

防御式编程 Defensive Programming

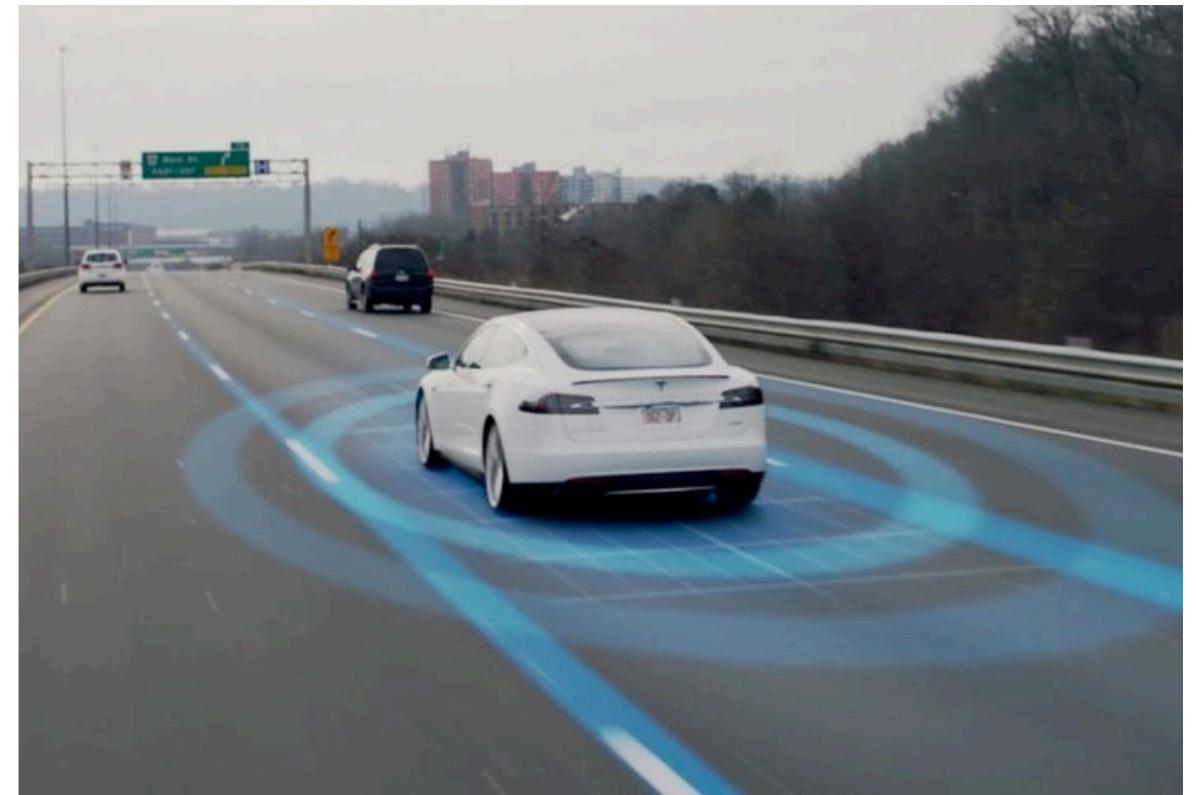
1

什么是防御式编程



防御性驾驶

- 你永远都不知道另一位司机会做什么
- 你永远都不知道另一位行人会做什么
- 确保其他人做出危险行为时不受伤害
- 时刻观察四周做好准备应对不同情况



防御式编程

- 你永远都不知道传入数据会是什么
- 程序不应因传入数据错误而被破坏
- 也不应该因其他程序产生的错误数据而被破坏
- 针对程序外部的保护



问题

- 如何保护程序免受外部数据的影响?
- 如果遇到非法输入
 - 没有提示，自动过滤？
 - 有提示，输出错误信息？

1

防御式编程思想

如何处理非法数据

- 检查所有来源于外部数据的值
 - 从文件、用户、网络或其他接口获得的数据
 - 检查是否处于允许范围内
 - 数据值：符合类型、长度范围
 - 字符串：符合长度范围
 - 日期：符合格式

如何处理非法数据

- 检查所有来源于外部数据的值
- 检查子程序所有输入参数的值
 - 应确保输入参数或类型符合预期
 - 还应确保输出参数或类型符合预期

```
void factorial(double n) {  
    int fac = 1;  
    for (int i = 1; i < n+1; i++) {  
        fac *= i;  
    }  
    return n;  
}
```

2

防御式编程方法

断言 (Assert)

- 在开发期间，能够让程序在运行时自检的代码
- 断言为真 (`assert(fac(4) == 24)`)，程序运行正常
- 断言为假，程序运行错误，需要调试
- 可以通过断言发生错误的位置判断程序错误的位置

断言能够检查到的错误

- 输入输出参数是否处于预期范围内
- 子程序开始或结束时文件或流是否处于打开/关闭状态
- 子程序开始或结束时文件或流是否处于预期位置（头/尾）
- 指针是否为空
- 数组是否已初始化

```
int fib(int x) {  
    assert(x >= 0);  
    if (x == 1) {  
        return 1;  
    } else if (x == 3) {  
        return 2;  
    } else {  
        return fib(x - 1) + fib(x - 2);  
    }  
}
```

```
#include <assert.h>  
#include "fib.h"
```

```
int main() {  
    assert(fib(1) == 1);  
    assert(fib(2) == 1);  
    assert(fib(10) == 55);  
    return 0;  
}
```

什么时候使用断言

- 主要用于程序开发和维护阶段
- 在开发和维护阶段应被编译到代码中
- 在开发完成后应选择移除出可执行代

码中，可能会导致性能下降

- 不要将执行代码直接放入断言语句中

什么时候使用断言

- 使用断言保护程序逻辑的先验条件
(Precondition)和后验条件(Postcondition)
- 先验条件：调用方法前要确保输入参数符合类型与数值预期
- 后验条件：调用方法后要确保输出参数符合类型与数值预期

先验条件和后验条件

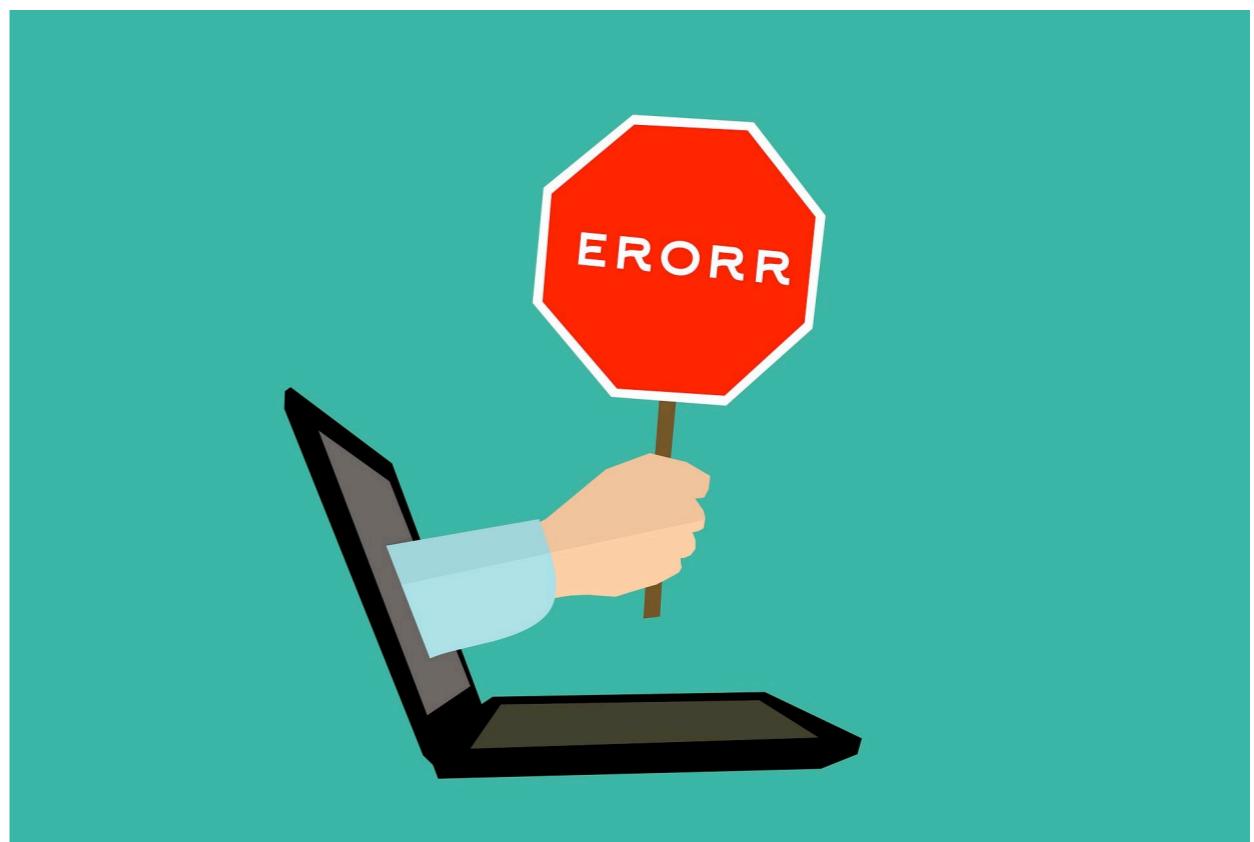
```
1 #include <stdio.h>
2 #include <assert.h>
3
4 double computeFahrenheit(double celcius) {
5     //PreCondition
6     assert(celcius >= -273.15);           //over absolute zero
7
8     double fahrenheit = 0.;
9     fahrenheit = celcius * 1.8 + 32;
10
11    //PostCondition
12    assert(fahrenheit >= -459.67);        //over absolute zero
13
14    return fahrenheit;
15 }
16
17 int main(){
18     printf("%f", computeFahrenheit(30));
19 }
```

断言的好处与局限性

- 可以在代码开发期间尽可能多地排查出问题与错误
- 需要在代码上线前想到可能出错的点
- 如有输入输出问题，将会中断程序
- 需要错误处理机制

错误处理(Error Handling)

- 应对处理能够预测的数据错误
 - 断言主要用于代码中不应发生的错误
- 常见处理方式
 - 返回中立值
 - 换用下一个正确数据（实时系统）
 - 返回与前次相同数据（数据处理）
 - 换用最接近的合法数据（预先处置）



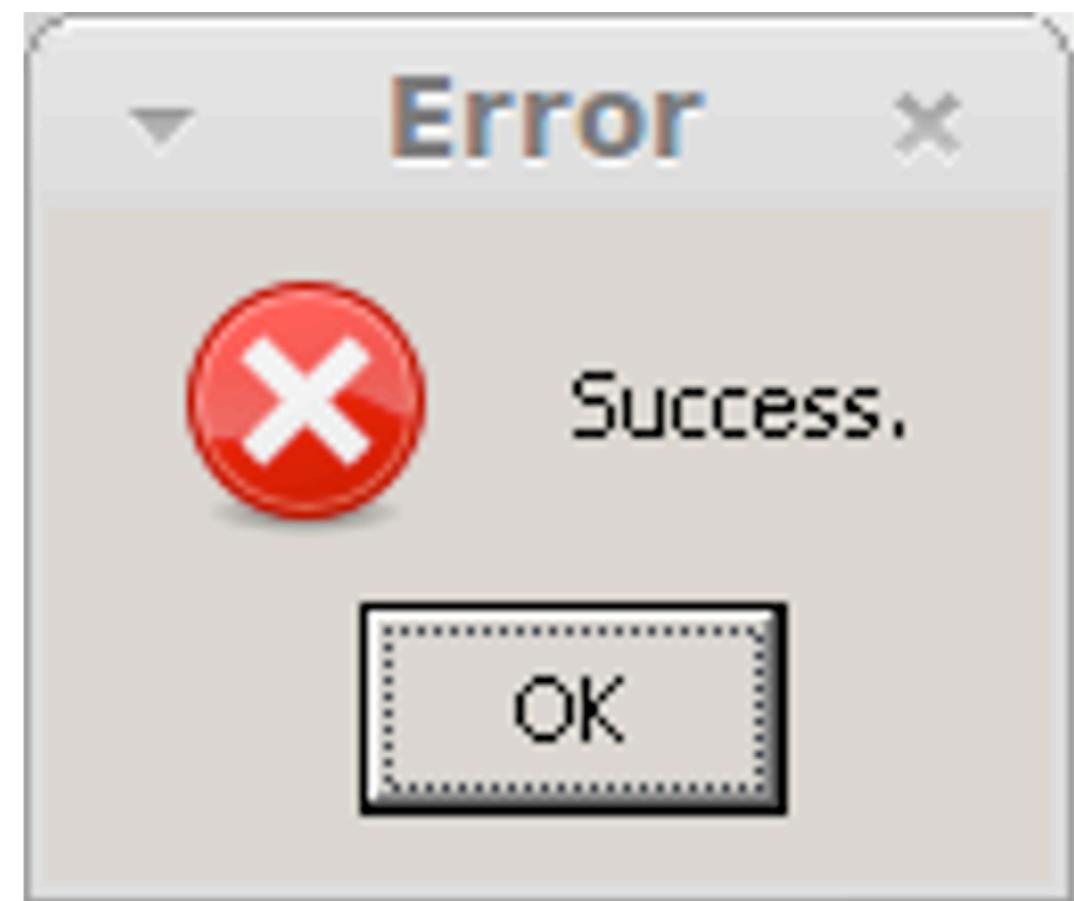
错误处理(Error Handling)

- 其他处理方式
 - 将警告信息记录到日志文件中
 - 返回一个错误码
 - 调用错误处理的接口
 - 显示出错信息
 - 在局部处理错误
 - 程序退出



错误处理的选择

- 正确性(Correctness) 指 永不返回不准确的结果
 - 常用于医疗和生命安全相关场景
- (Robustness) 指 尽可能保证程序的正常运行 哪怕中间出现了问题
 - 常用于日常生活消费相关场景



错误处理架构设计

- 采用一致的方式处理非法输入与输出
- 高可靠性系统应设计不同层级的错误

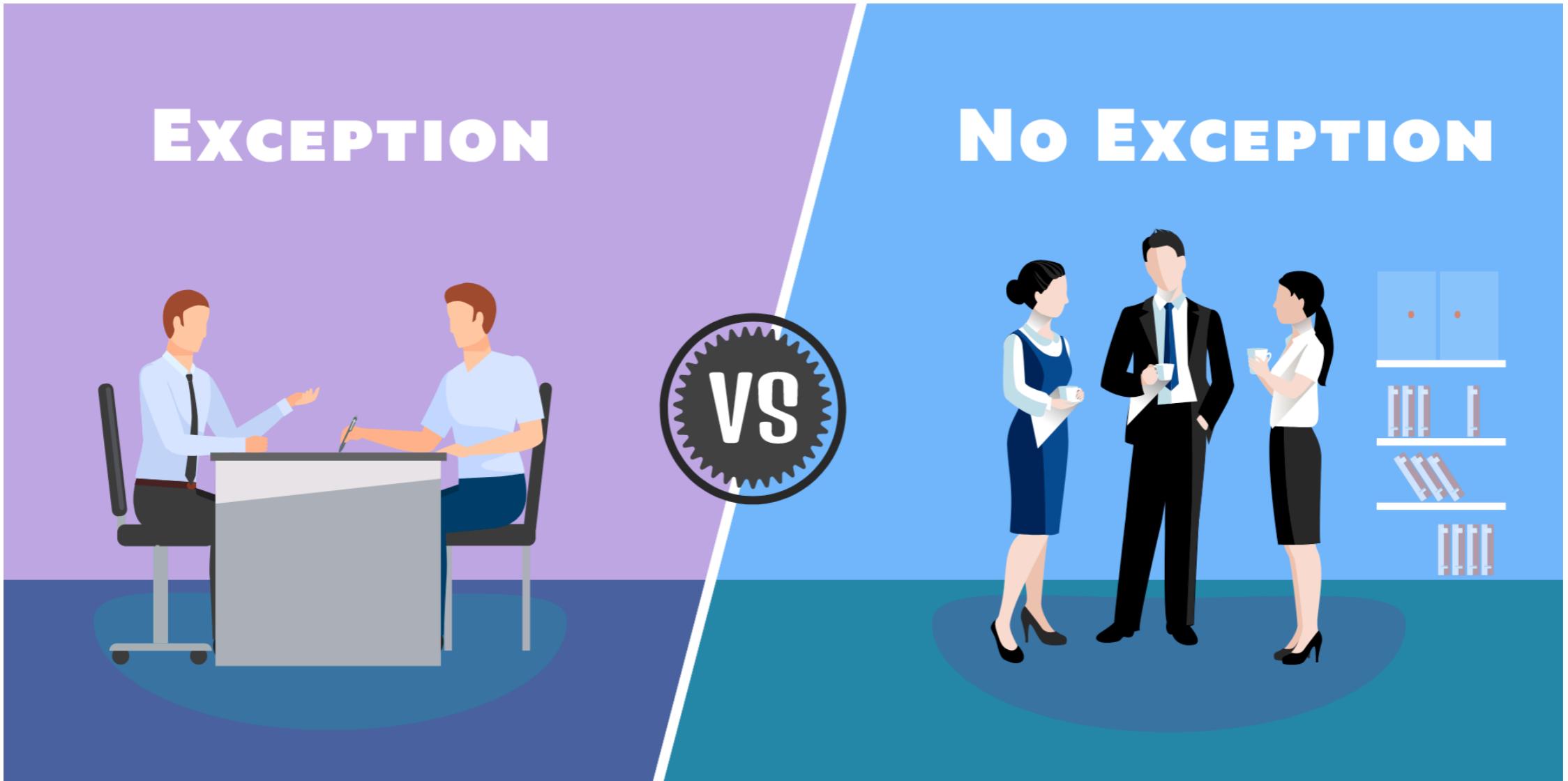
处理，分级逐层处理

- 可以设置通用的错误处理方式
- 需要统一、明确的架构

3

防御式编程使用场景

异常 (Exception)



异常 (Exception)

- 把代码中的错误或异常事件传递给调用方代码的一种特殊手段
- 子程序抛出 (throw) 异常
- 调用其他子程序处理 (handle)



异常 (Exception)

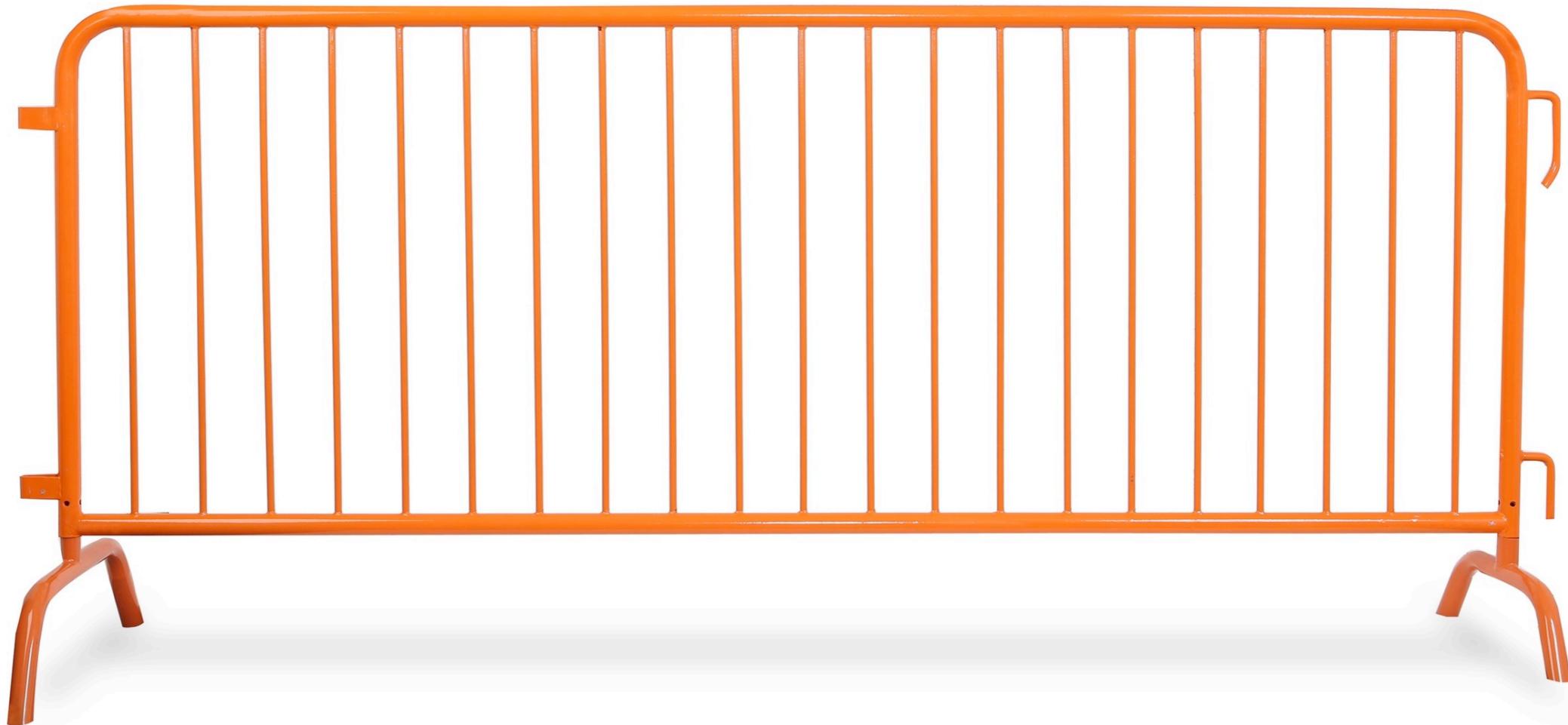
- 用异常通知程序的其它部分，发生了不可忽略的错误
- 只在真正例外的情况下抛出异常
- 不能用异常来推卸责任
- 避免在构造函数或析构函数中抛出异常
- 在恰当的抽象层次抛出异常
- 在异常消息中加入关于异常发生的全部信息
- 避免使用空的catch语句
- 考虑创建一个集中的异常报告机制
- 考虑异常的替换方案

C 语言中的异常处理

- 限制较多
- 直接终止程序
- 打印错误代码 `printf("Error Code: %d", code);`
- 使用 `goto` 语句进行跳转
 - 无条件直接跳转的系统指令
 - 但是会破坏代码的结构

```
#include<stdio.h>
#include<stdlib.h>
int main(){
    int a = 0;
    int b = 0;
    printf("请输入两个值:\n");
    printf("a = ");
    scanf("%d",&a);
    printf("b = ");
    scanf("%d",&b);
    if(b==0){
        goto Error;
    }
    printf("a/b = %d\n",a/b);
    return 0;
Error:
    printf("除数不能为0,程序异常退出!\n");
    exit(-1);
}
```

隔离 (Barricade)



隔离 (Barricade)

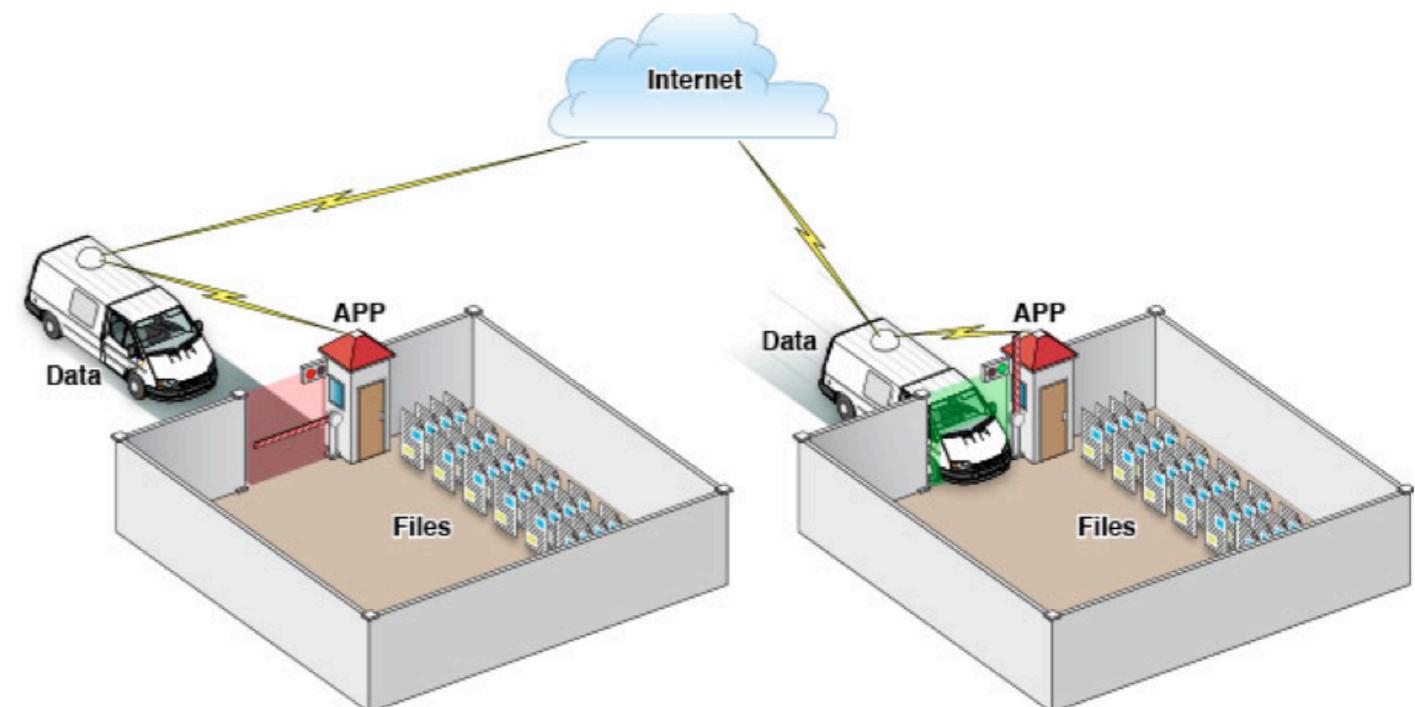
- 容损策略
- 把某些接口选定位“安全”区域的边界
- 对穿越安全区域边界的数据进行合法性校验
 - 假设数据不安全
 - 中间区域进行数据校验
 - 通过校验后发送到程序内部

隔离的例子

- Linux 中有 CGroup 机制
 - 位于 /sys/fs/cgroup
- 一个以文件系统为形式的资源隔离机制
- 可以限制并控制一个进程群组的资源，如CPU、内存等
- 每个进程都可以看成一个子系统
 - 与外部隔离
 - 资源控制器

隔离的例子

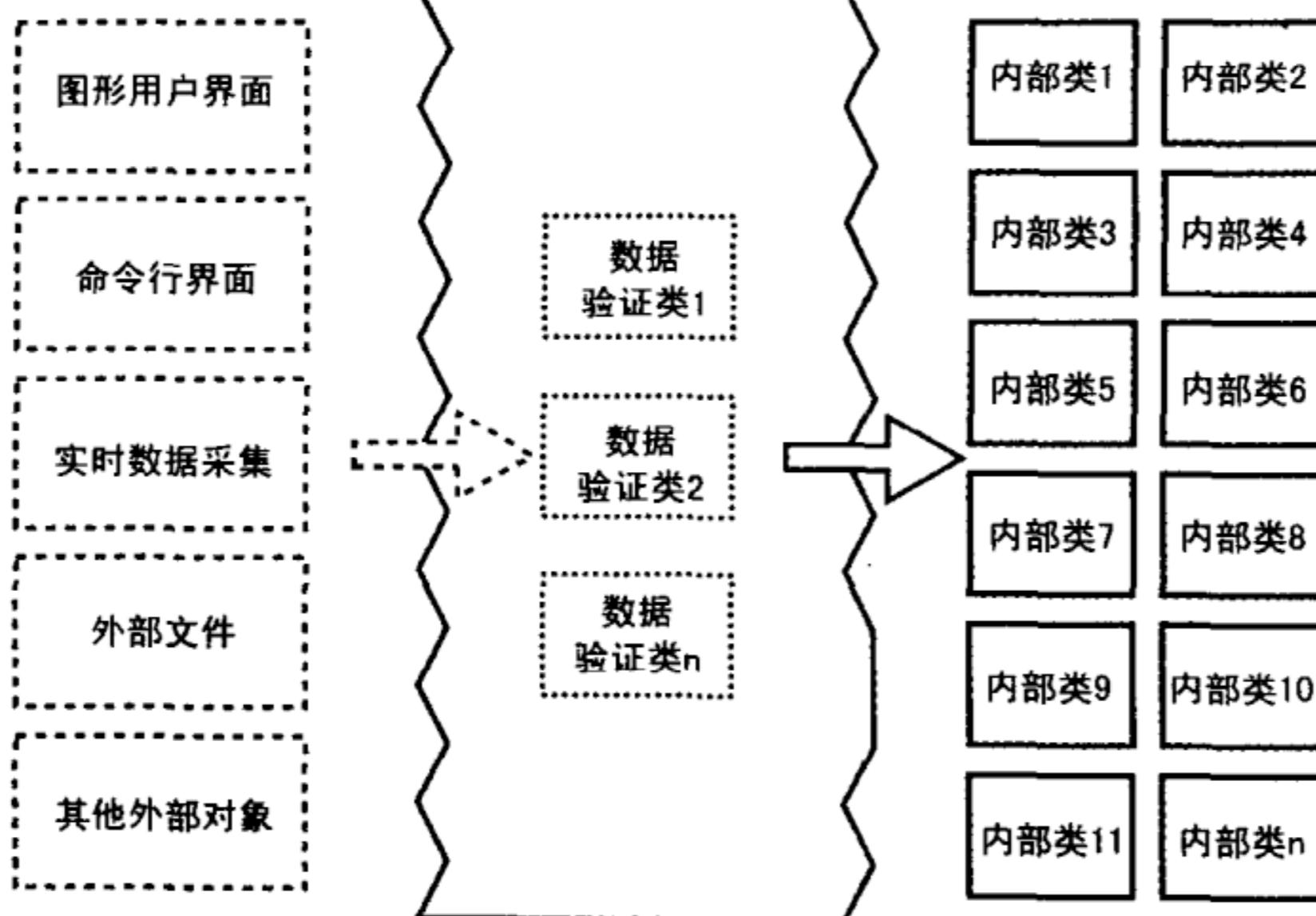
- 沙盒（Sandbox）
 - 一种安全体系
 - 对应用程序操作进行限制
- 每个程序都有自己独立的存储空间
- 不能访问其他程序的存储空间
- 所有数据请求都要通过权限检测



隔离应使用的技术

- 隔离外部的程序应使用错误处理
 - 假定所有外部环境都是不安全的
 - 输入是失控的、无法预测的
- 隔离内部的程序应使用断言
 - 传入的数据已被过滤
 - 应该不会出现错误或危险的数据

隔离应使用的技术



假定此处的数据是
肮脏且不可信的。

这些类要负责清理数据。
它们构成了隔栏。

这些类可以假定数据
都是干净且可信的。

4

辅助调试代码

辅助调试代码

- 使用辅助调试代码，以帮助快速检测错误
 - 断言代码
 - 错误处理代码
- 应尽早引入辅助调试代码，以帮助开发者在早期就可发现、定位错误

保留防御式代码

- 业界矛盾
 - 在开发阶段，希望尽可能多地检测并显示出错误，引入大量防御式代码
 - 在发布阶段，希望错误尽可能少的出现

保留防御式代码

- 保留检测重要错误的代码
- 去掉检测细微错误的代码
- 保留有可能导致程序稳妥崩溃的代码
- 去掉有可能导致程序硬性崩溃的代码
- 留好接口记录错误信息
- 确认错误信息都是对维护人员友好的

对防御式编程采取防御的姿态

- 过度的防御式编程也会引发问题
- 如果在程序每一个函数里都加入防御式代码，将会导致程序变得臃肿，运行缓慢
- 增加产品复杂度，增大维护难度
- 应在适当的地方使用防御式编程
- 要考虑好什么地方需要进行防御

