C++异常机制

c++标准库中定义了12种异常，很多大公司的C++编码规范明确禁止使用异常。

C++异常机制概述

用于在程序中处理异常事件。

异常处理过程：

1. 异常事件发生时，程序使用throw关键字抛出异常表达式，由操作系统为程序设置当前异常对象
2. 包含了异常出现点的最内层的**try**块，依次匹配catch语句中的异常对象
3. 若匹配成功，则执行catch块内的异常处理语句，然后接着执行**try...catch...**块之后的代码。
4. 如果在当前的try...catch...块内找不到匹配该异常对象的catch语句,则由更外层的try...catch...块来处理该异常；
5. 如果当前函数内所有的try...catch...块都不能匹配该异常，则递归回退到调用栈的上一层去处理该异常。
6. 如果一直退到主函数main()都不能处理该异常，则调用系统函数terminate()终止程序。

throw 关键字

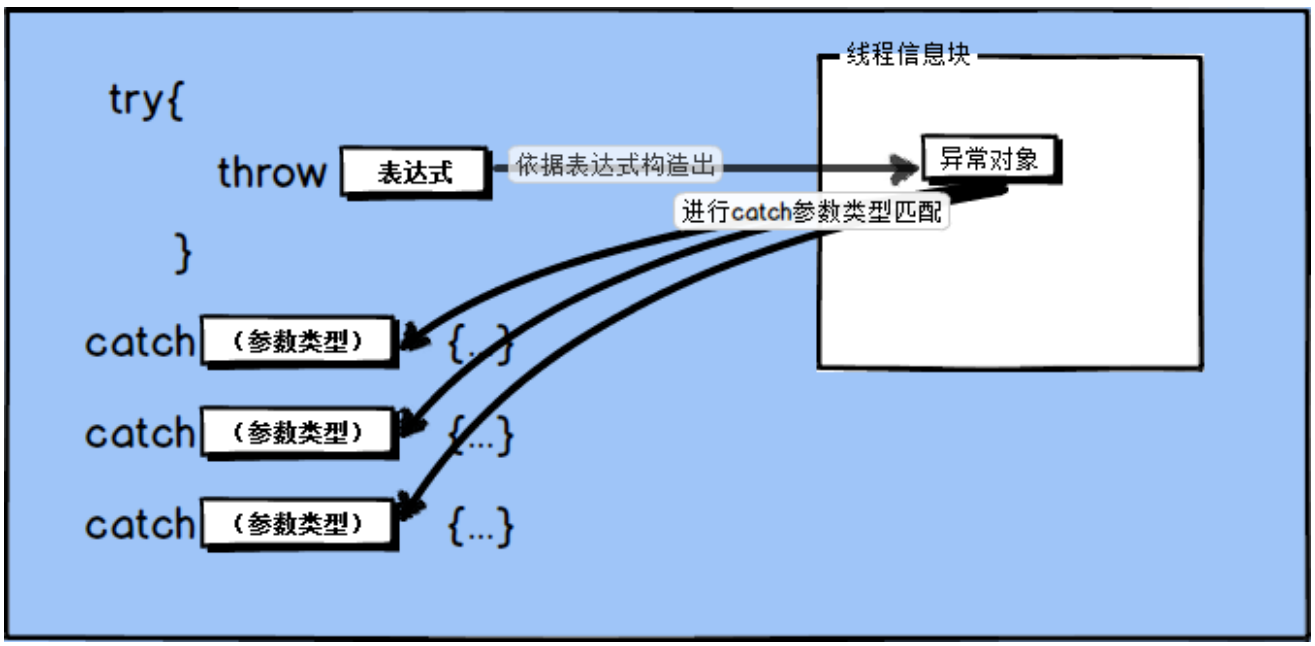
在上面这个示例中，**throw**是个关键字，与抛出表达式构成了throw语句。其语法为

```
throw 表达式;
```

throw语句必须包含在try块中

执行throw语句时，throw表达式将作为对象被复制构造为一个新的对象，称为异常对象。

异常对象放在内存的特殊位置，该位置既不是栈也不是堆，在window上是放在线程信息块TIB中。这个构造出来的新对象与本级的try所对应的catch语句进行类型匹配，类型匹配的原则在下面介绍。



异常对象

异常对象是一种特殊的对象，编译器依据异常抛出表达式复制构造异常对象，这要求抛出异常表达式不能是一个不完全类型

在函数中返回局部变量的引用或指针几乎肯定会造成错误，同样的道理，在throw语句中抛出局部变量的指针或引用也几乎是错误的行为。如果指针所指向的变量在执行catch语句时已经被销毁，对指针进行解引用将发生意想不到的后果。

throw出一个表达式时，该表达式的静态编译类型将决定异常对象的类型。所以当throw出的是基类指针的解引用，而该指针所指向的实际对象是派生类对象，此时将发生派生类对象切割。

除了抛出用户自定义的类型外，C++标准库定义了一组类，用户报告标准库函数遇到的问题。这些标准库异常类只定义了几种运算，包括创建或拷贝异常类型对象，以及为异常类型的对象赋值。

标准异常类	描述	头文件
exception	最通用的异常类，只报告异常的发生而不提供任何额外的信息	exception
runtime_error	只有在运行时才能检测出的错误	stdexcept
rang_error	运行时错误：产生了超出有意义值域范围的结果	stdexcept
overflow_error	运行时错误：计算上溢	stdexcept
underflow_error	运行时错误：计算下溢	stdexcept
logic_error	程序逻辑错误	stdexcept
domain_error	逻辑错误：参数对应的结果值不存在	stdexcept
invalid_argument	逻辑错误：无效参数	stdexcept
length_error	逻辑错误：试图创建一个超出该类型最大长度的对象	stdexcept
out_of_range	逻辑错误：使用一个超出有效范围的值	stdexcept
bad_alloc	内存动态分配错误	new
bad_cast	dynamic_cast类型转换出错	type_info

catch 关键字

catch语句匹配被抛出的异常对象。

在进行异常对象的匹配时，编译器不会做任何的隐式类型转换或类型提升。除了以下几种情况外，异常对象的类型必须与catch语句的声明类型完全匹配：

- 允许从非常量到常量的类型转换。
- 允许派生类到基类的类型转换。
- 数组被转换成指向数组（元素）类型的指针。
- 函数被转换成指向函数类型的指针。

寻找catch语句的过程中，匹配上的未必是类型完全匹配那项，而在是最靠前的第一个匹配上的catch语句（我称它为最先匹配原则）。所以，派生类的处理代码catch语句应该放在基类的处理catch语句之前，否则先匹配上的总是参数类型为基类的catch语句，而能够精确匹配的catch语句却不能够被匹配上。

使用catch(...){}可以捕获所有类型的异常，根据最先匹配原则，catch(...){}应该放在所有catch语句的最后面，否则无法让其他可以精确匹配的catch语句得到匹配。通常在catch(...){}语句中执行当前可以做的处理，然后再重新抛出异常。注意，catch中重新抛出的异常只能被外层的catch语句捕获。

异常机制与构造函数

异常机制的一个合理的使用是在构造函数中。

构造函数没有返回值，所以应该使用异常机制来报告发生的问题。更重要的是，**构造函数抛出异常表明构造函数还没有执行完，其对应的析构函数不会自动被调用，因此析构函数应该先析构所有所有已初始化的基对象，成员对象，再抛出异常。**

```
myClass::myClass(type1 pa1)
    try:  _myClass_val (初始化值)
{
  /*构造函数的函数体 */
}
  catch ( exception& err )
{
  /* 构造函数的异常处理部分 */
};
```

异常机制与析构函数

析构函数被期望不向外抛出异常。

析构函数中向函数外抛出异常，将直接调用terminator()系统函数终止程序。

如果一个析构函数内部抛出了异常，就应该在析构函数的内部捕获并处理该异常，不能让异常被抛出析构函数之外。可以如此处理：

- 若析构函数抛出异常，调用std::abort()来终止程序。
- 在析构函数中catch捕获异常并作处理。

noexcept操作符

在C++11中，编译器并不会在编译期检查函数的noexcept声明，因此，被声明为noexcept的函数若携带异常抛出语句还是可以通过编译的。在函数运行时若抛出了异常，编译器可以选择直接调用terminate()函数来终结程序的运行，因此，noexcept的一个作用是**阻止异常的传播,提高安全性**

上面一点提到了，我们不能让异常逃出析构函数，因为那将导致程序的不明确行为或直接终止程序。实际上出于安全的考虑，C++11标准中让类的析构函数默认也是noexcept的。 同样是为了安全性的考虑，经常被析构函数用于释放资源的delete函数，C++11也默认将其设置为noexcept。

常量表达式的结果会被转换成bool类型，noexcept(bool)表示函数不会抛出异常，noexcept(false)则表示函数有可能会抛出异常。故若你想更改析构函数默认的noexcept声明，可以显式地加上noexcept(false)声明，但这并不会带给你什么好处。

异常处理的性能分析

当抛出一个异常时，必须确定异常是不是从try块中抛出。异常处理机制为了完善异常和它的处理器之间的匹配，需要存储每个异常对象的类型信息以及catch语句的额外信息。由于异常对象可以是任何类型（如用户自定义类型），并且也可以是多态的，获取其动态类型必须要使用运行时类型检查（RTTI），此外还需要运行期代码信息和关于每个函数的结构。

当异常抛出点所在函数无法解决异常时，异常对象沿着调用链被传递出去，程序的控制权也发生了转移。转移的过程中为了将异常对象的信息携带到程序执行处（如对异常对象的复制构造或者catch参数的析构），在时间和空间上都要付出一定的代价，本身也有不安全性，特别是异常对象是个复杂的类的时候。

异常处理技术在不同平台以及编译器下的实现方式都不同，但都会给程序增加额外的负担，当异常处理被关闭时，额外的数据结构、查找表、一些附加的代码都不会被生成，正是因为如此，对于明确不抛出异常的函数，我们需要使用noexcept进行声明。