的返回值为： 2

也许大多数朋友都估计到，上面的 test 函数返回值是 2 。也即是说， finally 内部使用 return 语句后，它影响（覆盖了） try 区域中 return 语句的返回值。这真是一种特别糟糕的情况，虽然它表面上看起来不是那么严重，但是这种程序极易给它人造成误解（使得阅读该代码的人总得担心或考虑，是否有其它地方影响了这里的 return 的返回值）。

之所以出现这种现象的真正原因是，由于 finally 区域中的代码先于 return 语句（ try 作用域中的）被执行，但是，如果此时在 finally 内部也有一个 return 语句，这将会导致该函数直接就返回了，而致使 try 作用域中的 return 语句再也得不到执行机会（实际就是无效代码，被覆盖了）。

面对上述情况，其实更合理的做法是，既不在 try block 内部中使用 return 语句，也不在 finally 内部使用 return 语句，而应该在 finally 语句之后使用 return 来表示函数的结束和返回，把上面的程序改造一下，代码如下

```
import java.io.*;

public class Trans

{
```

```
public static void main(String[] args)
{
try
{
System.out.println("test 的返回值为： " + test());
}
catch(Exception e)
{
e.printStackTrace();
}
}
public static int test() throws RuntimeException
{
int ret = 1;
try
{
System.out.println("in try block");
}
catch(RuntimeException e)
{
System.out.println("in catch block");
e.printStackTrace();
throw e;
}
```

```
finally

{

ret = 2;

System.out.println("in finally block!");

}

// 把 return 语句放在最后，这最为妥当

return ret;

}

}
```

另一种更糟糕的情况

上面刚刚讲到， finally 内部使用 return 语句会覆盖 try 区域中 return 语句的返回值。不仅与此， finally 内部使用 return 语句还会导致出现另外一种更为糟糕的局面。到底是何种糟糕情况呢？还是先看看下面的示例程序再说吧！代码如下：

```
import java.io.*;

public class Trans

{

public static void main(String[] args)

{

try

{

System.out.println("test 的返回值为： " + test());

}

catch(Exception e)

{
```

```
e.printStackTrace();

}

}

public static int test() throws RuntimeException

{

int ret = 0;

try

{

System.out.println("in try block");

// 这里会导致出现一个运行态异常

int i=4,j=0;

ret = i/j;

}

catch(RuntimeException e)

{

System.out.println("in catch block");

e.printStackTrace();

// 异常被重新抛出，上层函数可以进一步处理此异常

throw e;

}

finally

{

System.out.println("in finally block!");

// 注意，这里有一个 return 语句
```

```
return ret;

}

}

}
```

是不是觉得上面示例程序中的代码写的挺好的，挺简洁的，还挺严谨的，应该不会有什么 BUG ！阿愚告诉你，错了！绝对错了！而且问题很严重！要不，朋友们在编译运行此程序前，先预期一下它的运行结果，大家是不是觉得运行流程应该是这样子的： 首先在终端输出“ in try block ”，接着，由于程序运行时出现了一个被 0 处的异常（ *ArithmeticException* ）；于是，进入到 catch block 中，这里的代码将继续向终端输出了“ in catch block ”信息，以及输出异常的堆栈信息等；接着，由于异常在 catch block 中又被重新抛出了，所以控制权返回到 main 函数的 catch block 中；对了，补充一点，也许大家会觉得，由于异常的 rethrow ，使得控制权离开 test 函数作用域的时候， finally 内的代码会被执行，也即“ in finally block ”信息也会被打印到终端上了 。仅仅如此吗？不妨看一下实际的运行结果，如下：

in try block

in catch block

java.lang.ArithmeticException: / by zero

at Trans.test(Trans.java:27)

at Trans.main(Trans.java:10)

in finally block!

test 的返回值为： 0

看了实际的运行结果，是不是觉得大吃一惊！被重新抛出（ rethrow ）的异常居然“丢弃了”，也即 main 函数中，并没有捕获到 test 函数中的任何异常，这也许大大出乎当初写这段代码的程序员的预料吧！究竟原因何在呢？其实， 罪魁祸首就是 finally block 内部的那条 return 语句 。因为这段程序的运行流程基本如刚才我们预期描述的那样，但是有一点是不对的，那就是当 test 函数中把异常重新抛出后，这将导致又一次的 catch 的搜索和匹配过程，以及 test 函数中的 UnWinding 操作，也即 finally 被调用执行，但是由于 finally 内部的 return 语句，不仅使得它结束了 test 函数的执行（并返回一个值给上层函数），而且这还使得上面的那个对异常进行进一步的操作过程给终止了（也即控制权进入到 catch block 的过程）。瞧瞧！后果严重吧！

是呀！如果 Java 程序员不注意这种问题，养成一个严谨的、好的编程习惯，它将会导致实际的许多 Java 应用系统中出现一些莫名奇妙的现象（总感觉系统中出现了某类异常，有一些问题，但上层的模块中却总是捕获不到相关的异常，感觉一些良好！其实不然，它完全是由于 finally 块中的 return 语句不小心把异常给屏蔽丢弃掉了）。

总结

- finally 区域内的代码总在 return 之前被执行；

- 强烈建议不要在 finally 内部使用 return 语句。它不仅会影响函数的正确返回值，而且它可能还会导致一些异常处理过程的意外终止，最终导致某些异常的丢失。

关于 Java 异常处理模型的阐述暂时就到此为止吧！虽然，有关 Java 的异常处理知识和技巧，还有不少方面需要深入讨论的，例如异常的嵌套，异常的分类，函数接口中有关异常的声明等等，阿愚打算在后面“相爱篇”的一些文章中，将继续讨论这些有关的东东。

接下来一篇文章中，会详细讨论类 Unix 操作系统中对异常处理的支持，方法，以及思想等。感兴趣的朋友， Let's go!