

Exception Handling

异常处理

董洪伟

<http://hwdong.com>

Error Handling/错误处理

- 看似正确的程序在特殊情况下可能出错。如两个整数相加的程序，用户输入的是字符串，如果程序没有考虑到该情况，则会出错！
- 可以在各种极端情况下都能正确应对的程序是健壮的(Robust)
- 大型程序通常是分层的，或者是模块化的。有底层的库（API函数）调用，有高层的用户接口。不同层次的模块发现异常情况后该如何处理？

Error Handling/错误处理

- 来看一种普通的航空订票系统：它的最高端是由一些GUI组件所组成，用来在用户的屏幕上显示内容并与用户交互。这些高端组件与那些封装了数据库API的数据存取对象相互作用。再往底层一些，那些数据库API与数据库引擎相交互，然而数据库引擎自己又会调用系统服务来处理底层的硬件资源，比如物理内存，文件系统和安全模型。

Error Handling/错误处理

- 一般情况下，极其严格的运行期错误会在这些底层代码中被检测出来，但是它们不能-----或者说不应该----试图自己处理这些错误。解决这些严格的运行期错误的责任应该由高端组件来承担。为了解决一个错误，高端组件必须得到错误发生的通知。本质上，错误处理包括错误检测和通知高端组件。这些高端组件依次处理错误并且试图从错误中恢复。

Error Handling-抛出和捕获

- 库的作者可以检查出运行时错误,但一般说根本不知道如何处理;库的用户知道怎样处理错误,但却无法去检查它们. 如`fopen()`函数知道文件是否正常打开了,但用户不知道哪些文件可以正常打开,用户要依赖`fopen()`去发现打开文件错误.
- 一个函数在发现自己无法处理的错误时,可以抛出(*throw*)一个异常,希望它的高层调用者能够处理,调用者可以捕获(*catch*)自己能够处理的错误 ----这就是C++异常处理的过程.

传统处理错误技术:

- [1] terminate the program/终止程序
- [2] return a value representing “error,”/返回表示错误的值
- [3] return a legal value and leave the program in an illegal state, ,”/返回合法值,但使程序处于错误状态(如设置一个全局错误变量)
- [4] call a function supplied to be called in case of “error.”/当错误发生时,调用预定的错误处理程序

[1]终止程序-exit()和abort()

- 在标准C的函数库中有两个函数用来终止一个程序：exit()和abort()。
- exit()首先会清空流和关闭打开的文件。abort()却不一样，它表示程序被意外终止，不会清空流和关闭打开的文件。
- 是异常未被捕获情况下缺省发生的事情.是一种最残酷的方法。
- 突然终止可能使一些资源不能得到正确的释放，数据未及时保存。如股票交易等。

[2]返回表示错误的值

- 错误码很难统一。因为一个库的实现者可能选择返回值0来代表一个错误，然而另一个实现者却选择0来代表成功并且用那些非0值代表出现错误。微软从来就没有有一个统一的错误码查询表，也不可能做到。
- 对于每个调用都要检查返回值,可使代码倍增.
- 有时根本不能返回值。如构造函数。

[3]使程序处于错误状态

- 同[2]有相同的问题,而且调用者有时可能未意识到程序已处于非法状态. (如C的<errno.h> 中定义了一个全局变量**errno**)
- 在一个多线程环境中, 被一个线程赋予的一个错误码**errno**, 有可能不经意的被另一个线程所改写, 而调用者还未对**errno**进行检查。

[4]当错误发生时,调用预定的错误处理程序

- 如果缺少关于异常情况的信息,该错误处理程序仍然无能为力,也只能采用类似前三种的方法.

异常处理机制是一种更规范的错误处理技术

- 异常处理是一种把控制权从异常发生的地点转移到一个匹配的handler的机制。它将发现错误和处理错误的部分相互分离,即将错误处理代码从”正常”代码中分离出来,使程序更易阅读.
- 异常对象可以是内在类型变量或用户定义类型的对象, 可提供更多信息,异常处理使系统从错误中恢复过来,提高了程序的容错能力.
- 异常处理机制由四部分: try 块, 一个或多个和try 块相关的处理器handler (catch块), throw语句, 以及异常对象自己。

异常处理

- 异常处理的默认相应方式是终止程序,而传统的程序可以”装糊涂”继续运行.
- 异常处理可以处理同步异常,如数组范围检查和I/O出错,不能处理异步异常.单击鼠标(交互行为)、网络消息达到、I/O完成等。
- 异常处理机制是基于堆栈回退的非局部控制机制，可以完成局部变量的销毁工作。
- 如果程序的某些部分出现无法处理的情况，可通知高层的环境去处理它以便从这个“异常”中恢复过来。异常多用于程序不同组件之间。

异常处理-示例

```
#include <iostream.h>
int fun(){
    char *buf;
    try {
        buf = new char[512];
        if( buf == 0 )
            throw "Memory allocation failure!" ;
    }
    catch( char * str ) {
        cout << "Exception raised: " << str << '\n';
    }
    // ...
    return 0;
}
```

异常四要素：

- **异常对象**：可以是任何类型，但一般不会用int、string等，而是用一个用户定义对象表示异常。因为要使它们好分辨，同时包含更多异常信息。
- **try块**放上要检查异常的代码
- 每个**catch块**指定捕获的异常和处理这些异常的异常处理器
- **throw**用来抛出异常对象或者说明抛出哪些异常对象。

Stack Unwinding /栈回退

- 当一个异常被抛出，运行时机制首先在当前的作用域寻找合适的handler。如果不存在这样一个handler，那么将会离开当前的作用域，进入更外围的一层继续寻找。这个过程不断的进行下去直到合适的handler被找到为止。此时堆栈已经被解开，并且所有的局部对象被销毁。
- 如果始终都没有找到合适的handler，那么程序将会终止。

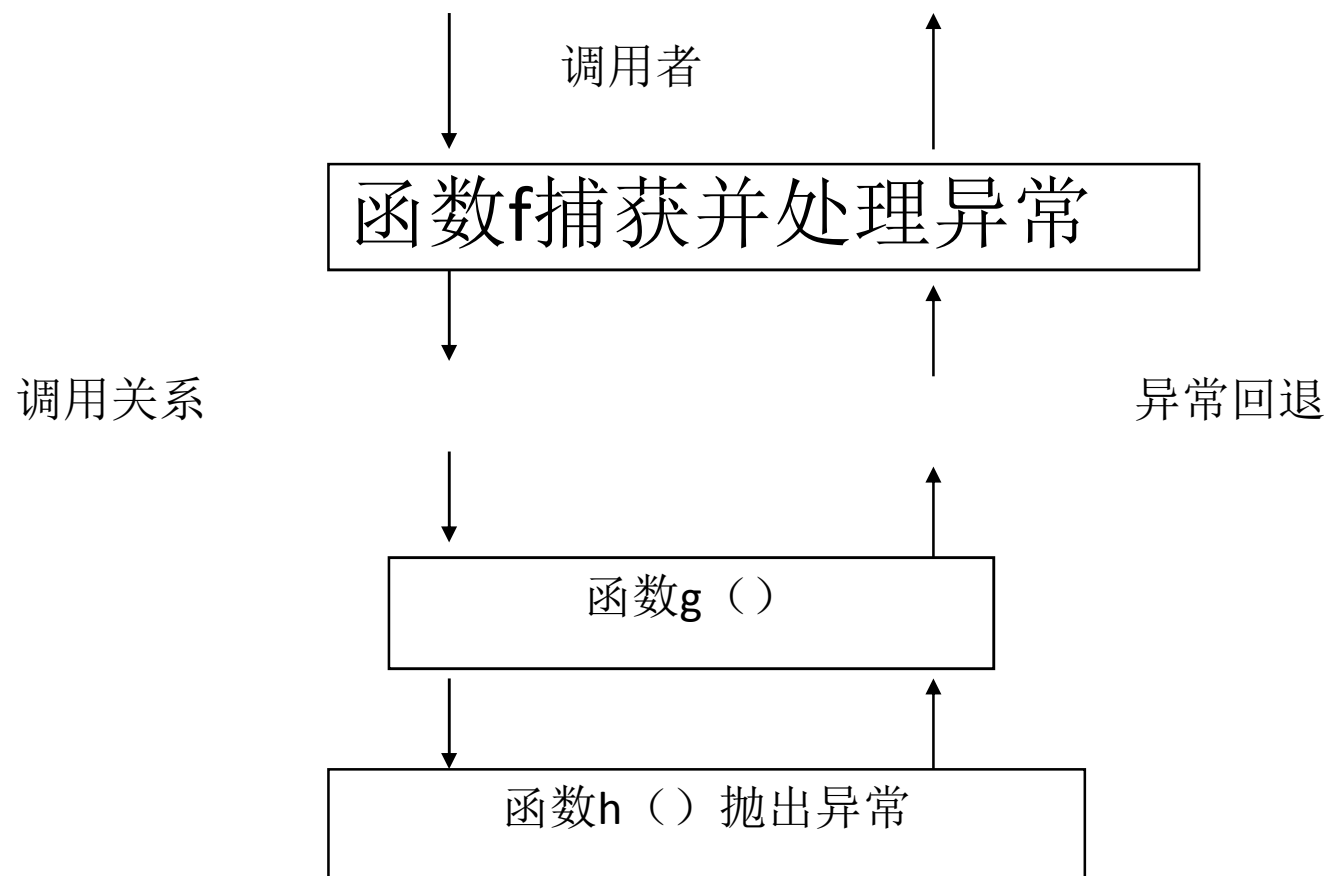
捕获任何异常

- 注意，C++保证局部对象被适当的销毁仅仅是在抛出的异常被处理的情况下。一个未被捕获异常是否引起局部对象的销毁由实现决定的。为了保证局部对象的析构函数在异常未被捕获情况下也能够被正常调用，你应该在`main()`里加入捕获任何异常的`catch`语句。

捕获任何异常

```
int main(){
    try {
        //...
    }
    catch(std::exception& stdexc) // exception handler
    {
        //...
    }
    catch(...) //
    {
        //...
    }
    return 0;
}
```

stack unwinding-类似函数调用返回



Grouping of Exceptions/异常的组织

- 一个异常就是某个表示异常类的对象。
- 异常类型可以是任何类型，但习惯上将它们组织成一定结构如层次结构。从而对异常可以分类。如某数学库的异常可能：

```
class Matherr { };  
class Overflow: public Matherr { };  
class Underflow: public Matherr { };  
class Zerodivide: public Matherr { };  
// ...
```

异常的组织

- 可以处理所有的数学异常，也可以处理特定的数学异常

```
void f(){  
    try {  
        // ...  
    }  
    catch (Overflow) {  
        // handle Overflow or anything derived from Overflow  
    }  
    catch (Matherr) {  
        // handle any Matherr that is not Overflow  
    }  
}
```

某些算术错误未作为异常

- 对某些算术错误，由于许多流水线系统结构中都是非同步的操作，所以未作为异常。如除0

Derived Exceptions/派生的异常

- 由于异常通常被组织成一个层次结构，即异常类型之间是派生关系。并且捕获和命名异常的语义等同于函数接受参数的语义，即用实际参数对形式参数初始化。那么抛出的异常在被捕获时就会产生切割问题。
- 解决方法与函数参数一样，即传递异常对象的指针或引用。

异常对象被切割

```
void g() throw Overflow
{
    //...
    throw Overflow();
}
void f(){
    try {
        g() ;
    }
    catch (Matherr m) {
        // ...
    }
}
```

避免切割：传异常的引用

```
int add(int x, int y){
    if ((x>0 && y>0 && x>INT_MAXy) || (x<0 && y<0 && x<INT_MINy))
        throw Int_overflow("+", x, y);
    return x+y; // x+y will not overflow
}

void f(){
    try {
        int i1 = add(1,2); int i2 = add(INT_MAX,-2);
        int i3 = add(INT_MAX,2); // here we go!
    }
    catch (Matherr& m) {
        // ...
        m.debug_print();
    }
}
```


Exceptions in Constructors

构造函数内的异常

- 异常为从构造函数内报告错误提供了一个解决方案。由于构造函数不能返回值，所以传统的解决方法可能有：
 - [1] 设置对象的状态为错误状态，指望用户会检查它。
 - [2] 设置一个非局部变量如`errno`，指望用户会检查它。
 - [3] 构造函数不做初始化，依赖用户以后调用专门的初始化函数。
 - [4] 标记对象为“未初始化的”，让第一个调用该对象的函数负责去完成实际的初始化，并在初始化失败时报告错误。

异常很容易地解决了构造函数出现的异常情况

```
class Vector {  
public:  
    class Size { };  
    enum { max = 32000 };  
    Vector::Vector(int sz){  
        if (sz<0 || max<sz)  
            throw Size() ;  
        // ...  
    }  
    // ...  
};
```

```
Vector* f(int i){  
    try {  
        Vector* p = new Vector(i) ;  
        // ...  
        return p;  
    }  
    catch(Vector::Size) {  
        // deal with size error  
    }  
}
```

Exceptions and Member Initialization


异常和成员的初始化

- 如果一个成员的初始化(直接或间接)抛出异常,那么会出现什么问题?
- 按照默认约定,这个异常将传到调用这个成员所属类的构造函数的位置.
- 然而,构造函数本省也可以通过将完整的函数体-包括成员初始化列表—包含在一个`try`块中,自己设法捕获这种异常.例如:

包含整个构造函数体的try块

```
class X {  
    Vector v;  
    // ...  
public:  
    X(int) ;  
    // ...  
};
```

```
X::X(int s)  
    try :v(s) // initialize v by s  
    {  
        // ...  
    }  
    catch (Vector ::Size) {  
        // ...  
    }
```



初始化v抛出的异常在X的构造函数内得到处理

Exceptions and Copying/异常与复制

- 与其他构造函数一样,复制构造函数也可以通过抛出异常的方式发出复制失败的信号.在这种情况下,没有创建新对象.
- 在抛出异常前,复制函数需要释放它已经申请到的资源.见 § E.2和 E.3
- 对于赋值运算符函数情况也是类似的.

Exceptions in Destructors

析构函数内的异常

- 从异常处理的角度看,析构函数可能在2种情况下被调用:
 - [1] 正常调用:作为某个正常的作用域正常退出的结果,或作为一个`delete`操作的结果.
 - [2] 异常处理期间的调用 *Call during exception handling*: 在栈回退的过程中,异常处理退出一个作用域,局部对象的析构函数被调用.
- 对于后一种情况,决不能让抛出异常,因为对象得不到完整的销毁. 如果真这样,那么就被当成异常处理机制的失败,并调用 *`std::terminate()`*.

如果一个析构函数要调用一个可能抛出异常的函数,那么它可以保护自己

```
X::~~X()  
    try {  
        f(); // might throw  
    }  
    catch (...) {  
        // do something  
    }
```

Exception Specifications

异常规范

- 将可能抛出的异常作为函数声明的一部分，称为异常规范或异常描述。如：

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

- 如果一个函数抛出了一个未在异常规范中的异常，会引发对std::unexpected(),的调用， std::unexpected(),的缺省行为是std::terminate(), 它将转而调用abort().
- 如果函数声明中不带异常规范，表明该函数可能抛出任何异常，不抛出异常的函数可以用空表表明。

int f(); // can throw any exception
int g()throw (); //no exception thrown

Checking Exception Specifications

检查异常规范

- 编译器不可能捕捉到所有违反异常规范的情况。
- 一个函数声明包含了异常规范，那么这个函数所有的函数声明（包括定义）都必须有完全一致的异常规范。如：

```
int f() throw (std::bad_alloc);
```

```
int f() // error: 与声明不一致
```

```
{
```

```
    // ...
```

```
}
```

覆盖虚函数时，异常规范必须至少与那个被覆盖的虚函数的异常规范一样是受限的（即子集）。

```
class B {  
public:  
    virtual void f() ; // can throw anything  
    virtual void g() throw(X,Y) ;  
    virtual void h() throw(X) ;  
};  
class D : public B {  
public:  
    void f() throw(X) ; // ok  
    void g() throw(X) ; // ok: D::g() is more restrictive than B::g()  
    void h() throw(X,Y) ; // error: D::h() is less restrictive than B::h()  
};
```

受限的给不那么受限的赋值

- 同样，你可以用一个指向更受限的 *异常规范* 的函数的指针去给一个不那么受限的 *异常规范* 的函数指针赋值，反过来不行。

void f() throw(X) ;

*void (*pf1)() throw(X,Y) = &f; // ok*

*void (*pf2)() throw() = &f; // error: f() 不如 pf2 受限*

- 特别的，你不能用一个没有 *异常规范* 的函数指针给一个带有 *异常规范* 的函数指针赋值

void g() ; // might throw anything

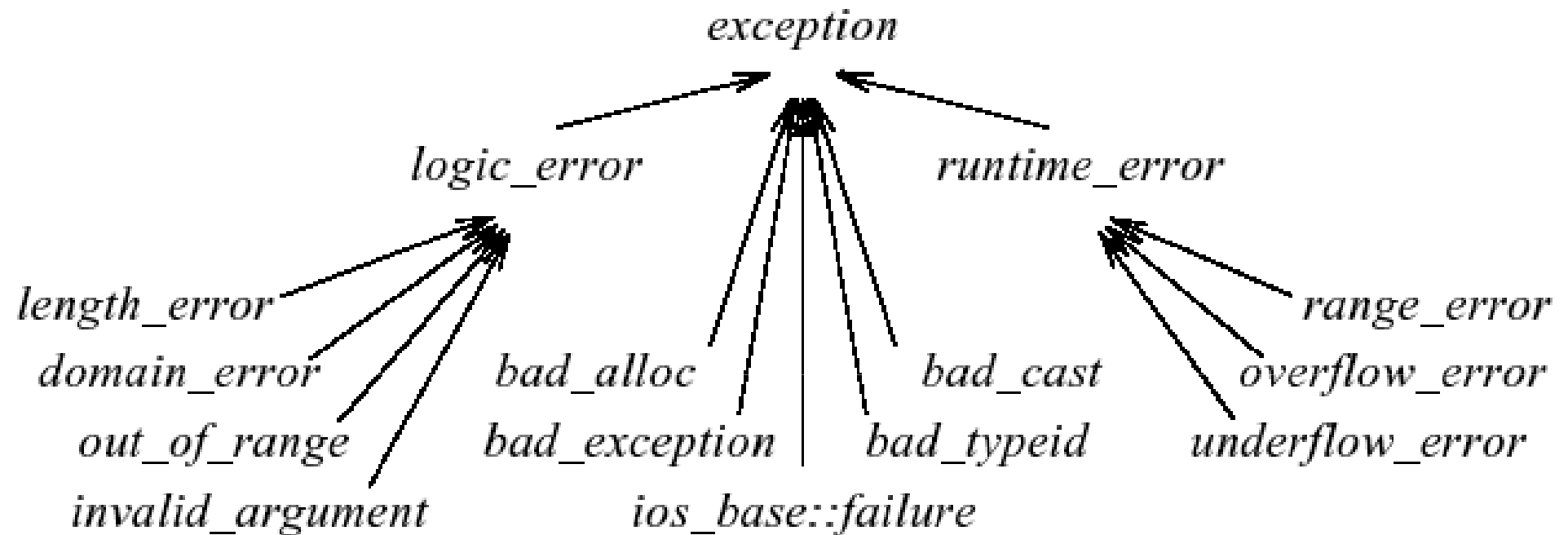
*void (*pf3)() throw(X) = &g; // error: g() 不如 pf3 受限*

Uncaught Exceptions/未捕获异常

- 如果抛出的一个异常未被捕获，函数 *std::terminate()* 将被调用。
- 当异常处理机制发现堆栈损坏，或在某个异常被抛出而导致的堆栈回退过程中，被调用的析构函数企图抛出异常时，也会调用 *std::terminate()*
- 未预期的异常由 *set_unexpected()* 确定的 *_unexpected_handler* 处理。
- *_unexpected_handler* 由 `<exception>` 内的 *std::set_terminate()* 设定。

```
typedef void(*terminate_handler)();  
terminate_handler set_terminate(terminate_handler);
```

Standard Exceptions/标准异常



标准异常的根是*exception*

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

常见的标准异常

Standard Exceptions (thrown by the language)			
Name	Thrown by	Reference	Header
<i>bad_alloc</i>	<i>new</i>	§6.2.6.2, §19.4.5	<new>
<i>bad_cast</i>	<i>dynamic_cast</i>	§15.4.1.1	<typeinfo>
<i>bad_typeid</i>	<i>typeid</i>	§15.4.4	<typeinfo>
<i>bad_exception</i>	<i>exception specification</i>	§14.6.3	<exception>

Standard Exceptions (thrown by the standard library)			
Name	Thrown by	Reference	Header
<i>out_of_range</i>	<i>at()</i>	§3.7.2, §16.3.3, §20.3.3	<stdexcept>
	<i>bitset<>::operator[]()</i>	§17.5.3	<stdexcept>
<i>invalid_argument</i>	<i>bitset constructor</i>	§17.5.3.1	<stdexcept>
<i>overflow_error</i>	<i>bitset<>::to_ulong()</i>	§17.5.3.3	<stdexcept>
<i>ios_base::failure</i>	<i>ios_base::clear()</i>	§21.3.6	<ios>

没有向填加任何新函数，只是适当的定义了所需的虚函数

捕获所有标准异常

```
void f()
{
    try {
        // use standard library
    }
    catch (exception& e) { // "standard library exception
        // ...
    }
    catch (...) { // other exception
        // ...
    }
}
```