C++异常处理：try,catch,throw,finally的用法 - 北岛知寒 - 博客园

写在前面

所谓异常处理，即让一个程序运行时遇到自己无法处理的错误时抛出一个异常，希望调用者可以发现处理问题.

异常处理的基本思想是简化程序的错误代码，为程序健壮性提供一个标准检测机制.

也许我们已经使用过异常，但是你习惯使用异常了吗？

现在很多软件都是n*36524*小时运行，软件的健壮性至关重要.

内容导读

本文包括2个大的异常实现概念：C++的标准异常和SEH异常.

C++标准异常：

也许你很高兴看到错误之后的Heap/Stack中对象被释放，可是如果没有呢？

又或者试想一下一个能解决的错误，需要我们把整个程序Kill掉吗？

在《C++标准异常》中我向你推荐这几章：

<使用异常规格编程> <构造和析构中的异常抛出> <使用析构函数防止资源泄漏>,以及深入一点的<抛出一个异常的行为>.

SEH异常：

我要问你你是一个WIN32程序员吗？如果不是，那么也许你真的不需要看.

SEH是Windows的结构化异常，每一个WIN32程序员都应该要掌握它.

SEH功能强大，包括Termination handling和Exception handling两大部分.

强有力的维护了代码的健壮，虽然要以部分系统性能做牺牲(其实可以避免).

在SEH中有大量的代码，已经在Win平台上测试过了.

这里要提一下：在__finally处理中编译器参与了绝大多数的工作，而Exception则是OS接管了几乎所有的工作，也许我没有提到的是：

对__finally来说当遇到ExitThread/ExitProcess/abort等函数时，finally块不会被执行.

另:<使用析构函数防止资源泄漏>这个节点引用了More effective C++的条款9.

用2个列子，讲述了我们一般都会犯下的错误，往往这种错误是我们没有意识到的但确实是会给我们的软件带来致命的Leak/Crash，但这是有解决的方法的，那就是使用“灵巧指针”.

如果对照<More effective C++>的37条条款，关于异常的高级使用，有以下内容是没有完成的：

使用构造函数防止资源Leak（More effective C++ #10）

禁止异常信息传递到析构Function外 (More effective C++ #11)

通过引用捕获异常 （More effective C++ #13）

谨慎使用异常规格 （More effective C++ #14）

了解异常处理造成的系统开销 （More effective C++ #15）

限制对象数量 （More effective C++ #26）

灵巧指针 （More effective C++ #28）

C++异常

C++引入异常的原因:

例如使用未经处理的pointer变的很危险，Memory/Resource Leak变的更有可能了.

写出一个具有你希望的行为的构造函数和析构函数也变的困难（不可预测），当然最危险的也许是我们写出的东东狗屁了，或者是速度变慢了.

大多数的程序员知道Howto use exception 来处理我们的代码，可是很多人并不是很重视异常的处理（国外的很多Code倒是处理的很好，Java的Exception机制很不错）.

异常处理机制是解决某些问题的上佳办法，但同时它也引入了许多隐藏的控制流程；有时候，要正确无误的使用它并不容易.

在异常被throw后，没有一个方法能够做到使软件的行为具有可预测性和可靠性

对C程序来说，使用Error Code就可以了，为什么还要引入异常？因为异常不能被忽略.

如果一个函数通过设置一个状态变量或返回错误代码来表示一个异常状态，没有办法保证函数调用者将一定检测变量或测试错误代码.

结果程序会从它遇到的异常状态继续运行，异常没有被捕获，程序立即会终止执行.

在C程序中，我们可以用int setjmp(jmp_buf env);和 void longjmp(jmp_buf env, int value);

这2个函数来完成和异常处理相识的功能，但是MSDN中介绍了在C++中使用longjmp来调整stack时不能够对局部的对象调用析构函数，

但是对C++程序来说，析构函数是重要的（我就一般都把对象的Delete放在析构函数中）.

所以我们需要一个方法：

能够通知异常状态，又不能忽略这个通知.

并且Searching the stack以便找到异常代码时.

还要确保局部对象的析构函数被Call.

而C++的异常处理刚好就是来解决这些问题的.

有的地方只有用异常才能解决问题，比如说，在当前上下文环境中，无法捕捉或确定的错误类型，我们就得用一个异常抛出到更大的上下文环境当中去.

还有，异常处理的使用呢，可以使出错处理程序与“通常”代码分离开来，使代码更简洁更灵活.

另外就是程序必不可少的健壮性了，异常处理往往在其中扮演着重要的角色.

C++使用throw关键字来产生异常，try关键字用来检测的程序块，catch关键字用来填写异常处理的代码.

异常可以由一个确定类或派生类的对象产生。C++能释放堆栈，并可清除堆栈中所有的对象.

C++的异常和pascal不同，是要程序员自己去实现的，编译器不会做过多的动作.

throw异常类编程,抛出异常用throw， 如：

throw ExceptionClass(“my throw“);

例句中，ExceptionClass是一个类，它的构造函数以一个字符串做为参数.

也就是说，在throw的时候，C++的编译器先构造一个ExceptionClass的对象，让它作为throw的值抛出去,同时，程序返回，调用析构.

看下面这个程序：

```
#include <iostream.h>
class ExceptionClass
{
     char* name;
public:
     ExceptionClass(const char* name="default name")
     {
           cout<<"Construct "<<name<<endl;
           this->name=name;
     }
     ~ExceptionClass()
     {
           cout<<"Destruct "<<name<<endl;
     }
     void mythrow()
     {
           throw ExceptionClass("my throw");
     }
}

void main()
{
     ExceptionClass e("Test");
     try
     {
           e.mythrow();
     }
     catch(...)
     {
           cout<<”**********”<<endl;
     }
}
```

这是输出信息：

```
Construct Test
Construct my throw
Destruct my throw
***************
Destruct my throw    （这里是异常处理空间中对异常类的拷贝的析构）
Destruct Test
====================================
```

不过一般来说我们可能更习惯于把会产生异常的语句和要throw的异常类分成不同的类来写，下面的代码可以是我们更愿意书写的.

```
class ExceptionClass
{
public:
         ExceptionClass(const char* name="Exception Default Class")
     {
               cout<<"Exception Class Construct String"<<endl;
     }
          ~ExceptionClass()
     {
                cout<<"Exception Class Destruct String"<<endl;
     }
          void ReportError()
     {
                cout<<"Exception Class:: This is Report Error Message"<<endl;
     }
};

class ArguClass
{
          char* name;
public:
          ArguClass(char* name="default name")
     {
                cout<<"Construct String::"<<name<<endl;
                this->name=name;
     }
     ~ArguClass()
     {
                cout<<"Destruct String::"<<name<<endl;
     }
```

```
    void mythrow()
    {
            throw ExceptionClass("my throw");
    }
};

_tmain()
{
    ArguClass e("haha");
        try
    {
            e.mythrow();
    }
    catch(int)
    {
            cout<<"If This is Message display screen, This is a Error!!"<<endl;
    }
    catch(ExceptionClass pTest)
    {
            pTest.ReportError();
    }
    catch(...)
    {
            cout<<"***************"<<endl;
    }
}
```

输出Message：

```
Construct String::haha
Exception Class Construct String
Exception Class Destruct String
Exception Class:: This is Report Error Message
Exception Class Destruct String
Destruct String::haha
```

使用异常规格编程

如果我们调用别人的函数，里面有异常抛出，用去查看它的源代码去看看都有什么异常抛出吗？这样就会很烦琐.

比较好的解决办法，是编写带有异常抛出的函数时，采用异常规格说明，使我们看到函数声明就知道有哪些异常出现。

异常规格说明大体上为以下格式：

```
void ExceptionFunction(argument…)
throw(ExceptionClass1, ExceptionClass2, ….)
```

所有异常类都在函数末尾的throw()的括号中得以说明了，这样，对于函数调用者来说，是一清二楚的。

注意下面一种形式：

```
void ExceptionFunction(argument…) throw()
```

表明没有任何异常抛出.

而正常的void ExceptionFunction(argument...)则表示：可能抛出任何一种异常，当然，也可能没有异常，意义是最广泛的.

异常捕获之后，可以再次抛出，就用一个不带任何参数的throw语句就可以了.

构造和析构中的异常抛出

这是异常处理中最要注意的地方了

先看个程序，假如我在构造函数的地方抛出异常，这个类的析构会被调用吗？可如果不调用，那类里的东西岂不是不能被释放了？

```
#include <iostream.h>
#include <stdlib.h>

class ExceptionClass1
{
    char* s;
public:
    ExceptionClass1()
    {
          cout<<"ExceptionClass1()"<<endl;
          s=new char[4];
          cout<<"throw a exception"<<endl;
          throw 18;
    }
    ~ExceptionClass1()
    {
          cout<<"~ExceptionClass1()"<<endl;
          delete[] s;
    }
};

void main()
{
    try
    {
```

```
            ExceptionClass1 e;
    }
    catch(...)
    {}
}
```

结果为：

```
ExceptionClass1()
throw a exception
```

在这两句输出之间，我们已经给S分配了内存，但内存没有被释放（因为它是在析构函数中释放的）.

应该说这符合实际现象，因为对象没有完整构造.

为了避免这种情况，我想你也许会说：应避免对象通过本身的构造函数涉及到异常抛出.

即：既不在构造函数中出现异常抛出，也不应在构造函数调用的一切东西中出现异常抛出.

但是在C++中可以在构造函数中抛出异常，经典的解决方案是使用STL的标准类auto_ptr.

其实我们也可以这样做来实现：

在类中增加一个 Init()以及 UnInit();成员函数用于进行容易产生错误的资源分配工作，而真正的构造函数中先将所有成员置为NULL，然后调用 Init();

并判断其返回值/或者捕捉 Init()抛出的异常，如果Init();失败了，则在构造函数中调用 UnInit(); 并设置一个标志位表明构造失败.

UnInit()中按照成员是否为NULL进行资源的释放工作.

那么，在析构函数中的情况呢？

我们已经知道，异常抛出之后，就要调用本身的析构函数，如果这析构函数中还有异常抛出的话，则已存在的异常尚未被捕获，会导致异常捕捉不到.

标准C++异常类

C++有自己的标准的异常类.

一个基类

exception 是所有C++异常的基类.

```
class exception {
public:
   exception() throw();
   exception(const exception& rhs) throw();
   exception& operator=(const exception& rhs) throw();
   virtual ~exception() throw();
   virtual const char *what() const throw();
};
```

派生了两个异常类

logic_erro 报告程序的逻辑错误，可在程序执行前被检测到.

runtime_erro 报告程序运行时的错误，只有在运行的时候才能检测到.

以上两个又分别有自己的派生类：

由logic_erro派生的异常类

```
domain_error            报告违反了前置条件
invalid_argument        指出函数的一个无效参数
length_error             指出有一个产生超过NPOS长度的对象的企图（NPOS为size_t的最大可表现值
out_of_range             报告参数越界
bad_cast                在运行时类型识别中有一个无效的dynamic_cast表达式
bad_typeid               报告在表达式typeid(*p)中有一个空指针P
```

由runtime_error派生的异常

```
range_error          报告违反了后置条件
overflow_error      报告一个算术溢出
bad_alloc           报告一个存储分配错误
```

使用析构函数防止资源泄漏

这部分是一个经典和很平常就会遇到的实际情况，下面的内容大部分都是从More Effective C++条款中得到的.

假设，你正在为一个小动物收容所编写软件，小动物收容所是一个帮助小狗小猫寻找主人的组织.

每天收容所建立一个文件，包含当天它所管理的收容动物的资料信息，你的工作是写一个程序读出这些文件然后对每个收容动物进行适当的处理（appropriate processing）.

完成这个程序一个合理的方法是定义一个抽象类，ALA（"Adorable Little Animal"），然后为小狗和小猫建立派生类.

一个虚拟函数processAdoption分别对各个种类的动物进行处理：

```
class ALA {
public:
  virtual void processAdoption() = 0;
  ...
};
class Puppy: public ALA {
public:
  virtual void processAdoption();
  ...
};
class Kitten: public ALA {
public:
  virtual void processAdoption();
  ...
};
```

你需要一个函数从文件中读信息，然后根据文件中的信息产生一个puppy（小狗）对象或者kitten(小猫)对象.

这个工作非常适合于虚拟构造器（virtual constructor），在条款25详细描述了这种函数.

为了完成我们的目标，我们这样声明函数：

```
// 从s中读动物信息，然后返回一个指针
// 指向新建立的某种类型对象
ALA * readALA(istream& s);
```

你的程序的关键部分就是这个函数，如下所示：

```
void processAdoptions(istream& dataSource)
{
    while(dataSource)
    {
          ALA *pa = readALA(dataSource); //得到下一个动物
          pa->processAdoption(); //处理收容动物
          delete pa; //删除readALA返回的对象
    }
}
```

这个函数循环遍历dataSource内的信息，处理它所遇到的每个项目.

唯一要记住的一点是在每次循环结尾处删除ps.

这是必须的，因为每次调用readALA都建立一个堆对象.如果不删除对象，循环将产生资源泄漏。

现在考虑一下，如果pa->processAdoption抛出了一个异常，将会发生什么？

processAdoptions没有捕获异常，所以异常将传递给processAdoptions的调用者.

转递中，processAdoptions函数中的调用pa->processAdoption语句后的所有语句都被跳过，这就是说pa没有被删除.

结果，任何时候pa->processAdoption抛出一个异常都会导致processAdoptions内存泄漏，很容易堵塞泄漏.

```
void processAdoptions(istream& dataSource)
{
    while(dataSource)
    {
          ALA *pa = readALA(dataSource);
          try
          {
                pa->processAdoption();
          }
          catch(...)
          {
                              // 捕获所有异常
                delete pa;               // 避免内存泄漏
                                           // 当异常抛出时
                throw;                   // 传送异常给调用者
          }
          delete pa;                 // 避免资源泄漏
    }                         // 当没有异常抛出时
}
```

但是你必须用try和catch对你的代码进行小改动.

更重要的是你必须写双份清除代码，一个为正常的运行准备，一个为异常发生时准备.

在这种情况下，必须写两个delete代码.

象其它重复代码一样，这种代码写起来令人心烦又难于维护，而且它看上去好像存在着问题.

不论我们是让processAdoptions正常返回还是抛出异常，我们都需要删除pa，所以为什么我们必须要在多个地方编写删除代码呢？

我们可以把总被执行的清除代码放入processAdoptions函数内的局部对象的析构函数里，这样可以避免重复书写清除代码.

因为当函数返回时局部对象总是被释放，无论函数是如何退出的.

（仅有一种例外就是当你调用longjmp时。Longjmp的这个缺点是C++率先支持异常处理的主要原因）

具体方法是用一个对象代替指针pa，这个对象的行为与指针相似。当pointer-like（类指针）对象被释放时，我们能让它的析构函数调用delete.

替代指针的对象被称为smart pointers（灵巧指针），下面有解释，你能使得pointer-like对象非常灵巧.

在这里，我们用不着这么聪明的指针，我们只需要一个pointer-lik对象，当它离开生存空间时知道删除它指向的对象.

写出这样一个类并不困难，但是我们不需要自己去写。标准C++库函数包含一个类模板，叫做auto_ptr，这正是我们想要的.

每一个auto_ptr类的构造函数里，让一个指针指向一个堆对象（heap object），并且在它的析构函数里删除这个对象.

下面所示的是auto_ptr类的一些重要的部分：

```
template<class T>
class auto_ptr
{
public:
      auto_ptr(T *p = 0): ptr(p) {}       // 保存ptr，指向对象
      ~auto_ptr() { delete ptr; }         // 删除ptr指向的对象
private:
      T *ptr;            // raw ptr to object
};
```

auto_ptr类的完整代码是非常有趣的，上述简化的代码实现不能在实际中应用.

（我们至少必须加上拷贝构造函数，赋值operator以及下面将要讲到的pointer-emulating函数）

但是它背后所蕴含的原理应该是清楚的：用auto_ptr对象代替raw指针，你将不再为堆对象不能被删除而担心，即使在抛出异常时，对象也能被及时删除.

(因为auto_ptr的析构函数使用的是单对象形式的delete，所以auto_ptr不能用于指向对象数组的指针.

如果想让auto_ptr类似于一个数组模板，你必须自己写一个。在这种情况下，用vector代替array可能更好)

```
auto_ptr
template<class T>
class auto_ptr
{
public:
     typedef T element_type;
     explicit auto_ptr(T *p = 0) throw();
     auto_ptr(const auto_ptr<T>& rhs) throw();
     auto_ptr<T>& operator=(auto_ptr<T>& rhs) throw();
     ~auto_ptr();
     T& operator*() const throw();
     T *operator->() const throw();
     T *get() const throw();
     T *release() const throw();
};
```

使用auto_ptr对象代替raw指针，processAdoptions如下所示：

```
void processAdoptions(istream& dataSource)
{
     while(dataSource)
     {
           auto_ptr<ALA> pa(readALA(dataSource));
           pa->processAdoption();
     }
}
```

这个版本的processAdoptions在两个方面区别于原来的processAdoptions函数：

pa被声明为一个auto_ptr对象，而不是一个raw ALA*指针.

在循环的结尾没有delete语句.

其余部分都一样，因为除了析构的方式，auto_ptr对象的行为就象一个普通的指针。是不是很容易.

隐藏在auto_ptr后的思想是：

用一个对象存储需要被自动释放的资源，然后依靠对象的析构函数来释放资源，这种思想不只是可以运用在指针上，还能用在其它资源的分配和释放上.

想一下这样一个在GUI程序中的函数，它需要建立一个window来显式一些信息：

```
// 这个函数会发生资源泄漏，如果一个异常抛出
void displayInfo(const Information& info)
{
       WINDOW_HANDLE w(createWindow());//在w对应的window中显式信息
       destroyWindow(w);
}
```

很多window系统有C一like接口，使用象like createWindow 和 destroyWindow函数来获取和释放window资源.

如果在w对应的window中显示信息时，一个异常被抛出，w所对应的window将被丢失，就象其它动态分配的资源一样.

解决方法与前面所述的一样，建立一个类，让它的构造函数与析构函数来获取和释放资源：

```
//一个类，获取和释放一个window 句柄
class WindowHandle
{
public:
       WindowHandle(WINDOW_HANDLE handle): w(handle) {}
       ~WindowHandle() { destroyWindow(w); }
        operator WINDOW_HANDLE() { return w; }       // see below
private:
       WINDOW_HANDLE w;
       // 下面的函数被声明为私有，防止建立多个WINDOW_HANDLE拷贝
     //有关一个更灵活的方法的讨论请参见下面的灵巧指针
       WindowHandle(const WindowHandle&);
       WindowHandle& operator=(const WindowHandle&);
};
```

这看上去有些象auto_ptr，只是赋值操作与拷贝构造被显式地禁止（参见More effective C++条款27），有一个隐含的转换操作能把WindowHandle转换为WINDOW_HANDLE.

这个能力对于使用WindowHandle对象非常重要，因为这意味着你能在任何地方象使用raw WINDOW_HANDLE一样来使用WindowHandle.

（参见More effective C++条款5，了解为什么你应该谨慎使用隐式类型转换操作）

通过给出的WindowHandle类，我们能够重写displayInfo函数，如下所示：

```
// 如果一个异常被抛出，这个函数能避免资源泄漏
void displayInfo(const Information& info)
{
     WindowHandle w(createWindow());
     //在w对应的window中显式信息;
}
```

即使一个异常在displayInfo内被抛出，被createWindow 建立的window也能被释放.

资源应该被封装在一个对象里，遵循这个规则，你通常就能避免在存在异常环境里发生资源泄漏.

但是如果你正在分配资源时一个异常被抛出，会发生什么情况呢？

例如当你正处于resource-acquiring类的构造函数中.

还有如果这样的资源正在被释放时，一个异常被抛出，又会发生什么情况呢？

构造函数和析构函数需要特殊的技术.

你能在More effective C++条款10和More effective C++条款11中获取有关的知识.

抛出一个异常的行为

个人认为接下来的这部分其实说的很经典，对我们理解异常行为/异常拷贝是很有帮助的.

条款12：理解“抛出一个异常”与“传递一个参数”或“调用一个虚函数”间的差异

从语法上看，在函数里声明参数与在catch子句中声明参数几乎没有什么差别：

```
class Widget { ... };                 //一个类，具体是什么类在这里并不重要
void f1(Widget w);                    // 一些函数，其参数分别为
void f2(Widget& w);                   // Widget, Widget&,或
void f3(const Widget& w);             // Widget* 类型
void f4(Widget *pw);
void f5(const Widget *pw);
catch(Widget w) ...                  //一些catch 子句，用来
catch(Widget& w)   ...               //捕获异常，异常的类型为
catch(const Widget& w) ...            // Widget, Widget&, 或
catch(Widget *pw) ...                 // Widget*
catch(const Widget *pw) ...
```

你因此可能会认为用throw抛出一个异常到catch子句中与通过函数调用传递一个参数两者基本相同.

这里面确有一些相同点，但是他们也存在着巨大的差异.

让我们先从相同点谈起.

你传递函数参数与异常的途径可以是传值、传递引用或传递指针，这是相同的.

但是当你传递参数和异常时，系统所要完成的操作过程则是完全不同的.

产生这个差异的原因是：你调用函数时，程序的控制权最终还会返回到函数的调用处，但是当你抛出一个异常时，控制权永远不会回到抛出异常的地方。

有这样一个函数，参数类型是Widget，并抛出一个Widget类型的异常：

```
// 一个函数，从流中读值到Widget中
istream operator>>(istream& s, Widget& w);
void passAndThrowWidget()
{
    Widget localWidget;
    cin >> localWidget;          //传递localWidget到 operator>>
    throw localWidget;           // 抛出localWidget异常
}
```

当传递localWidget到函数operator>>里，不用进行拷贝操作，而是把operator>>内的引用类型变量w指向localWidget，任何对w的操作实际上都施加到localWidget上.

这与抛出localWidget异常有很大不同.

不论通过传值捕获异常还是通过引用捕获（不能通过指针捕获这个异常，因为类型不匹配）都将进行lcalWidget的拷贝操作，也就说传递到catch子句中的是localWidget的拷贝.

必须这么做，因为当localWidget离开了生存空间后，其析构函数将被调用.

如果把localWidget本身（而不是它的拷贝）传递给catch子句，这个子句接收到的只是一个被析构了的Widget，一个Widget的“尸体”.

这是无法使用的。因此C++规范要求被做为异常抛出的对象必须被复制.

即使被抛出的对象不会被释放，也会进行拷贝操作.

例如如果passAndThrowWidget函数声明localWidget为静态变量（static）：

```
void passAndThrowWidget()
{
    static Widget localWidget;        // 现在是静态变量（static） 一直存在至程序结束
    cin >> localWidget;               // 象以前那样运行
    throw localWidget;                // 仍将对localWidget进行拷贝操作
}
```

当抛出异常时仍将复制出localWidget的一个拷贝.

这表示即使通过引用来捕获异常，也不能在catch块中修改localWidget;仅仅能修改localWidget的拷贝.

对异常对象进行强制复制拷贝，这个限制有助于我们理解参数传递与抛出异常的第二个差异：抛出异常运行速度比参数传递要慢.

当异常对象被拷贝时，拷贝操作是由对象的拷贝构造函数完成的.

该拷贝构造函数是对象的静态类型（static type）所对应类的拷贝构造函数，而不是对象的动态类型（dynamic type）对应类的拷贝构造函数.

比如以下这经过少许修改的passAndThrowWidget:

```
class Widget { ... };
class SpecialWidget: public Widget { ... };
void passAndThrowWidget()
{
    SpecialWidget localSpecialWidget;
    ...
    Widget& rw = localSpecialWidget;      // rw 引用SpecialWidget
    throw rw;                             //它抛出一个类型为Widget的异常
}
```

这里抛出的异常对象是Widget，即使rw引用的是一个SpecialWidget.

因为rw的静态类型（static type）是Widget，而不是SpecialWidget.

你的编译器根本没有主要到rw引用的是一个SpecialWidget。编译器所注意的是rw的静态类型（static type）.

这种行为可能与你所期待的不一样，但是这与在其他情况下C++中拷贝构造函数的行为是一致的.

（不过有一种技术可以让你根据对象的动态类型dynamic type进行拷贝，参见条款25）

异常是其它对象的拷贝，这个事实影响到你如何在catch块中再抛出一个异常.

比如下面这两个catch块，乍一看好像一样：

```
catch(Widget& w)                    // 捕获Widget异常
{
    ...                               // 处理异常
    throw;                            // 重新抛出异常，让它
}                                   // 继续传递
catch(Widget& w)                    // 捕获Widget异常
{
    ...                               // 处理异常
    throw w;                          // 传递被捕获异常的
}                                   // 拷贝
```

这两个catch块的差别在于第一个catch块中重新抛出的是当前捕获的异常，而第二个catch块中重新抛出的是当前捕获异常的一个新的拷贝.

如果忽略生成额外拷贝的系统开销，这两种方法还有差异么？
当然有。第一个块中重新抛出的是当前异常（current exception）,无论它是什么类型.

特别是如果这个异常开始就是做为SpecialWidget类型抛出的，那么第一个块中传递出去的还是SpecialWidget异常，即使w的静态类型（static type）是Widget.

这是因为重新抛出异常时没有进行拷贝操作.

第二个catch块重新抛出的是新异常，类型总是Widget，因为w的静态类型（static type）是Widget.

一般来说，你应该用throw来重新抛出当前的异常，因为这样不会改变被传递出去的异常类型，而且更有效率，因为不用生成一个新拷贝.

（顺便说一句，异常生成的拷贝是一个临时对象.

正如条款19解释的，临时对象能让编译器优化它的生存期（optimize it out of existence），

不过我想你的编译器很难这么做，因为程序中很少发生异常，所以编译器厂商不会在这方面花大量的精力）

让我们测试一下下面这三种用来捕获Widget异常的catch子句，异常是做为passAndThrowWidgetp抛出的：

```
catch (Widget w) ...                // 通过传值捕获异常
catch (Widget& w) ...               // 通过传递引用捕获异常
catch (const Widget& w) ...         //通过传递指向const的引用捕获异常
```

我们立刻注意到了传递参数与传递异常的另一个差异.

一个被异常抛出的对象（刚才解释过，总是一个临时对象）可以通过普通的引用捕获.

它不需要通过指向const对象的引用（reference-to-const）捕获.

在函数调用中不允许转递一个临时对象到一个非const引用类型的参数里（参见条款19），但是在异常中却被允许.

让我们先不管这个差异，回到异常对象拷贝的测试上来.

我们知道当用传值的方式传递函数的参数，我们制造了被传递对象的一个拷贝（参见Effective C++ 条款22），并把这个拷贝存储到函数的参数里.

同样我们通过传值的方式传递一个异常时，也是这么做的。当我们这样声明一个catch子句时：

```
catch (Widget w) ...                // 通过传值捕获
```

会建立两个被抛出对象的拷贝，一个是所有异常都必须建立的临时对象，第二个是把临时对象拷贝进w中.

同样，当我们通过引用捕获异常时：

```
catch (Widget& w) ...               // 通过引用捕获
catch (const Widget& w) ...         file://也通过引用捕获
```

这仍旧会建立一个被抛出对象的拷贝：拷贝是一个临时对象.

相反当我们通过引用传递函数参数时，没有进行对象拷贝.

当抛出一个异常时，系统构造的（以后会析构掉）被抛出对象的拷贝数比以相同对象做为参数传递给函数时构造的拷贝数要多一个.

我们还没有讨论通过指针抛出异常的情况，不过通过指针抛出异常与通过指针传递参数是相同的.

不论哪种方法都是一个指针的拷贝被传递.

你不能认为抛出的指针是一个指向局部对象的指针，因为当异常离开局部变量的生存空间时，该局部变量已经被释放.

Catch子句将获得一个指向已经不存在的对象的指针。这种行为在设计时应该予以避免.

对象从函数的调用处传递到函数参数里与从异常抛出点传递到catch子句里所采用的方法不同，

这只是参数传递与异常传递的区别的一个方面，第二个差异是在函数调用者或抛出异常者与被调用者或异常捕获者之间的类型匹配的过程不同.

比如在标准数学库（the standard math library）中sqrt函数：

```
double sqrt(double);      // from <cmath> or <math.h>
```

我们能这样计算一个整数的平方根，如下所示：

```
int i;
double sqrtOfi = sqrt(i);
```

毫无疑问，C++允许进行从int到double的隐式类型转换，所以在sqrt的调用中，i 被悄悄地转变为double类型，并且其返回值也是double.

（有关隐式类型转换的详细讨论参见条款5）一般来说，catch子句匹配异常类型时不会进行这样的转换.

见下面的代码：

```
void f(int value)
{
    try
    {
          if(someFunction())         // 如果 someFunction()返回
          {
```

```
                throw value;              //真，抛出一个整形值
                ...
          }
    }
    catch(double d)              // 只处理double类型的异常
    {
          ...
    }
    ...
}
```

在try块中抛出的int异常不会被处理double异常的catch子句捕获.

该子句只能捕获真真正正为double类型的异常；不进行类型转换.

因此如果要想捕获int异常，必须使用带有int或int&参数的catch子句.

不过在catch子句中进行异常匹配时可以进行两种类型转换.

第一种是继承类与基类间的转换.

一个用来捕获基类的catch子句也可以处理派生类类型的异常.

例如在标准C++库（STL）定义的异常类层次中的诊断部分（diagnostics portion ）(参见Effective C++ 条款49).

捕获runtime_errors异常的Catch子句可以捕获range_error类型和overflow_error类型的异常，

可以接收根类exception异常的catch子句能捕获其任意派生类异常.

这种派生类与基类（inheritance_based）间的异常类型转换可以作用于数值、引用以及指针上：

```
catch (runtime_error) ...               // can catch errors of type
catch (runtime_error&) ...              // runtime_error,
catch (const runtime_error&) ...        // range_error, or overflow_error
catch (runtime_error*) ...              // can catch errors of type
catch (const runtime_error*) ...        // runtime_error*,range_error*, oroverflow_error*
```

第二种是允许从一个类型化指针（typed pointer）转变成无类型指针（untyped pointer），

所以带有const void* 指针的catch子句能捕获任何类型的指针类型异常：

catch (const void*) ... file://捕获任何指针类型异常

传递参数和传递异常间最后一点差别是catch子句匹配顺序总是取决于它们在程序中出现的顺序.

因此一个派生类异常可能被处理其基类异常的catch子句捕获，即使同时存在有能处理该派生类异常的catch子句，与相同的try块相对应.

例如：

```
try
{
    ...
}
catch(logic_error& ex)                // 这个catch块 将捕获
{
    ...                                  // 所有的logic_error
}                                     // 异常，包括它的派生类
catch(invalid_argument& ex)           // 这个块永远不会被执行
{
    ...                                   //因为所有的invalid_argument异常 都被上面的catch子句捕获
}
```

与上面这种行为相反，当你调用一个虚拟函数时，被调用的函数位于与发出函数调用的对象的动态类型（dynamic type）最相近的类里.

你可以这样说虚拟函数采用最优适合法，而异常处理采用的是最先适合法.

如果一个处理派生类异常的catch子句位于处理基类异常的catch子句前面，编译器会发出警告.

（因为这样的代码在C++里通常是不合法的）

不过你最好做好预先防范：不要把处理基类异常的catch子句放在处理派生类异常的catch子句的前面.

上面那个例子，应该这样去写：

```
try
{
    ...
}
catch(invalid_argument& ex)            // 处理 invalid_argument
{
    ...
}
catch(logic_error& ex)                 // 处理所有其它的
{
    ...                                   // logic_errors异常
}
```

综上所述，把一个对象传递给函数或一个对象调用虚拟函数与把一个对象做为异常抛出，这之间有三个主要区别.

第一、异常对象在传递时总被进行拷贝；当通过传值方式捕获时，异常对象被拷贝了两次.

对象做为参数传递给函数时不需要被拷贝.

第二、对象做为异常被抛出与做为参数传递给函数相比，前者类型转换比后者要少（前者只有两种转换形式）.

最后一点，catch子句进行异常类型匹配的顺序是它们在源代码中出现的顺序，第一个类型匹配成功的catch将被用来执行.

当一个对象调用一个虚拟函数时，被选择的函数位于与对象类型匹配最佳的类里，即使该类不是在源代码的最前头.

灵巧指针

第一次用到灵巧指针是在写ADO代码的时候，用到com_ptr_t灵巧指针；但一直印象不是很深；

其实灵巧指针的作用很大，对我们来说垃圾回收，ATL等都会使用到它.

在More effective 的条款后面特意增加这个节点，不仅是想介绍它在异常处理方面的作用，还希望对编写别的类型代码的时候可以有所帮助.

smart pointer（灵巧指针）其实并不是一个指针,其实是某种形式的类.

不过它的特长就是模仿C/C++中的指针，所以就叫pointer 了.

所以希望大家一定要记住两点：smart pointer是一个类而非指针，但特长是模仿指针.

那怎么做到像指针的呢？

C++的模板技术和运算符重载给了很大的发挥空间.

首先smart pointer必须是高度类型化的（strongly typed ），模板给了这个功能.

其次需要模仿指针主要的两个运算符->和*，那就需要进行运算符重载.

详细的实现：

```
template<CLASS  T> class SmartPtr
{
public:
        SmartPtr(T* p = 0);
        SmartPtr(const SmartPtr& p);
        ~SmartPtr();
        SmartPtr& operator =(SmartPtr& p);
        T& operator*() const {return *the_p;}
        T* operator->() const {return the_p;}
private:
        T *the_p;
}
```

这只是一个大概的印象，很多东西是可以更改的.

比如可以去掉或加上一些const ，这都需要根据具体的应用环境而定.

注意重载运算符*和->，正是它们使smart pointer看起来跟普通的指针很相像.

而由于smart pointer是一个类，在构造函数、析构函数中都可以通过恰当的编程达到一些不错的效果.

举例：

比如C++标准库里的std::auto_ptr 就是应用很广的一个例子.

它的实现在不同版本的STL 中虽有不同，但原理都是一样，大概是下面这个样子：

```
template<CLASS  X> class auto_ptr
{
public:
    typedef X element_type;
    explicit auto_ptr(X* p = 0) throw():the_p(p) {}
    auto_ptr(auto_ptr& a) throw():the_p(a.release()) {}
    auto_ptr& operator =(auto_ptr& rhs) throw()
    {
          reset(rhs.release());
          return *this;
    }
    ~auto_ptr() throw() {delete the_p;}
    X& operator* () const throw() {return *the_p;}
    X* operator-> () const throw() {return the_p;}
    X* get() const throw() {return the_p;}
    X* release() throw()
    {
          X* tmp = the_p;
          the_p = 0;
          return tmp;

    }
    void reset(X* p = 0) throw()
    {
          if(the_p!=p)
          {
                delete the_p;
                the_p = p;
          }
    }
private:
    X* the_p;
};
```

关于auto_ptr 的使用可以找到很多的列子，这里不在举了.

它的主要优点是不用 delete ，可以自动回收已经被分配的空间，由此可以避免资源泄露的问题.

很多Java 的拥护者经常不分黑白的污蔑C++没有垃圾回收机制，其实不过是贻笑大方而已.

抛开在网上许许多多的商业化和非商业化的C++垃圾回收库不提， auto_ptr 就足以有效地解决这一问题.

并且即使在产生异常的情况下， auto_ptr 也能正确地回收资源.

这对于写出异常安全（exception-safe ）的代码具有重要的意义.

在使用smart pointer 的过程中，要注意的问题：

针对不同的smart pointer ，有不同的注意事项。比如auto_ptr ，就不能把它用在标准容器里，因为它只在内存中保留一份实例.

把握我前面说的两个原则：smart pointer 是类而不是指针，是模仿指针，那么一切问题都好办.

比如，smart pointer 作为一个类，那么以下的做法就可能有问题.

```
SmartPtr p;
if(p==0)
if(!p)
if(p)
```

很显然， p 不是一个真正的指针，这么做可能出错.

而SmartPtr 的设计也是很重要的因素.

您可以加上一个bool SmartPtr::null() const 来进行判断.

如果坚持非要用上面的形式， 那也是可以的,我们就加上operator void* ()试试：

```
template<CLASS  T> class SmartPtr
{
public: ...
        operator void*() const {return the_p;}
... private:
        T* the_p;
};
```

这种方法在basic_ios 中就使用过了。这里也可以更灵活地处理，比如类本身需要operator void*()这样地操作，

那么上面这种方法就不灵了。但我们还有重载operator !()等等方法来实现.
总结smart pointer的实质：
smart pointer 的实质就是一个外壳，一层包装。正是多了这层包装，我们可以做出许多普通指针无法完成的事，比如前面资源自动回收，或者自动进行引用记数，比如ATL 中CComPtr 和 CComQIPtr 这两个COM 接口指针类.

然而也会带来一些副作用，正由于多了这些功能，又会使 smart pointer 丧失一些功能.

WIN结构化异常

对使用WIN32平台的人来说，对WIN的结构化异常应该要有所了解的。WINDOWS的结构化异常是操作系统的一部分，而C++异常只是C++的一部分，当我们用C++编写代码的时候，我们选择C++的标准异常（也可以用MS VC的异常），编译器会自动的把我们的C++标准异常转化成SEH异常。

微软的Visual C++也支持C + +的异常处理，并且在内部实现上利用了已经引入到编译程序和Windows操作系统的结构化异常处理的功能。

SEH实际包含两个主要功能：结束处理（termination handling）和异常处理（exceptionhandling）.

在MS VC的FAQ中有关于SEH的部分介绍，这里摘超其中的一句：

“在VC5中，增加了新的/EH编译选项用于控制C++异常处理。C++同步异常处理(/EH)使得编译器能生成更少的代码，/EH也是VC的缺省模型。”

一定要记得在背后的事情：在使用SEH的时候，编译程序和操作系统直接参与了程序代码的执行。

Win32异常事件的理解

我写的另一篇文章：内存处理和DLL技术也涉及到了SEH中的异常处理。

Exception（异常处理） 分成软件和硬件exception2种.如：一个无效的参数或者被0除都会引起软件exception，而访问一个尚未commit的页会引起硬件exception.

发生异常的时候，执行流程终止，同时控制权转交给操作系统，OS会用上下文（CONTEXT）结构把当前的进程状态保存下来，然后就开始search 一个能处理exception的组件，search order如下：

首先检查是否有一个调试程序与发生exception的进程联系在一起，推算这个调试程序是否有能力处理

如上面不能完成，操作系统就在发生exception event的线程中search exception event handler

search与进程关联在一起的调试程序

系统执行自己的exception event handler code and terminate process

结束处理程序

利用SEH，你可以完全不用考虑代码里是不是有错误，这样就把主要的工作同错误处理分离开来.

这样的分离，可以使你集中精力处理眼前的工作，而将可能发生的错误放在后面处理.

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（table）来支持处理SEH的数据结构.

编译程序还必须提供回调（callback）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.

结束处理程序代码初步

一个结束处理程序能够确保去调用和执行一个代码块（结束处理程序，termination handler），

而不管另外一段代码（保护体， guarded body）是如何退出的。结束处理程序的语法结构如下： __try

```
{
   file://保护块
}
__finally
{
```

```
  file://结束处理程序
}
```

在上面的代码段中，操作系统和编译程序共同来确保结束处理程序中的__finally代码块能够被执行，不管保护体（try块）是如何退出的。

不论你在保护体中使用return，还是goto，或者是longjump，结束处理程序（finally块）都将被调用。

我们来看一个实列：（返回值：10, 没有Leak，性能消耗：小）

```
DWORD Func_SEHTerminateHandle()
{
    DWORD dwReturnData = 0;
    HANDLE hSem = NULL;
    const char* lpSemName = "TermSem";
    hSem =  CreateSemaphore(NULL, 1, 1, lpSemName);
    __try
    {
          WaitForSingleObject(hSem,INFINITE);
          dwReturnData = 5;
    }
    __finally
    {
          ReleaseSemaphore(hSem,1,NULL);
          CloseHandle(hSem);
    }
    dwReturnData += 5;
    return dwReturnData;
}
```

这段代码应该只是做为一个基础函数，我们将在后面修改它，来看看结束处理程序的作用.

在代码加一句：（返回值：5, 没有Leak，性能消耗：中下）

```
DWORD Func_SEHTerminateHandle()
{
    DWORD dwReturnData = 0;
    HANDLE hSem = NULL;
    const char* lpSemName = "TermSem";
    hSem =  CreateSemaphore(NULL, 1, 1, lpSemName);
    __try
    {
          WaitForSingleObject(hSem,INFINITE);
          dwReturnData = 5;
          return dwReturnData;
    }
    __finally
    {
          ReleaseSemaphore(hSem,1,NULL);
          CloseHandle(hSem);
    }
    dwReturnData += 5;
    return dwReturnData;
}
```

在try块的末尾增加了一个return语句.

这个return语句告诉编译程序在这里要退出这个函数并返回dwTemp变量的内容，现在这个变量的值是5.

但是，如果这个return语句被执行，该线程将不会释放信标，其他线程也就不能再获得对信标的控制.

可以想象，这样的执行次序会产生很大的问题，那些等待信标的线程可能永远不会恢复执行.

通过使用结束处理程序，可以避免return语句的过早执行.

当return语句试图退出try块时，编译程序要确保finally块中的代码首先被执行.

要保证finally块中的代码在try块中的return语句退出之前执行.

在程序中，将ReleaseSemaphore的调用放在结束处理程序块中，保证信标总会被释放.

这样就不会造成一个线程一直占有信标，否则将意味着所有其他等待信标的线程永远不会被分配CPU时间.

在finally块中的代码执行之后，函数实际上就返回.

任何出现在finally块之下的代码将不再执行，因为函数已在try块中返回,所以这个函数的返回值是5，而不是10.

读者可能要问编译程序是如何保证在try块可以退出之前执行finally块的.

当编译程序检查源代码时，它看到在try块中有return语句.

这样，编译程序就生成代码将返回值（本例中是5）保存在一个编译程序建立的临时变量中.

编译程序然后再生成代码来执行finally块中包含的指令，这称为局部展开.

更特殊的情况是，由于try块中存在过早退出的代码，从而产生局部展开，导致系统执行finally块中的内容.

在finally块中的指令执行之后，编译程序临时变量的值被取出并从函数中返回.

可以看到，要完成这些事情，编译程序必须生成附加的代码，系统要执行额外的工作.

在不同的CPU上，结束处理所需要的步骤也不同.

例如，在Alpha处理器上，必须执行几百个甚至几千个CPU指令来捕捉try块中的过早返回并调用finally块.

在编写代码时，就应该避免引起结束处理程序的try块中的过早退出，因为程序的性能会受到影响.

后面，将讨论__leave关键字，它有助于避免编写引起局部展开的代码.

设计异常处理的目的是用来捕捉异常的—不常发生的语法规则的异常情况（在我们的例子中，就是过早返回）.

如果情况是正常的，明确地检查这些情况，比起依赖操作系统和编译程序的SEH功能来捕捉常见的事情要更有效.

注意当控制流自然地离开try块并进入finally块（就像在Funcenstein1中）时，进入finally块的系统开销是最小的.

在x86CPU上使用微软的编译程序，当执行离开try块进入finally块时，只有一个机器指令被执行，读者可以在自己的程序中注意到这种系统开销.

当编译程序要生成额外的代码，系统要执行额外的工作时系统开销就很值得注意了.

修改代码：（返回值：5，没有Leak，性能消耗：中）

```
DWORD Func_SEHTerminateHandle()
{
     DWORD dwReturnData = 0;
     HANDLE hSem = NULL;
     const char* lpSemName = "TermSem";
     hSem =  CreateSemaphore(NULL, 1, 1, lpSemName);
     __try
     {
           WaitForSingleObject(hSem,INFINITE);
           dwReturnData = 5;
           if(dwReturnData == 5)
                goto ReturnValue;
           return dwReturnData;
     }
     __finally
     {
           ReleaseSemaphore(hSem,1,NULL);
           CloseHandle(hSem);
     }
     dwReturnData += 5;
ReturnValue:
     return dwReturnData;
}
```

代码中，当编译程序看到try块中的goto语句，它首先生成一个局部展开来执行finally块中的内容.

这一次，在finally块中的代码执行之后，在ReturnValue标号之后的代码将执行，因为在try块和finally块中都没有返回发生.

这里的代码使函数返回5,而且，由于中断了从try块到finally块的自然流程，可能要蒙受很大的性能损失（取决于运行程序的CPU）

写上面的代码是初步的，现在来看结束处理程序在我们代码里面的真正的价值：

看代码：（信号灯被正常释放，reserve的一页内存没有被Free，安全性：安全）

```
DWORD TermHappenSomeError()
{
     DWORD dwReturnValue = 9;
     DWORD dwMemorySize = 1024;

     char* lpAddress;
     lpAddress = (char*)VirtualAlloc(NULL, dwMemorySize, MEM_RESERVE, PAGE_READWRITE);
}
```

finally块的总结性说明

我们已经明确区分了强制执行finally块的两种情况：

从try块进入finally块的正常控制流.

•局部展开：从try块的过早退出（goto、longjump、continue、break、return等）强制控制转移到finally块.

第三种情况，全局展开（globalunwind），在发生的时候没有明显的标识，我们在本章前面Func_SEHTerminate函数中已经见到.在Func_SEHTerminate的try块中，有一个对TermHappenSomeError函数的调用。TermHappenSomeError函数会引起一个内存访问违规（memory access violation），一个全局展开会使Func_SEHTerminate函数的finally块执行.

由于以上三种情况中某一种的结果而导致finally块中的代码开始执行。为了确定是哪一种情况引起finally块执行，可以调用内部函数AbnormalTermination：这个内部函数只在finally块中调用，返回一个Boolean值.指出与finally块相结合的try块是否过早退出。换句话说，如果控制流离开try块并自然进入finally块，AbnormalTermination将返回FALSE。如果控制流非正常退出try块—通常由于goto、return、break或continue语句引起的局部展开，或由于内存访问违规或其他异常引起的全局展开—对AbnormalTermination的调用将返回TRUE。没有办法区别finally块的执行是由于全局展开还是由于局部展开.

但这通常不会成为问题，因为可以避免编写执行局部展开的代码.（注意内部函数是编译程序识别的一种特殊函数。编译程序为内部函数产生内联（inline）代码而不是生成调用函数的代码。例如，memcpy是一个内部函数（如果指定/Oi编译程序开关）。当编译程序看到一个对memcpy的调用，它直接将memcpy的代码插入调用memcpy的函数中，而不是生成一个对memcpy函数的调用。其作用是代码的长度增加了，但执行速度加快了。

在继续之前，回顾一下使用结束处理程序的理由：

•简化错误处理，因所有的清理工作都在一个位置并且保证被执行。

•提高程序的可读性。

•使代码更容易维护。

•如果使用得当，具有最小的系统开销。

异常处理程序

异常是我们不希望有的事件。在编写程序的时候，程序员不会想去存取一个无效的内存地址或用0来除一个数值。不过，这样的错误还是常常会发生的。CPU负责捕捉无效内存访问和用0除一个数值这种错误，并相应引发一个异常作为对这些错误的反应。CPU引发的异常，就是所谓的硬件异常（hardwareexception）。在本章的后面，我们还会看到操作系统和应用程序也可以引发相应的异常，称为软件异常（softwareexception）。

当出现一个硬件或软件异常时，操作系统向应用程序提供机会来考察是什么类型的异常被引发，并能够让应用程序自己来处理异常。下面就是异常处理程序的语法：

```
__try
{
     //保护块
```

```
}
__except(异常过虑器)
{
    //异常处理程序
}
```

注意__except关键字。每当你建立一个try块，它必须跟随一个finally块或一个except块。一个try块之后不能既有finally块又有except块。但可以在try-except块中嵌套try-finally块，反过来也可以。
异常处理程序代码初步
与结束处理程序不同，异常过滤器（exceptionfilter）和异常处理程序是通过操作系统直接执行的，编译程序在计算异常过滤器表达式和执行异常处理程序方面不做什么事。

下面几节的内容举例说明try-except块的正常执行，解释操作系统如何以及为什么计算异常过滤器，并给出操作系统执行异常处理程序中代码的环境。

本来想把代码全部写出来的，但是实在是写这边文挡化的时间太长了，所以接下来就只是做说明，而且try和except块比较简单。

尽管在结束处理程序的try块中使用return、goto、continue和break语句是被强烈地反对，但在异常处理程序的try块中使用这些语句不会产生速度和代码规模方面的不良影响。

这样的语句出现在与except块相结合的try块中不会引起局部展开的系统开销.

当引发了异常时，系统将定位到except块的开头，并计算异常过滤器表达式的值，过滤器表达式的结果值只能是下面三个标识符之一，这些标识符定义在windows的Except.h文件中。标识符定义为：

```
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
```

下面将讨论这些标识符如何改变线程的执行。

下面的流程概括了系统如何处理一个异常的情况：(这里的流程假设是正向的)

开始 -> 执行一个CPU指令 -> {是否有异常被引发} -> 是 -> 系统确定最里层的try 块 -> {这个try块是否有一个except块} -> 是 -> {过滤器表达式的值是什么} ->异常执行处理程序 -> 全局展开开始 -> 执行except块中的代码 -> 在except块之后执行继续

EXCEPTION_EXECUTE_HANDLER

在异常过滤器表达式的值如果是EXCEPTION_EXECUTE_HANDLER，这个值的意思是要告诉系统：“我认出了这个异常.

即，我感觉这个异常可能在某个时候发生，我已编写了代码来处理这个问题，现在我想执行这个代码”

在这个时候，系统执行一个全局展开，然后执行向except块中代码（异常处理程序代码）的跳转.

在except块中代码执行完之后，系统考虑这个要被处理的异常并允许应用程序继续执行。这种机制使windows应用程序可以抓住错误并处理错误，再使程序继续运行，不需要用户知道错误的发生。但是，当except块执行后，代码将从何处恢复执行？稍加思索，我们就可以想到几种可能性:

第一种可能性是从产生异常的CPU指令之后恢复执行。这看起来像是合理的做法，但实际上，很多程序的编写方式使得当前面的指令出错时，后续的指令不能够继续成功地执行。

代码应该尽可能地结构化，这样，在产生异常的指令之后的CPU指令有望获得有效的返回值。例如，可能有一个指令分配内存，后面一系列指令要执行对该内存的操作。

如果内存不能够被分配，则所有后续的指令都将失败，上面这个程序重复地产生异常。

所幸的是，微软没有让系统从产生异常的指令之后恢复指令的执行。这种决策使我们免于面对上面的问题。

第二种可能性是从产生异常的指令恢复执行。这是很有意思的可能性。

如果在except块中有这样的语句会怎么样呢：在except块中有了这个赋值语句，可以从产生异常的指令恢复执行。

这一次，执行将继续，不会产生其他的异常。可以做些修改，让系统重新执行产生异常的指令。

你会发现这种方法将导致某些微妙的行为。我们将在EXCEPTION_CONTINUE_EXECUTION一节中讨论这种技术。

第三种可能性是从except块之后的第一条指令开始恢复执行。这实际是当异常过滤器表达式的值为EXCEPTION_EXECUTE_HANDLER时所发生的事。在except块中的代码结束执行后，控制从except块之后的第一条指令恢复。

c++异常参数传递

从语法上看，在函数里声明参数与在catch子句中声明参数是一样的，catch里的参数可以是值类型，引用类型，指针类型.例如：

```
try
{
    .....
}
catch(A a)
{
    .....
}
catch(B& b)
{
    .....
}
catch(C* c)
{
    .....
}
```

尽管表面是它们是一样的，但是编译器对二者的处理却又很大的不同.

调用函数时，程序的控制权最终还会返回到函数的调用处，但是抛出一个异常时，控制权永远不会回到抛出异常的地方.

```
class A;
void func_throw()
{
    A a;
    throw a;  //抛出的是a的拷贝，拷贝到一个临时对象里
}
try
{
    func_throw();
}
```

```
catch(A a)  //临时对象的拷贝
{
    ...
}
```

当我们抛出一个异常对象时，抛出的是这个异常对象的拷贝。当异常对象被拷贝时，拷贝操作是由对象的拷贝构造函数完成的.

该拷贝构造函数是对象的静态类型(static type)所对应类的拷贝构造函数，而不是对象的动态类型(dynamic type)对应类的拷贝构造函数。此时对象会丢失RTTI信息.
异常是其它对象的拷贝，这个事实影响到你如何在catch块中再抛出一个异常。比如下面这两个catch块，乍一看好像一样:

```
catch(A& w)  // 捕获异常
{
      // 处理异常
      throw; // 重新抛出异常，让它继续传递
}
catch(A& w)  // 捕获Widget异常
{
      // 处理异常
      throw w; // 传递被捕获异常的拷贝
}
```

第一个块中重新抛出的是当前异常(current exception),无论它是什么类型。（有可能是A的派生类）
第二个catch块重新抛出的是新异常，失去了原来的类型信息.
一般来说，你应该用throw来重新抛出当前的异常，因为这样不会改变被传递出去的异常类型，而且更有效率，因为不用生成一个新拷贝.
看看以下这三种声明:

```
catch (A w) ... // 通过传值
catch (A& w) ... // 通过传递引用
catch (const A& w) ... //const引用
```

一个被异常抛出的对象(总是一个临时对象)可以通过普通的引用捕获;它不需要通过指向const对象的引用(reference-to-const)捕获.

在函数调用中不允许转递一个临时对象到一个非const引用类型的参数里，但是在异常中却被允许.
回到异常对象拷贝上来,我们知道，当用传值的方式传递函数的参数，我们制造了被传递对象的一个拷贝，并把这个拷贝存储到函数的参数里.

同样我们通过传值的方式传递一个异常时，也是这么做的当我们这样声明一个catch子句时:
catch (A w) ... // 通过传值捕获
会建立两个被抛出对象的拷贝，一个是所有异常都必须建立的临时对象，第二个是把临时对象拷贝进w中。实际上，编译器会优化掉一个拷贝。同样，当我们通过引用捕获异常时，
catch (A& w) ... // 通过引用捕获
catch (const A& w) ... //const引用捕获
这仍旧会建立一个被抛出对象的拷贝:拷贝是一个临时对象。相反当我们通过引用传递函数参数时，没有进行对象拷贝.

话虽如此，但是不是所有编译器都如此,VS200就表现很诡异.

通过指针抛出异常与通过指针传递参数是相同的.

不论哪种方法都是一个指针的拷贝被传递,你不能认为抛出的指针是一个指向局部对象的指针，因为当异常离开局部变量的生存空间时，该局部变量已经被释放.

Catch子句将获得一个指向已经不存在的对象的指针。这种行为在设计时应该予以避免.

另外一个重要的差异是在函数调用者或抛出异常者与被调用者或异常捕获者之间的类型匹配的过程不同.

在函数传递参数时，如果参数不匹配，那么编译器会尝试一个类型转换，如果存在的话。而对于异常处理的话，则完全不是这样。见一下的例子：

```
void func_throw()
{
    CString a;
    throw a;  //抛出的是a的拷贝，拷贝到一个临时对象里
}
try
{
    func_throw();
}
catch(const char* s)
{
    ...
}
```

抛出的是CString，如果用const char*来捕获的话，是捕获不到这个异常的.

尽管如此，在catch子句中进行异常匹配时可以进行两种类型转换.第一种是基类与派生类的转换，一个用来捕获基类的catch子句也可以处理派生类类型的异常.

反过来，用来捕获派生类的无法捕获基类的异常.

第二种是允许从一个类型化指针(typed pointer)转变成无类型指针(untyped pointer)，所以带有const void* 指针的catch子句能捕获任何类型的指针类型异常:
catch (const void*) ... //可以捕获所有指针异常
另外，你还可以用catch(...)来捕获所有异常，注意是三个点.
传递参数和传递异常间最后一点差别是catch子句匹配顺序总是取决于它们在程序中出现的顺序.

因此一个派生类异常可能被处理其基类异常的catch子句捕获，这叫异常截获，一般的编译器会有警告.

```
class A
{
public:
    A()
    {
        cout << "class A creates" << endl;
    }
    void print()
    {
        cout << "A" << endl;
    }
```

```
    ~A()
    {
          cout << "class A destruct" << endl;
    }
};
class B: public A
{
public:
    B()
    {
          cout << "class B create" << endl;
    }
    void print()
    {
          cout << "B" << endl;
    }
    ~B()
    {
          cout << "class B destruct" << endl;
    }
};
void func()
{
    B b;
    throw b;
}
try
{
    func();
}
catch(B& b)  //必须将B放前面，如果把A放前面，B放后面，那么B类型的异常会先被截获。
{
    b.print();
}
catch(A& a)
{
    a.print() ;
}
```

相反的是，当你调用一个虚拟函数时，被调用的函数位于与发出函数调用的对象的动态类型(dynamic type)最相近的类里.

你可以这样说虚拟函数匹配采用最优匹配法，而异常处理匹配采用的是最先匹配法.

附：

异常的描述

函数和函数可能抛出的异常集合作为函数声明的一部分是有价值的，例如

```
void f(int a) throw(x2,x3);
```

表示f()只能抛出两个异常x2,x3,以及这些类型派生的异常，但不会抛出其他异常.

如果f函数违反了这个规定，抛出了x2,x3之外的异常,例如x4,那么当函数f抛出x4异常时，

会转换为一个std::unexpected()调用，默认是调用std::terminate(),通常是调用abort().

如果函数不带异常描述，那么假定他可能抛出任何异常,例如:

```
int f();  //可能抛出任何异常
```

不带任何异常的函数可以用空表表示:

```
int g() throw();  // 不会抛出任何异常
```