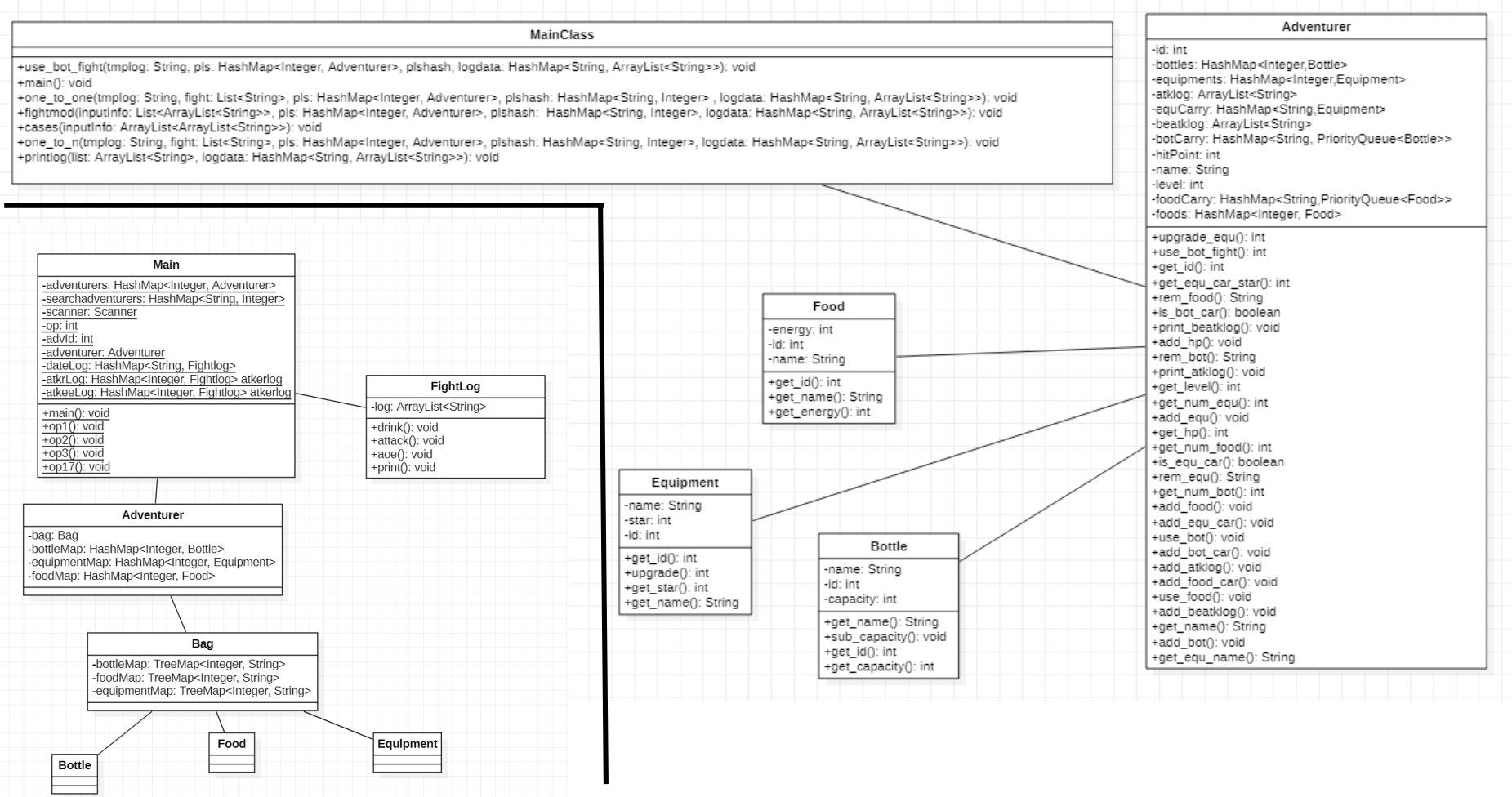
《面向对象程序先导》
Lec5-JAVA程序常见错误分析

北京航空航天大学
吴际
2023.10.13

内容

- 架构设计问题点评
- 常见逻辑性错误
- 黑盒测试
- 调试方法
- 作业内容介绍

两个示例



一些抽样发现的问题：

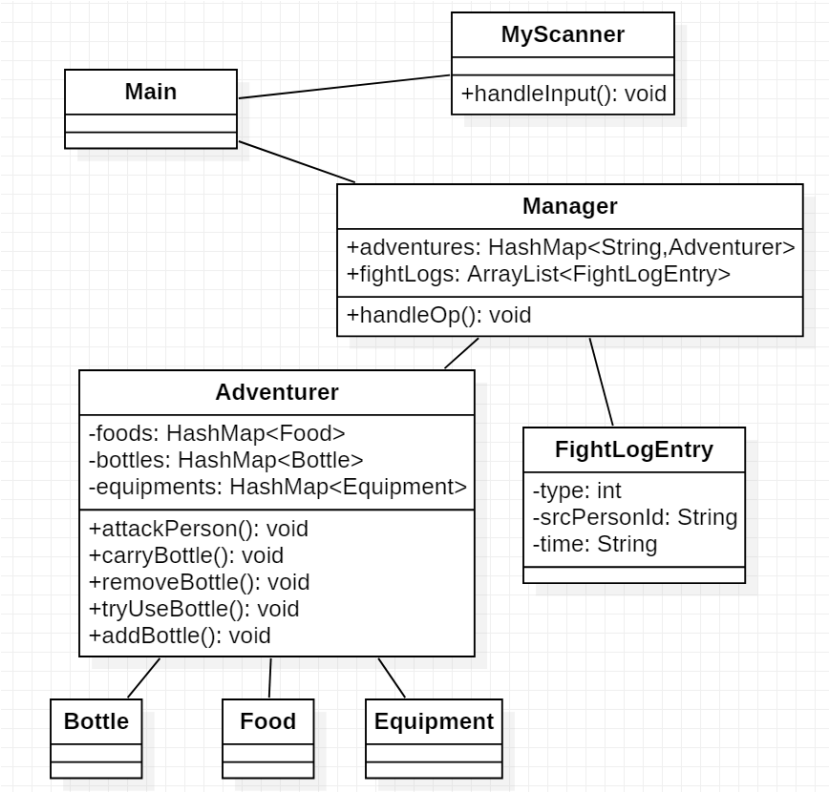
(1). 主要代码逻辑都写在了Main类，类内方法职责杂糅，代码量庞大且臃肿。有同学甚至一个main函数写接近500行

(2). 输入解析相关代码分布零散

(3). 缺少FightLog类，将相关逻辑和别的逻辑掺杂在一块

没有做到各司其职，层次结构不清晰！

推荐的层次化架构



各司其职：该是谁干的事，代码就写到对应的那个类里！

Main：负责代码顶层逻辑，调输入接口+调Manager处理输入（只有短短几行，只负责调用，组织起来各个顶层模块）

MyScanner：输入处理类，集中进行输入处理

Manager：进行指令的具体分类处理，其中维护了冒险者集合和战斗日志集合

FightLogEntry：战斗日志项，存有战斗日志的相关信息，战斗日志内容解析逻辑

Adventurer：其中有各个方法来实现冒险者的各个操作，如：加瓶子，使用瓶子，攻击另一个冒险者等

要追求高内聚，低耦合！各部分逻辑不胡乱掺杂

多种输入指令的处理

- switch case
- 把每种指令的处理封装成一个方法
  - Command1，command2,…
- Myscanner做初步的输入提取和切分
- Manager根据输入指令判断调用command<x>(…)
  - switch (operator){
    - case 1: command1(…);break;
    - case 10: command10(…);break;
    - …
  - }

- 按照指令单独设计类CommandUtil
  - 提供command方法
- Manager类管理m个commandUtil对象
  - ArrayList<CommandUtil>
- 根据输入指令自动获得cmd对象
  - cmdUtil = cmdUtilArray.get(operator -1);
  - cmdUtil.command(…);
- 请同学们思考并预习
  - 能否使用接口或继承来实现这些CommandUtil?

常见的逻辑性错误

- 常见的数据流错误主要包括：
  - 引用错误
  - 运算错误
- 常见的控制流错误主要包括：
  - 差一错误
  - 意外的分支
- 常见的综合性错误主要包括：
  - 输入输出错误
  - 拷贝错误
  - String类相关错误

引用错误·空指针异常

- 如果一个对象为 null， 访问其成员就会产生空指针异常

```
public class NullPointerExample {
    public static void main(String[] args) {
            String str = null;
            int len = str.length(); // 这里会抛出空指针异常
        }
}
```

```
java.lang.NullPointerException
    org.apache.catalina.connector.CoyoteWriter.write(CoyoteWriter.java:180)
    com.kaikeba.mvc.DispatcherServlet.service(DispatcherServlet.java:42)
    javax.servlet.http.HttpServlet.service(HttpServlet.java:741)
    org.apache.tomcat.websocket.server.WsFilter.doFilter(WsFilter.java:53)
    com.e.filter.CharSetFilter.doFilter(CharSetFilter.java:16)
```

引用错误·索引越界异常

- 访问数组或容器中的元素时，索引超出正常索引范围会产生索引越界异常

```java
public static void main(String[] args) {

        List<Object> list = new ArrayList();

        list.add("添加的第一个元素python");
        list.add("添加的第二个元素java");
        list.add("添加的第三个元素Javascript");
        list.add("添加的第四个元素C++");

        System.out.println(list.size());    //打印结果为：4

        for (int i = 0;i <= list.size();i++) {
            System.out.println(list.get((i)));
        }
}
```

```
Exception in thread "main" java.lang.IndexOutOfBoundsException: Index: 4, Size: 4
```

```java
String name = "";
for (Message message : lys) {
    name = us.getAllUser(sql,message.getUserId())
                                        .get(0).getName();
    names.add(name);
}
```

```java
String name = "";
for (Massage massage : lys) {
    List<User> userList = us.getAllUser(sql, massage.getUserId());
    if(userList!=null&& userList.size()>0) {
        name = userList.get(0).getName();
        names.add(name);
    }
```

1.接收获取到的List
2.判断List是否为空，且长度是否大于0
3.重新运行代码，程序正常

引用错误·类型转换错误

- 引用数据类型在转换时，如果两个引用的类型不匹配，就会产生类型转换错误
  - java.lang.ClassCastException

```
public class ClassCastExceptionExample {
    public static void main(String[] args) {
        Shape shape = new Circle();
        Rectangle rectangle = (Rectangle) shape; // 类型转换错误
    }
}
```

- 类型匹配
  - 类型相同
  - 类型具有父类与子类关系

```
public boolean equals (Object o){
    if (o==null)return false;
    Point other = (Point)o;
    return (this.x==other.x) && (this.y == other.y);
}

public boolean equals (Object o){
   if (o==null)return false;
   if(o instanceof Point){
      Point other = (Point)o;
      return (this.x==other.x) && (this.y == other.y);
   }
   else return false;
}
```

运算错误·逻辑运算短路求值

```
String test = null;
// ...
if (test == null || test.length() > 5){
    //...
}
```

```
String test = null;
// ...
if (test.length() > 5 || test == null){
    //...
}
```

哪个会触发java.lang.NullPointerException?

Java 中，逻辑运算符 || 以及 && 存在短路行为

运算错误·隐式强转与类型提升

```
public static void main(String[] args) {
    byte b = 10;
    b += 1020;
    System.out.println(b);
}
```

```
public static void main(String[] args) {
    byte b = 10;
    int a = 1020;
    System.out.println(b+a);
}
```

- 在赋值语句中以及函数返回语句中，可能发生隐式强制转换
  - 隐式强转可能导致溢出或精度丢失
- Java 中范围较小的数字类型转换到范围较大的数字类型时触发类型转换
  - 类型提升
  - 由算数运算符触发

boolean	8bit/1byte
byte	8bit/1byte
char	16bit/2byte
short	16bit/2byte
float	32bit/4byte
int	32bit/4byte
long	64bit/8byte
double	64bit/8byte

控制流错误·差一错误

- 差一错误（off-by-one）是一种常见的逻辑错误，指某一数值（或某一过程的循环执行次数）错误地与其期望值差了1
- 差一错误会导致许多问题，例如索引越界、死循环等

```
int[] a,i;
a = new int[8];
for (i == 0; i<=8;i++){
    //...
    a[i] = 0;
}
```

控制流错误·差一错误

- 有如下数组char[]：

```
public class OffByOneExample {
    public static void main(String[] args) {
        char[] string = new char[10];
    }
}
```

- 使用整型变量 iterator 去遍历：

```
// ...
char[] string = new char[10];
int iterator = 0;
// 如何安全遍历string?
```

- 为了顺利遍历string，必须做如下约定：
  - 必须确保能访问到第0个元素
  - 确保iterator 小于数组长度，否则触发越界异常
  - 当数组 string 遍历完成后，应及时跳出循环
- 如不遵守上述约定，所引发的错误常称为差一错误
- Java 类库提供的迭代器也采用了类似的约定

控制流错误·差一错误的防范

- 只需对下述三种状态进行检测：
  - 零状态：迭代还未发生时，检查程序状态是否符合预期
  - 一状态：进行了一次迭代，检查迭代结果以及迭代后的程序状态
  - 终止状态：反复迭代，观察迭代能否在预期次数时终止
- 对于遍历数组 string 的例子，只需进行如下测试：
  - 零状态：无需检查(或者检查string != null && string.length > 0)
  - 一状态：进行一次迭代，观察是否得到了第0个元素
  - 终止状态：迭代10次，观察第10次迭代是否正常得到最后一个元素，再进行一次迭代，观察有无引发空指针异常，观察迭代是否已经终止
- 类似地，如果我们遇到“差 k 错误”，也可以采用这三个步骤来测试程序

综合错误·输入输出错误

- 在进行输入输出操作时，我们需要保证目标输入输出流是打开状态
- Scanner 可能被多个对象共享访问，任何一个对象对于 Scanner 的操作都会改变其状态
  - 推荐编写统一的输入类，保证 Scanner 内部状态正确
- Scanner.nextLine() 方法常被用来吸收换行符，并配合 next 等方法一同使用
  - 如果忘记调用 nextLine ，则会引发错误

```
public class NextLineExample {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("请输入你的年龄：");
        int age = sc.nextInt();
        System.out.println("请输入你的姓名：");
        String name = sc.next();
        System.out.println("请输入你的工资：");
        String salary = sc.nextLine();
        System.out.println("你的信息如下：");
        System.out.println("姓名："+name+"\n"+"年龄："+age+"\n"+"工资："+salary);
    }
}
```

综合错误·输入输出错误·例

- next类方法
  - next, nextInt, nextDouble, nextFloat
  - 扫描System.in直到获得了有效输入，并把有效输入之前的其他符号过滤掉（如空格和回车等）
  - 无法获得带有空格的输入
- nextLine方法
  - 获得回车之前的整行输入（吸收回车）
- Bug原因：输入姓名后按下的回车不会被next吸收，nextLine 在吸收后直接返回
- 解决办法
  - next后添加一个 nextLine 方法
  - 或者都使用next类方法

综合错误·输入输出错误·例

- 正确代码如下：

```
public class NextLineExample {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("请输入你的年龄：");
        int age = sc.nextInt();
        System.out.println("请输入你的姓名：");
        String name = sc.next();
        System.out.println("请输入你的工资：");
        int salary = sc.nextInt();
        System.out.println("你的信息如下：");
        System.out.println("姓名："+name+"\n"+"年龄："+age+"\n"+"工资："+salary);
    }
}
```

```
Exception in thread "main" java.util.InputMismatchException
at java.util.Scanner.throwFor(Unknown Source)
at java.util.Scanner.next(Unknown Source)
at java.util.Scanner.nextInt(Unknown Source)
```

直接赋值、浅拷贝、深拷贝

- Java程序中除基础类型变量外，赋值和传参都是传递一个引用值（地址值），不会复制对象
  - 产生对象共享访问
  - 一个对象可能被其他对象修改，导致状态变化
- 如果这种对象状态变化是不受控制的，则需要限制对象共享
  - 使用一个对象的拷贝来进行传递处理
  - Object类内置有clone 方法
- 浅拷贝：对于对象中的引用类型属性，复制其引用值（地址值）
  - 默认的clone方法是浅拷贝
- 深拷贝：对于对象中的引用类型属性，复制其内容（层次迭代）

由于浅拷贝导致“某一对象被意外共享”所引发的错误称为拷贝错误

综合错误·拷贝错误·例

```
class Address {
    private String city;
    public Address(String city) {
        this.city = city;
    }
    public void setCity(String city) {
        this.city = city;
    }
}
```

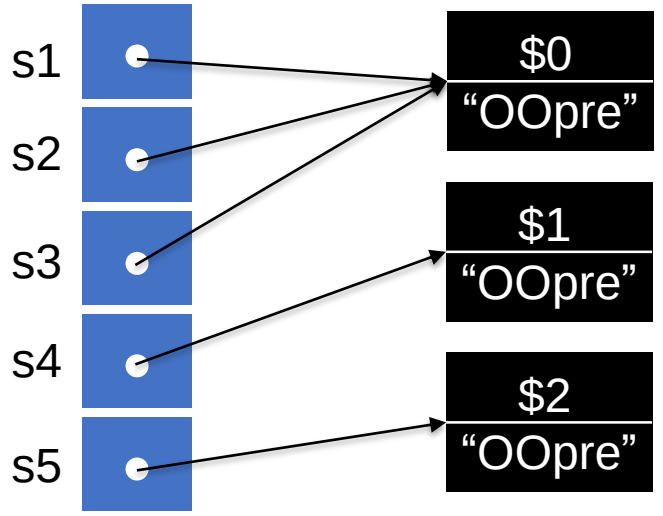
```
class Factory implements Cloneable {
    private String name;
    private Address address;
    public Factory(String name, Address address) {
        this.name = name;
        this.address = address;
    }
    public Address getAddress() {
        return address;
    }
    @Override
    public Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}
```

综合错误·拷贝错误·例

```
public class Test {
    public static void main(String[] args) throws CloneNotSupportedException {
        Address addr = new Address("CityA");
        Factory fact1 = new Factory("SunFact", addr);
        Factory clonedFact = (Factory) fact1.clone();
        // 修改克隆的对象的属性，原对象的属性也会改变
       clonedFact.getAddress().setCity("CityB");
    }
}
```

综合错误·String类的使用不当

- String是普遍使用的一个类
  - String s1 = “OOpre"; // 直接创建
  - String s2 = “OOpre"; // 直接创建
  - String s3 = s1; // 引用赋值
  - String s4 = new String(“OOpre"); // String 对象创建
  - String s5 = new String(“OOpre"); // String 对象创建
- 这五个对象引用是否存在共享?
- String对象的相等比较
  - equals：比较的是内容
  - ==：比较的是引用地址
- String对象是不可变对象
  - 对String对象的任意改变都会返回一个新的String类型对象
  - 如：String 的 replace 方法




```
System.out.println(s1.equals(s2));
System.out.println(s1==s2);
System.out.println(s1.equals(s3));
System.out.println(s1==s3);
System.out.println(s1.equals(s4));
System.out.println(s1==s4);
```

综合错误·错误扩散与捕捉处理

- 程序运行时出现错误或异常后，可能会通过方法调用和对象共享传播，导致问题变得更加复杂
- 需要程序主动来捕捉错误或异常，及时进行处理
  - 否则，成为未捕获异常

```
class ExceptionExample {
    void m() {
        int data = 50 / 0;
    }
    void n() {
        m();
    }
    void p() {
        n();
    }
}
```

```
public class Test {
    public static void main(String args[]) {
        ExceptionExample obj = new ExceptionExample();
        try {
            obj.p();
        } catch (Exception e) {
            System.out.println("exception handled");
            e.printStackTrace();
        }
    }
}
```

综合错误·错误扩散与捕捉处理

```
class CallChainExample {
    int intNum() {
        return new Random().nextInt();
    }
    long seed() {
        return (intNum()+1) * System.currentTimeMillis();
    }
    void func(int data) {
        System.out.println(data / seed());
    }
}
```

CallChainExample类的三个方法可能会触发什么异常？如何处理？

- 如果某个方法运行时触发了异常，而其调用方法没有捕捉和进行处理，那么错误会继续传播到更高层次的调用方法中，直至main
  - try {} catch(Exception e) {}
- 如果一个对象的属性取值发生了错误，所有访问该对象的方法都可能受到影响，如果不进行捕捉，该错误会传播到其他对象
  - 同学们能否自己举一个例子?
  - 如何捕捉?

黑盒测试

- 黑盒测试是不关注程序内部实现结构的测试方法
- 黑盒测试的重点
  - 覆盖功能
  - 构造数据
- 测试是保证程序质量必不可少的一环
  - 发现执行失败的功能或场景
- 黑盒测试的设计依据
  - 软件需求（作业指导书）
  - 功能及其关系
  - 功能的输入和输出
    - 范围要求和可能抛出的异常等

黑盒测试·等价类划分

- 按照程序的功能逻辑，把功能的输入划分为若干等价类，在每个等价类中只需使用若干代表性数据进行测试
- 等价类：有效等价类、无效等价类
- 例如，三角形判定程序，输入为三条边的长度（整数）
- 同学们在自行测试自己编写的程序时要注意利用等价类划分来提高测试效率和效果

		有效等价类型	号码	无效等价类	号码
输入条件	输入三个整数	整数	1	一边为非整数：a为非整数	12
				一边为非整数：b为非整数	13
				一边为非整数：c为非整数	14
				两边为非整数：a,b为非整数	15
				两边为非整数：b,c为非整数	16
				两边为非整数：a,c为非整数	17
				三边a，b，c均为非整数	18
		三个数	2	只给一边：只给a	19
				只给一边：只给b	20
				只给一边：只给c	21
				只给两边：只给ab	22
				只给两边：只给b,c	23
				只给两边：只给ac	24
				给出三个以上	25
		非零数	3	一边为零：a为0	26
				一边为零：b为0	27
				一边为零：c为0	28
				二边为零：a,b为0	29
				二边为零：b,c为0	30
				二边为零：a,c为0	31
				三边a,b,c均为0	32
		正数	4	一边<0：a<0	33
				一边<0：b<0	34
				一边<0：c<0	35
				二边<0：a<0且b<0	36
				二边<0：a<0且c<0	37
				二边<0：b<0且c<0	38
				三边均<0：a<0且b<0且c<0	39
输出条件	构成一般三角形	a+b>c	5	a+b<0	40
				a+b=0	41
		b+c>a	6	b+c<a	42
				b+c=a	43
		a+c>b	7	a+c<b	44
				a+c=b	45
	构成等腰三角形	a=b且b=c且a=c且两边之和大于第三边	8 9 10		
	构成等腰三角形	a=b=c	11		

调试

- 测试能够发现程序问题，但不能确认bug的确切位置
- 给定发现问题的输入数据，通过调试来在代码中找到 bug
  - 一个bug可能分散在多处代码位置
- 测试本质上是试错的过程
  - 即通过等价类划分等手段找到引发错误的输入数据
- 调试本质上是一个逻辑推理过程
  - 基于程序中的控制流和数据流来**追踪**和**推理**错误的根源

调试·数据化简

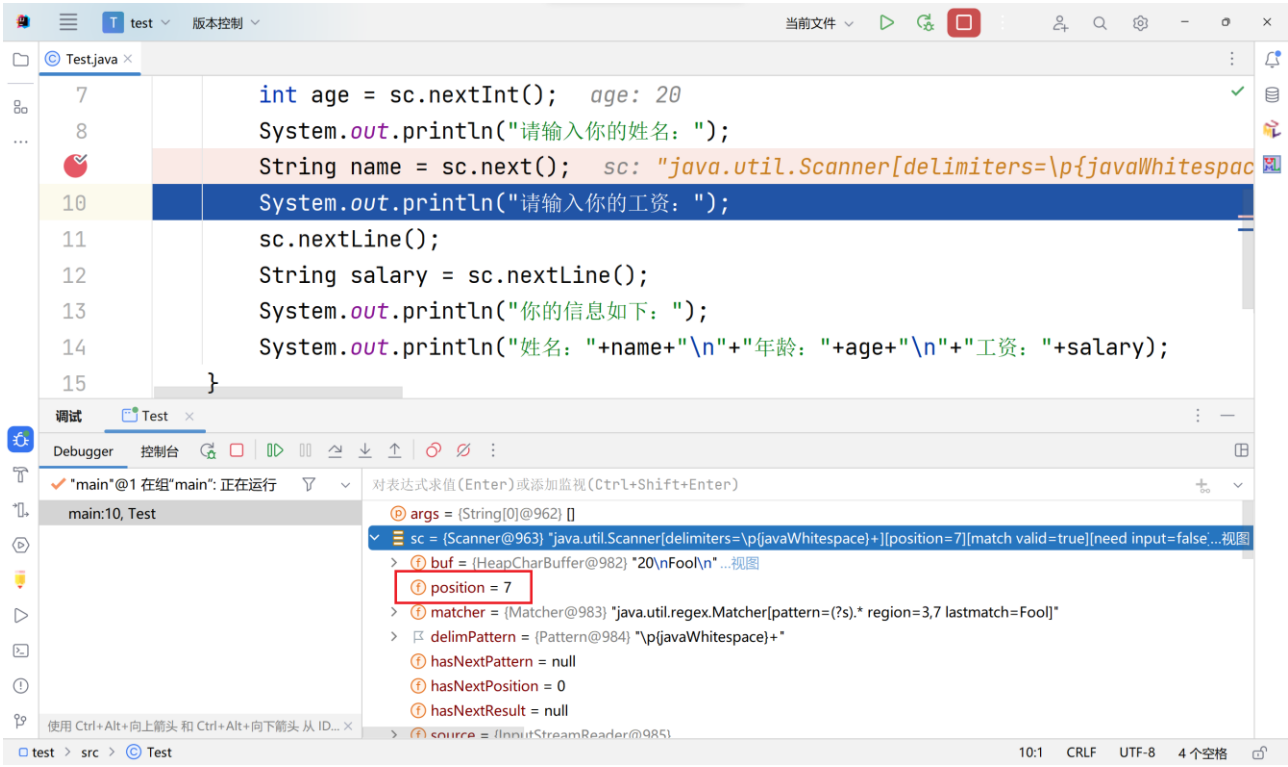
- 数据化简：发现问题的输入数据往往比较“臃肿”，难以直接调试，需要进行化简并确保仍能重现错误
  - 一个后台服务运行了一周，爆出了错误
  - 一周内产生了若干TB的访问日志
- 数据化简是一个十分重要、极具挑战的工作
  - 极大节约调试时间和复杂度
  - 基本做法1（经验分析）：分析输入数据中违背约束或不常见特征
  - 基本做法2（对比分析）：把成功运行数据与失败运行数据进行对比

调试·断点设置

- 调试需要细致观察和追踪程序运行路径和变量的取值
  - 断点（breakpoint）
- 断点分类
  - 行断点：程序执行到相应的代码行就会暂停执行
  - 条件断点：程序执行时如果设定的条件满足就会暂停执行
- 断点设置反应了调试人员的bug位置推理
  - 需要动态调整
  - 经验原则1：如果当前位置发现变量取值已经出错，那就在调用该方法的上一层调用处之前设置断点
  - 经验原则2：如果涉及复杂的处理逻辑（循环+分支处理），最好设置条件断点

调试·监视内存信息

- 在断点处要细致观察程序执行状态
  - 调用栈及参数传递
  - 当前对象的属性取值
  - 当前方法的局部变量取值
  - 上一层方法对应的对象和局部变量取值等
  - 更上一层...
- 变量如果是一个对象
  - 可以按照管理层次关系逐层查看对象内容
- IDEA 或其他IDE工具内置的调试器都提供了相应的信息查看功能



调试·插入打印语句调试

- 断点是个灵活的调试机制，但需要手工来查看和分析
  - 有可能会对程序执行造成影响
- 可以使用打印输出来获得程序执行的中间过程信息
  - 调用层次
  - 变量取值，特别是分支判定相关的变量取值
- 进一步可以编写程序来分析打印出的内容，缩小bug的可疑范围

作业内容介绍

- 修改有错误的程序并提交
  - 不设置强测
  - 不设置junit任务
  - 每个人拿到的bug会有差异
- 可能的BUG类别
  - 浅克隆/深克隆的错误使用
  - 迭代访问
  - 字符串处理
- 定位BUG并修正
  - 阅读指导书所介绍的功能和代码，编写测试数据发现问题
  - 调试修复问题