2023年春季学期

# 软件设计实践

## 面向鲁棒性的设计

谢 涛　　马 郓

# 本讲概览

- 正确性和鲁棒性
  - 鲁棒性原则（Postel's Law 伯斯塔尔法则）
  - 软件中的“问题”
  - 提高正确性和鲁棒性的方法
- 防御性编程
  - 断言
  - 错误和异常处理
  - 隔栏

# 正确性和鲁棒性

- 软件的正确性（Correctness）是指软件按照“规约”执行的能力，是最重要的软件质量指标
  - 软件的行为要严格符合规约中定义的行为
  - 永不给用户错误的结果
  - 让开发者变得更容易：输入错误，直接结束
  - 对内的实现，倾向于正确
- 软件的鲁棒性（Robustness）是指在异常情况下，软件能够正常运行的能力
  - 未被规约覆盖的情况即为“异常情况”，出现规约定义之外的情形时，软件要做出恰当的反应
  - 尽可能保持软件运行而不是总是退出
  - 让用户变得更容易：出错也可以容忍
  - 对外的接口（界面/文件/API/用户输入），倾向于鲁棒

# 正确性和鲁棒性

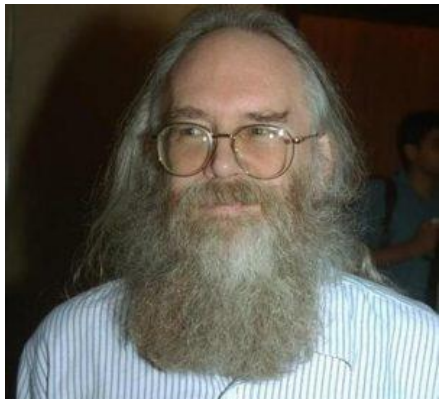
## ❑ 示例

| 问题 | 正确性做法 | 鲁棒性做法 |
| --- | --- | --- |
| 视频文件有坏帧 | 停止播放，提示用户文件损坏 | 跳过坏帧，从下一处正确的帧继续播放 |
| 用户输入了奇怪格式的日期 | 提示用户输入错误 | 尝试用不同的日期格式从输入中识别数字，并告知用户解析结果 |
| 代码中括号不匹配 | 报编译错误 | 尝试补充不匹配的括号继续编译 |

# 鲁棒性原则

## ❑ Postel's Law（伯斯塔尔法则）

*Be conservative in what you do, be liberal in what you accept from others.*



Jon Postel
(1943-1998)

- IANA（互联网编号分配机构）创始人
- SMTP、FTP、UDP等协议的发明人

- 对自己的代码要保守，对用户的行为要开放
  - “严于律己，宽以待人”
- 总是假定用户恶意、假定自己的代码可能失败
- 把用户想象成傻瓜，可能输入任何东西
  - 返回给用户的错误提示信息要详细、准确、无歧义

# 软件中的“问题”

- Problem
- Mistake
- Defect
- Bug
- Fault
- Error
- Failure
- Exception
- Anomaly

# Mistake->Fault(Bug)->Error->Failure

## ❑ 例：查找数组中的元素

- 程序员出现过失/过错（mistake），写出了有bug/fault的代码，程序存在缺陷（defect）

```
//从x中查找n最后一次出现的位置，返回其下标；如果x中没有n，返回-1
int findLast(vector<int> x, int n) {
    if (x.size() == 0) return -1;
    for(int i = x.size()-1; i > 0; i--) {
        if (x[i] == n)
            return i;
    }
    return -1;
}
```

- x=[ ], n=1时，循环没执行，所以没有执行到fault代码 (i>0)
- x=[1,2,3], n=3时，执行了fault代码 (i>0)，但是因为刚开始遍历就找到被查找的元素返回了结果，所以没有引发error
- x=[1,2,3], n=5时，执行了fault代码(i>0)，由于未遍历到首个元素所以引发了error，但是因为返回值与预期结果一致，所以没有导致failure
- x=[1,2,3], n=1时，执行了fault代码(i>0)，引发了error，导致了failure（预期返回0，但返回了-1）

# Fault(Bug)->Error->Failure

- 根据IEEE 610.12-1990标准
  - Fault（故障）
    - A static defect in software (incorrect lines of code)
    - 代码实现中的错误
    - 通常也被称为Bug
  - Error（错误）
    - An incorrect internal state (unobserved)
    - 不正确的内部运行状态
  - Failure（失效）
    - External, incorrect behavior with respect to the expected behavior (observed)
    - 运行时表现出来的、外在的、与规约不一致的行为

# Fault(Bug)->Error->Failure

## ❑ 导致Failure的必要条件（RIP模型）

### ➢ Reachability（可达性）

- **执行到包含Fault的代码**
- 例如，死代码（dead code）中的Fault永远不会导致Failure

### ➢ Infection（感染性）

- **执行Fault的代码后，程序状态是错误的（error）**
- 例如，错误状态与运行环境有关，在特定环境下才会出错

### ➢ Propagation（传播性）

- **错误的状态（error）需要有路径传播到程序的“输出”，被外部所感知的**
- 例如，通过对错误状态进行检测和处理可以在内部将错误状态消除，使外界感受不到Failure

# Mistake与Defect

- 过失(mistake)用于指代程序员的错误行为
- 缺陷(defect)是软件的一种内在性质，可能会导致程序的执行结果与预期不一致
  - 此处的预期结果不一定是规约所描述的，而是指最终用户所希望的结果
- 程序员的过失(mistake)可能导致软件出现缺陷(defect)
  - 例如，数组的下标越界，访问空指针等
- 并非所有缺陷(defect)都是由于过失(mistake)所导致
  - 程序的外部输入可能不符合规约的前置条件，而开发时未对非预期的输入进行处理，从而导致Failure
    - 除数为0，动态内存分配需要的空间太大，读取的文件不存在，等

# 软件中的“问题”

- Problem（问题）笼统/泛指各种不正确的情况
- Mistake（过失）侧重描述人的行为存在失误
- Defect（缺陷）泛指各类设计和实现中存在的问题，导致bug的根源
- Bug（…）通常与Fault表示同样的含义
- Fault（故障）通常指代码实现中的问题
- Error（错误）不正确的内部运行状态
- Failure（失效）运行时表现出来的、外在的、与规约不一致的行为
- Exception（例外/异常）一种应对故障、处理错误的编程机制
- Anomaly（异常现象）常用于算法领域描述与正常分布不一致的情况

**注意：同一个单词在不同文献中可能有不同的含义，需结合上下文语境甄别其具体意义**

# 提高正确性和鲁棒性的方法

- 故障拒绝（Fault Avoidance）
  - 在设计和开发时采用一定的手段来减少错误或缺陷
  - 防御性编程、代码审查、形式化验证
- 故障检测（Fault Detection）
  - 交付前通过一定的手段将故障暴露出来并予以解决
  - 测试，调试，测试驱动的开发
- 容错（Fault Tolerance）
  - 在运行时出现故障能够通过一定的手段消除其影响
  - 冗余，备份，重试

# 本讲概览

- 正确性和鲁棒性
  - 鲁棒性原则（Postel's Law 伯斯塔尔法则）
  - 软件中的“问题”
  - 提高正确性和鲁棒性的方法

- 防御性编程
  - 断言
  - 错误和异常处理
  - 隔栏

## 防御性编程（Defensive programming）

- 子程序不因传入错误数据而被破坏，哪怕是由其它子程序产生的错误数据
- 其核心是承认程序都会有问题，都需要被修改，这是保护的基础
  - 类似于驾驶汽车，永远不能确定其他司机会做什么，同时要保护好自己，哪怕是其他司机犯的错误
- 常用技术
  - 输入检查
  - 断言
  - 错误处理
  - 异常处理
  - 隔栏

# 输入检查

- 检查所有来源于外部的数据
  - 文件
  - 用户输入
  - 网络
  - 其它外部接口
- 检查函数的所有输入参数的值是否满足规约的前置条件
  - 类型是否一致
  - 参数值是否合法
  - 长度是否符合要求

# 断言

- 断言（Assertion）是在开发期间使用的、让程序在运行时自检的代码
  - 用来检查永远不应该发生的状况
  - 断言为真，则表明程序运行正常；断言为假，则说明程序发生错误
  - 断言只在开发阶段被编译到目标代码中，而在生成产品代码时不编译到产品代码中

## C++中的断言

- 在头文件<cassert>中定义的assert宏
- assert(表达式) 当表达式为false时程序会终止运行，并且会输出源文件、错误的代码、以及行号
- 在源文件起始位置加入#define NDEBUG可停用assert

```
#include <iostream>
#include <cassert>
using namespace std;
void func1(int n) {
    assert(n==1);
    cout << "func1 finish" <<endl;
}
int main() {
    func1(1);
    cout << "first" <<endl;
    func1(2);
    cout << "second"<<endl;
    return 0;
}
```

输出结果

```
func1 finish
first
Assertion failed: n==1, file example1.cpp, line 6
```

## ❑ 断言的使用技巧

- 可用断言在函数开始处检查传入参数的合法性

- 每个assert只检验一个条件；同时检验多个条件时，若断言失败，无法直接定位错误
  - 不推荐: assert(nOffset>=0 && nOffset+nSize<=m_nInfomationSize);

- 不要在断言中使用改变环境的语句，因为assert仅在debug阶段生效，如果这么做，会使程序在真正运行时出错
  - 错误: assert(i++ < 100)

# 错误处理

- 如何处理预料中可能要发生的错误？
  - 理想情况：希望在发生错误情况时，不只是简单地终止程序运行，而是能够反馈错误情况的信息，并且能够对程序运行中已发生的事情做些处理
- 直观的思路
  - 在预计会发生异常的地方，加入相应的代码
  - 利用特殊的返回值或全局变量表明出现的错误
  - *问题？*

```
int errorcode = 0;

double divide(int n, int m) {
    if (m == 0) {
        errorcode = 1;
        return 0;
    }
    else
        return (double)n/m;
}
int main() {
    int n, m;
    cin >> n >> m;
    double r = divide(n, m);
    if (errorcode == 1)
        cout << "Divided by zero!" << endl;
    else
        cout << r << endl;
}
```

- 异常（Exception）是把代码中的错误或不正常事件传递给调用方代码的一种特殊手段
  - 当一个函数出现自己无法处理的错误时，可以抛出异常，然后由该函数的直接或者间接调用者处理这个错误
  - 异常与Bug的区别
    - “异常” 是在程序开发中必须考虑的一些特殊情况，是程序运行时就可预料的执行分支（注：异常是不可避免的，如程序运行时产生除 0 的情况、打开的外部文件不存在、数组访问的越界等）
    - “Bug”是程序的缺陷，是程序运行时不被预期的运行方式（注：Bug是人为的、可避免的；如使用野指针、动态分配内存使用结束后未释放等）

## C++异常处理

### throw关键字

```
throw 表达式;
```

- 抛出一个异常，异常是一个表达式
- 表达式的类型可以是一个基本类型，也可以是个类

### try…catch语句

```
try {
   语句组
}
catch(异常类型) {
   异常处理代码
}
```

- catch 用于捕捉异常
- try 中包含会出现异常的代码或者函数，后面通常会跟一个或者多个catch语句块，根据异常类型来选择执行哪个catch语句块

## 示例

```
int main() {
    double m ,n;
    cin >> m >> n;
    try {
        cout << "before dividing." << endl;
        if(n == 0)
            throw -1; //抛出int类型异常
        else
            cout << m / n << endl;
        cout << "after dividing." << endl;
    }
    catch(double d) {
        cout << "catch(double) " << d << endl;
    }
    catch(int e) {
        cout << "catch(int) " << e << endl;
    }
    cout << "finished" << endl;
    return 0;
}
```

输入：
9 6
输出：
before dividing.
1.5
after dividing.
finished

输入：
3 0
输出：
before dividing.
catch(int) -1
finished

## 使用catch(...)捕获任意类型的异常

```
int main() {
    double m ,n;
    cin >> m >> n;
    try {
        cout << "before dividing." << endl;
        if(n == 0)
            throw -1; //抛出int类型异常
        else if( m == 0 )
            throw -1.0; //抛出double类型异常
        else
            cout << m / n << endl;
        cout << "after dividing." << endl;
    }
    catch(double d) {
        cout << "catch(double) " << d << endl;
    }
    catch(...) {
        cout << "catch(...) " << endl;
    }
    cout << "finished" << endl;
    return 0;
}
```

输入：
9 0
输出：
before dividing.
catch(...)
finished

输入：
0 6
输出：
before dividing.
catch(double) -1
finished

# C++异常处理

## 异常的抛出规则

- 异常时通过抛出对象而引发的，该对象的类型决定了应该激活哪个catch的处理代码
- 被选中的处理代码的调用链是，找到于该类型匹配且离抛出异常位置最近的那一个catch
- 抛出异常对象后会生成一个异常对象的拷贝，因为抛出的异常对象可能是一个临时对象，所以会调用复制构造函数生成一个拷贝对象

## 异常的匹配规则

- 首先检查throw本身是否在try块内部，如果是，再在当前函数栈中查找匹配的catch语句。如果有匹配的直接跳到catch的地方执行
- 如果没有匹配的catch块，则退出当前函数栈，在调用函数的栈中查找匹配的catch
- 如果到达main函数的栈，都没有匹配的catch，就会终止程序
- 找到匹配的catch会直接跳到catch语句执行，执行完后，会继续沿着catch语句后面执行

## 示例

```
class CException{
public :
    string msg;
    CException(string s):msg(s) { }
};
double Devide(double x, double y) {
    if(y == 0)
        throw CException("devided by zero");
    cout << "in Devide" << endl;
    return x / y;
}
int CountTax(int salary) {
    try {
        if(salary < 0)
            throw -1;
        cout << "counting tax" << endl;
    }
    catch (int i) {
        cout << "salary < 0" << endl;
    }
    cout << "tax counted" << endl;
    return salary * 0.15;
}
```

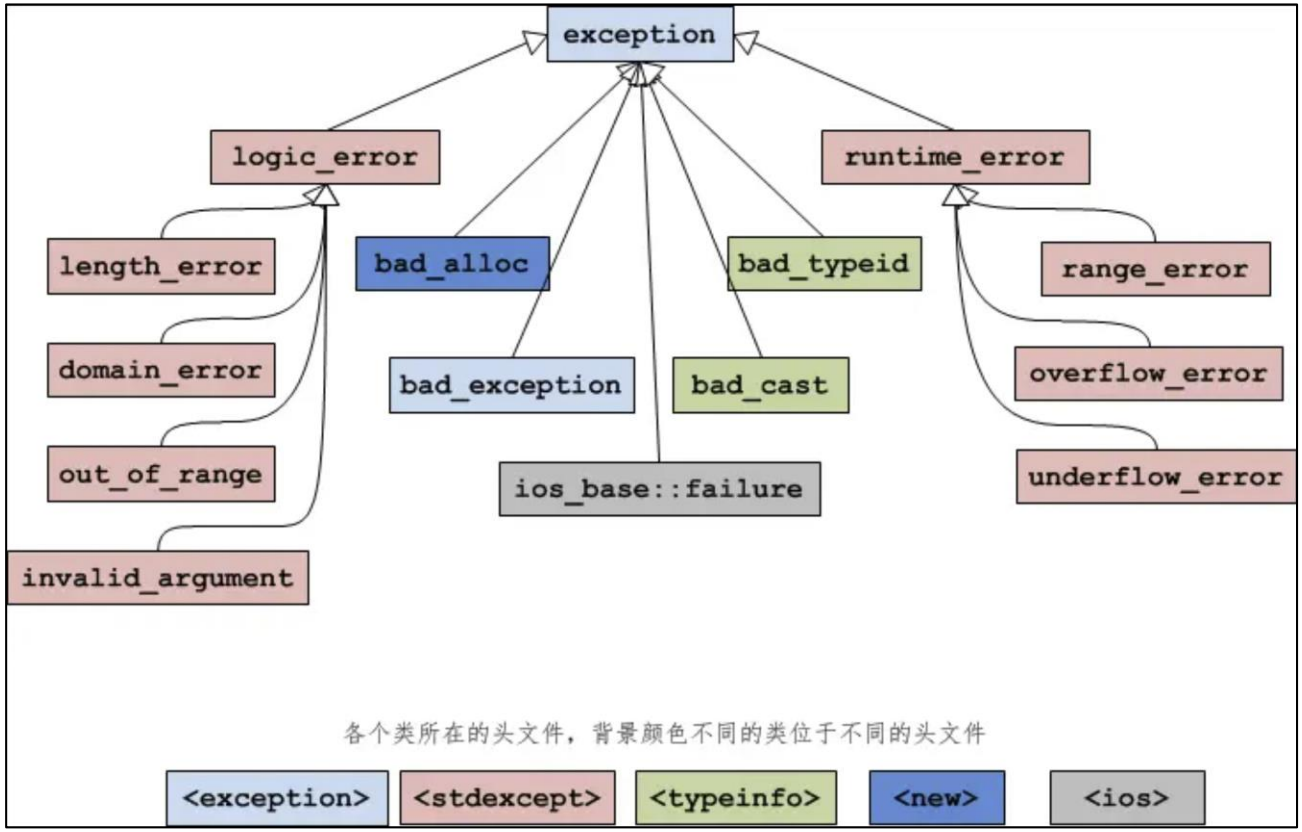
```
int main() {
    double f = 1.2;
    try {
        CountTax(-1);
        f = Devide(3,0);
        cout << "try end" <<endl;
    }
    catch(CException e) {
        cout << e.msg << endl;
    }
    cout << "f=" << f << endl;
    cout << "finished" << endl;
    return 0;
}
```

输出结果

```
salary < 0
tax counted
devided by zero
f=1.2
finished
```

## C++标准异常类

- C++ 标准库中有一些类用于异常处理
  - what() 是异常类提供的一个公共方法，用于返回异常产生的原因
  - 各个异常类的具体用于请参照CppReference



## C++标准异常类 bad_alloc

- 在用new运算符进行动态内存分配时，如果没有足够的内存，则会抛出此异常

```
#include <iostream>
#include <new>
using namespace std;
int main (){
    try {
        char *p = new char[0x7fffffff];
        //无法分配这么多空间，抛出异常
    }
    catch (bad_alloc& e) {
        cerr << e.what() << endl; //what函数返回异常原因
    }
    return 0;
}
```

## C++标准异常类 out_of_range

- 用vector或string的at成员函数根据下标访问元素时，如果下标越界，就会抛出此异常

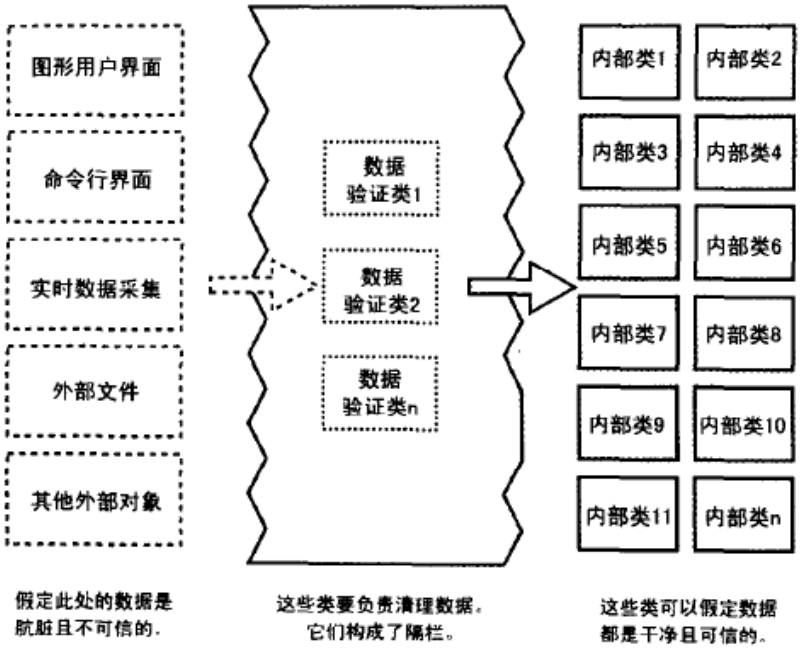
```
int main (){
    vector<int> v(10);
    try {
        v.at(100)=100; //抛出out_of_range异常
    }
    catch (out_of_range& e) {
        cerr<< e.what() << endl;
    }

    string s = "hello";
    try {
        char c = s.at(100); //抛出out_of_range异常
    }
    catch (out_of_range& e) {
        cerr<< e.what() << endl;
    }
    return 0;
}
```

## 示例：抛出不同类型的异常

```
#include <iostream>
#include <string>
#include <stdexcept>
#include <cmath>
using namespace std;
void f(const string& input) {
    try {
        int val = stoi(input);
        cout << sqrt(val) << endl;
    } catch (const invalid_argument& e) {
        cout << "Not a integer: " << e.what() << endl;
    } catch (const out_of_range& e) {
        cout << "Number out of range " << e.what() << endl;
    } catch (const exception& e) {
        cout << "Unexpected : " << e.what() << endl;
    }
}
int main() {
    string s;
    cin >> s;
    f(s);
    return 0;
}
```

# 隔栏

- 如果所有的代码都做异常和错误处理，会使代码变得臃肿，可读性下降
- 隔栏（barricade）是在设计上简化错误处理的策略，将程序的外部和内部进行隔离
  - 把某些接口选定为安全区域的边界，对穿越安全区域边界的数据进行合法性校验，并当数据非法时进行处理



- 隔栏的使用使断言和错误处理有了清晰的区分
  - 隔栏外部的程序使用错误处理技术，外部的数据是不安全的
  - 隔栏内部的程序就应该使用断言技术，因为传入程序内部的数据都已经过了隔栏的处理，应该是正确的，如果出错，则说明程序本身出错

# 防御式编程的使用考虑

- 防御式编程的矛盾
  - 在产品开发阶段，希望显示出的错误越多越好，引入很多防御性的代码
  - 在产品发布阶段，希望错误尽可能偃旗息鼓，尽量不要在使用中出现
  - 如何权衡？
- 过度的防御式编程也会引入问题
  - 如果在程序的每一个想到的地方都进行参数检查、错误保护等，那么程序将变得臃肿而缓慢
  - 更糟糕的是，防御式编程引入的额外代码增加了软件的复杂度，反而容易造成错误

**保留防御性代码的建议**

1. 保留那些检查重要错误的代码
2. 去掉检查细微错误的代码
3. 去掉可以导致程序硬性崩溃的代码
4. 保留可以让程序稳妥崩溃的代码
5. 为你的技术支持人员记录错误信息
6. 确认留在代码中的错误消息是友好的

# 上机安排/期中考试

- 4月22日上机：模拟期中考试
  - 13:20—15:00，共100分钟
  - 共9道程序填空题，无选择题
  - 全真模拟正式考试环境（断网、分发离线版CppReference）
  - 模拟考试后请填写问卷，反馈题量和难度

- 4月29日和5月6日上机：因五一和校庆暂停

- 5月13日上机：正式期中考试
  - 时间和题量待模拟考试后反馈确定
  - 根据答题的相对情况阶梯赋分

# 谢谢

## 欢迎在课程群里填写问卷反馈