复旦大学计算机科学技术学院

编程方法与技术

E.1. 上节课复习

周扬帆

2021-2022第一学期

Java注解

```
@Override
void printRecord() {
        super.printRecord();
        System.out.println("Address: " + address);
}
```

- 加入源代码的特殊语法元数据
- 用于标注类、方法、变量、参数和包
- 编译器可以获取并做相应处理
- 标注可以被嵌入到字节码中
- 运行时可以获取到标注内容
  - 可通过反射获取标注内容

元注解

- @Retention
  - 标识这个注解的作用期
  - 代码到编译/编译后到加载/加载到运行
  - 默认是RetentionPolicy.CLASS
- @Documented
  - 标记这些注解是否包含在用户文档中
- @Target
  - 标记注解的应该是哪种东西（方法、属性、类...）
- @Inherited
  - 默认作用于子类
  - 如果父类有这个注解，子类如果没有任何注解的话，会自动应用父类这个注解

自定义注解

- 实现代码来处理注解
  - 反射!
- 方法的注解

```
@Retention(RetentionPolicy.RUNTIME)
public @interface YZ {
   int id();
   String msg();
}
public class A {
     @YZ(id=1, msg="hello")
     void test();
     //...
}
```

调用公有方法　　反射

```
try {
    Method testMethod = A.class.getDeclaredMethod("test");
    if ( testMethod != null ) {
        YZ anno = testMethod.getAnnotation(YZ.class);
        System.out.println("check value:"+anno.id());
    }
} catch (NoSuchMethodException e) {
    e.printStackTrace();
}
```

得到属性值

Java序列化

- 序列化Serialization
  - 一个对象可以被表示为一个字节序列
  - 该字节序列包括
    - 对象的类型信息
    - 对象中数据的类型
    - 对象的数据
- 反序列化Deserialization: 逆过程
  - 将字节序列转换成Java对象

read/writeObject读写数据

- 例：保存为文件
- 将FileOutputStream封装为ObjectOutputStream
- 调用其writeObject方法将可序列化对象写入到输出流

```
public static void main(String[] args) {
  //...
  try {
     FileOutputStream fileOut = new FileOutputStream(“student.ser");
     ObjectOutputStream outStream = new ObjectOutputStream(fileOut);
     outStream.writeObject(obj);
     outStream.close();
     fileOut.close();
  } catch(Exception e) {
     // ...
  }
}
```

对象输出流

write/readObject读写数据

- 例：从文件中导入
- 将FileInputStream封装为ObjectInputStream
- 调用其readObject方法将输入流写入到可序列化对象

```
public static void main(String[] args) {
  //...
  try {
       FileInputStream fileIn =new FileInputStream("student.ser");
       ObjectInputStream in = new ObjectInputStream(fileIn);
       obj = (Student) in.readObject();
       in.close();
       fileIn.close();
  } catch(Exception e) {
     // ...
  }
}
```

对象输入流

自定义序列化实例

```
public class ClassA implements Serializable {
    ......

    public transient ClassC unserializableObject = new ClassC();

    public ClassA() {
        unserializableObject.c = 6;
    }

    private void writeObject(ObjectOutputStream outputStream) throws IOException {
        outputStream.defaultWriteObject();
        outputStream.writeInt(unserializableObject.c);
    }

    private void readObject(ObjectInputStream inputStream)
            throws IOException, ClassNotFoundException {
        inputStream.defaultReadObject();
        unserializableObject = new ClassC();
        unserializableObject.c = inputStream.readInt();
    }
}
```

会被序列化的成员

- transient关键字
  - 修饰的属性强制不被序列化
  - 反序列化后变成默认值

```
public class Employee implements java.io.Serializable
{
        public String address;
        public transient int SSN;
        public int studentID;
        public void printInfo() {
                // …
        }
}
```

- 什么用?
  - 节省空间 （可以算出来的）
  - 没用的 （如和运行环境相关的）

复旦大学计算机科学技术学院

编程方法与技术

E.2. Safety介绍

周扬帆

2021-2022第一学期

Safety v.s. Security

- Security
  - Attacks
  - Resilience against attacks
  - 在有坏蛋的情况下，保证系统的“安全”
    - 减少、去除harm
- Safety
  - 关注不要产生坏的后果
  - 可能没有主动的attacks
    - ？
    - 猪头程序员
    - 没想到的运行环境

安全攸关软件事故

2003年美加停电事故：由于软件故障，美国和加拿大发生大面积停电事故，造成至少11人丧生

事故原因：电网管理**软件**内部实现存在重大缺陷，无法正确处理并行事件。

2006年巴西空难：由于防撞系统问题，巴西两架飞机相撞，造成154名人员丧生

事故原因：防碰撞**硬件系统故障**，但软件系统没有对故障处理的机制

2005年，东京证券交易所出现了人类历史上最长停机事故，造成的资金和信誉损失难以估算

事故原因：由于输入错误的升级指令，导致**软件运行环境**和预期的不一致

安全攸关软件事故

事故原因分析

2003年美加停电事故：由于软件故障，美国和加拿大发生大面积停电事故，造成至少11人丧生

软件自身实现问题

事故原因：电网管理**软件内部实现**存在重大缺陷，无法正确处理并行事件。

2006年巴西空难：由于防撞系统问题，巴西两架飞机相撞，造成154名人员丧生

软件设计对外部环境元素变化考虑不足

事故原因：防碰撞**硬件系统故障**，但软件系统没有对故障处理的机制

2005年，东京证券交易所出现了人类历史上最长停机事故，造成的资金和信誉损失难以估算

事故原因：由于输入错误的升级指令，导致**软件运行环境**和预期的不一致

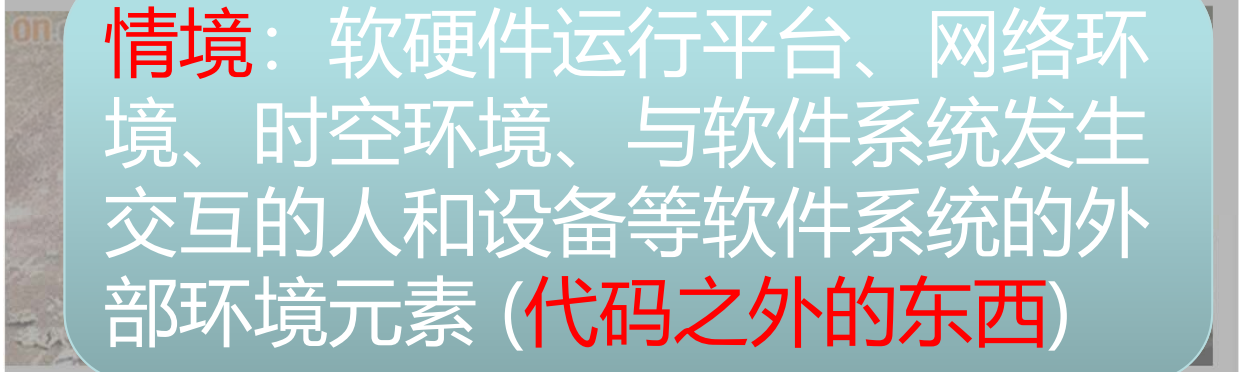
安全攸关软件事故

事故原因分析

2003年美加停电事故：由于软件故障，美国和加拿大发生大面积停电事故，造成至少11人丧生

事故原因：电网管理**软件内部实现**存在重大缺陷，无法正确处理并行事件。

2006年巴西空难：由于防撞系统问题，巴西两架飞机相撞，造成154名人员丧生



事故原因：防碰撞**硬件系统故障**，但软件系统没有对故障处理的机制

2005年，东京证券交易所出现了人类历史上最长停机事故，造成的资金和信誉损失难以估算

事故原因：由于输入错误的升级指令，导致**软件运行环境**和预期的不一致

情境：软硬件运行平台、网络环境、时空环境、与软件系统发生交互的人和设备等软件系统的外部环境元素 (代码之外的东西)

Safety v.s. Security

- Security
  - Attacks
  - Resilience against attacks
  - 在有坏蛋的情况下，保证系统的“安全”
    - 减少、去除harm
- Safety
  - 关注不要产生坏的后果
  - 可能没有主动的attacks
    - ?
    - 猪头程序员
    - 没想到的运行环境

复旦大学计算机科学技术学院

编程方法与技术

E.3. 编程语言的safety保障机制

周扬帆

2021-2022第一学期

Safe Programming

- 任何编程语言都可以写出unsafe的代码
- 讨论
  - 你们写C/JAVA/Python程序，觉得哪种语言容易写出有问题的程序，为什么?
- Safe的编程语言
  - 程序员不容易犯错
- Unsafe的编程语言
  - 程序员容易发错
- 什么叫容易/不容易犯错?

C语言Unsafe的部分

- **指针越界**

```
int *p = malloc(10*sizeof(int));
p[10] = 1;
```

- **指针随意cast类型**
  - double a = …;
    int i = *(int *)&a;
  - unsigned int i = …;
    [(unsigned int*)i)[0] = 1;
  - int add(int, int);
    int (*p) (int) =(int (*) (int)) add;
    p(0);

C语言Unsafe的部分

- **程序员需管理数据空间**
  - char buff[6];
    strcpy(buff, argv[1]);
  - char *src="hello world";
    char *dest=(char *)malloc(strlen(src));
    memcpy(dest, src, sizeof(src));
  - char *p = (char *)malloc(1024);
    //free(p)
    return;

Safety保障的代价

- 更严苛的规矩（类型系统）

  char *p = 0x4551FE99; X

- 提供更高层次的抽象，不能灵活操作底层

  int i = 10;

- …

- 设计一种语言，比C语言更多的限制/抽象
  - 灵活性差
  - 慢
  - 写更多代码
  - …

语言Safety保障的实现

- 语言本身需要做什么?
- 例子

  a[i] = (byte)b; 考虑类型、考虑对类型的操作

  - byte是一种数据类型
  - b是byte或者可以cast成byte的数据类型
  - a是一个可以存byte数据的数组/指针
  - i是一个可以作为index的数

  - a不是null
  - i不是负数
  - a可以至少有i个元素（的存储能力）

语言Safety保障的实现

语言Safety保障的实现

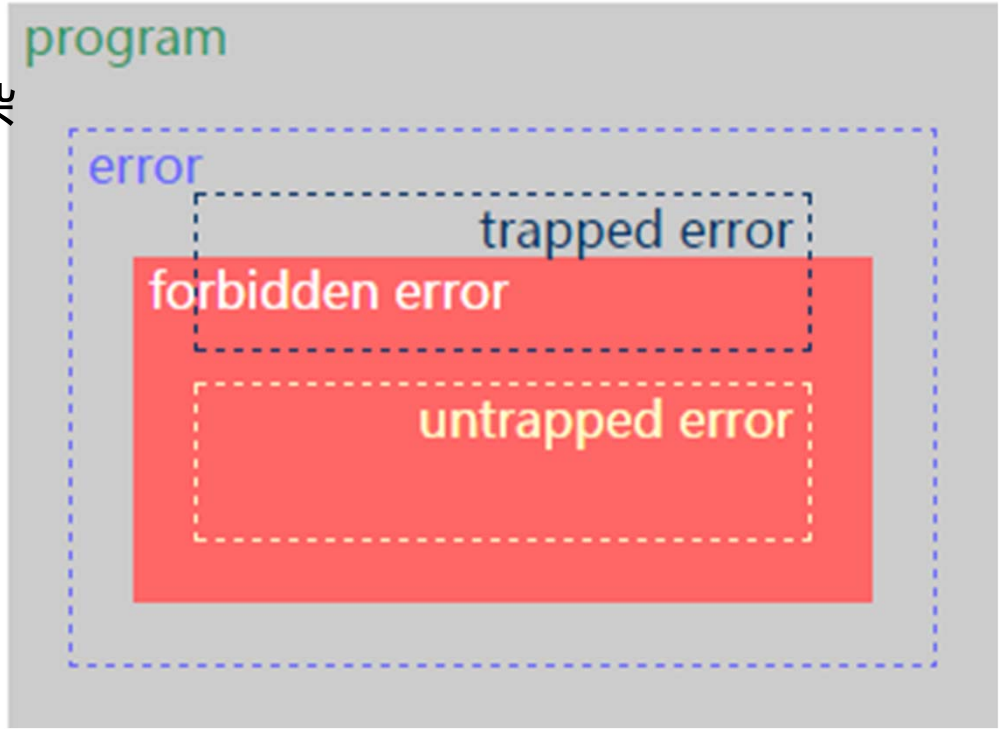
- **程序是什么?**
  - **信息如何表示**
  - **信息如何处理**

1. 一组基本类型构成的“基本类型集合”
2. “基本类型集合”上定义的一系列组合、运算、转换方法

数据类型抽象

- 编程语言的类型系统
  - 类型就是集合
  - 类型是语言的核心
    - 语法只是实现手段

类型有哪些，类型如何复合成复杂类型
类型操作的属性
    1 + 2
    1 + "Hello world"
等等



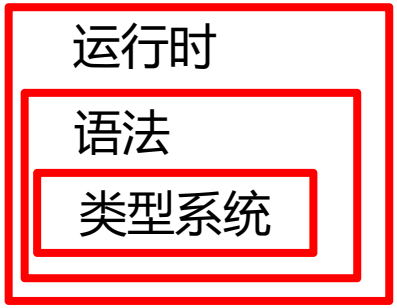
语言Safety保障的实现

- 语言的设计层面及编译(语法)检查
  - 让你做不了容易出错的事
    - 没有指针、有自动内存管理
    - 数据类型抽象(如boolean)
    - 面向对象的支持
    - 不能乱cast （类型安全）

IDE/Compiler

Runtime Environment

- 运行时
  - Exception handling



复旦大学计算机科学技术学院

编程方法与技术

E.4. Java语言的safety保障机制

周扬帆

2021-2022第一学期

数据类型抽象

- 8种基本类型
  - 整形：int 4字节 默认0
  - 短整形：short 2字节 默认0
  - 长整形：long 4字节 默认0
  - 浮点形：float 4字节 默认0.0f
  - 双精度浮点形：double 8字节 默认0L

Type Name	Size (in bits)
Integrals:	
byte	8
short	16
int	32
long	64
Floating Points:	
float	32
double	64
Characters:	
char	16
Booleans:	
boolean	n/a

数据类型抽象

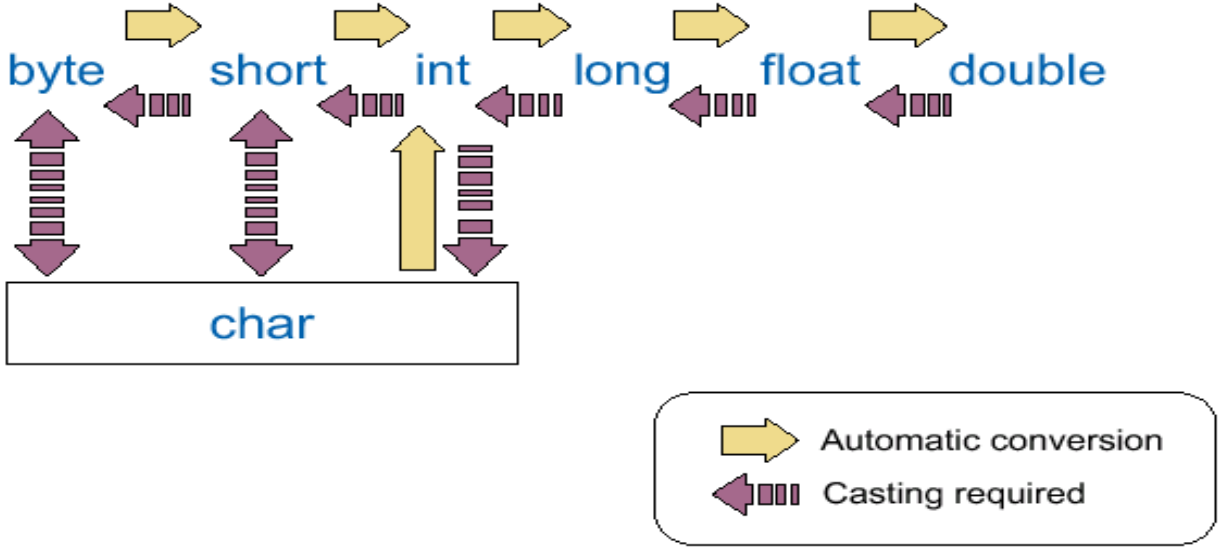
- 8种基本类型
  - 字符形：char 2字节 默认‘u0000’
  - 字节形：byte 1字节 默认0
  - 布尔形：boolean 1个bit 默认false (由于填充的关系，通常是1字节)

Type Name	Size (in bits)
Integrals:	
byte	8
short	16
int	32
long	64
Floating Points:	
float	32
double	64
Characters:	
char	16
Booleans:	
boolean	n/a

数据类型抽象

- 低级(规模"小")的类型可以自动转换为高级(规模"大")的类型
- 高级(规模"大")的类型需显式地转换为低级(规模"小")的类型

为什么



隐式类型转换精度不丢失： 强

数据类型抽象

- 指定存储大小
- 默认值
- Casting的规矩
- 数据操作的实现方式

```
int x, y;
x = 10;
y = x;
x++;
x = x + 1;
```

数据类型抽象

- 还有什么数据类型?
  - 数组、类-对象
- 数组
  - 元素都是同一种类型 (对safety的好处?)
  - 长度在创建时确定，并保持不变 (对safety的好处?)

    char s[] = new char[20];
    - 数组变量名，自带长度属性

      s.length
    - 越界检查（编译时能搞定所有的越界检查吗? ）

      s[21] = 1;

数据类型抽象

- 还有什么数据类型?
  - 数组、类-对象
- 类-对象
  - 类有数据和处理数据的方法（函数）
    - 函数在语言中的目的?

```
public class HumanBeing {
        private float energy = 0;
        public int Sleep() {
                energy += 50;
        }
}
```

结构化程序设计

- 分支关键字
  - if else
  - switch case break
- 循环关键字
  - while循环
  - for循环
  - do while循环
  - break continue控制循环过程
- 调用和返回return
- 异常处理try catch finally

结构化编程

非结构化编程?
有什么不好?

结构化程序设计

- goto有什么不好
  - 从一个block的中间跳到另一个block的中间
  - 变量状态、可读性?
- 从一个函数的中间跳到另一个函数的中间
  - 怎么办?
    - 函数指针
    - setjmp
  - 变量状态、可读性?

```
        void (*p)(void) = test1;
        p = p + 4;
        p();
//      longjmp(env, 2);
```

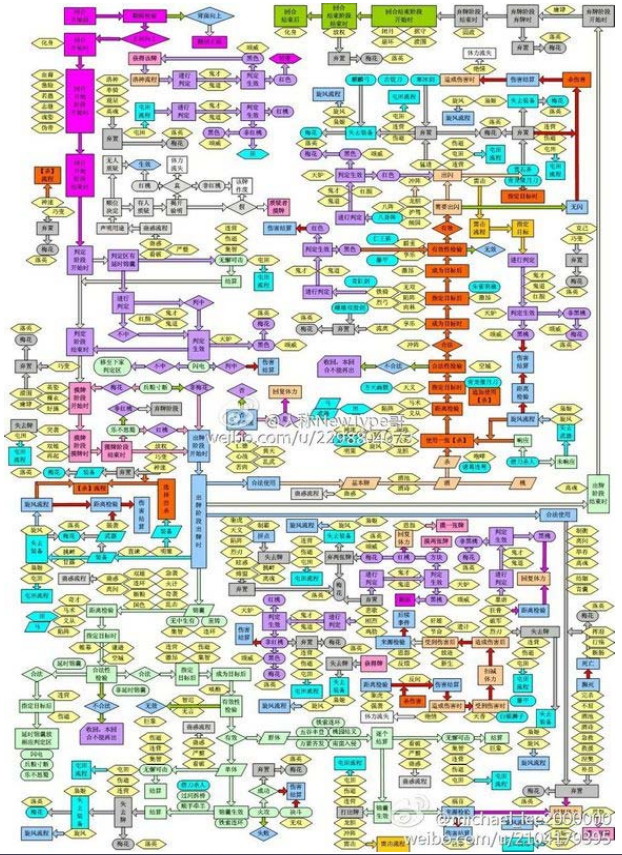
```
#include <setjmp.h>
#include <stdio.h>
#include <stdlib.h>

jmp_buf env;
void test1()
{
        int j = 999;
        int i = setjmp(env);
        printf("here1\n");
        if( i != 0) {
                printf("done! %d\n", j);
                exit(0);
        }

}
void test2()
{
        printf("here2\n");
        longjmp(env, 2);
}
void main() {
        test1();
        test2();
}
```

面向对象程序设计

- 面向对象和safety什么关系
  - 封装是什么？ - 面向对象 = 面向数据结构
  - 把数据结构和数据结构中数据的处理方法放在一起
    - 对safety的好处?
- 对比面向过程
  - 程序就是一个过程procedure
  - 过程中，各个环节都可能使用某个数据结构
    - 如果更改了?
- 封装
  - 就近原则



面向对象程序设计

- 对象的数据大部分情况需要初始化
  - 设定初始值
  - 初始化存储空间
  - 可以写一个initDataStructure()方法，让程序员new后必须调
    - → 有什么不好
    - → 万一忘了调了，就开始用怎么办?
  - **构造方法**
    - → 对象new的时候一定被调

面向对象程序设计

- 访问控制
  - public private protected default
  - 有些方法、数据不需要被其他人访问、修改
    - 内部用
    - 避免出错

面向对象程序设计

- **继承**
  - **继承的本质?**
  - **如果父类有抽象方法**

    abstract void onClick();
    - **有abstract方法，不能被new / 回忆C++的virtual**
    - **子类要能被new, 要实现具体的方法**
  - **只有抽象方法的接口（interface）/ 回忆C++的纯虚类**

```
interface Comparable {
    boolean compare(Comparable b);
}
class MyInteger implements Comparable {
    public boolean compare(MyInteger b) {
            …
    }
}
```

面向对象程序设计

- **多态**
  - **子类是一种父类，能被当作父类来使用**
    - A a = new B(); //if A is a super class of B
  - **里式替换法则：类型系统上的一个形式化的属性**
  - **易于扩展**
- **继承、多态**
  - **开闭原则**
    - Software entities (classes, modules, functions) should be open for extension but closed for modification
    - 原因?
    - 和继承、多态什么关系

面向对象程序设计

- 继承、多态
  - 如果父类的某个方法不允许覆盖

    final void onClick();
    - 保证父类某个方法一定会被调用，进而保障父类数据结构可控
  - 如果某种类不想被继承，免得瞎改

    final class XXX {

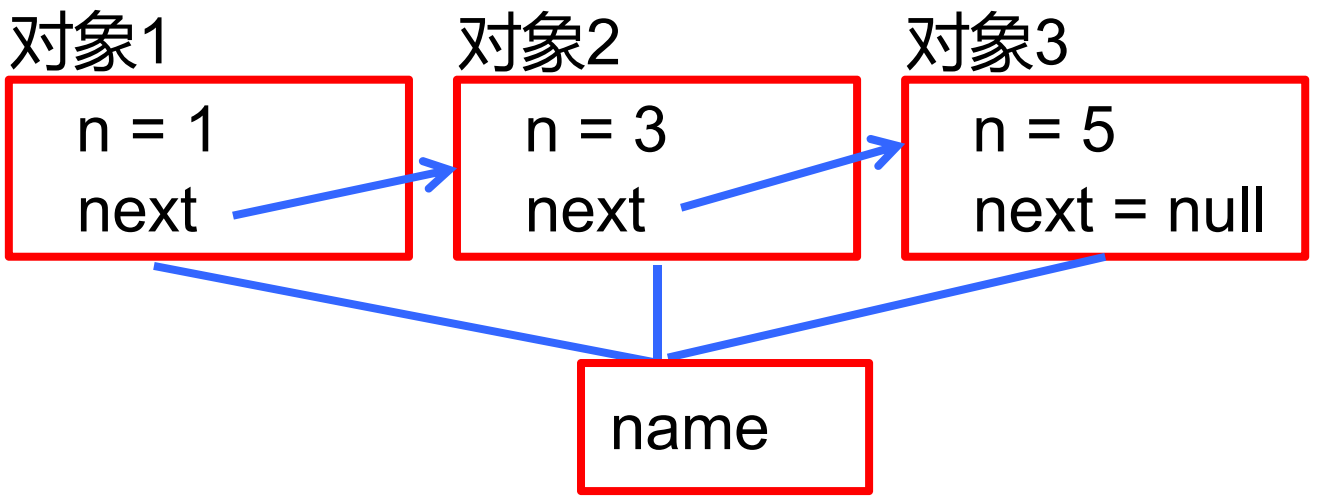
    …

    }

面向对象程序设计

- Java是相对比较纯粹的面向对象语言
  - 无全局变量？？
  - static变量
    - 有些变量是本类所有对象**共用**的， 在对象中**共享**一个公共变量
    - 节约空间/方便对象共享数据
    - 访问不需要对象，通过类访问Ele.name
    - safety的考虑?

```
class Ele {
    public static String name;
    ……
}
```



Java泛型类

- 类的泛型 （模板类）
  - 类的实现中，把某些用到的数据类型抽象为泛型（模板）
  - 在类创建的时候才指定类型
  - 此模板可以接受合适类型的对象

```
class Datum <T> {
        private T var;
        public T getVar() {
                return var;
        }
        public void setVar(T var2) {
                var = var2;
        }
}
```

可做变量定义

可做返回值定义

可做参数定义

Java泛型类

- **目的?**
  - **相对于使用父类类型，更加安全**
  - **编译的时候可以进行类型检查，避免代码写错**

```
class Datum <T> {
        private T var;
        public T getVar() {
                return var;
        }
        public void setVar(T var2) {
                var = var2;
        }
}
public class Test {
        public static void main(String args[]) {
                Datum <Integer> datum = new Datum <Integer>();
                datum.setVar("Wrong");
                System.out.println(2 *(Integer) datum.getVar());
        }   }
```

语法错好过运行错

语法检查不通过，因为类型不符合

语言Safety保障的实现

- 语言的设计层面及编译检查
  - 让你做不了容易出错的事
    - 没有指针、有自动内存管理
    - 数据类型抽象(如boolean)
    - 面向对象的支持
    - 不能乱cast

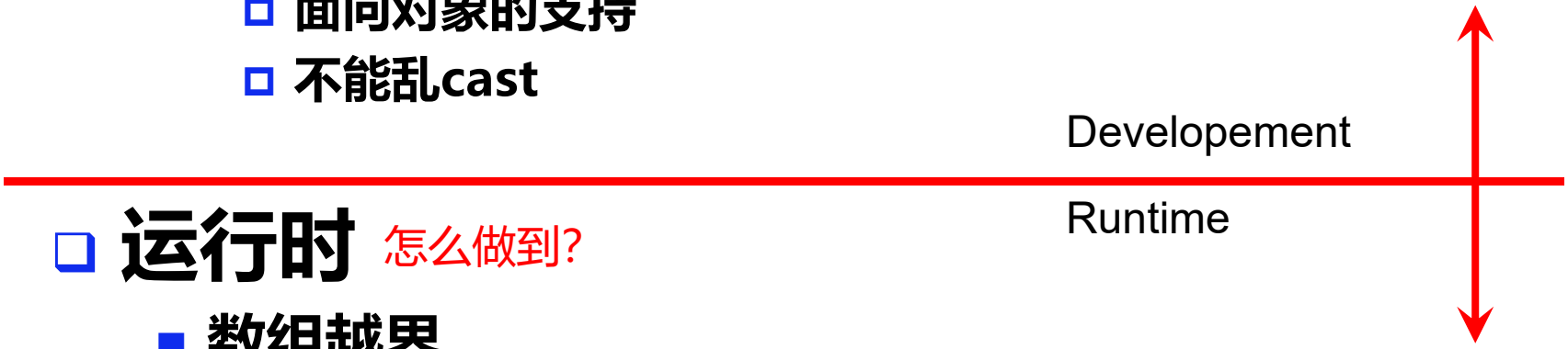
IDE/Compiler

- 运行时
  - 数组越界
  - 文件找不到了，U盘拔出来了

Runtime Environment

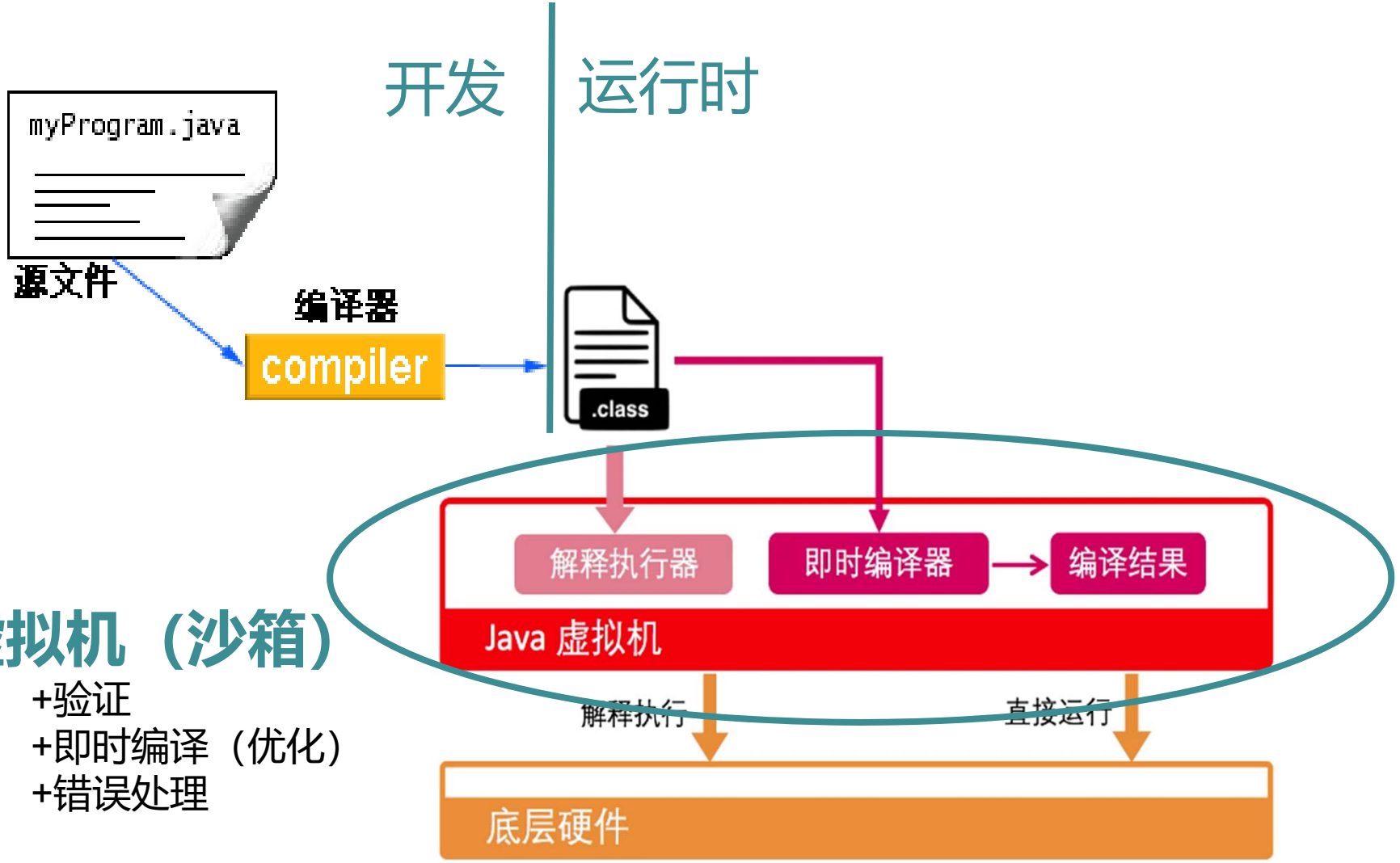
语言Safety保障的实现

- 语言的设计层面及编译检查
  - 让你做不了容易出错的事
    - 没有指针、有自动内存管理
    - 数据类型抽象(如boolean)
    - 面向对象的支持
    - 不能乱cast



- 运行时 怎么做到?
  - 数组越界
  - 文件找不到了，U盘拔出来了

Java特性



虚拟机（沙箱）

+验证
+即时编译（优化）
+错误处理

Runtime Environment和Safety

- 代码检查、内存保护
  - 没有野指针
- 内存管理
  - 垃圾回收，只new不用delete
- 运行时错误检查
  - Exception Handling
- …

垃圾回收

- 没有指针，内存自动管理
  - new了怎么自动delete呢?
  - GC- Garbage Collection
  - 可轻松实现内存管理的原因：无灵活的指针操作
- 垃圾回收
  - 谁需要垃圾回收：堆(heap)/栈(stack)?
  - 怎么做：错误回收的后果?
    - 引用计数
    - 可达性分析

垃圾回收机制

- **如何进行内存回收**
  - 标记清除算法、复制算法、标记整理算法
  - 分代回收算法
  - 垃圾回收器
- **时间停止 – Stop-the-world**
  - GC的时候，所有线程都暂停，直至GC结束
  - 因此GC需要优化、节省Stop-the-World时间

可达性分析

- GC Root: 正运行的程序可访问的引用变量
- 从GC Root出发找它们引用的对象
  - 递归 → 引用链
- 引用链上没有的对象:
  - 从GC Root无可达路径
  - 可GC

垃圾回收：可达性分析

- **GC Root: 正运行的程序可访问的引用变量**
  - 虚拟机栈(局部变量)中的对象引用
  - 本地方法栈中native方法的对象引用
  - 方法区中类的静态属性的对象引用
  - 方法区中常量的对象引用
- **从GC Root出发找它们引用的对象**
  - 递归 → 引用链
- **引用链上没有的对象:**
  - 从GC Root无可达路径
  - 可GC

垃圾回收机制

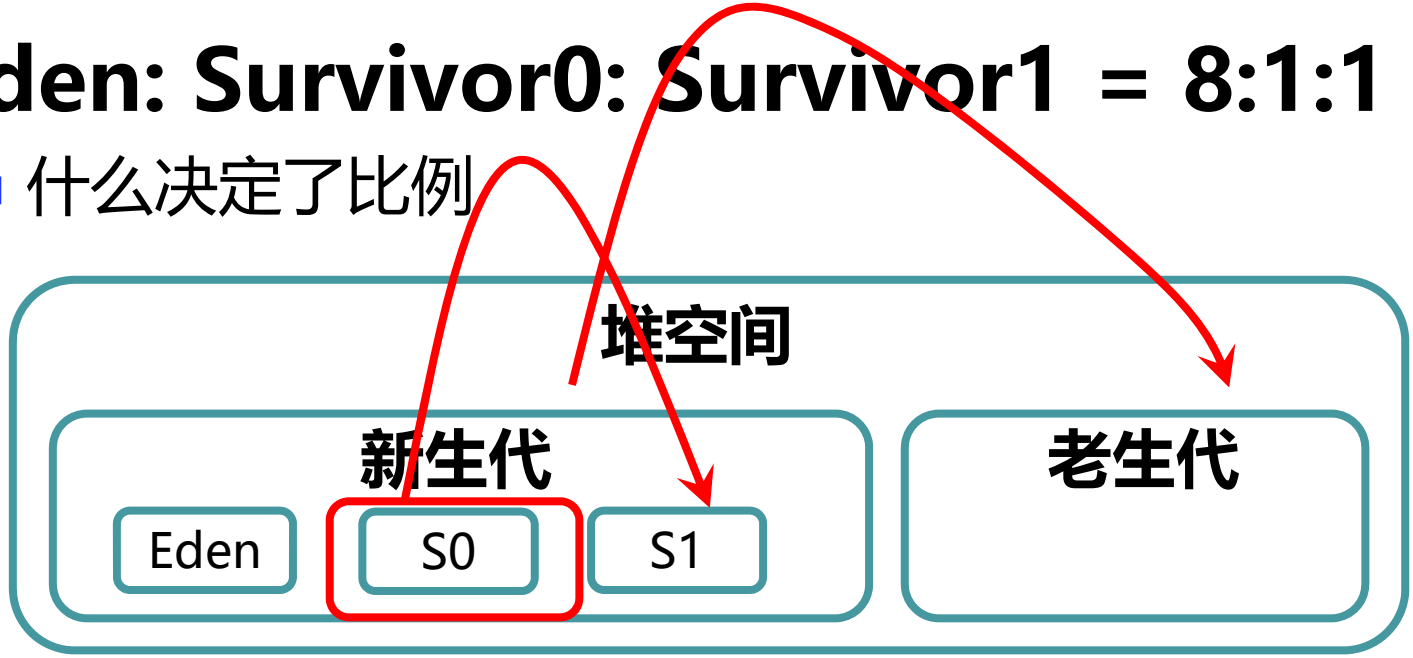
- **哪些内存需要回收**
  - 错误回收 → 程序出错
  - 经典算法：引用计数法、可达性分析
- **如何进行内存回收**
  - 标记清除算法、复制算法、标记整理算法
    - STW？时间？
    - 存储效率？
  - 分代回收算法
  - 垃圾回收器

垃圾回收机制

- **哪些内存需要回收**
  - 错误回收 → 程序出错
  - 经典算法：引用计数法、可达性分析
- **如何进行内存回收**
  - 标记清除算法、复制算法、标记整理算法
  - 分代回收算法
  - 垃圾回收器

新生代GC: Minor GC

- 对象创建在Eden
- Eden回收，存活 → S0
- S0回收，存活 → S1, 交换S0/S1
- 采用复制算法原因?
- Eden: Survivor0: Survivor1 = 8:1:1
  - 什么决定了比例

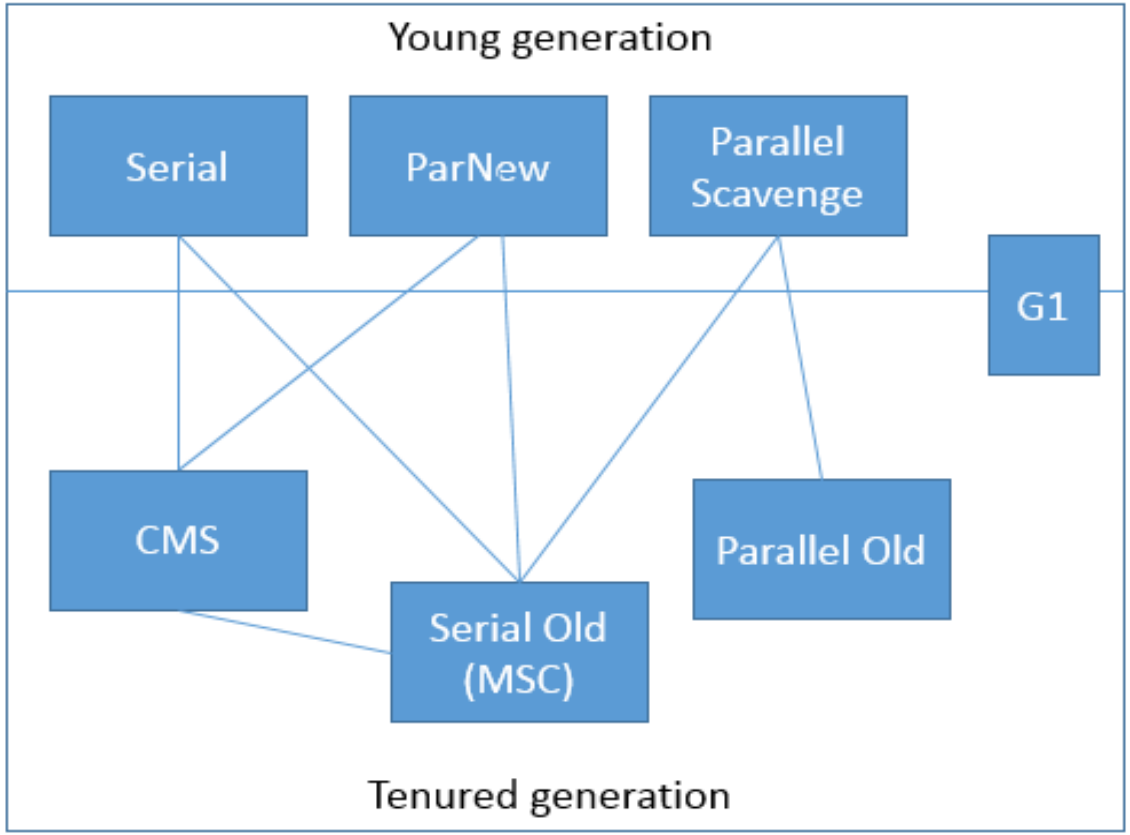


垃圾回收机制

- **哪些内存需要回收**
  - 错误回收 → 程序出错
  - 经典算法：引用计数法、可达性分析
- **如何进行内存回收**
  - 标记清除算法、复制算法、标记整理算法
  - 分代回收算法
  - 垃圾回收器

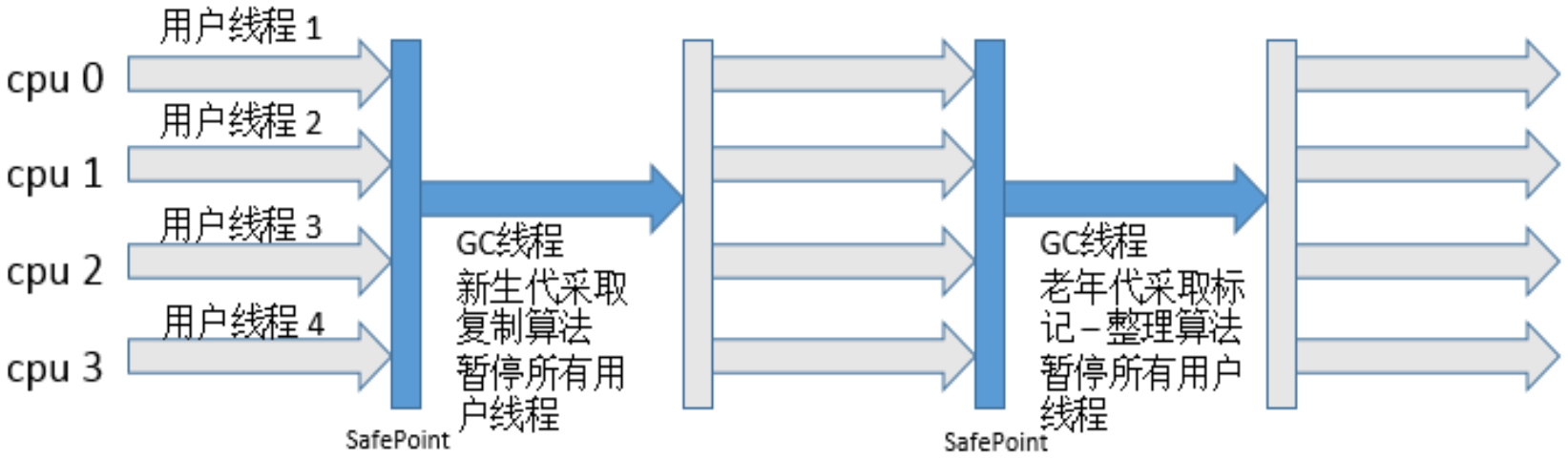
垃圾回收器概览

- 垃圾回收器
  - 虚拟机对垃圾回收算法的实现



Serial/Serial Old回收器

- 单线程回收器
- Serial回收器针对新生代采用Copying算法
- Serial Old回收器针对老生代采用Mark-Compact算法



Serial/Serial Old 收集器运行示意图

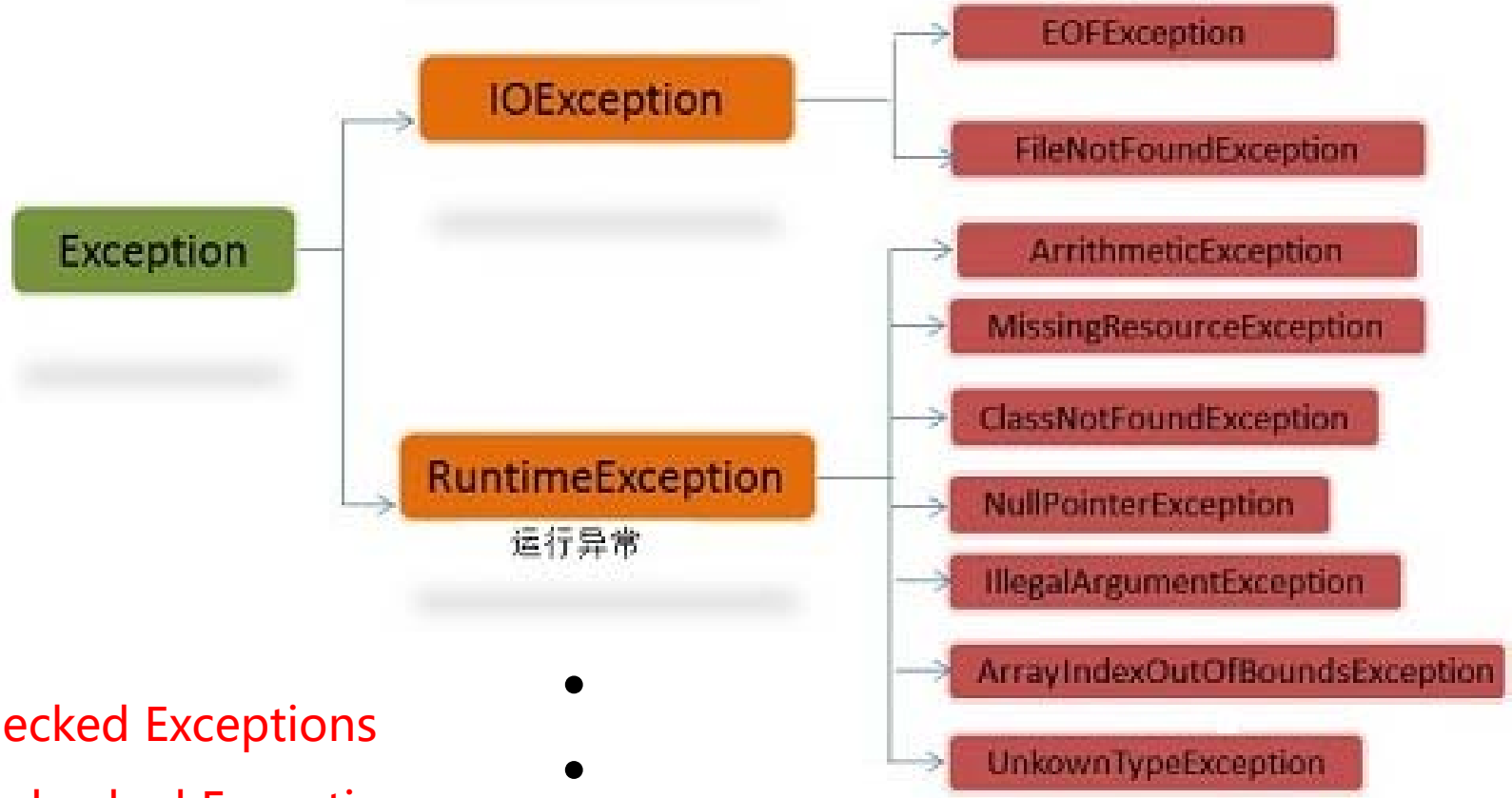
Runtime Environment和Safety

- 代码检查、内存保护
  - 没有野指针
- 内存管理
  - 垃圾回收，只new不用delete
- 运行时错误检查
  - Exception Handling
- …

Java异常

- 程序运行时会遇见很多异常
  - 文件找不到
  - 读写文件时发生IO错误
  - 网络连接失败
  - 参数非法
  - 空引用
  - 数组越界
- Safety机制
  - 通过异常的捕捉处理、防止程序奔溃

Java异常



checked Exceptions

unchecked Exception

Java异常

- Checked exceptions
  - 提供机制，强制程序员写异常处理
  - 什么是需要强制的?
- Unchecked exceptions
  - 提供机制，让程序员可以在发生异常后，进行处理

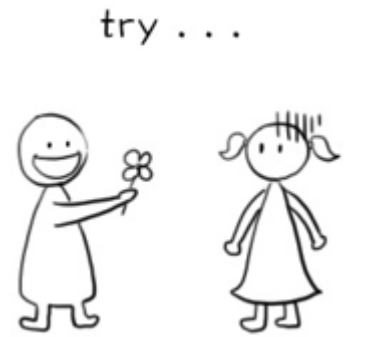
程序员

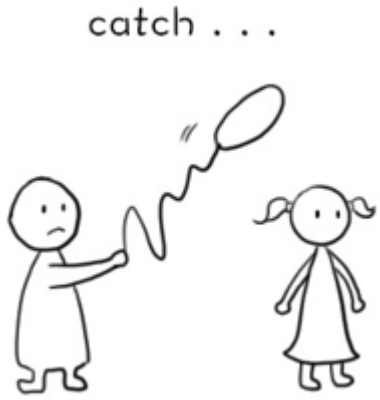
环境

Java异常的捕捉

- **try/catch/finally**

```
try {
        //可能会抛出异常的语句
}
catch (XXException e) {
        //异常处理的语句
}
catch (XXException e) {
        //异常处理的语句
}
finally  {
        //最后需要执行的语句
}
```





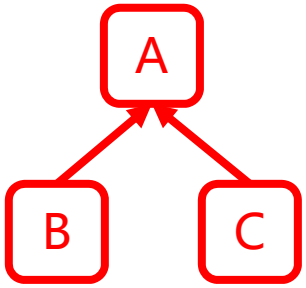
异常捕捉示例

```
public static void copy(String sFile, String dFile) {
        File srcFile = new File(sFile);
        File destFile = new File(dFile);
        FileInputStream fin = null;
        FileOutputStream fout = null;
        try {
                fin = new FileInputStream(srcFile);
                if (!destFile.exists()) {
                        destFile.createNewFile();
                }
                fout = new FileOutputStream(destFile);
                byte[] bytes = new byte[1024];
                while (fin.read(bytes) != -1) {
                        fout.write(bytes);
                        fout.flush();
                }
        } catch (FileNotFoundException e) {
                System.out.println("Can find the source file: " + sFile);
        } catch (IOException e) {
                System.out.println("IO Exception caught.");
        }

        ...
}
```

Java异常示例

- 数组index越界，index负值
- null引用
- 创建大小为负数的数组
- 找不到类加载，找不到方法执行，找不到成员变量
- 执行了虚的类方法
- 错误的casting




```
B b = new B();
A a = b;
C c = (C)a;
```

子类可以cast成父类，父类需要显式cast成子类

？？

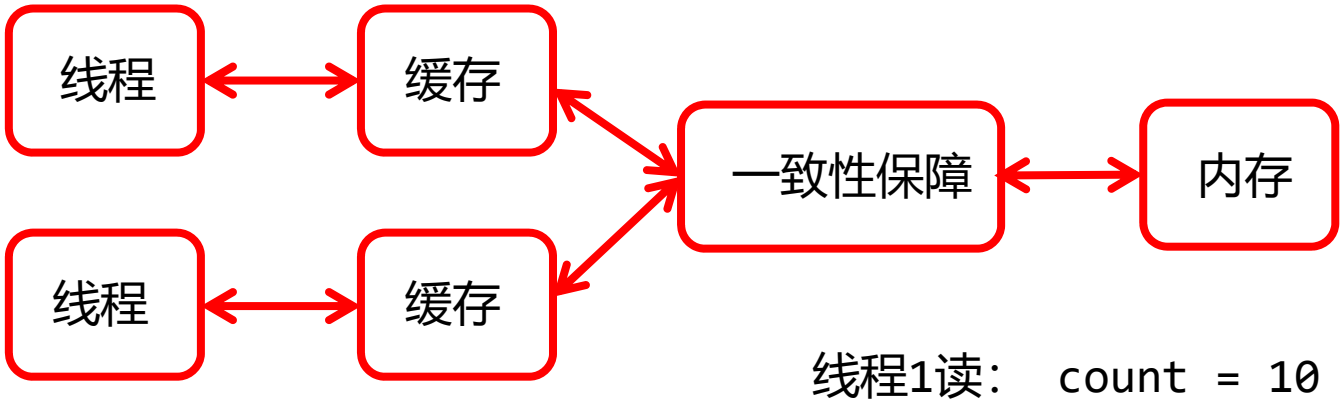
- ……

内存/线程的safety

```
public class SyncDemo {
        public volatile static int count = 0;
        public static void inc() {
            try {
                Thread.sleep(1);
            } catch (InterruptedException e) {
            }
            count++;
        }
        public static void main(String[] args) {
            for (int i = 0; i < 5000; i++) {
                new Thread(new Runnable() {
                    public void run() {
                            SyncDemo.inc();
                     }
                }).start();
            }
            // … try {Thread.sleep(5000); } catch (InterruptedException e) { …}
            System.out.println("Counter.count=" + SyncDemo.count);
        }
}
```

内存/线程的safety

- 缓存一致性



线程1读： count = 10
线程2读： count = 10
线程1写： count = 11
此时线程1cache的count是11
此时线程2cache的count是10

内存/线程的safety

- 内存模型
  - 非原子操作(int i,j)
    - → i ++ ;
    - → i = i+1;
    - → j = i;
  - 原子操作
    - → i = 10;
- Java提供了某种锁机制

synchronized关键字

- 可修饰方法

```
public synchronized void myMethod(... ) {
        ...
}
```

  - 给这个方法加上一个锁(lock) – 对象的monitor
  - 同时，只有一个线程可以在执行这个方法
    - → 谁执行，谁获得锁
    - → return释放锁
  - 请测试一下，如果一个类两个方法都用synchronized修饰，能不能同时进入

- 可修饰块(block)

```
synchronized（对象的引用）{
        // 执行时，该共享对象被锁
        // 只能同时被一个线程执行
}
```

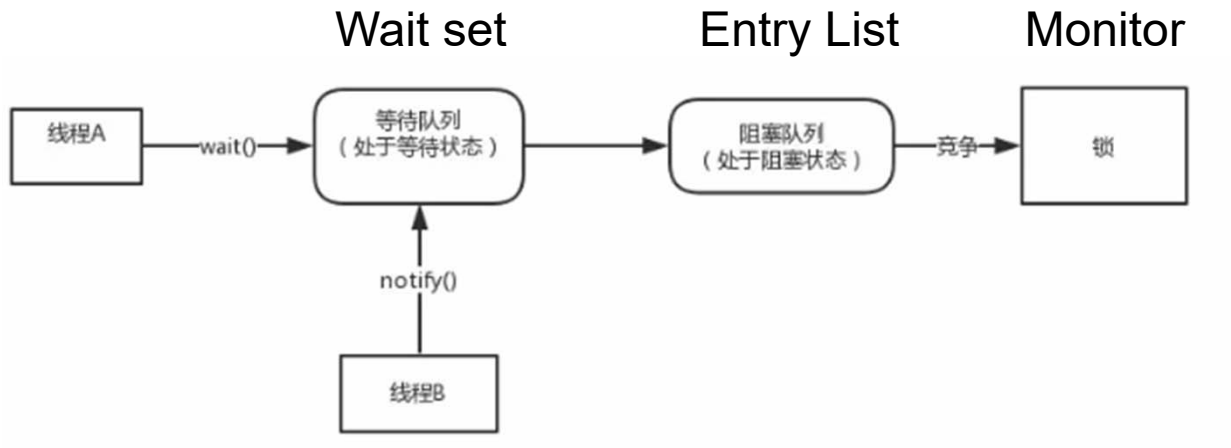
wait/notify/notifyAll机制

- wait()
  - 必须持有monitor, 否则抛异常
  - 交出去monitor
  - 然后在wait set上等着被notify
  - notify了之后去entry list等重新获得monitor
- notify()/notifyAll()
  - 必须持有monitor, 否则抛异常
  - wait set上的一个/所有线程被notify
  - 不交出去monitor

```
lock.wait();
...

synchronized (lock) {
    ...
}
```

wait/notify/notifyAll机制



内存/线程的safety

- 锁（monitor）机制保障原子性
  - 问题?
    - 大部分情况需要程序员自己考虑加锁、死锁、慢
- 多线程还有很多safety的challenges
  - 比如乱序执行

Original code

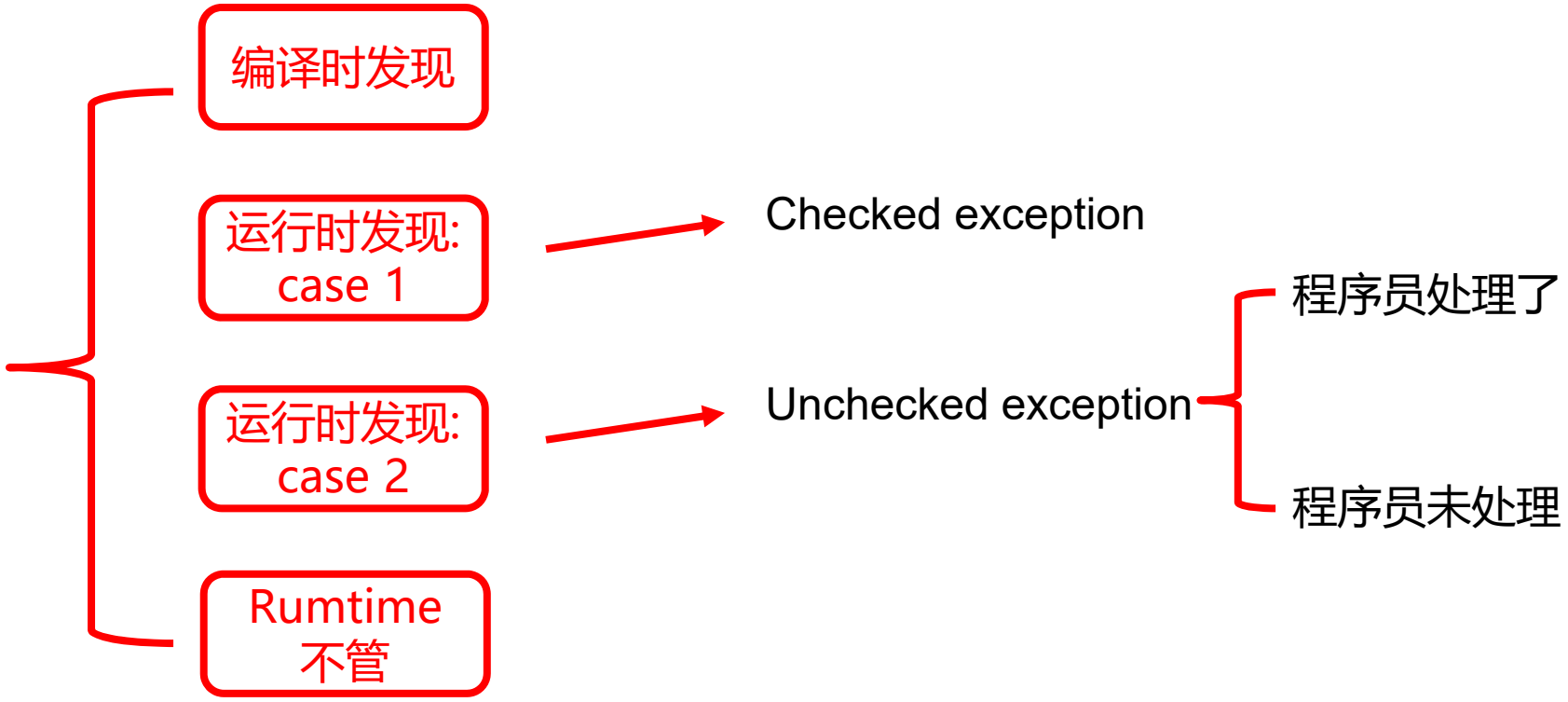
Initially, A == B == 0

Thread 1	Thread 2
1: r2 = A;	3: r1 = B
2: B = 1;	4: A = 2

May observe r2 == 2, r1 == 1

- JSR133
  - 内存模型的标准文件

Safety威胁的处理



复旦大学计算机科学技术学院

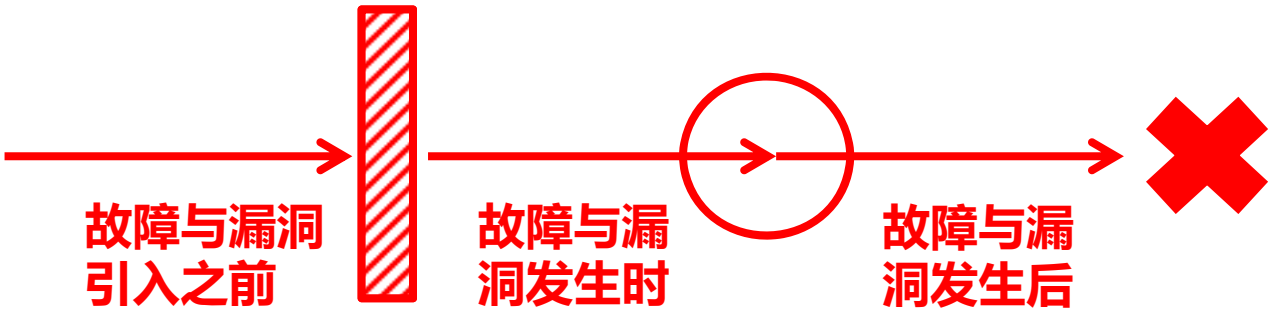
编程方法与技术

E.5. 程序设计中的safety保障机制

周扬帆

2021-2022第一学期

Safety保障



- 防: 防止软件故障的发生
  - 软件采取的主动(proactive)性的预防策略
- 容: 降低故障对软件/系统的破坏
  - 软件采取的被动(reactive)性的防御策略
- 除: 排除已经发生的故障
  - 定位其所在, 以协助人工干预或采用自动化手段, 更正软件

Safety保障

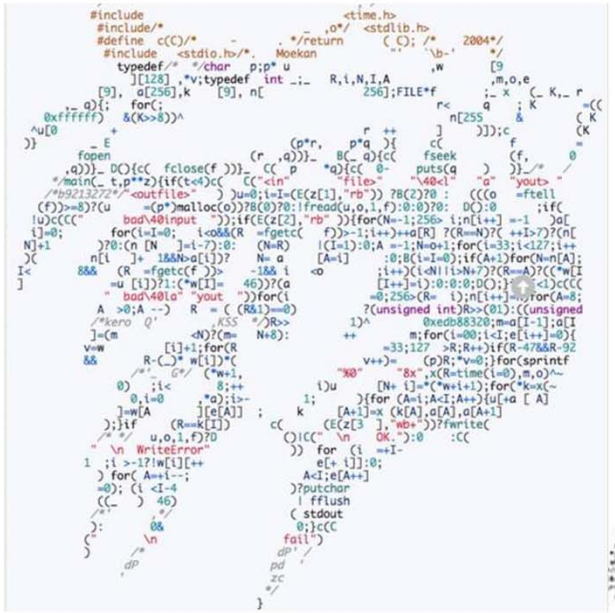
- 代码风格
- 程序的组织/命名空间
- 测试、排错
- Safety保障的软件设计

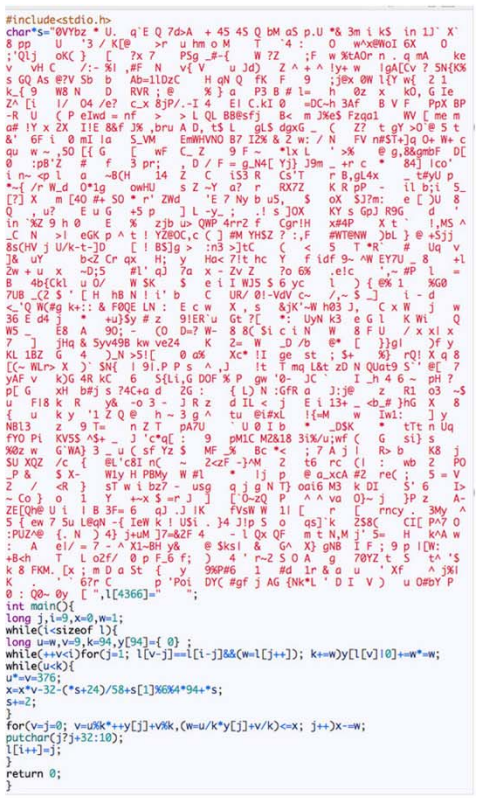
Safety保障

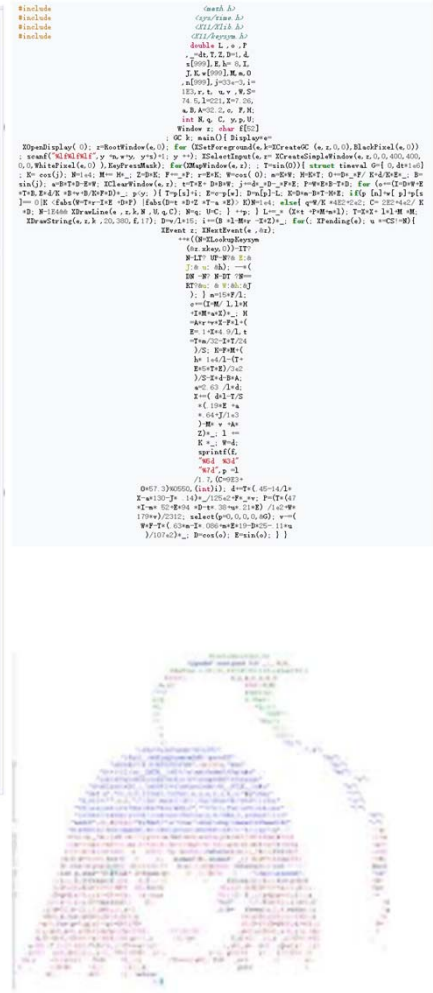
- 代码风格
- 程序的组织/命名空间
- 测试、排错
- Safety保障的软件设计

IOCCC

- The International Obfuscated C Code Contest
  - www.ioccc.org/
  - 一年一次







JAVA代码风格

- 为什么要代码风格coding style
  - 人要活得有style, 我有我风格?
  - No!

- 软件生命周期
  - 大部分时候，代码是要维护的：修bug，改进
  - 大部分时候，维护代码的人 != 开发代码的人
- 代码风格
  - 协助别人理解
  - 协助自己理解（遗忘）

命名规范

- **命名规范: 有利于理解代码**

文件名: StudentRecord.java		MyClass.java
函数名: getRecordbyStudentID	v.s.	naJilu
变量名: studentName		stringA

- **c/c++命名规范**
  - **匈牙利命名法则**
    - → **微软**
    - → **nStudentNumber, pName, m_pName, g_pName**
    - → **[作用域+下划线]+变量类型+变量描述**
    - → **变量描述：词组或词，首字母大写**
    - → **某些批评：太麻烦，废话太多**

命名规范

- 蛇形命名法：snack_case
  - like_this，常见于Linux内核，C++标准库等
- 驼峰命名法：CamelCase
  - 小驼峰：likeThis， 第一个字母小写
  - 大驼峰（帕斯卡命名法）：LikeThis，第一个字母大写

命名规范

- Java风格惯例
  - 类和接口名: 大驼峰
    - → StudentHall
  - 变量和方法: 小驼峰
    - → getRoomNumber
    - → roomNumber
  - 常量: 蛇形全大写 MAX_VALUE
    - → static final int MIN_WIDTH = 14;
  - 包名:全部使用英文小写字母
    - → 项目：<域名反转>.<团队名>.<父项目名>.<子项目名>

Java命名规范

- 遵从一般人的风格
  - 类名: 大驼峰
  - 变量和方法: 小驼峰
- 名字取容易理解的词

文件名: StudentRecord.java
函数名: getRecordbyStudentID
变量名: studentName

- 尽量不要用拼音
  - String xueShengXingming
  - String studentName

Java命名规范

- **除非是作用域很小的变量，否则避免单字母随便拎过来就用**

```
for(int i = 0; i < studentNum; i++)
```

```
int a[] = new int [n];
//此处省去30行

//看到这里就费事去理解a是什么了
caculateValueofP(a);
a[m] = 30;
```

Java命名规范

- 变量和常量: 名词词组
  - studentName
- 方法: 动词词组
  - getStudentName
- 类和接口: 名词词组
- 对象名: 对象属于变量，名词词组

排版风格

- 排版也有助于理解代码的结构

```
for(;;){int i = j = 10; do{
i --; j++}while(  i != 0)
```

- K&R缩进风格

```
if (<cond>) {
        <body>
}
```

  - 缩进8或者4个空格，或者一个tab
  - { 在上一行末
  - } 独立一行

排版风格

- **BSD缩进风格**

```
if (<cond>)
{
        <body>
}
```

  - **缩进8或者4个空格，或者一个tab**
  - **{ 和 }都独立一行**
- **其他缩进风格**

Java排版风格

- Java多采用K&R风格
- 缩进用tab还是用空格?
  - 随便, 保持统一
- 一行不超过一条语句
  - if/else单独一行
  - for/while/do单独一行
  - switch/case/default/单独一行

Java排版风格

- 长行切断换行规则
  - 切在比较好理解的分界点
  - 下一行的缩进要考虑上一行，建议比上一行缩进一个位置 （tab或者4/8个空格）
  - if (((a == b) && (c ==d)) || (a > c)
    || c< 0)

Java排版风格

- 大部分操作符前后留空格
  - int i = 10; 等号前后都有空格
  - 一元操作符除外: !b
- 小括号
  - if ((a == b) && (d == e))
    - → (前面不是括号或者一元操作符，便和前面间隔空格
    - →) 后面不是括号，便和后面间隔空格
  - 函数的(前面不留空格
- 中括号，前后没有空格

……

Java注释风格

- 原则
  - 多注释
  - 注释形式统一

    如/*　　*/ 用于类、方法、域的注释

    // 用于程序内部逻辑的注释

    todo: 用于表示未完成功能

    fixme: 用于表示是临时代码，需要更改
  - 注释内容准确简洁

    注释一定要准确，不模棱两可

Java注释风格

- 一般需注释
  - 类（接口）的注释
  - 构造函数的注释
  - 方法的注释
  - 域变量的注释

Java注释风格

- 一般需注释
  - 典型算法必须有注释
  - 在代码不明晰处必须有注释
  - 在代码修改处加上修改标识的注释
  - 在循环和逻辑分支组成的代码中加注释
  - 为他人提供的接口必须加详细注释。

Java逻辑代码注释经验

- 多注释永远是个梦，除非...
- 不要在实现好之后注释
  - 一般码农哪有那闲功夫
- 在写代码之前注释
  - 用注释理清代码逻辑
  - 再开始写代码

成年人程序员

- 不耍酷
  - 唯恐别人看不懂代码
- 不赌一把
  - 不确定的事要确定下来
    - →如多打几个括号，多写casting

Safety保障

- 代码风格
- 程序的组织/命名空间
- 容错设计模式
- 测试、排错

Java程序源文件

- 放在一个文件的坏处
  - 不方便模块化分工
  - 不方便编译
  - 不方便动态加载
  - 不方便版本控制

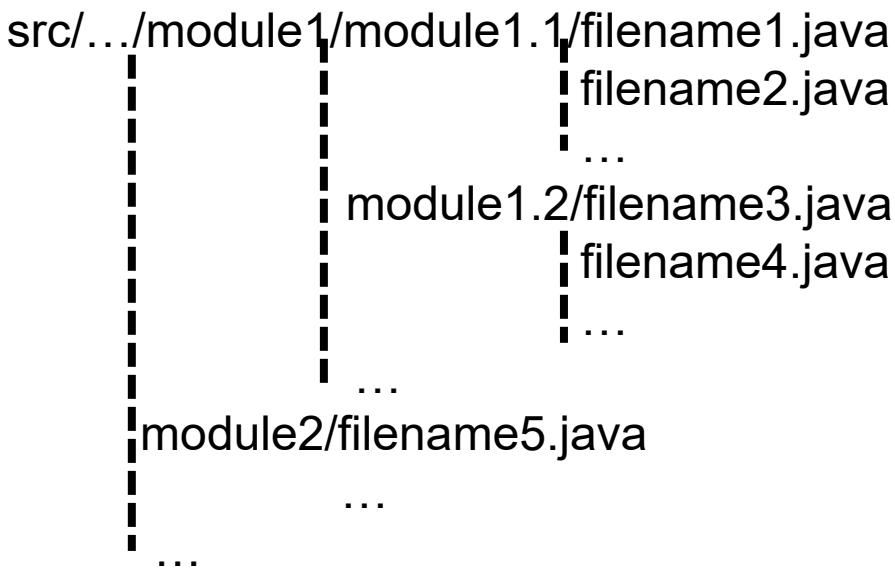
  …
- Java程序一般由多个源文件构成
- 源文件怎么组织

Java项目的源代码组织

- 最直接的做法：一个类对应一个源文件
  - 好处1：方便程序员管理
  - 好处2：方便JVM加载
  - 坏处：琐碎、麻烦
- Java的折中
  - 一个源文件可以包含多个类
  - 但是每个文件只能包含一个public类

Java项目的源代码组织

- 多个源文件如何组织其层次结构
  - 扁平化处理？放在同一个文件夹中
  - 缺点：太乱、不方便模块化管理等
  - 根据逻辑关系，按模块分目录，树状管理



方便理解、方便分工 …

命名空间

- 不同团队需要协调类的名字
  - 不能重名，否则JVM出错
- 保证不重名太麻烦，甚至不可行
- 需引入命名空间
  - 在一个命名空间内不重名
  - 不同命名空间，可以重名
  - 方法：Java包(package)

Java包（package）

- 包是一组类（源文件）的集合
- 一般一个功能模块放在一个包里
  - 如日志处理
  - 如网络功能
- 指定包名的语句
  - package pkg1[.pkg2[.pkg3...]]
  - 如package abc.de.fg
  - 源文件的第一条语句

Java包（package）

- 指定了包名的源文件需放在特定目录
  - 如用package abc.de.fg指定了包名的文件
  - 放在 abc/de/fg/ 目录下
  - 方便JVM查找
- 不指定包名的文件，属于“无名包”
  - 只能放在源文件根目录下
- Java类库（JDK）都会指定在JDK定义的包中
  - 如之前用过的Random类：java.util.Random

Safety保障

- 代码风格
- 程序的组织/命名空间
- Safety保障的软件设计
- 测试、排错

好读、好理解 → 好维护
是关键的safety保障机制

Safety保障的软件设计

- **Sanity check ?**

```
if(p == null) {
        return;
}
p.func();
```

```
if(index <= 0) {
        return;
}
array[index] = xxx;
```

- **在程序合适的位置，判断数据的完整性、正确性**

Safety保障的软件设计

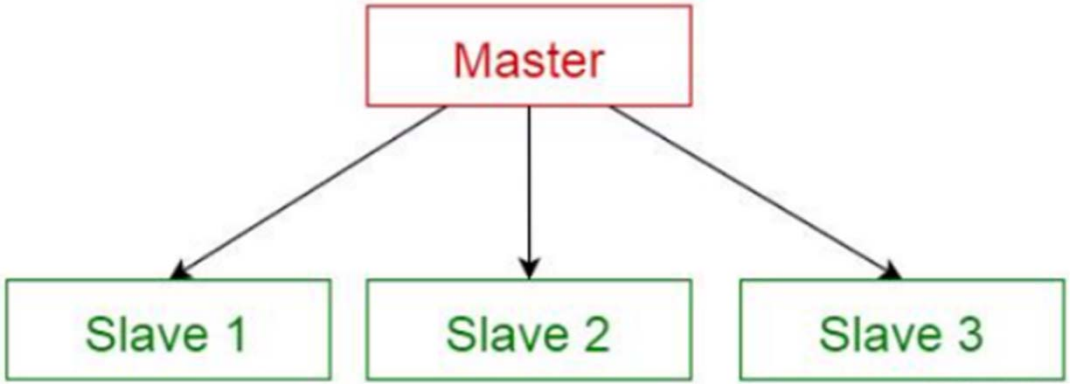
- 如何应对failure
  - 语言的Exception handling
- 主备模式: 同样功能用多个服务实现
  - 简单的客户-单服务模式什么不足?
  - 把同样功能实现在不同机器/进程/模块上
- 场景?
  - 功能足够重要
  - 坏的原因?

Safety保障的软件设计

- 怎么知道坏了?
  - Heartbeat / Watchdog
  - Acknowledgement
- 坏了的服务怎么办?
  - 恢复
  - 数据状态一致性
    - → 回滚Rollback
    - → 检查点Checkpoint
    - → MessageLog / Message Replay
    - → Restart / Data Reset

Safety保障的软件设计

- 主从模式
  - 多个同样的功能在多个服务上实现
  - 场景
    - 负载大、负载不可预测
  - 设计考虑?
    - Master要简单不能成为瓶颈
    - 负载均衡
    - 负载监控



Safety保障的软件设计

- 实在服务不来怎么办
  - Shed load/Deferrable Work
  - Slow it down
- 主从变体
  - 多个从同时算结果：Voting / NV

Safety保障的软件设计

- Escalation
- Audit
- Quarantine
- Routine
- Let sleeping dogs lie
- …

Safety保障

- 代码风格
- 程序的组织/命名空间
- Safety保障的软件设计
- 测试、排错

好读、好理解、预计了各种可能的环境、人的影响safety因素 → 人无完人

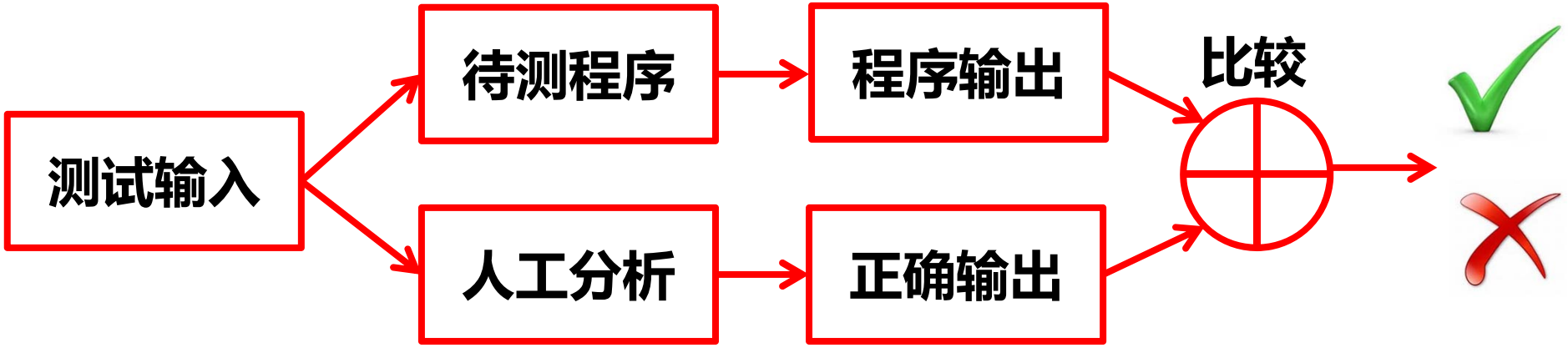
测试

- 怎么发现程序的bug： 以排序为例

人来看：排序结果确实是有序的

启发：人来看的过程

- 给程序一组输入 (test inputs)
- 程序产生一组输出 (test results)
- 判断输出是不是符合预期



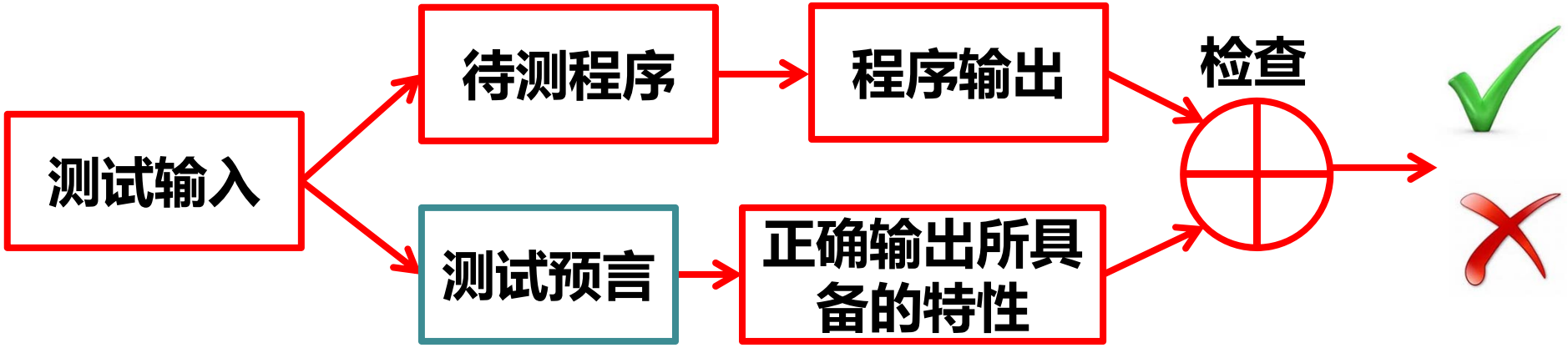
测试

- 怎么发现程序的bug： 以排序为例

机器来看：排序结果确实是有序的

测试预言 (test oracle)

- 给程序一组输入 (test inputs)
- 程序产生一组输出 (test results)
- 判断输出是不是符合预期



测试思考

1. 测试新的排序算法，我们可以利用一个现有的排序算法作为测试预言。用正确的程序来做测试预言，对于一般的软件来说，可行性如何

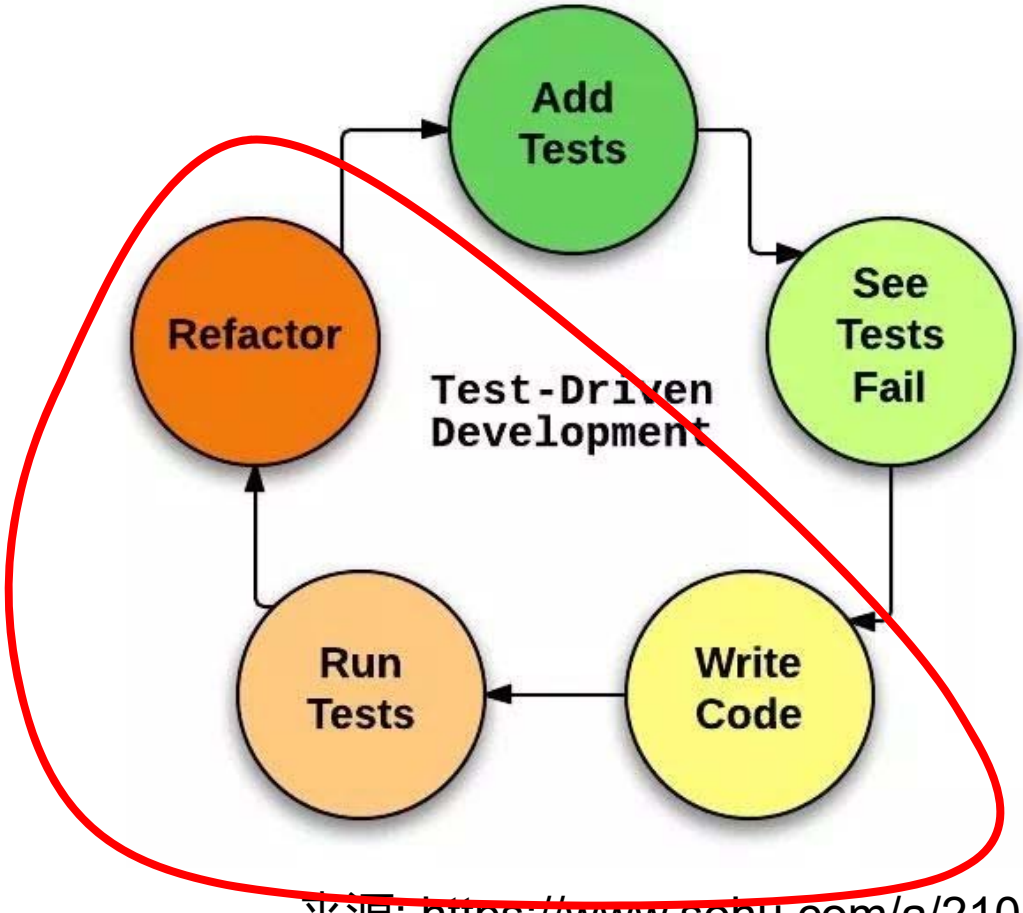
2. 测试能否保证程序100%正确

3. 工业上一般如何评估一个程序测试的充分性

4. 除了输出结果正确，测试还需要关注什么

进阶：测试驱动的开发

- 面向通过测试用例写程序



来源: https://www.sohu.com/a/210366371_505788

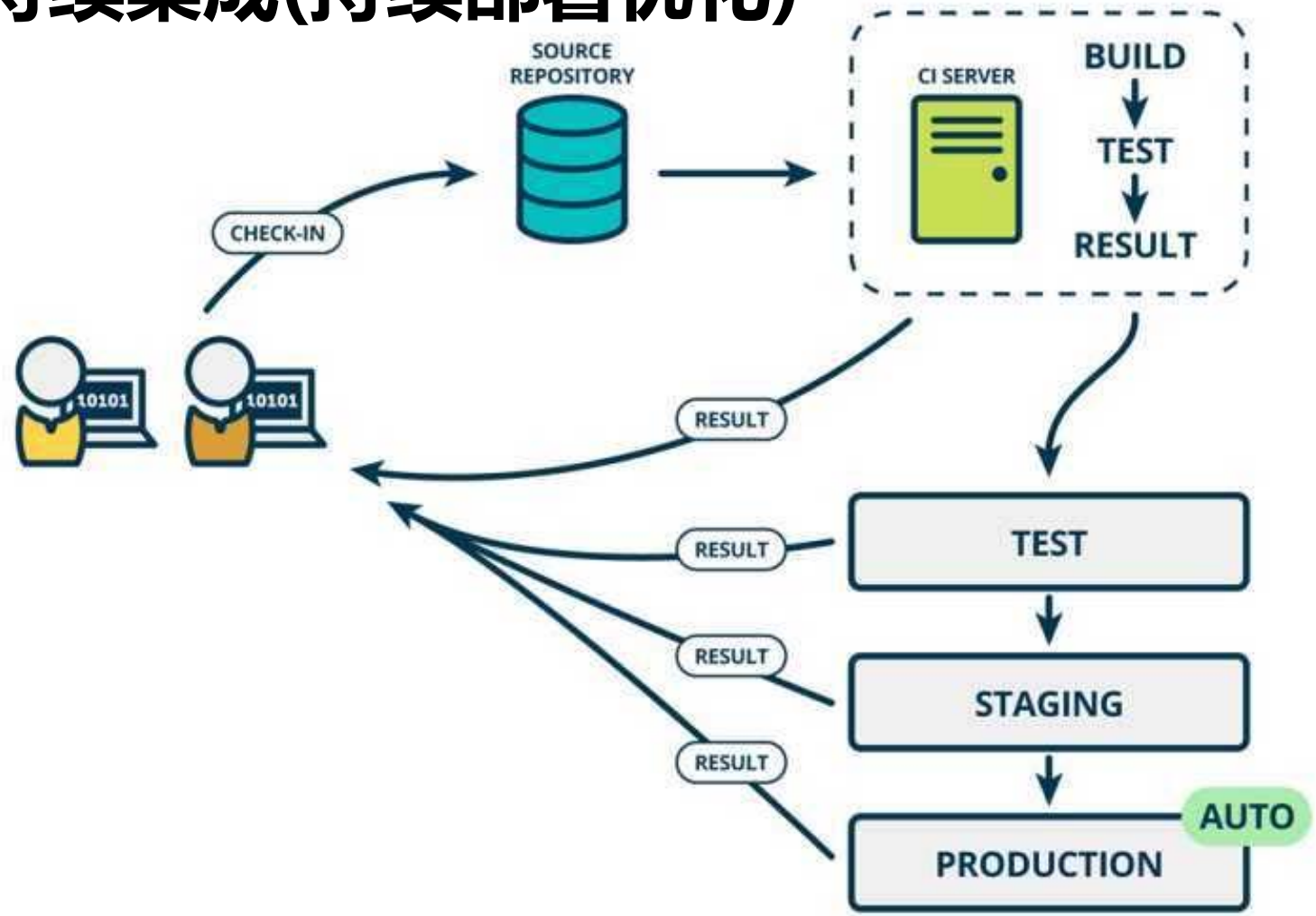
进阶：持续集成(CI)

- 测试自动化 –(系统越来越复杂)

来源: https://zhuanlan.zhihu.com/p/32549586

进阶：持续集成(CI)

- 持续集成(持续部署优化)



来源: https://zhuanlan.zhihu.com/p/32549586

Safety保障

- 代码风格
- 程序的组织/命名空间
- Safety保障的软件设计
- 测试、排错
  - Watch, 单步
  - Reasoning, narrow down
  - 动态分析

复旦大学计算机科学技术学院

编程方法与技术

E.6. safety总结

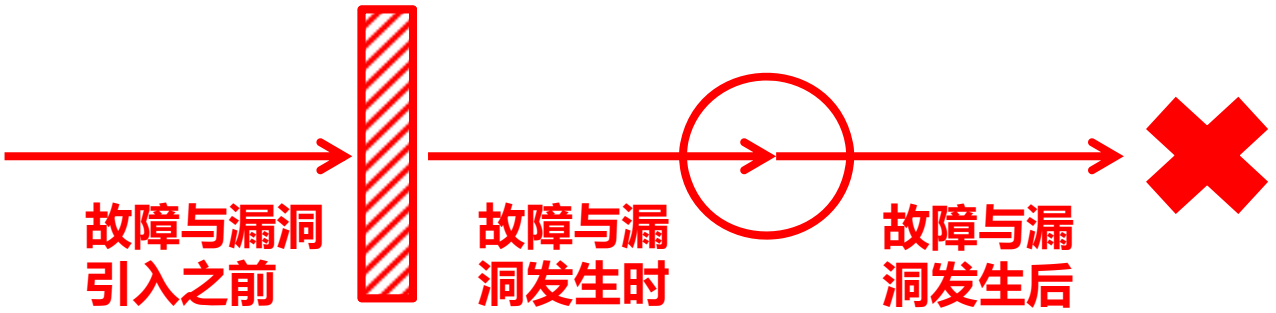
周扬帆

2021-2022第一学期

Safety v.s. Security

- Security
  - Attacks
  - Resilience against attacks
  - 在有坏蛋的情况下，保证系统的“安全”
    - 减少、去除harm
- Safety
  - 关注不要产生坏的后果
  - 可能没有主动的attacks
    - ？
    - 猪头程序员
    - 没想到的运行环境

Safety保障



- 防: 防止软件故障的发生
  - 软件采取的主动(proactive)性的预防策略
- 容: 降低故障对软件/系统的破坏
  - 软件采取的被动(reactive)性的防御策略
- 除: 排除已经发生的故障
  - 定位其所在, 以协助人工干预或采用自动化手段, 更正软件

Safety v.s. Security

- 猪头程序员
  - Java的语言Safety保障
    - 各种限制、抽象 + 没有指针
  - Exception handling
  - 代码风格保障
  - 程序组织保障
- 没想到的运行环境
  - Exception handling
  - 容错pattern
- 人无完人
  - 测试、排错、软件工程

祝各位
考试顺利！！

欢迎对“分析、理解软件和系统”和“编程”感兴趣的人加入我的研究小组