C++编程(12)

唐晓晟
北京邮电大学电信工程学院

第14章 异常处理

- 错误处理
- 异常的结组
- 捕捉异常
- 资源管理
- 不是错误的异常
- 异常的描述
- 未捕捉的异常
- 异常和效率
- 处理错误的其他方式
- 标准异常
- 忠告

14.1 错误处理

- 前面8.3节，已经介绍了异常处理的基本语法、语义和使用风格等方面的问题
- 异常的概念就是为了帮助程序员更好的进行错误处理，其基本想法是：让一个函数在发现了自己无法处理的错误时抛出(throw)一个异常，希望它的直接或者间接调用者能够处理这个问题，希望处理这类问题的函数可以表明它将要捕捉(catch)这个异常

错误处理

- 异常处理机制提供了一种更规范的错误处理风格，能明确地把错误代码从正常代码中分离出来，使程序更容易读，也更容易用工具进行处理，也能简化分别写出的代码片断之间的相互关系
- 异常处理机制可以看作是编译时的类型检查和歧义性控制机制在运行时的对应物，使得设计过程更为重要，虽然需要做的工作增加，但也使得代码的可用性大为提高
- 错误处理仍然是一件很困难的工作

14.1.1 关于异常的其他观点

- C++异常处理机制的设计是为了处理错误和其他非正常的情况，特别的，希望能够支持由分别开发的组件组合而成的程序中的错误处理
- 这个机制的设计只是为了处理同步异常，例如数组范围检查和I/O错误，异步问题的处理不能由这种机制处理(有些系统提供了信号机制)
- 标准C++中没有线程或者进程的概念，但是C++的异常处理设计能够有效用于并发程序，要求程序员遵守基本的并发规则
- 抛出大量异常会使得程序结构模糊

14.2 异常的结组(Grouping of Exceptions)

- 一个异常也就是某个用于表示异常发生的类的一个对象
- 一个throw的作用就是导致堆栈的一系列回退，直到找到某个适当的catch
- 异常经常可以自然地形成一些族，意味着可以借助于继承来描述异常的结构
- 异常的类层次结构表示对于代码的健壮性可能很重要，考虑：若没有结组机制，如何处理来自数学库的所有异常

异常的结组示例

```
class Matherr{};
class Overflow:public Matherr{};
class Underflow:public Matherr{};
class Zerodivide:public Matherr{};
void f(){
    try {
    // …
    }
    catch(Overflow){
    // 处理Overflow或者任何由Overflow派生的异常
    }
    catch(Matherr){    // 处理所有不是Overflow的Matherr
    }
}
```

异常的结组示例

```
void g()
{
    try{
        //...
    }
    catch(Overflow){/*...*/}
    catch(Underflow){/*...*/}
    catch(Zerodivice){/*...*/}
    //…
}
```

若不使用结组机制，需要列举所有异常

缺点：

1 忘记列出某个异常

2 当向数学库添加新异常时，每段试图处理所有数学异常的代码都必须修改，其结果就是，某个库一旦发布，就无法再加入任何新异常了

注意：

无论是内部的数学操作，还是基本的数学库，都没有将算术错误报告为异常(比如除0，在许多流水线机器系统结构中都是非同步的操作)

14.2.1 派生的异常

- 由于异常的类层次结构，很多情况下，异常捕捉代码块只针对基类(而不是特定的子类)异常进行捕捉
- 捕捉和命名异常的语义等同于函数接受参数语义

派生的异常示例

```
class matherr{
    //...
    virtual void debug_print() const {
        ceer << "Math error";
    }
};

class Int_overflow:public Matherr{
    const char* op;
    int a1, a2;
public:
    Int_overflow(const char* p,int a,int b){
        op = p; a1 = a; a2 = b;     }
    virtual void debug_print() const {
        cerr << op << a1 << a2;     }
};
```

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

即使g()抛出的实际上是一个Int_overflow，这里m仍然一个Matherr对象，也即：Int_overflow所携带的附加信息将是不可访问的

若需要访问附加信息，catch后面括号中应该声明为Matherr的引用或者指针

14.2.2 多个异常的组合

- 并不是每组异常都具有树形结构，也常有一个异常同属于两个组的情况，如下，这样的一个Netfile_err异常能够被处理网络的异常捕获，也能被处理文件系统异常的函数捕获
- 对于错误处理的这种非层次性结构的组织形式很重要，否则用户或许根本意识不到自己捕获了某个异常

```
class Netfile_err:public Network_err, public File_system_err
{/*...*/};
```

14.3 捕捉异常

```
void f() {
    try{
        throw E();
    }
    catch(H){
        //何时运行到这里？
    }
}
```

[1] 如果H是和E相同的类型
[2] 如果H是E的无歧义的公用基类
[3] 如果H和E是指针类型，且[1]或[2]对它们所引用的类型成立
[4] 如果H是引用类型，且[1]或[2]对H所引用的类型成立

此外，还可以给用于捕捉异常的类型加上const，限制我们不能修改捕捉到的异常

从原则上说，异常在抛出时被复制，异常处理器得到的只是原始异常的一个副本(甚至已经被复制多次)，因此，我们不能抛出一个不允许复制的异常

14.3.1 重新抛出

- 在捕捉到了一个异常之后，很多情况下，处理器并不能完成对这个错误的全部处理，典型的方式是：该处理器做完局部能够做的事情，然后再一次抛出这个异常，这将使得错误恢复动作分布在几个处理器里
- 重新抛出采用一个不带运算对象的throw语句
- 重新抛出的异常就是当初捕捉到的那个异常，而不仅仅是异常处理器能访问到的那个部分

重新抛出示例

```
void h() {
    try { // 可能抛出Matherr的代码
    }
    catch(Matherr){
        if(can_handle_it_completed){
            //处理Matherr
            return;
        }
        else{
            //完成这里能做的事情
            throw;
        }
    }
}
```

14.3.2 捕捉所有异常

- 对于函数，省略号”　”表示“任何参数”，同样，catch(　)表示要“捕捉所有异常”

```
void m()
{
    try {
        // 一些代码
    }
    catch(...) {//处理所有异常
        // 清理工作
        throw;
    }
}
```

14.3.2.1 处理器的顺序

- 派生异常可能被多于一个异常类型的处理器捕捉，因此，在写try语句时处理器的排列顺序非常重要

```
void f() {
    try{
        //...
    }
    catch(std::ios_base::failure){
        //处理任何流io错误
    }
    catch(std::exception& e){
        //处理所有标准库异常
    }
    catch(...){
        //处理任何其他异常
    }
}
```

处理器的顺序

```
void f() {
    try{
        //...
    }
    catch(...){
        //处理任何其他异常
    }
    catch(std::exception& e){
        //处理所有标准库异常
    }
    catch(std::bad_cast){
        //处理dynamic_cast失败
    }
}
```

如果次序安排的不好，有些异常处理代码将永远不会运行

左例中，bad_cast部分将永远不被执行，即使将catch(…)部分去掉也不行

14.4 资源管理

- 当函数申请了某种资源(打开文件或者分配空间)时，一个很重要的问题就是需要正确释放这些资源
- 但是如果程序在运行过程中产生了异常，就会导致程序的流程不是所希望那样，此时需特别注意资源问题
- 异常产生时，实现会“向上穿过堆栈”去为某个异常查找对应处理器，叫做“堆栈回退”，此时，会对所有局部对象调用析构函数

资源管理示例

```
void use_file(const char* fn)
{
    FILE* f = fopen(fn, "r");
    // 使用f资源
    fclose(f);
}

// 若在使用f资源的过程中产生了异常，则fclose(f)将不会被调用，资源释放失败
```

```
void use_file(const char* fn) {
    FILE* f = open(fn, "r");
    try {
        // 使用f
    }
    catch(...){
        fclose(f);
        throw;
    }
    fclose(f);
}

// 一种解决方法，但是非常罗嗦
```

资源管理示例

考虑问题的一般形式：

```
void acquire() {
    // 申请资源1
    // ...
    // 申请资源n
    // 使用资源
    // 释放资源n
    // ...
    // 释放资源1
}
// 一般是按照申请的相反顺序来释放资源，这和构造和析构对象的顺序很相似
```

定义一个类File_ptr

```
class File_ptr{
    FILE* p;
public:
    File_ptr(const char* n,
        const char* a){p=fopen(n,a);}
    File_ptr(FILE* pp) { p=pp; }
    // 适当的复制机制
    ~File_ptr() { if(p) fclose(p); }
    operator FILE*() { return p; }
};
```

这样处理，主程序只需要一句话

```
File_ptr f(fn,"r");
```

析构函数总会被调用，资源总会被释放

14.4.1 构造函数和析构函数的使用

- 利用局部对象管理资源的技术通常被称为“资源申请即初始化(resource acquisition is initialization)”
- 只有在一个对象的构造函数执行完毕时，这个对象才被看作已经建立起来了，只有在此之后，异常产生时的堆栈回退才为该对象调用析构函数
- 一个由子对象组成的对象的构造将一直持续到它所有的子对象都完成了构造

示例

```
class X{
    File_ptr aa;
    Lock_ptr bb;
public:
    X(const char* x,
      const char* y)
       :aa(x,"rw"), // 申请'x'
        bb(y)// 申请y
    {}
};
```

考虑X，构造函数需要两种资源，文件x和锁y，这些请求都可能失败并抛出异常，若普通设计，程序员需要写大量代码

为此，可以使用两个类(子对象)来实现该类，这使得程序员根本不必去考虑任何异常情况

若在创建了aa但还没有bb的情况下出现了异常，aa的析构函数会被调用，bb的则不会

示例

```
class Y{
    int* p;
    void init();
public:
    Y(int s) {
        p=new int(s);
        init();
    }
    ~Y(){delete[] p;}
    //...
};

// 一段有问题的代码，可能
导致存储流失
```

```
class Z{
    vector<int> p;
    void init();
public:
    Z(int s) : p(s) { init(); }
    // ...
};


// 改进后的代码
```

14.4.2 auto_ptr

- 标准库提供的auto_ptr支持“资源申请即初始化”的技术
- 下例中，Rectangle和由pc所指向的Circle都将被删除，无论是否有异常被抛出

```
void f(Point p1, Point p2, auto_ptr<Circle> pc, Shape* pb)
{
    auto_ptr<Shape> p(new Rectangle(p1,p2)); // p指向一个矩形
    auto_ptr<Shape> pbox(pb);

    p->rotate(45); // auto_ptr<Shape>的使用与Shape*一样
    // ...
    if(in_a_mess) throw Mess();
} // 记住，退出时需要删除pb
```

auto_ptr

- 为了获得这种所有权语义(ownership semantics)，也常被称为破坏性复制语义(destructive copy semantic)，auto_ptr具有与常规指针很不一样的复制语义：在将一个auto_ptr复制给另一个之后，原来的auto_ptr将不再指向任何东西
- 因为复制auto_ptr将导致对它自身的修改，所以const auto_ptr就不能复制

auto_ptr的声明

```
// auto_ptr在<memory>中声明
template<class X> class std::auto_ptr{
    template<class Y> struct auto_ptr_ref{/*...*/}; // 协助类
    X* ptr;
public:
    typedef X element_type;
    explicit auto_ptr(X* p=0) throw(){ptr=p;}
        // throw()表示不抛出异常
    ~auto_ptr() throw() { delete ptr; }
    //注意，复制构造函数和赋值都用非const参数
    auto_ptr(auto_ptr& a) throw(); // 复制，而后a.ptr=0
    template<class Y> auto_ptr(auto_ptr<Y>& a) throw();
    // 复制，而后a.ptr=0
    auto_ptr& operator=(auto_ptr& a) throw(); // 复制，而后a.ptr=0
    template<class Y> auto_ptr& operator=(auto_ptr<Y>& a) throw();
    // 复制，而后a.ptr=0
```

auto_ptr的声明

```
    // 运算符重载
    X& operator*() const throw() { return *ptr; }
    X* operator->() const throw() { return ptr; }
    X* get() const throw() { return ptr; } // 提取指针
    X* release() throw() { X* t=ptr; ptr=0; return t; }
    // 放弃所有权
    void reset(X* p=0) throw() { if(p!=ptr) {delete ptr; ptr=p; }}

    auto_ptr(auto_ptr_ref<X>) throw(); // 从auto_ptr_ref复制
    template<class Y> operator auto_ptr_ref<Y>() throw();
    // 复制到auto_ptr_ref
    template<class Y> operator auto_ptr<Y>() throw();
    // 从auto_ptr的破坏性复制
};
```

auto_ptr说明

- auto_ptr_ref的用途就是为普通auto_ptr实现破坏性复制语义，这使得const auto_ptr不可能复制
- 如果指针D*能转换到B*，那么这里的模板构造函数和模板赋值都能将auto_ptr<D>转换到auto_ptr<B>

auto_ptr示例

```
void g(Circle* pc)
{
    auto_ptr<Circle> p2(pc); // 现在p2负责删除
    auto_ptr<Circle> p3(p2); // 现在p3负责删除(且p2不再负责)
    p2->m = 7; // 程序错误：p2.get() == 0
    Shape* ps = p3.get(); // 从auto_ptr抽取指针
    auto_ptr<Shape> aps(p3); // 转移所有权，并转换类型
    auto_ptr<Circle> p4(pc); // 程序错误：现在p4也负责删除
} // p4(多个auto_ptr拥有同一个对象)的效果是无定义的
// 最可能的情况是被删除两次

vector<auto_ptr<Shape>>& v; // 危险，在容器里使用auto_ptr
sort(v.begin(), v.end()); // 该排序可能造成v混乱
// auto_ptr的破坏性复制语义意味着它不满足标准容器或标准算法
```

14.4.3 告诫

- 并不是所有的程序都需要有从所有破坏中恢复的能力
- 并不是所有资源都重要的需要使用“资源申请即初始化”技术、auto_ptr和catch( )等去保护它们
- 当然，标准库的设计需要好好设计资源的处理问题

14.4.4 异常和new

- 若X的构造函数抛出异常，会如何？
- 标准内存申请p1，p2不会产生存储流失
- 若采用放置语法进行非标准分配，则需要进行非标准的释放

```
void f(Arena& a, X* buffer)
{
    X* p1 = new X;
    X* p2 = new X[10];

    X* p3 = new(&buffer[10])X;
    // 将X放入buffer(无需分配存储)
    X* p4 = new(&buffer[11])X[10];

    X* p5 = new(a)X;
    // 在Arena a中分配存储(从a中释放)
    X* p6 = new(a)X[10];
}
```

14.4.5 资源耗尽

- 讨论资源申请失败时应该做什么
- 两种风格
  - 唤醒：请求某个调用程序纠正问题，而后继续执行
  - 终止：结束当前计算并返回某个调用程序
- C++中，唤醒模型由函数调用机制支持，而终止模型由异常处理机制支持

示例

```
void* operator new(size_t size) // 某个new的实现
{
   for(;;){
      if(void* p = malloc(size)) return p; // 申请内存
      if(_new_handler == 0) throw bad_alloc(); // 无处理器
      _new_handler(); // 寻求帮助，如果也找不到内存
   }                              // 或许会抛出一个异常，否则无穷循环
}
```

// _new_handler()是当new无法申请到内存时，系统内定的处理函数，当然，也可以用自己的函数替代之

// set_new_handler(&my_new_handler);

每个C++实现都要求保留足够内存，在存储耗尽的情况下还能抛出bad_alloc，当然，全部耗尽的情形还是可能发生的

示例

```
void my_new_handler();
void f()
{
    void(*oldnh)() = set_new_handler(&my_new_handler);
    try{
        // ...
    }
    catch(bad_alloc){
        // ...
    }
    catch(...) {
        set_new_handler(oldnh); // 重置处理器
        throw; // 重新抛出
    }
    set_new_handler(oldnh); // 重置处理器
} // 也可以使用"资源申请即初始化"技术避免使用catch语句
```

14.4.6 构造函数里的异常

- 异常也为从构造函数里报告出错的问题提供了一个解决方案，构造函数无法返回一个独立的值供调用程序检查，传统的解决方法有
  - 返回一个处于错误状态的对象，相信用户有办法检查其状态
  - 设置一个非局部变量，指出创建失败，相信用户能去检查这个变量
  - 在构造函数中不做初始化，依靠用户在第一次使用对象之前调用某个初始化函数
  - 将对象标记为“未初始化的”，让对这个对象调用的第一个成员函数去完成实际的初始化工作，并让这个函数在初始化失败时报告错误

构造函数里的异常

- 异常处理机制使构造失败的信息能从构造函数内部传出来

```
class Vector{
public:
    class Size{};
    enum {max = 32000};
    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){
        // 处理错误
    }
}
```

14.6.1 异常和成员初始化

- 当成员的初始式抛出异常，该异常将传到调用这个成员的类的构造函数的位置
- 构造函数可以通过将完整的函数体包在一个try块里，自己设法捕捉这种异常

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

```
X::X(int s)
try
    :v(s) { // 用s初始化v
}
catch(Vector::Size){
    // ...
}
```

14.4.6.2 异常和复制

- 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一个失败信号
- 在这种情况下，实际上并没有创建对象
- 在抛出异常之前，复制构造函数就需要释放它已经申请到的所有资源
- 复制赋值函数与复制构造函数类似

14.4.7 析构函数里的异常

- 从异常处理的角度看，析构函数可能在两种情况下被调用
  - 正常调用
  - 在异常处理中被调用
- 对于后一种情况，绝不能让析构函数里抛出异常，如果抛出了，就将被认为式异常处理机制的一次失败，并调用std::terminate()，这么做的原因是没有一种普遍有效的方式来确定应该偏向处理那个异常(导致析构函数被调用的异常和析构函数抛出的异常)

析构函数里的异常

- 如果某个析构函数要调用一个可能抛出异常的函数，它可以保护自己
- 如果有某个异常已经被抛出，但是尚未被捕捉，标准库函数uncaught_exception()就会返回true，析构函数能够根据此信息来进行不同的动作

```
X::~X()
try {
    f();
}
catch(...){
    //
}
```

14.5 不是错误的异常

- 有些异常是预期内的，就算被捕捉到也不会对程序行为产生不良影响

```
void f(Queue<X>& q)
try {
    for(;;) {
        X m = q.get();
        // ...
    }
}
catch(Queue<X>::Empty) {
    return;
}
```

换一种看法，我们也可以认为异常处理机制只不过是另外一种控制结构

需要注意的是，异常处理是一种结构化更差的机制，通常在实际抛出时效率也更低

注意不要滥用

示例

- 有时候，采用异常作为返回方式是一种很优雅的技术
- 任何有助于维持一种关于一个错误是什么以及它被如何处理的清晰模型，都是非常宝贵的

```
void fnd(Tree* p, const string& s)
{
    if(s==p->str) throw p;
    if(p->left) fnd(p->left,s);
    if(p->right) fnd(p->right,s);
}
```

```
Tree* find(Tree* p,
          const string& s)
{
    try{ fnd(p,s); }
    catch(Tree* q){
        return q;
    }
    return 0;
}
```

14.6 异常的描述

- 将可能抛出的异常集合作为函数声明的一部分是有价值的，它可以有效的为调用者提供一种保证

```
void f(int a) throw(x2,x3);
// 保证f函数只会抛出异常x2、x3以及从这些类型派生的异常
```

- 若函数不遵循自己作出的保证，这个企图将会被转换为一个对std::unexpected()的调用，其默认意义是std::terminate()，通常将被转为对abort()的调用

异常的描述

```
void f() throw(x2,x3)
{ /*...*/ }
```

等价于

```
void f()
try
 {
    // ...
 }
catch(x2){throw;}
catch(x3){throw;}
catch(...){
   std::unexpected();
}
```

这里最重要的优点在于函数的声明属于界面，而界面是函数的调用者可以看到的

另外，函数的定义一般不是可用的东西，即使有源码，也不希望经常去查看

最后，带有异常描述的函数也比手工写出的等价版本更短更清晰

若函数不抛出任何异常，可以声明为:

```
int g() throw();
```

14.6.1 对异常描述的检查

- 编译时不可能捕捉到所有违反界面描述的情况，但是编译时可以完成大部分工作

```
int f() throw(std::bad_alloc);
int f() // 错误：缺少异常描述
{/*...*/}

void f() throw(X);
void (*pf1)() throw(X,Y) = &f; // ok
void (*pf2)() throw() = &f;  // 错误：f()不如pf2那么受限

void g(); // 可能抛出任何异常
void (*pf3)() throw(X) = &g; //错误：g()不如pf3那么受限
```

对异常描述的检查

```
class B{
public:
    virtual void f();
    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); // 可以，更受限
    void h() throw(X,Y); // 错误
};
```

异常描述不是函数类型的一部分，typedef不能带有异常描述

```
typedef void(*PF)()
throw(X);
// Error
```

14.6.2 未预期的异常

- 异常描述可能导致调用unexpected()，这是设计者所不希望看到的
- 一个设计良好的子系统Y常常将它的所有异常都从一个类Yerr派生出来

```
class Some_Yerr:public Yerr{/*…*/};
void f() throw(Xerr, Yerr, exception);
// 能把所有的Yerr传给它的调用者，这样f()中的Yerr都不会触发unexpected()
```

- 标注库抛出的所有异常都是从exception派生出来的

14.6.3 异常的映射

- 有时候，希望能将unexpected()的行为方式修改为其他的能够接受的方式，而不是简单的终止程序
- 最简单方式就是将标准库异常std::bad_exception加入某个异常描述，此时，unexpected()将直接抛出一个bad_exception，而不是去调用某个试图应付困难的函数

```
class X{}; class Y{};
void f() throw(X, std::bad_exception)
{
    throw Y();
}
```

虽然没有调用terminate()，bad_exception异常仍然是很粗鲁的，特别是丢失了引起问题的那个异常的所有信息

14.6.3.1 异常的用户映射

- 未预期异常的响应由_unexpected_handler决定，它又是通过<exception>中的std::set_unexpected()设置的
- 重新定义unexpected()函数时，我们先使用“资源申请即初始化”技术，为unexpected()函数定义一个类STC

```
typedef void(*unexpected_handler)();
unexpected_handler set_unexpected(unexpected_handler);
class STC{
    unexpected_handler old;
public:
    STC(unexpected_handler f){old=set_unexpected(f);}
    ~STC(){set_unexpected(old);}
};
```

异常的用户映射示例

假设void g() throw(Yerr);是可能引发调用unexpected()的函数，假设引发的原因是现在应用到网络环境中...

定义一个函数，使它具有所希望的unexpected()的意义

```
class Yunexpected:public Yerr{};
void throwY() throw(Yunexpected) { throw Yunexpected();}
// 在用作unexpected()函数时，throwY将所有未预期的异常都映射为Yunexpected

void networked_g() throw(Yerr){
    STC xx(&throwY); // 并没有违反异常描述，
                     // 因为Yunexpected是从Yerr派生出来的
    g();
} // 现在unexpected()抛出Yunexpected，但是丢失了异常的类型
```

14.6.3.2 找回异常的类型

```
class Yunexpected:public Yerr{
public:
    Network_exception* pe;
    Yunexpected(Network_expection* p):pe(p?p->clone():0){}
    ~Yunexpected(){delete pe;}
}; // clone用于在自由存储区为异常建立一个副本，异常处理完毕仍然存在
void throwY() throw(Yunexpected) {
    try{
        throw; //重新抛出将立即被捕捉
    }
    catch(Network_exception& p){
        throw Yunexpected(&p);
    }
    catch(...){
        throw Yunexpected(0);
    }}
```

throwY由unexpected()调用，而unexpected()是由一个catch(...)处理器调用的，所以现在一定存在着某个能够重新抛出的异常

实际的异常是Network_exception

14.7 未捕捉的异常

- 如果抛出的一个异常未被捕捉，那么就会调用函数std::terminate()
- 未捕捉的异常由_uncaught_handler确定，它由<exception>里的std::set_terminate()设置
- 提出terminate()的原因是，少数情况下，必须能用不那么精细的错误处理技术终止异常处理过程
- 如果希望在发生未捕捉异常时保证进行清理工作，可以为main()添加一个捕捉一切的处理器(但是仍然无法捕捉在全局变量的构造和析构中抛出的异常)

14.8 异常和效率

- 原则上说，当没有异常时，异常处理的实现在运行时没有任何开销，此外，也可以做到使抛出异常并不总比调用函数的开销更大
- 尽管异常处理会影响效率，但是，那些能够代替异常的东西也不是没有代价的
- 需要系统化地处理错误的地方，最好使用异常处理机制
- 异常描述可能非常有助于改进所生成的代码

14.9 处理错误的其他方式

- 使用异常处理方式，需要一种整体策略
- 异常处理策略最好使在设计的初始阶段予以考虑，而且必须是简单而明确的
- 成功的容错系统只能是多层次的，不可能有某种单一机制能够处理所有错误
- 将系统划分为一些独立的子系统，使它们或是成功结束，或是以某种定义良好的方式失败，是至关重要的

14.10 标准异常

- 标准异常、抛出它们的函数、运算符和一般性功能

标准异常(由语言抛出)			
名字	抛出	参考	头文件
bad_alloc	new	6.2.6.2 19.4.5	<new>
bad_cast	dynamic_cast	15.4.1.1	<typeinfo>
bad_typeid	typeid	15.4.4	<typeinfo>
bad_exception	异常描述	14.6.3	<exception

标准异常

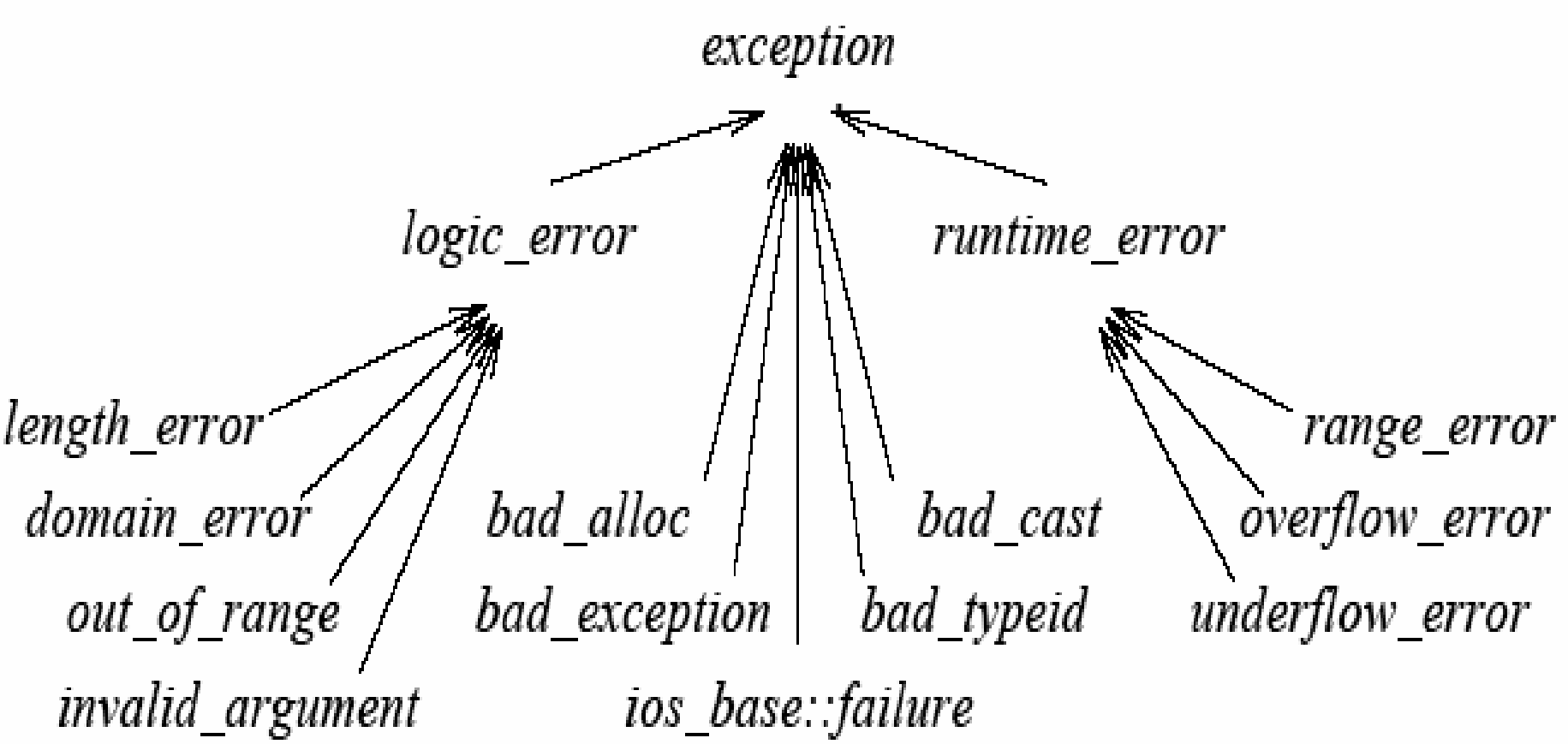
标准异常(由标准库抛出)			
名字	抛出	参考	头文件
out_of_range	at() bitset<>::operator[]()	3.7.2 16.3.3 20.3.3	<stdexcept> <stdexcept>
invalid_argument	按位设置构造函数	17.5.3.1	<stdexcept>
overflow_error	bitset<>::to_ulong()	17.5.3.3	<stdexcept>
ios_base:failure	ios_base::clear()	21.3.6	<ios>

标准异常

- 所有库异常都是同一个类层次结构中的部分，其根是标准库异常类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:
    // ...
};
```

标准异常类层次结构图


exception
logic_error
runtime_error
length_error
domain_error
out_of_range
invalid_argument
bad_alloc
bad_exception
ios_base::failure
bad_cast
bad_typeid
range_error
overflow_error
underflow_error

14.11 忠告

- [1] 用异常做错误处理
- [2] 当更局部的控制机构足以应付时，不要使用异常。
- [3] 采用“资源申请即初始化”技术去管理资源
- [4] 并不是每个程序都要求具有异常时的安全性
- [5] 采用“资源申请即初始化”技术和异常处理器去维持不变式
- [6] 尽量少用try块，用“资源申请即初始化”技术，而不是显式的处理器代码
- [7] 并不是每个函数都需要处理每个可能的错误
- [8] 在构造函数里通过抛出异常指明出现失败

忠告

- [9] 在从赋值中抛出异常之前，使操作对象处于合法状态
- [10] 避免从析构函数里抛出异常
- [11] 让***main***( ) 捕捉并报告所有的异常
- [12] 使正常处理代码和错误处理代码相互分离
- [13] 在构造函数里抛出异常之前，应保证释放在此构造函数里申请的所有资源
- [14] 使资源管理具有层次性
- [15] 对于主要界面使用异常描述

忠告

- [16] 当心通过***new***分配的内存在发生异常时没有释放，并由此而导致存储的流失
- [17] 如果一函数可能抛出某个异常，就应假定它一定会抛出这个异常
- [18] 不要假定所有异常都是由***exception***类派生出来的
- [19] 库不应该单方面终止程序。相反，应该抛出异常，让调用者去做决定
- [20] 库不应该生成面向最终用户的错误信息。相反，它应该抛出异常，让调用者去做决定
- [21] 在设计的前期开发出一种错误处理策略