Morden cpp notes — exception

Sunday, December 6, 2020 9:27 AM

在C++之前，处理异常的手段是错误码：

```
int n = read_data(fd, ...); //读取数据.
if (n==0) {
    ...  //返回值不对，处理
}
if(errno == EAGAIN) {
    ...  // 处理错误.
}
```

这种方式，正常的业务逻辑代码与错误处理代码混在了一起，看起来很乱。另一个问题是它可以被忽略。

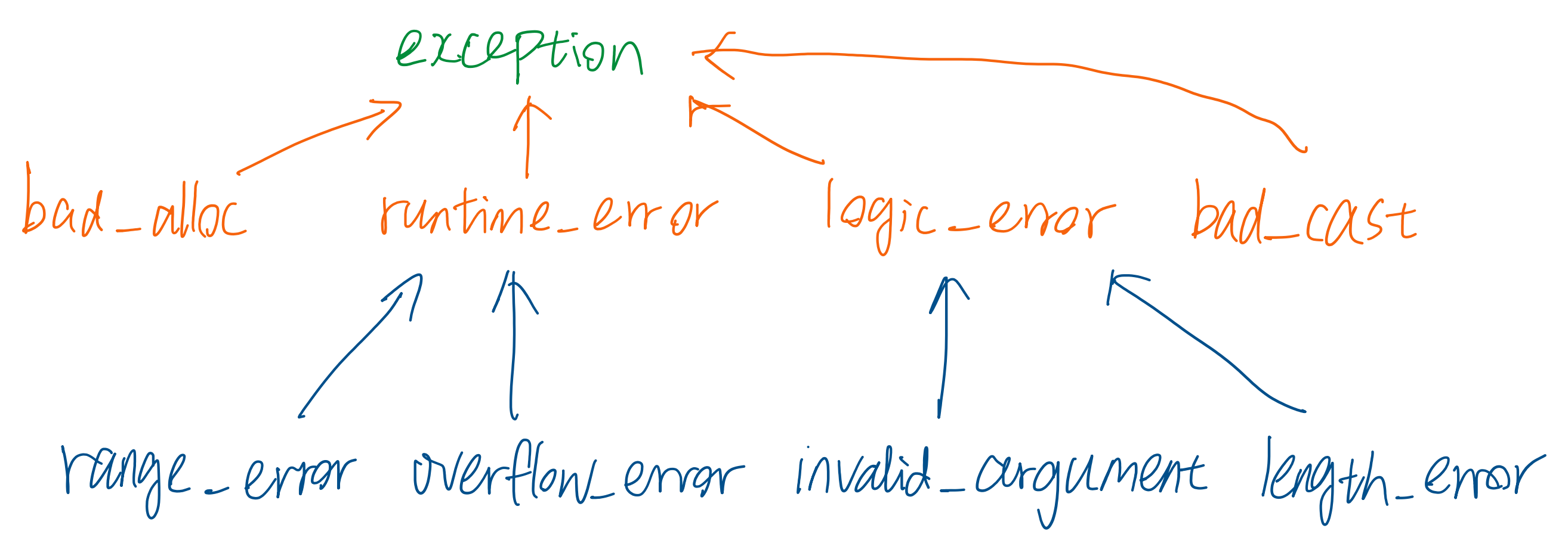
作为一种新的处理方式，异常就是针对错误码的缺陷而设计的

- 异常的处理流程是完全独立的。throw 抛出异常后，错误处理代码集中在 catch中，彻底分离业务逻辑和错误逻辑。
- 异常是绝对不能被忽略的。
- 异常可以用在错误码无法使用的场合，因为有的函数根本就没有返回值，全局的 errno 太不优雅。

异常的使用方式：用try 把可能发生异常的代码块包起来，然后编写 Catch块捕捉异常并处理。

```
try
{
   int n = read_data(fd, ...); //读取数据,可能抛出异常.
   ...
}
catch(...)
{
   ... //集中处理各种错误.
}
```

C++为异常处理设计了一个配套的异常类型体系，定义在<stdexcept>里.



建议选择上面的第一层或者第二层某个类型作为基类，不要再加深层次。
Eg. 可以从 runtime_error派生出自己的异常类.

```
class my-exception: public std::runtime_error
{
  public:
    using this_type = my_exception; //给自己起个别名.
    using super-type = std::runtime_error; //给父类也起个别名.

public:
    my_exception(const char* msg); //构造函数
        super_type(msg)     //别名也可以用于构造.
    {}

    my_exception() = default; //默认构造函数
   ~my_exception() = default; //默认析构函数.

Private:
    int code = 0; //其他的内部私有数据
};
```

抛出异常时，不要直接用throw关键字，而是封装成一个函数，这和不要直接用 new, delete关键字是类似的，通过引入中间层来获得更多的可读性，安全性和灵活性.
抛异常的函数没有返回值，所以应该用属性做编译阶段优化：

```
[[noreturn]]                    //属性标签
void raise(const char* msg)     //函数封装 throw,没有返回值.
{
  throw my-exception(msg); //抛出异常.
}
```

使用catch捕获异常的时候，注意C++允许编写多个catch块，捕获不同异常，但是异常只能按照 catch的顺序依次匹配，而不会去找最佳匹配. 所以，建议只用一个catch块，绕过这个坑.
写catch块就像是写一个标准函数，所以入口参数也学写为 const & 形式，避免对象拷贝的代价：

```
try
{
  raise("error occured"); // 函数封装 throw, 抛出异常.
}
catch(const exception & e) // const & 捕获异常,可以用基类
{
  cout << e.what() << endl; // what()是exception的虚函数.
}
```

关于try-catch，还有一个很有用的形式，function-try. 就是把整个函数体视为一个大try 块，而catch放在后面，与函数体同级并列：

```
void some_function()
try          // 函数名后直接写try块
{ ...
}
catch(...)     // catch块与函数体同级并列.
{ ...
}
```

这样做，不仅能够捕获函数执行过程中所有可以产生的异常，而且少了一级缩进层次，处理逻辑更清晰.
Eg.

```
void f(int i)
try
{
   if(i < 0)
     throw "less than zero";
   std::cout << "greater than zero" << std::endl;
}
catch(const char* e)
{
  std::cout << e << std::endl;
}
int main() {
    f(1);
    f(-1);
    return 0;
}
```

Output:

```
greater than zero
less than zero.
```

Since you can't return a value inside a function-try block, so it makes no sense to use a function-try block for a non-void function.

使用异常的判断标准：

- 不允许被忽略的错误,
- 极少数情况下才发生
- 严重影响正常流程，很难恢复到正常状态.
- 无法本地处理.

Eg:

- 构造函数，内部初始化失败，无法创建，后面的逻辑也无法进行，可以用异常处理.
- 读写文件，通常文件系统很少出错. 如果用错误码来处理不存在，权限错误等，显得太啰嗦，这时也应使用异常.
- 相反的例子是 socket通信，因为收发数据失败很频繁，使用异常会增加很多处理成本，所以不应使用异常.

nonexcept 专门用来修饰函数，告诉编译器，这个函数不会抛出异常. 编译器看到 nonexcept，就得到了一个"保证"，可以对函数做优化，消除异常处理成本，和 const一样，noexcept要放在函数后面：

```
void func_nonexcept() noexcept //声明绝不会抛出异常
{
    cout << "noexcept" << endl;
}
```

如果 noexcept 函数中出现异常，会直接崩溃 (crash, core dump). 所以也不要都使用 noexcept，毕竟无法预测内部调用的那些函数是否会抛出异常.

- 使用异常的话，必须禁用裸指针，改为智能指针，确保出现异常的时候，资源得到正确的释放
- 在编码阶段应写好文档和注释，说清楚哪些函数，什么情况下会抛出什么样的异常，应如何处理
- boost.exception 库是对 C++ 标准异常的一个很好的补充
- C++ 异常处理机制没有保证代码最终执行的 finally.
- 重要的构造函数 (普通构造，拷贝构造，转移构造)，析构函数应尽量声明为 noexcept，优化性能. 而析构函数则必须保证绝对不会抛出异常
- noexcept 也可以当作运算符，指定在某个条件下才不会抛出异常. noexcept 相当于 noexcept(true). 还可以使用 noexcept(false) 标记异常.