# Object 类 : (重点)

**13366285946 Jack\_xie 微信号**

**继承 :** Object 类是所有类的祖宗类.

如果一个类没有显示继承类, 则该类默认继承自 Object 类.

继承的好处 : 子类继承父类, 那么子类就 `无条件` 拥有了父类中定义的所有数据(属性) 和行为(方法).

**说明 :** 构造方法是不可以被继承的, 因为构造方法就是用来创建 `当前` 类的对象.

**多态的基础是继承**, 如果一个对象使用多态的语法接收, 那么之后调用具体子类属性和方法时, 需要进行类型转换, 为了避免发生 ClassCastException 异常,

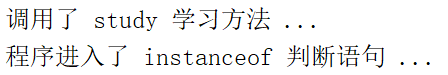
所以在转换之前会使用 **instanceof** 关键字进行对象具体类型判断.



## getClass() 方法 :

**说明 :** getClass() 完成的是单一具体类型判断.

public class Test1 {  
 public static void main(String[] args) {  
  
 // 1. 创建一个 Student 对象  
 // 接收方式一 : 使用子类的具体类型接收.  
 // Student stu = new Student();  
  
 // 多态 :  
 // 含义一 : 父类引用指向了子类对象  
 // 含义二 : 接口引用指向了实现类对象.  
 // Person p = new Teacher();  
 // Person p = new Student();  
 Person p = new GoodStudent();  
  
 // 编译 : 编译阶段程序是通过对象的左边真正接收的引用类型来查找对应的属性和方法.  
 // 多态的弊端 : instanceof 对象类型判断解决该问题  
 // 说明 : instanceof 判断类型包含父类类型. (继承关系)  
 if (p instanceof Student) {  
 Student stu = (Student) p; // ClassCastException 类转换异常.  
 stu.study();  
 System.out.println("程序进入了 instanceof 判断语句 ...");  
 }  
  
 // getClass() 方法  
 // 方式一 : 对象.getClass(); 可以获取到该对象的具体类型  
 // 方式二 : 类名.class 通过类名也可以获取到该类型对应的 Class 对象  
 // 说明 : getClass() 判断类型仅能判断单一的那个具体类型. (不包含继承关系)  
 if (p.getClass() == Student.class) {  
 Student stu = (Student) p;  
 System.out.println("程序进入了 getClass() 判断语句 ...");  
 }  
 }  
}

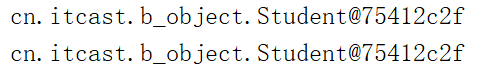


## toString() 方法 :

**需求 :** 定义一个 Student 类. 输出 Student 类型的对象, 希望查看该对象的数据信息. (属性信息)

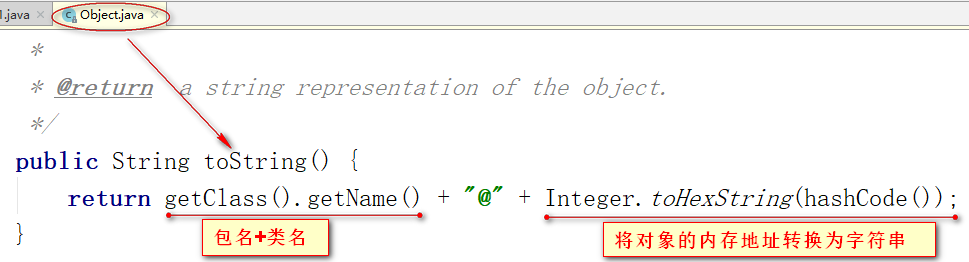
**说明 :** 在Java中, 如果直接输出一个对象名称, 其Java底层是调用该对象的 toString() 方法.

public class Test1 {  
 public static void main(String[] args) {  
  
 // stu1 是对象类型. (引用类型) reference / pointer 地址 (NullPointerException 空指针异常)  
 /\*  
 Student stu = null;  
 stu.getName();  
 \*/  
 Student stu1 = new Student("张三", 18);  
  
 // 请问 : 如果在程序中, 直接输出对象, 结果是 `类型@地址`. cn.itcast.b\_object.Student@75412c2f  
 // 需求 : 希望输出对象, 能够查看该对象的所有 `数据 / 属性`  
 System.out.println(stu1); // Student{name = '张三', age = 18}  
 System.out.println(stu1.toString());  
 }  
}



**思考 :** 对象为 Student 类型的对象, 调用 toString() 方法, 为什么调用到了 Object 类中 ???

**stu1.toString(); 自定义对象的数据输出.**



**原因 :** 由于 Student 类没有重写从 Object 类中继承而来的 toString() 方法, 因为调用该方法时, 在 Student 类中查找不到, 程序就会继续去 Student 类的父类中进行查询. 一旦找到, 就执行该父类的行为.

**需求 :** 希望执行 Student 对象的 toString() 方法, 输出**自定义数据格式.**

**解决 :** Student 类重写 Object 类中继承而来的 toString() 方法.

**重写 :** 子类重写父类继承而来的方法, 此时, 程序执行时, 就会执行子类重写的该方法, 而不会去执行父类中的该方法. 如果子类想要执行父类中继承而来的方法, 请使用 super.方法名(); 执行.

*// 重写 Object 类继承而来的 toString() 方法.*@Override  
public String toString() {  
 return "Student{name = '"+name+"', age = "+age+"}";  
}

*// 重写 Object 类继承而来的 toString() 方法.*@Override  
public String toString() {  
 return "Student{name = '" + this.name + "', age = " + this.age + "}";  
}

**请问 :** this 是谁 ??? 调用该方法的 `当前对象`.

*// alt + insert -> toString() 方法*@Override  
public String toString() {  
 return "Student{" +  
 "name='" + name + '\'' +  
 ", age=" + age +  
 '}';  
}

**总结 :** toString() 是 Object 定义的所有对象输出规则. 每一个类都应该重写该方法, 实现自定义类的数据输出.

## 1.3 equals() 方法 :

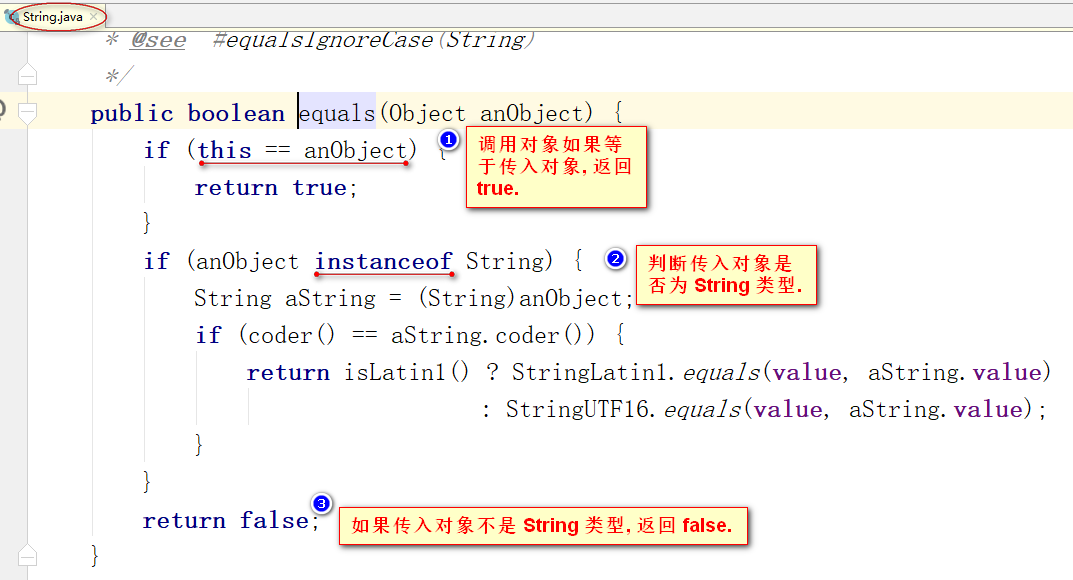
**需求 :** 比较两个 Student 类型的对象是否相等. (如果两个学生对象的姓名和年龄都相等, 那么这两个对象我们认为就相等.)

public class Test1 {  
 public static void main(String[] args) {  
  
 *// 1. 创建两个 Student 对象  
 // new 一个对象, 在内存中就会产生一块新的空间.* Student stu1 = new Student("张三", 18);  
 Student stu2 = new Student("张三", 18);  
 *// Student stu2 = new Student("李四", 20);  
  
 // 需求 : 判断两个 Student 对象是否相等 ??? 相等的含义 : (如果 `姓名和年龄` 都相同, 就相等.)  
  
 // 方式一 : == 符号判断 (比较的两个对象的内存地址)* System.*out*.println(" == 符号 : " + (stu1 == stu2)); // false  
  
 *// 方式二 : 调用 Object 类提供的 equals 方法. 判断两个对象是否相等.* System.*out*.println("equals 方法 : " + stu1.equals(stu2)); // false  
 }  
}

**String 类的 equals 方法 :**

public class Test3 {  
 public static void main(String[] args) {  
  
 *// String 类的 equals 方法.  
 // 含义 : 判断两个字符串的内容是否相等.* String str1 = "hello"; *// 字面值, 常量.* String str2 = new String("hello");  
  
 System.*out*.println(str1 == str2); *// false 地址判断* System.*out*.println(str1.equals(str2)); *// true* }  
}

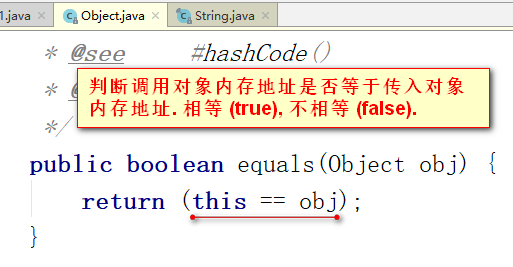
说明 : String 和 Student 两个类调用的 equals 方法, 都继承自 Object 类. 那为什么两个类的执行效果是不一样的 ???



String 类重写了 Object 类的 equals 方法, 实现了字符串数据内容判断.

String 类一个 final 最终类, 该类不能被继承. 因为 instanceof 和 getClass() 的使用在该类中就不存在本质的区别.

**区分 :** 如果一个类本身是可以被继承的, 不是最终类, 那么建议在判断类型时, 尽量使用 getClass() 方法实现判断.



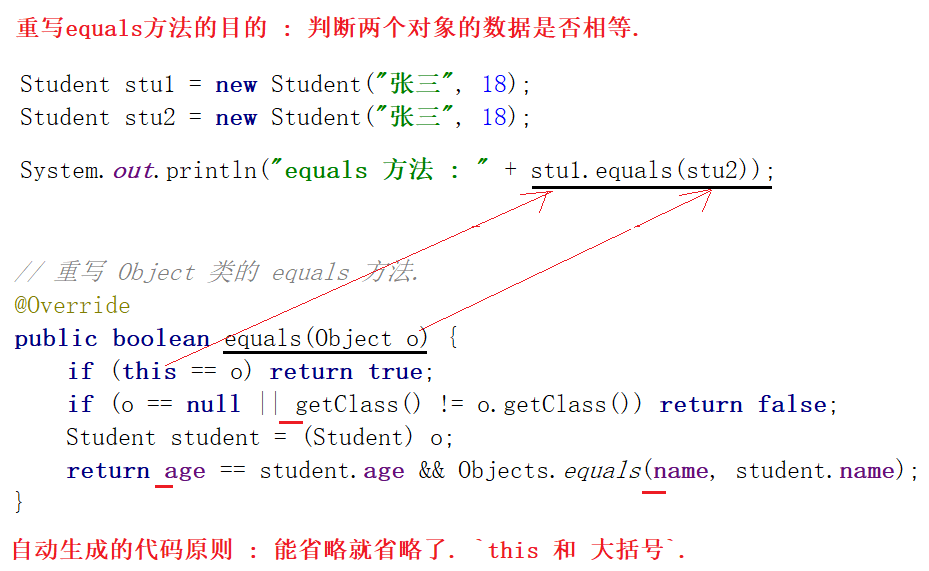
Student 类没有重写 Object 类的 equals 方法, 所以程序调用了 Object 父类的该方法, Object类的 equals 方法仅仅是比较两个对象的内存地址是否相等.

**需求 :** 希望两个对象数据相等, 就返回 true, 如果数据不相等, 就返回 false. 就类似于 String 类实现的效率.

**解决 :** Student 类重写 Object 类的 equals 方法. (因为子类重写父类的方法, 程序执行时就会执行子类重写的该方法.)

**重写版本一 :**

// 重写 Object 类的 equals 方法.  
@Override  
public boolean equals(Object obj) {  
 // 1. 先判断两个对象的内存地址是否相等.  
 if (this == obj) {  
 return true;  
 }  
 // 2. 判断传入参数的具体类型  
 // 条件一 : 传入参数 obj 对象不能为 null.  
 // 条件二 : 传入参数 obj 对象必须与 this (Student) 类型一模一样.  
 if (obj == null || obj.getClass() != this.getClass()) {  
 return false;  
 }  
 // 3. 类型强转, 并判断两个对象的具体数据  
 Student stu = (Student) obj; // 强转目的是调用子类的特有属性和行为  
 // 含义 : 两个对象的年龄要相等, 并且, 姓名也要相等  
 // 基本数据类型, 只有值判断, 没有方法判断. 只有引用类型才拥有方法判断.  
 if (this.age == stu.age && this.name.equals(stu.name)) {  
 return true;  
 }  
 // 4. 如果两个对象的年龄或相等, 只要有不同, 就返回 false.  
 return false;  
}



stu1 = cn.itcast.c\_object.Student@58ceff1

stu2 = cn.itcast.c\_object.Student@7c30a502

== 符号 : false

this = cn.itcast.c\_object.Student@58ceff1

obj = cn.itcast.c\_object.Student@7c30a502

equals 方法 : true

**小心 : 判断语句的 `大括号` 省略.**

public class Test4 {  
 public static void main(String[] args) {  
  
 // flag 表示标记  
 boolean flag = false;  
  
 // 判断  
 // 如果判断失去了 `大括号`, 只有第一条语句与该判断相关.  
   
 // 说明 : 如果判断仅有一条执行结果, 才允许省略 `大括号`.  
 if (flag == true)  
 System.out.println("A");

System.out.println("B");  
 System.out.println("C");  
 System.out.println("D");  
 System.out.println("E");  
 }  
}

**重写版本二 :**

// 重写 Object 类的 equals 方法.  
@Override  
public boolean equals(Object obj) {  
 // 1. 先判断两个对象的内存地址是否相等.  
 if (this == obj) return true;  
   
 // 2. 判断传入参数的具体类型  
 // 条件一 : 传入参数 obj 对象不能为 null.  
 // 条件二 : 传入参数 obj 对象必须与 this (Student) 类型一模一样.  
 if (obj == null || obj.getClass() != this.getClass()) return false;  
   
 // 3. 类型强转, 并判断两个对象的具体数据  
 Student stu = (Student) obj;   
 return this.age == stu.age && this.name.equals(stu.name);  
}

**重写版本三 :**

// 重写 Object 类的 equals 方法.  
@Override  
public boolean equals(Object obj) {  
 // 1. 先判断两个对象的内存地址是否相等.  
 if (this == obj) return true;  
  
 // 2. 判断传入参数的具体类型  
 // 条件一 : 传入参数 obj 对象不能为 null.  
 // 条件二 : 传入参数 obj 对象必须与 this (Student) 类型一模一样.  
 if (obj == null || obj.getClass() != this.getClass()) return false;  
  
 // 3. 类型强转, 并判断两个对象的具体数据  
 Student stu = (Student) obj;  
 return this.age == stu.age && Objects.equals(this.name, stu.name);  
}

**Objects 工具类的底层方法解释说明 :**

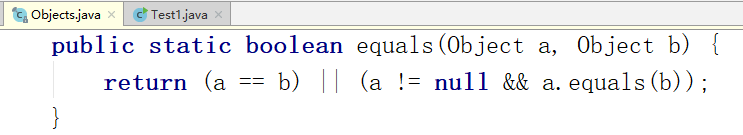
// 重写 Object 类的 equals 方法.  
@Override  
public boolean equals(Object obj) {  
 // 1. 先判断两个对象的内存地址是否相等.  
 if (this == obj) return true;  
  
 // 2. 判断传入参数的具体类型  
 // 条件一 : 传入参数 obj 对象不能为 null.  
 // 条件二 : 传入参数 obj 对象必须与 this (Student) 类型一模一样.  
 if (obj == null || obj.getClass() != this.getClass()) return false;  
  
 // 3. 类型强转, 并判断两个对象的具体数据  
 Student stu = (Student) obj;  
 // return this.age == stu.age && Objects.equals(this.name, stu.name);  
 // Objects 工具类的 equals 方法判断, 更加的谨慎.  
 return this.age == stu.age && ((this.name == stu.name) || (this.name != null && this.name.equals(stu.name)));   
 // (a == b) || (a != null && a.equals(b)); 底层实现.  
}

**总结 :** 重写 Object 类的 equals 方法, 两个对象最终比较的是什么 ??? 就是比较两个对象的所有数据.

**IDEA 最终自动生成的 `模板` :**

*// 重写 Object 类的 equals 方法.*@Override  
public boolean equals(Object o) {  
 if (this == o) return true; // 调用对象的内存地址 == 传入参数对象的内存地址  
 if (o == null || getClass() != o.getClass()) return false; // 传入参数对象 == null 或者 调用对象的类型 != 传入参数对象的类型  
 Student student = (Student) o; // 将传入参数对象强转为 Student 当前对象类型  
 return age == student.age && Objects.*equals*(name, student.name); // 当前对象的age == 强转对象的age 并且 两个对象的name必须一致.  
}

## 1.4 Objects 工具类 :

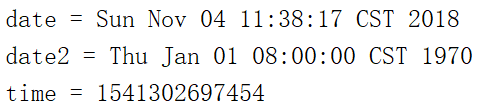


# Date, DateFormat, Calendar 日期类 : (了解)

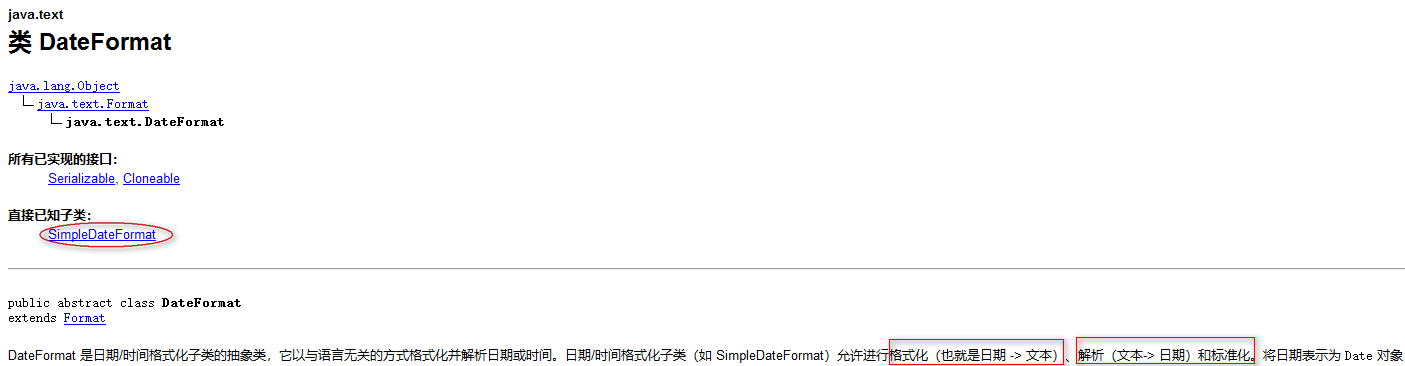
## 2.1 Date 日期类 :

**中国标准时间：China Standard Time UT+8:00**

public class Test1 {  
 public static void main(String[] args) {  
  
 *// 1. 创建一个表示当前的时期时间对象* Date date = new Date();  
 *// date = Sun Nov 04 10:50:02 CST 2018* System.*out*.println("date = " + date);  
  
 *// 2. Date(long date)* Date date2 = new Date(0L);  
 *// date2 = Thu Jan 01 08:00:00 CST 1970 (标准时间)* System.*out*.println("date2 = " + date2);  
  
 *// 3. 毫秒数 (从标准时间从现在为止, 共经过了多少毫秒) 1秒 == 1000毫秒* long time = date.getTime();  
 *// time = 1541300008821* System.*out*.println("time = " + time);  
 }  
}



## 2.2 DateFormat 日期格式化 :





**需求 :** 希望输出的日期时间格式自定义. 2018年11月4日 10:55:55

功能1 : 格式化, 将日期对象转换为字符串. String format(Date date)

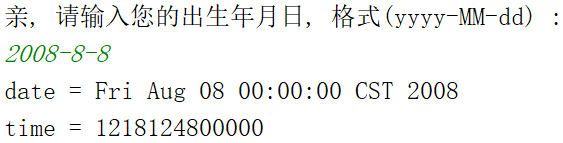
功能2 : 解析, 将字符串转换为日期对象. Date parse(String source);

public class Test2 {  
 public static void main(String[] args) {  
  
 *// 1. 创建一个日期格式化对象, 自定义显示 `模式`* SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy年MM月dd日 HH:mm:ss");  
  
 *// 2. 格式化 date = 2018年11月04日 11:03:15* String date = dateFormat.format(new Date());  
 System.*out*.println("date = " + date);  
 }  
}



**解析 :** 键盘录入了一个日期字符串, 如果后期需要进行 `日期时间` 相关的操作, 那么就必须将该字符串转换为 `日期时间` 对象.

public class Test3 {  
 public static void main(String[] args) throws ParseException {  
  
 *// 1. 创建一个键盘录入对象* Scanner sc = new Scanner(System.*in*);  
  
 *// 2. 提示, 并接收用户的输入* System.*out*.println("亲, 请输入您的出生年月日, 格式(yyyy-MM-dd) :");  
 String birthdayStr = sc.nextLine();  
  
 sc.close();  
  
 *// 3. 查看接收日期时间毫秒数. getTime();  
 // 3.1 将字符串对象转换为日期对象 (DateFormat 的 parse 方法可以完成)  
 // 如果输入字符串与解析模式不一致, 程序就会抛出 (Unparseable date: "2008年08月08")* SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");  
 *// 3.2 实现解析* Date date = dateFormat.parse(birthdayStr); *// 异常 : parseException 解析异常 throws ParseException  
 // 3.3 使用 date 对象调用 getTime() 获取毫秒数* System.*out*.println("date = " + date);  
 long time = date.getTime();  
 System.*out*.println("time = " + time);  
 }  
}



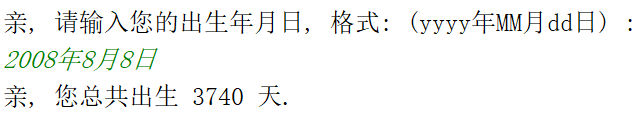
## 2.3 案例 : 计算总共活了多少天.

步骤分析 :

1. 获取用户的出生年月日.
2. 获取当前的日期对象.
3. 使用当前对象对象毫秒数 – 出生年月日对象时间毫秒数
4. 将毫秒数的差值转换为天数即可. 公式 : 毫秒数 / 1000 / 60 / 60 / 24;

**实现代码 :**

public class Test4 {  
 public static void main(String[] args) throws ParseException {  
  
 *// 1. 创建一个键盘录入对象* Scanner sc = new Scanner(System.*in*);  
  
 *// 2. 提示, 并接收用户输入* System.*out*.println("亲, 请输入您的出生年月日, 格式: (yyyy年MM月dd日) :");  
 String birthdayStr = sc.nextLine();  
 sc.close();  
  
 *// 3. 创建一个日期格式化对象 (SimpleDateFormat -> parse)* SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy年MM月dd日");  
  
 *// 4. 实现解析* Date birthday = dateFormat.parse(birthdayStr);  
  
 *// 5. 获取当前日期对象* Date now = new Date();  
  
 *// 6. 计算两个对象的时间毫秒数差值* long time = now.getTime() - birthday.getTime();  
  
 *// 7. 根据毫秒数计算天数* long day = time / 1000 / 60 / 60 / 24;  
  
 *// 8. 判断* if (day <= 0) {  
 System.*out*.println("亲, 您还没有出生呢.");  
 } else {  
 System.*out*.println("亲, 您总共出生 " + day + " 天.");  
 }  
 }  
}

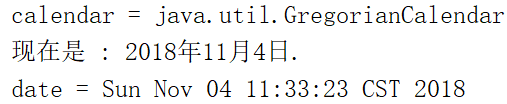


## 2.3 Calendar 日历类 :



**说明 :** 日历对象中已经包含了很多与 `**日期时间**` 相关的数据, 我们可以根据该对象直接使用获取.

import java.util.Calendar;  
import java.util.Date;  
  
public class Test5 {  
 public static void main(String[] args) {  
  
 *// 1. 需求, 获取当前日期时间的 `日历` 对象* Calendar calendar = Calendar.*getInstance*();  
 */\*  
 YEAR=2018,  
 MONTH=10,  
 DAY\_OF\_MONTH=4  
 HOUR=11  
 MINUTE=25  
 SECOND=16,  
 MILLISECOND=564  
 \*/* System.*out*.println("calendar = " + calendar.toString());  
  
 *// 需求 : 获取单独的年月日, 时分秒* int year = calendar.get(Calendar.*YEAR*);  
 int month = calendar.get(Calendar.*MONTH*); *// 从0开始计算* int day = calendar.get(Calendar.*DAY\_OF\_MONTH*);  
  
 System.*out*.println("现在是 : " + year + "年" + (month + 1) + "月" + day + "日.");  
  
 *// Calendar 类提供了一个 getTime(); 返回的是 Date 对象.* Date date = calendar.getTime();  
 System.*out*.println("date = " + date);  
 }  
}



**日历对象的修改和设置 :**

public class Test6 {  
 public static void main(String[] args) {  
  
 *// 1. 获取日历对象* Calendar calendar = Calendar.*getInstance*();  
 *printCalendar*(calendar);  
  
 *// 2. 回到 10 年之前* calendar.add(Calendar.*YEAR*, -10);  
 *printCalendar*(calendar);  
  
 *// 3. 去到一个月之后* calendar.add(Calendar.*MONTH*, 1);  
 *printCalendar*(calendar);  
  
 *// 4. 设置为 `2011年11月11日`  
 /\*  
 calendar.set(Calendar.YEAR, 2011);  
 calendar.set(Calendar.MONTH, 10);  
 calendar.set(Calendar.DAY\_OF\_MONTH, 11);  
 \*/* calendar.set(2011, 10, 11);  
 *printCalendar*(calendar);  
 }  
  
 *// 方法 : 输出日历对象的 `年月日`* public static void printCalendar(Calendar calendar) {  
 int year = calendar.get(Calendar.*YEAR*);  
 int month = calendar.get(Calendar.*MONTH*); *// 从0开始计算* int day = calendar.get(Calendar.*DAY\_OF\_MONTH*);  
  
 System.*out*.println("现在是 : " + year + "年" + (month + 1) + "月" + day + "日.");  
 }  
}

# System 系统类 : (了解)

## 3.1 currentTimeMillies 当前时间毫秒 :

**演示一 :**

long start = System.*currentTimeMillis*(); *// 获取当前时间的毫秒值*long time = new Date().getTime();  
  
System.*out*.println("start = " + start);  
System.*out*.println("time = " + time);

**演示二 :**

public class Test3 {  
 public static void main(String[] args) {  
  
 // 需求 : 计算某一段代码的执行效率  
 // 超级计算机 : 神威.太湖之光 12.5亿亿次 (天,地,人)  
  
 // 1. 记录开始时间  
 long start = System.currentTimeMillis(); // 获取当前时间的毫秒值  
  
 // 执行代码  
 long sum = 0;  
 for (int i = 0; i < 1000000000; i++) {  
 sum += i;  
 }  
  
 // 2. 记录结束时间  
 long end = System.currentTimeMillis();  
  
 // 3. 计算两个时间的差值  
 // 共耗时 : 283 毫秒.  
 System.out.println("共耗时 : " + (end - start) + " 毫秒.");  
 }  
}



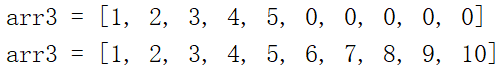
## arrayCopy 数组元素复制方法 :

**演示一 :**

public class Test1 {  
 public static void main(String[] args) {  
  
 *// 需求 : 数组拷贝. arrayCopy, 集合中底层存在大量的该方法实现.  
  
 // 1. 创建两个数组* int[] arr1 = {1, 2, 3, 4, 5};  
 int[] arr2 = {6, 7, 8, 9, 10};  
   
 *// 2. 实现 arrayCopy 方法的调用  
 // 静态方法 : static 修饰, 使用 `类名.方法名();` 调用 (类方法 Class Method)  
 // 非静态方法 : 直接使用 `对象名.方法名();` 调用 (对象方法 instance method)  
 /\*  
 arraycopy(Object src, int srcPos, Object dest, int destPos, int length);  
 1. src : 源数据 source  
 2. pos : 位置 position  
 3. dest : 目的地 destination  
 \*/* System.*arraycopy*(arr1, 0, arr2, 0, 3);  
  
 *// 3. 查看数组元素* System.*out*.println("arr1 = " + Arrays.*toString*(arr1)); *// [1, 2, 3, 4, 5]* System.*out*.println("arr2 = " + Arrays.*toString*(arr2)); *// [1, 2, 3, 9, 10]* }  
}

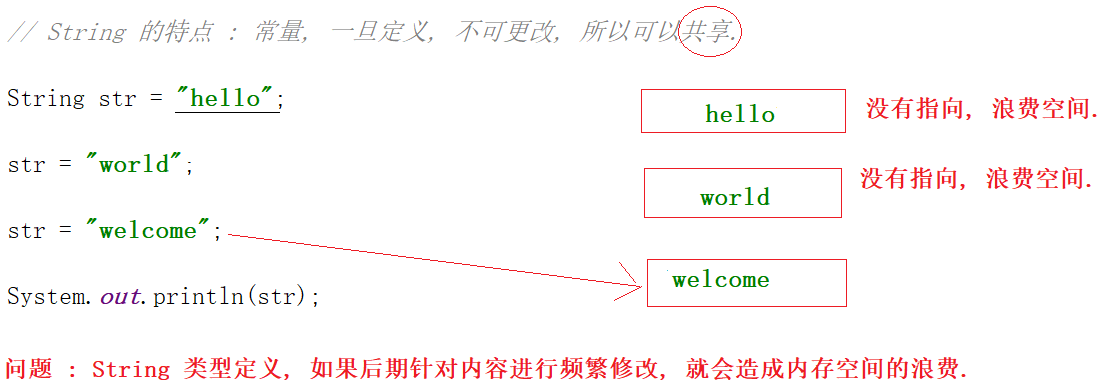
**演示二 :**

import java.util.Arrays;  
  
public class Test2 {  
 public static void main(String[] args) {  
   
 *// ArrayList 集合 : 可变数组. 底层就是 arraycopy 实现的. 因此添加和删除效率低, 但是查询快.  
  
 // 1. 创建两个数组* int[] arr1 = {1, 2, 3, 4, 5};  
 int[] arr2 = {6, 7, 8, 9, 10};  
  
 *// 2. 创建一个新数组* int[] arr3 = new int[10];  
  
 *// 需求 : 将 arr1 和 arr2 中的元素复制为 arr3 数组中.* System.*arraycopy*(arr1, 0, arr3, 0, arr1.length);  
 System.*out*.println("arr3 = " + Arrays.*toString*(arr3));  
  
 System.*arraycopy*(arr2, 0, arr3, arr1.length, arr2.length);  
 System.*out*.println("arr3 = " + Arrays.*toString*(arr3));  
 }  
}



# StringBuilder 字符串构造者类 : (重点)

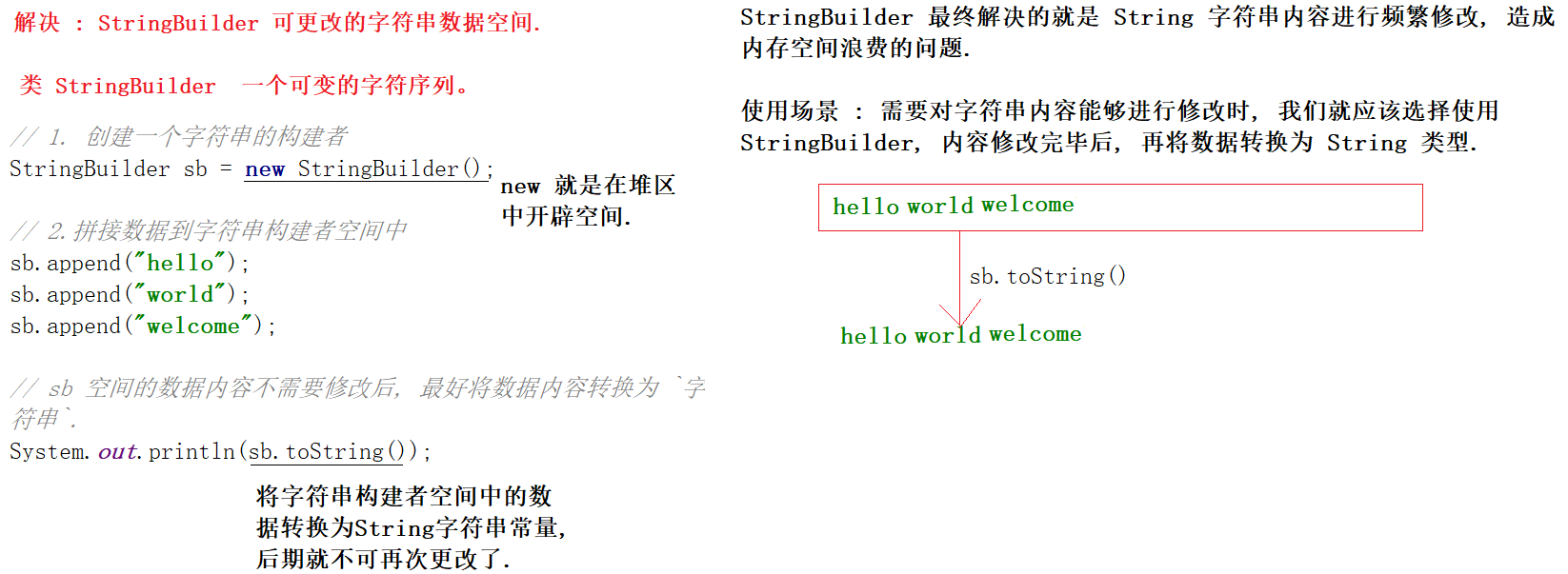
## 4.1 String类的使用说明 :

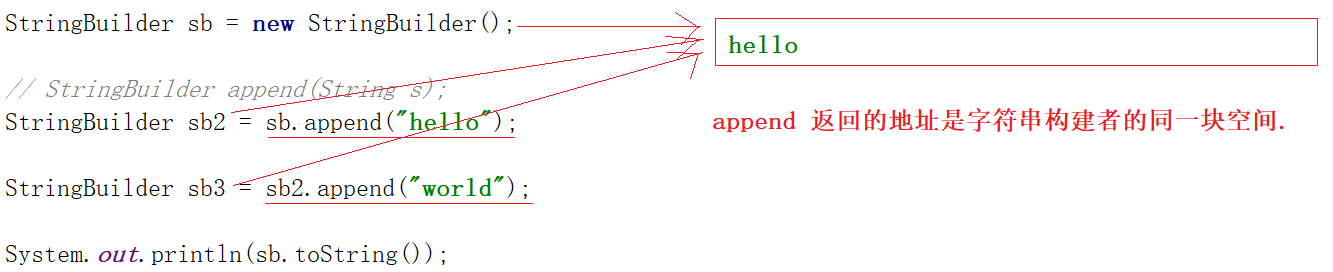


public class Test1 {  
 public static void main(String[] args) {  
  
 *// String 的特点 : 常量, 一旦定义, 不可更改, 所以可以共享.* String str = "hello";  
 str = "world";  
 str = "welcome";  
  
 System.*out*.println(str);  
 }  
}



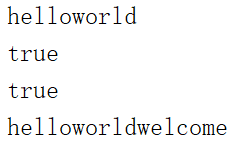
## 4.2 StringBuilder 使用说明 :





**链式编程代码书写实现 :**

public class Test4 {  
 public static void main(String[] args) {  
  
 StringBuilder sb = new StringBuilder();  
  
 *// StringBuilder append(String s);* StringBuilder sb2 = sb.append("hello");  
 StringBuilder sb3 = sb2.append("world");  
 System.*out*.println(sb.toString());  
  
 System.*out*.println(sb == sb2); *// true* System.*out*.println(sb == sb3); *// true  
  
 // 说明 : 如果一个方法返回的是自己本身的地址, 程序编写时, 我们就不需要去接收该返回结果.  
 // 好处 : 有了以上这个前提条件, 程序代码就可以实现 `链式编程`* StringBuilder sb4 = new StringBuilder();  
  
 *// 这就是 `链式编程`* sb4.append("hello").append("world").append("welcome");  
  
 System.*out*.println(sb4); *// sb4.toString(); 是 String 类型.* }  
}



# 包装类 : (了解)

说明 : 系统为我们提供了一些方法, 可以直接使用, 而基本数据类型是没有方法的.

基本数据类型 引用类型 (包装类型)

byte Byte

short Short

int Integer 整型

long Long

float Float

double Double

char Character 字符

boolean Boolean

集合的泛型定义 : <泛型> 泛型必须是引用类型, 不能定义为基本数据类型.

## 装箱与拆箱 : (基本数据类型,包装类型之间的自动转换)

装箱 : 将基本数据类型包装为引用类型. Integer num = 998;

拆箱 : 将包装类型拆解为基本数据类型. int value = num;

**装箱演示 :**

*// 自动装箱 : 编译器自动完成, 不需手动实现.*Integer num = 998;  
  
*// 包装方法底层废弃了, 不推荐使用.*Integer num2 = new Integer(998);  
  
*// 手动实现 : 底层自动实现数据的装箱.*Integer num3 = Integer.*valueOf*(998);

**拆箱演示 :**

*// 自动拆箱 : 将包装类型自动拆解为基本数据类型.*int num = Integer.*valueOf*(998);  
  
*// 手动拆箱 : value1.intValue()*Integer value1 = Integer.*valueOf*(998);  
int num2 = value1.intValue();

## 5.2 基本数据类型与字符串的转换 :

演示一 : 如何将基本数据类型转换为字符串.

String str1 = 998 + ""; *// 记住  
// 了解 :*String str2 = Integer.*toString*(998);  
String str3 = String.*valueOf*(998);

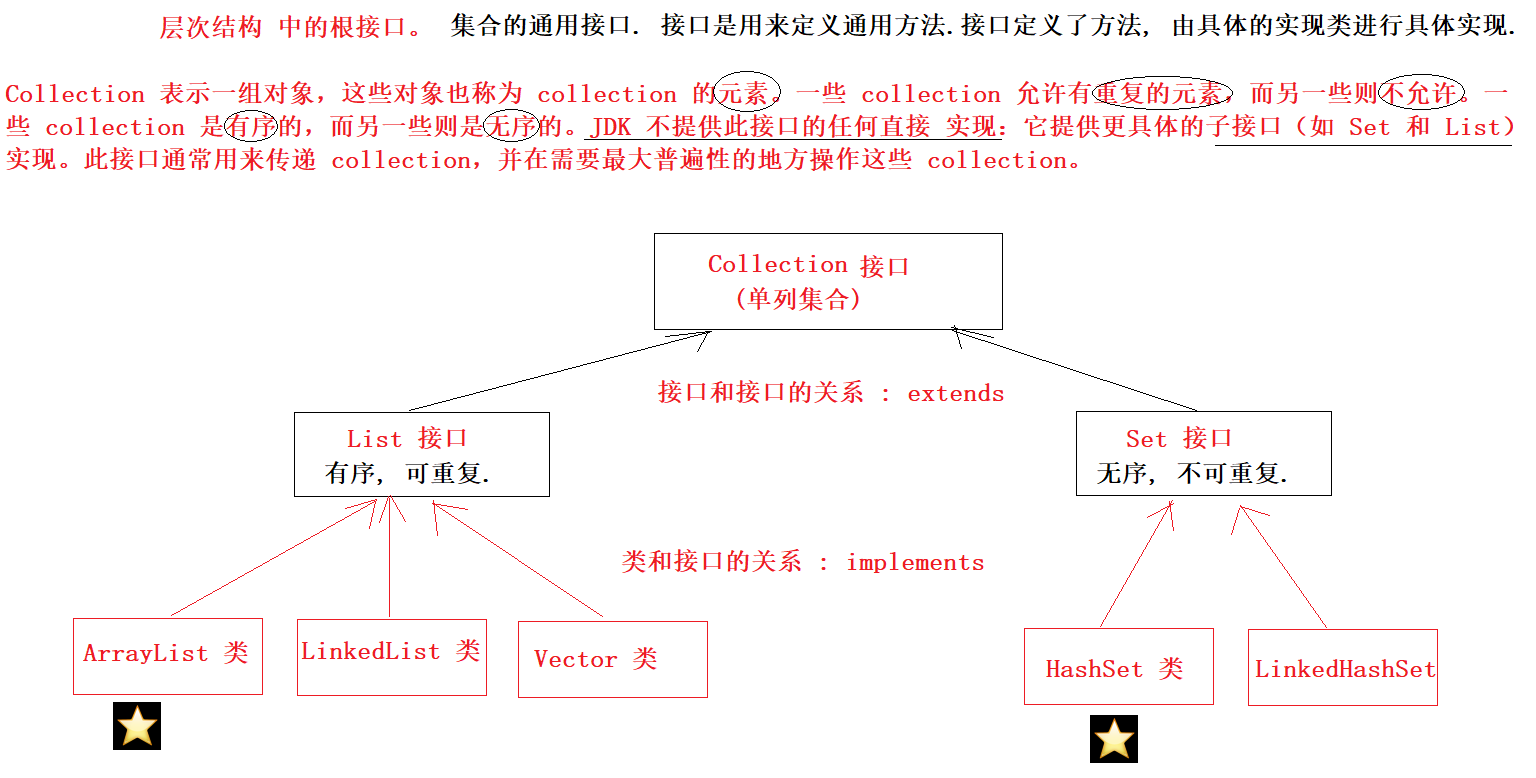
**包装类型的好处在于可以为基本数据类型提供相对应的 `数据和行为`.**

public class Test4 {  
 public static void main(String[] args) {  
  
 int maxValue = Integer.*MAX\_VALUE*;  
 System.*out*.println("maxValue = " + maxValue);  
  
 int minValue = Integer.*MIN\_VALUE*;  
 System.*out*.println("minValue = " + minValue);  
  
 int size = Integer.*SIZE*; *// byte = 8bit* System.*out*.println("size = " + size); *// 比特位数* Class<Integer> cls = Integer.*TYPE*;  
 System.*out*.println("cls = " + cls);  
 }  
}

演示二 : 如何将字符串转换为基本数据类型

String str = "998";  
  
*// 运算 : 998 + 2 = 1000 结果, 而不是 9982*String result = str + 2;  
System.*out*.println("result = " + result);  
  
*// 字符串转基本数据类型 :*int value = Integer.*parseInt*(str); *// 网页上输出的数据都是 String 类型的.*int sum = value + 2;  
System.*out*.println("sum = " + sum); *// sout 输出 v variable 变量*

# 6. Collection 接口 : (重要)





## 集合与数组的区别 :

数组 : 可以存储基本数据类型和引用类型. 长度固定, 一旦确定, 后期无法再次修改.

例如 : int[长度], float[长度], String[长度], Student[长度] …

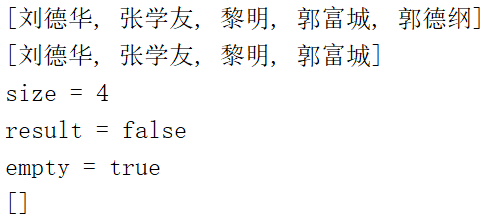
集合 : 仅能存储引用类型. 不能存储基本数据类型. 集合底层自动实现了 `可变长度`.

例如 : ArrayList<Integer>, ArrayList<Float>, ArrayList<String>, ArrayList<Student> …

## 6.2 Collection常用方法 :

**演示一 : 常用方法**

public class Test1 {  
 public static void main(String[] args) {  
  
 *// 需求 : 查看 Collection 接口中定义的方法.  
 // ctrl + 类名, 进入, 然后 alt + 7 查看该类的大纲.* Collection<String> c = new ArrayList<>(); *// 泛型的自动推导.  
  
 // 1. 添加* c.add("刘德华");  
 c.add("张学友");  
 c.add("黎明");  
 c.add("郭富城");  
 c.add("郭德纲");  
  
 *// 说明 : ArrayList 重写了 toString() 方法.* System.*out*.println(c); *// [刘德华, 张学友, 黎明, 郭富城] 该方法 ArrayList 最终还是从父类中继承而来的.  
  
 // 2. 删除* c.remove("郭德纲");  
 System.*out*.println(c);  
  
 *// 3. 长度* int size = c.size();  
 System.*out*.println("size = " + size);  
  
 *// 4. 包含* boolean result = c.contains("郭德纲");  
 System.*out*.println("result = " + result);  
  
 *// 5. 清空* c.clear();  
  
 *// 6. 判断集合是否为空* boolean empty = c.isEmpty();  
 System.*out*.println("empty = " + empty); *// 有元素, 返回false, 没有元素, 返回 true* System.*out*.println(c);  
 }  
}



**演示二 : addAll**

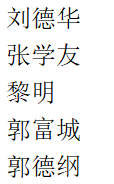
public class Test2 {  
 public static void main(String[] args) {  
  
 Collection<String> c = new ArrayList<>();  
 c.add("刘德华");  
 c.add("张学友");  
 c.add("黎明");  
 c.add("郭富城");  
  
 Collection<String> c2 = new ArrayList<>();  
 c2.add("貂蝉");  
 c2.add("西施");  
 c2.add("王昭君");  
 c2.add("杨玉环");  
  
 *// 需求 : 将 c2 集合中的所有元素全部添加到 c 集合中.* c.addAll(c2);   
 System.*out*.println(c);  
 }  
}

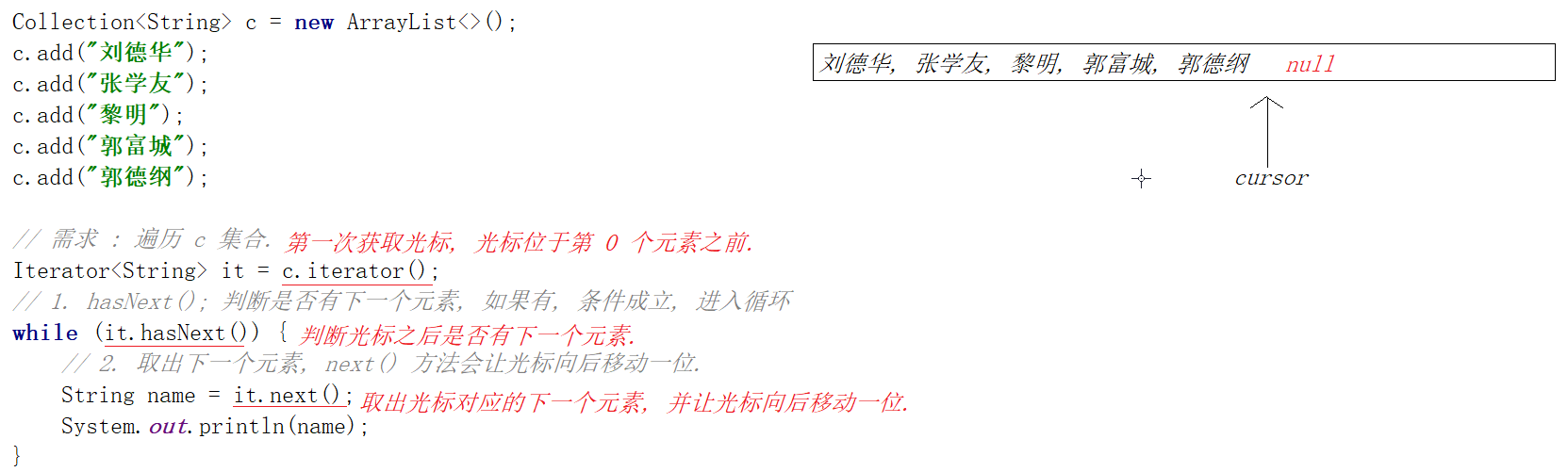


## 6.3 通用遍历方式 - Iterator迭代器

**理解 : 迭代就是遍历.**

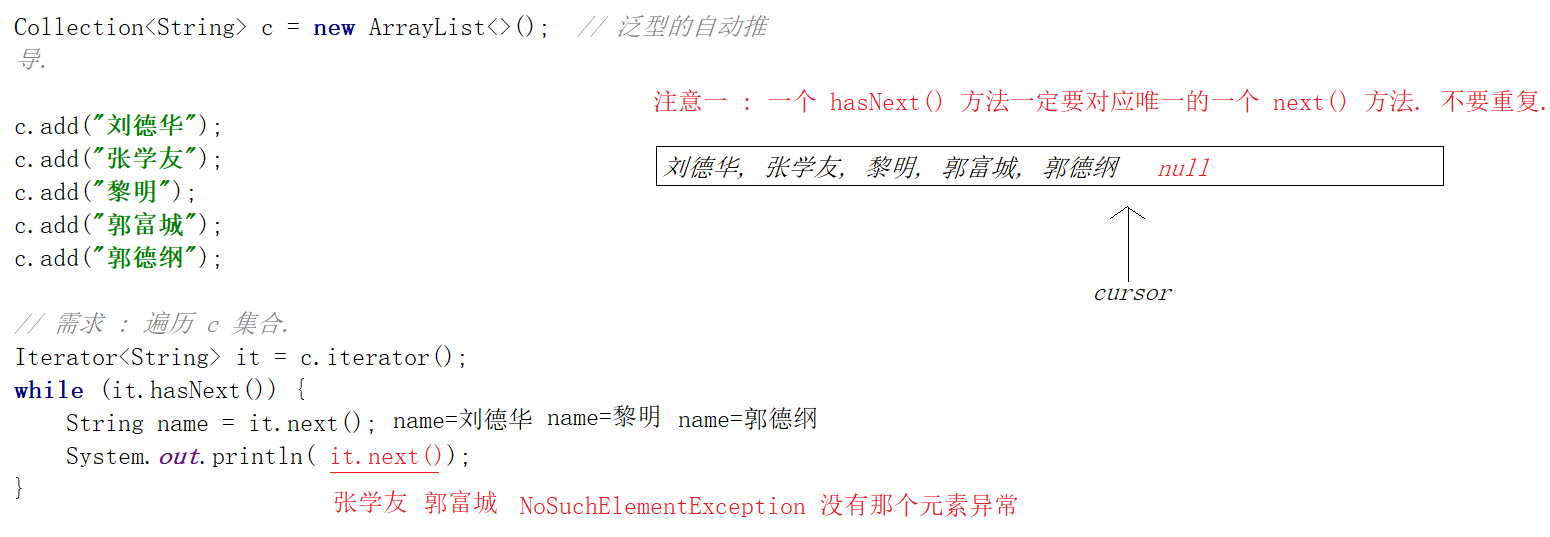
public class Test3 {  
 public static void main(String[] args) {  
  
 Collection<String> c = new ArrayList<>(); *// 泛型的自动推导.* c.add("刘德华");  
 c.add("张学友");  
 c.add("黎明");  
 c.add("郭富城");  
 c.add("郭德纲");  
  
 *// 需求 : 遍历 c 集合.  
 // size() + get(); 实现前提条件为 ArrayList 接口集合变量.  
 // Collection 集合定义了一个 iterator 迭代器的通用遍历方案.  
 // Iterator 是一个接口. c.iterator(); 返回的是 Iterator 该接口的实现类对象.* Iterator<String> it = c.iterator();  
 *// Iterator 接口提供的方法如下 : hasNext(); 判断 next(); 获取  
 // 1. hasNext(); 判断是否有下一个元素, 如果有, 条件成立, 进入循环* while (it.hasNext()) {  
 *// 2. 取出下一个元素, next() 方法会让光标向后移动一位.* String name = it.next();  
 System.*out*.println(name);  
 }  
 }  
}





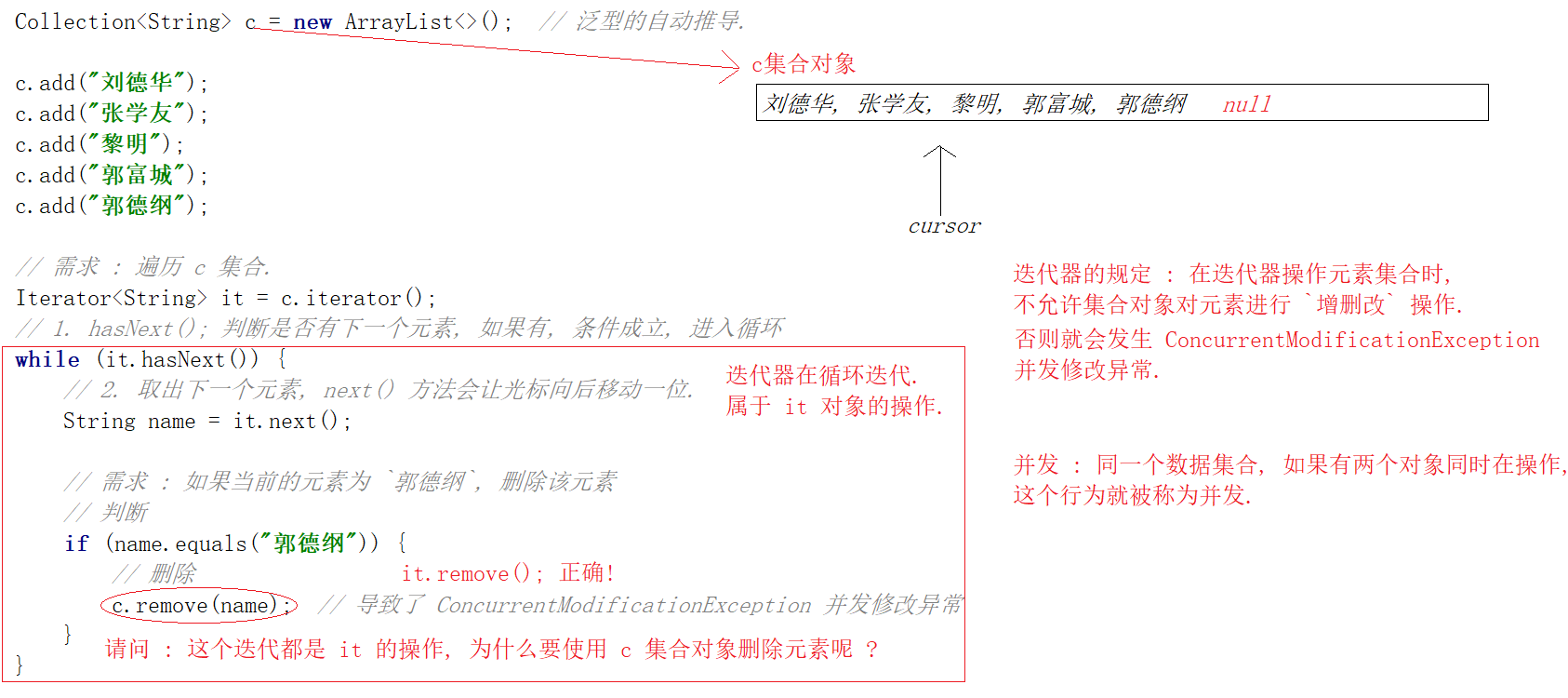
### 6.3.1 迭代器使用注意一 : 一个 hasNext 不要对应多个 next 方法

public class Test4 {  
 public static void main(String[] args) {  
  
 Collection<String> c = new ArrayList<>(); *// 泛型的自动推导.* c.add("刘德华");  
 c.add("张学友");  
 c.add("黎明");  
 c.add("郭富城");  
 c.add("郭德纲");  
  
 *// 需求 : 遍历 c 集合.* Iterator<String> it = c.iterator();  
 *// 1. hasNext(); 判断是否有下一个元素, 如果有, 条件成立, 进入循环* while (it.hasNext()) {  
 *// 2. 取出下一个元素, next() 方法会让光标向后移动一位.* String name = it.next();  
 System.*out*.println(it.next());  
 }  
 }  
}



### 6.3.2 迭代器使用注意二 : 迭代器迭代同时, 不要使用集合对元素进行 `增删改` 操作

public class Test5 {  
 public static void main(String[] args) {  
  
 Collection<String> c = new ArrayList<>(); *// 泛型的自动推导.* c.add("郭德纲");  
 c.add("刘德华");  
 c.add("张学友");  
 c.add("黎明");  
 c.add("郭富城");  
  
 *// 需求 : 遍历 c 集合.* Iterator<String> it = c.iterator();  
 *// 1. hasNext(); 判断是否有下一个元素, 如果有, 条件成立, 进入循环* while (it.hasNext()) {  
 *// 2. 取出下一个元素, next() 方法会让光标向后移动一位.* String name = it.next();  
  
 *// 需求 : 如果当前的元素为 `郭德纲`, 删除该元素  
 // 判断* if (name.equals("郭德纲")) {  
 *// 删除  
 // c.remove(name); // 导致了 ConcurrentModificationException 并发修改异常* it.remove();  
 }  
 }  
 System.*out*.println(c);  
 }  
}



## 6.4 增强 for 循环 – 迭代器的简化形式

格式 :

for (元素类型 变量名 : 集合) {

直接操作变量名.

}

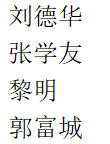
public class Test6 {  
 public static void main(String[] args) {  
  
 Collection<String> c = new ArrayList<>(); *// 泛型的自动推导.* c.add("刘德华");  
 c.add("张学友");  
 c.add("黎明");  
 c.add("郭富城");  
 c.add("郭德纲");  
  
 *// 需求 : 遍历 c 集合.  
 // 说明 : 增强for循环编译完毕后, 底层就是迭代器. 它就是迭代器的遍历简化形式.* for (String name : c) {  
 System.*out*.println(name);  
 }  
 }  
}

请问 : 增强 for 循环遍历集合元素, 可否对集合元素进行 `增删改` 操作 ??? 不可以.

*// 需求 : 遍历 c 集合.  
// 说明 : 增强for循环编译完毕后, 底层就是迭代器. 它就是迭代器的遍历简化形式.*for (String name : c) {  
 *// 需求 : 删除 `郭德纲`* if (name.equals("郭德纲")) {  
 c.remove(name); // ConcurrentModificationException 并发修改异常  
 }  
}

**增强for循环, 实现遍历数组.**

public class Test7 {  
 public static void main(String[] args) {  
  
 String[] names = {"刘德华", "张学友", "黎明", "郭富城"};  
  
 *// 简写方式 : 集合 / 数组.for 回车* for (String name : names) {  
 System.*out*.println(name);  
 }  
 }  
}



# 泛型 – 参数化类型 (了解)

例如 : 定义一个集合. 请问, 集合中存储什么类型的元素数据呢 ???

请问 : 定义 ArrayList 该类的工程师不知道存储什么类型的数据. 那怎么办 ???

解决 : 该工程师就不指定存储数据的类型, 谁用, 谁指定.

张三 : ArrayList<Apple>();

李四 : ArrayList<Orange>();

王五 : ArrayList<Girl>();

好处 : 保证类型的灵活性.

ArrayList<E> Element 元素

Comparable<T> Type 类型

HashMap<K,V> K key 键, V value 值

其它 : R, Result 结果.

结果 : 泛型的字母没有固定写法样式, 可以自己决定. 但一定要有意义.

所有字母的类型都是不确定的类型, 只有在创建的那一刻才真正确定. 并且将类型传递到具体的泛型上.

ArrayList<E> E -> String E -> Integer E -> Student

new ArrayList<String>(); new ArrayList<Integer>(); new ArrayList<Student>();

方法 : public static int sum(int a, int b) { return a + b; } a = 10; b = 20;

调用 : sum(10, 20);

## 7.1 泛型类 :

**泛型类定义 :**

*// 定义一个类*public class GenericClass<T> {  
 *// 属性* private T field;  
  
 *// 行为* public void setField(T field) {  
 this.field = field;  
 }  
  
 public T getField() {  
 return field;  
 }  
}

**测试泛型类中泛型的灵活性 :**

public class Test1 {  
 public static void main(String[] args) {  
  
 *// 1. 创建一个泛型类* GenericClass<String> g1 = new GenericClass<>();  
 g1.setField("张三");  
 String name = g1.getField();  
 System.*out*.println("name = " + name);  
  
 *// 2. 再创建一个泛型类* GenericClass<Integer> g2 = new GenericClass<>();  
 g2.setField(998);  
 Integer number = g2.getField();  
 System.*out*.println("number = " + number);  
 }  
}



## 7.2 泛型接口 :

**泛型接口的定义 :**

*// 定义了一个泛型接口, 该泛型其实是给实现类提供的.*public interface GenericInterface<T> {  
 *// 抽象方法* void show(T t);  
}

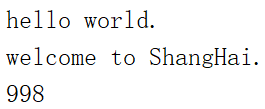
**接口实现类的定义 :**

*// 实现类确定接口的泛型方式一 : 实现同时, 直接定义.*public class GenericInterfaceImpl1 implements GenericInterface<String> {  
 @Override  
 public void show(String s) {  
 System.*out*.println(s);  
 }  
}

*// 实现类确定接口的泛型方式二 : 不直接指定, 创建实现类对象时再进行指定.*public class GenericInterfaceImpl2<T> implements GenericInterface<T> {  
 @Override  
 public void show(T t) {  
 System.*out*.println(t);  
 }  
}

**测试类演示 :**

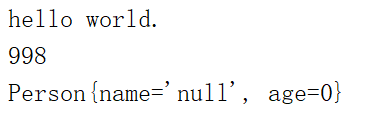
public class Test2 {  
 public static void main(String[] args) {  
  
 *// 实现类方式一 :* GenericInterfaceImpl1 g1 = new GenericInterfaceImpl1();  
 g1.show("hello world.");  
  
 *// 实现类方式二 :* GenericInterfaceImpl2<String> g2 = new GenericInterfaceImpl2<>();  
 g2.show("welcome to ShangHai.");  
  
 GenericInterfaceImpl2<Integer> g3 = new GenericInterfaceImpl2<>();  
 g3.show(998);  
 }  
}



## 7.3 泛型方法 : T

**泛型方法的定义 :**

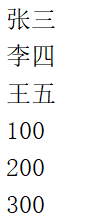
public class Test3 {  
 public static void main(String[] args) {  
  
 *printInfo*("hello world.");  
  
 *printInfo*(998); *// 泛型不能是基本数据类型. 998 被包装为了 Integer (自动包装)  
  
 printInfo*(new Student());  
 }  
  
 *// 定义一个方法, 此时需要为该方法定义一个泛型. 定义的语法: <>* public static <T> void printInfo(T t) {  
 System.*out*.println(t);  
 }  
}



## 7.4 泛型通配符 : ?

使用场景 : 必须在已经定义的接口或类上的泛型位置才能使用.

public class Test4 {  
 public static void main(String[] args) {  
  
 ArrayList<String> list = new ArrayList<>();  
 list.add("张三");  
 list.add("李四");  
 list.add("王五");  
 *printInfo*(list);  
  
  
 ArrayList<Integer> list2 = new ArrayList<>();  
 list2.add(100);  
 list2.add(200);  
 list2.add(300);  
 *printInfo*(list2);  
 }  
  
 *// 已经定义的接口. Collection<E>  
 // ? 表示 ? extends Object  
 // 翻译 : 不指定 Collection 集合中存储的是什么类型, 但是我知道该类型继承自 Object.* public static void printInfo(Collection<?> t) {  
 for (Object o : t) {  
 System.*out*.println(o);  
 }  
 }  
}



### 7.4.1 泛型上下限 : (了解)

/\*  
泛型通配符的 `上下限`  
  
上限 : extends ? extends Object ? extends Person ? extends Animal  
翻译 : ? 我不知道该泛型是什么类型, 但是我知道该泛型继承 (Object, Person, Animal).  
  
下限 : super ? super ArmyDog (军犬)  
翻译 : ? 我不知道该泛型是什么类型, 但是该泛型是 ArmyDog 的 super 类.  
 \*/

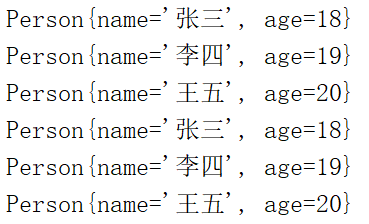
**上限 :**

**方法定义 :**

*// printInfo(Collection )*public static void printInfo(Collection<? extends Person> c) {  
 for (Person person : c) {  
 System.*out*.println(person);  
 }  
}

**传递演示 :**

ArrayList<Person> list1 = new ArrayList<>();  
list1.add(new Person("张三", 18));  
list1.add(new Person("李四", 19));  
list1.add(new Person("王五", 20));  
  
*printInfo*(list1);  
  
ArrayList<Student> list2 = new ArrayList<>();  
list2.add(new Student("张三", 18));  
list2.add(new Student("李四", 19));  
list2.add(new Student("王五", 20));  
  
*printInfo*(list2);  
  
*/\*  
ArrayList<String> list3 = new ArrayList<>(); 错误!  
list3.add("张三");  
  
printInfo(list3);  
\*/*



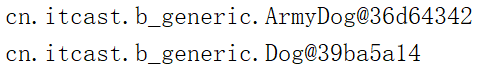
**下限 :**

**方法定义 :**

*// printInfo(Collection )  
// 说明 : 一个集合中不可能存储多个类型的元素, 基本都是存储同一个元素.*public static void printInfo2(Collection<? super ArmyDog> c) {  
 for (Object o : c) {  
 System.*out*.println(o);  
 }  
}

**传递演示 :**

ArrayList<ArmyDog> list4 = new ArrayList<>();  
list4.add(new ArmyDog());  
*printInfo2*(list4);  
  
ArrayList<Dog> list5 = new ArrayList<>();  
list5.add(new Dog());  
*printInfo2*(list5);  
  
*/\*  
ArrayList<Cat> list6 = new ArrayList<>(); 错误!  
list6.add(new Cat());  
  
printInfo2(list6);  
\*/*



## 7.5 泛型小结 :

**泛型的特点** : 泛型属于编译时期的类型, 在程序编译阶段是存在泛型, 但是, 程序一旦运行, 泛型就会被 `**自动擦除`**. 也就是说, 运行阶段不存在任何泛型.

泛型的好处 :

1. 类型的灵活性.
2. 统一了集合中元素的类型. 如果泛型不指定, 默认会被提升为 Object 类型.
3. 泛型可以将运行时期的类型转换异常提前到编译阶段进行语法检查.
4. 泛型可以省略元素类型强转问题.

# 8. 案例 – 斗地主 (版本一)

2♥, 2♦, 2♣, 2♠, 3♥, 3♦, 3♣, 3♠, 4♥, 4♦, 4♣, 4♠, 5♥, 5♦, 5♣, 5♠, 6♥, 6♦, 6♣, 6♠, 7♥, 7♦, 7♣, 7♠, 8♥, 8♦, 8♣, 8♠, 9♥, 9♦, 9♣, 9♠, 10♥, 10♦, 10♣, 10♠, J♥, J♦, J♣, J♠, Q♥, Q♦, Q♣, Q♠, K♥, K♦, K♣, K♠, A♥, A♦, A♣, A♠, 小☺, 大☠

张三丰 : [A♣, 8♣, 10♥, 8♠, 7♦, K♥, A♥, 3♠, A♦, 7♣, 10♣, K♠, 2♠, 5♠, 3♣, K♣, 6♥]

张无忌 : [J♠, 3♦, 8♥, Q♦, J♦, J♣, 9♦, 6♣, 9♥, 4♣, A♠, 4♥, 5♥, K♦, 5♦, 10♠, 4♠]

张学友 : [大☠, 2♦, 小☺, J♥, 9♣, 6♠, 6♦, 10♦, 4♦, 7♥, 3♥, 8♦, 2♥, 2♣, Q♣, 7♠, Q♥]

底牌 : [Q♠, 9♠, 5♣]

**思路说明 :**

步骤分析 :

步骤一 : 准备一副扑克牌.(54张)

步骤二 : 洗牌. Collections 集合工具类 方式: shuffle

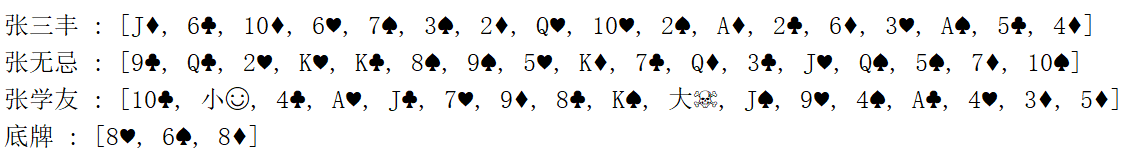
步骤三 : 准备三个玩家和一个底牌.

步骤四 : 发牌, 每人循环各拿一张, 最后三张作为底牌.

步骤五 : 看牌.

**具体代码实现 :**

import java.util.ArrayList;  
import java.util.Collections;  
  
public class PokerTest {  
 public static void main(String[] args) {  
  
 *// 步骤一 : 准备一副扑克牌.(54张)  
 // 1.1 创建一个 `ArrayList 有序列表集合`* ArrayList<String> pokers = new ArrayList<>(); *// String 是通用类型  
 // 1.2 定义一个 `数字集合` [2, 3, 4, 5, 6, 7, 8, 9, 10, J, Q, K, A]* ArrayList<String> numbers = new ArrayList<>();  
 *// 添加 2~10* for (int i = 2; i <= 10; i++) {  
 numbers.add(i + "");  
 }  
 *// 添加 J, Q, K, A* numbers.add("J");  
 numbers.add("Q");  
 numbers.add("K");  
 numbers.add("A");  
 System.*out*.println("numbers = " + numbers);  
  
 *// 1.3 定义一个 `花色集合`* ArrayList<String> colors = new ArrayList<>();  
 colors.add("♥");  
 colors.add("♦");  
 colors.add("♣");  
 colors.add("♠");  
 System.*out*.println("colors = " + colors);  
  
 *// 1.4 拼接扑克牌 2♥, 2♦, 2♣, 2♠, 3♥, 3♦, 3♣, 3♠,  
 // 先遍历数字集合* for (String number : numbers) {  
 *// 再遍历花色集合* for (String color : colors) {  
 *// 拼接扑克牌* String poker = number + color;  
 *// 最后将拼接完成的扑克牌存放到 pokers 集合中* pokers.add(poker);  
 }  
 }  
 *// 1.5 最后拼接 `大小王`* pokers.add("小☺");  
 pokers.add("大☠");  
 System.*out*.println("pokers = " + pokers);  
  
 *// 步骤二 : 洗牌. Collections 集合工具类 方式: shuffle* Collections.*shuffle*(pokers);  
 System.*out*.println("pokers = " + pokers);  
  
 *// 步骤三 : 准备三个玩家和一个底牌.* ArrayList<String> player1 = new ArrayList<>();  
 ArrayList<String> player2 = new ArrayList<>();  
 ArrayList<String> player3 = new ArrayList<>();  
 ArrayList<String> base = new ArrayList<>();  
  
 *// 步骤四 : 发牌, 每人循环各拿一张, 最后三张作为底牌.* for (int i = 0; i < pokers.size(); i++) {  
 *// 取出当前的扑克牌* String poker = pokers.get(i);  
 *// 判断最后三张* if (i >= pokers.size() - 3) { *// 54 - 3 == 51  
 // 底牌 (51, 52, 53)* base.add(poker);  
 } else {  
 *// 玩家  
 // 取模 / 取余 判断* if (i % 3 == 0) {  
 *// 玩家 1* player1.add(poker);  
 } else if (i % 3 == 1) {  
 *// 玩家 2* player2.add(poker);  
 } else {  
 *// 玩家 3* player3.add(poker);  
 }  
 }  
 }  
  
 System.*out*.println("-----------------------------------");  
  
 *// 步骤五 : 看牌.* System.*out*.println("张三丰 : " + player1);  
 System.*out*.println("张无忌 : " + player2);  
 System.*out*.println("张学友 : " + player3);  
 System.*out*.println("底牌 : " + base);  
 }  
}



# List 接口的特点 : 下标

**ArrayList 数组列表 : 下标操作**

List 接口的特点 : 有序, 可重复.

常用方法 :

1. 增 : void add(int index, E e);

2. 删 : E remove(int index);

3. 该 : E set(int index, E e);

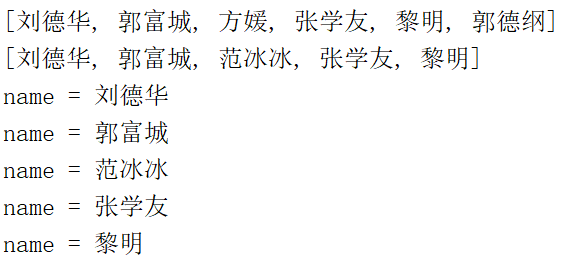
4. 查 : E get(int index);

遍历1 : Iterator 迭代器遍历.

遍历2 : 增强 for 循环.

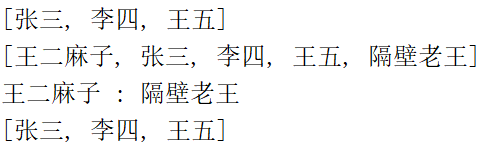
遍历3 : size() + get(); 实现遍历.

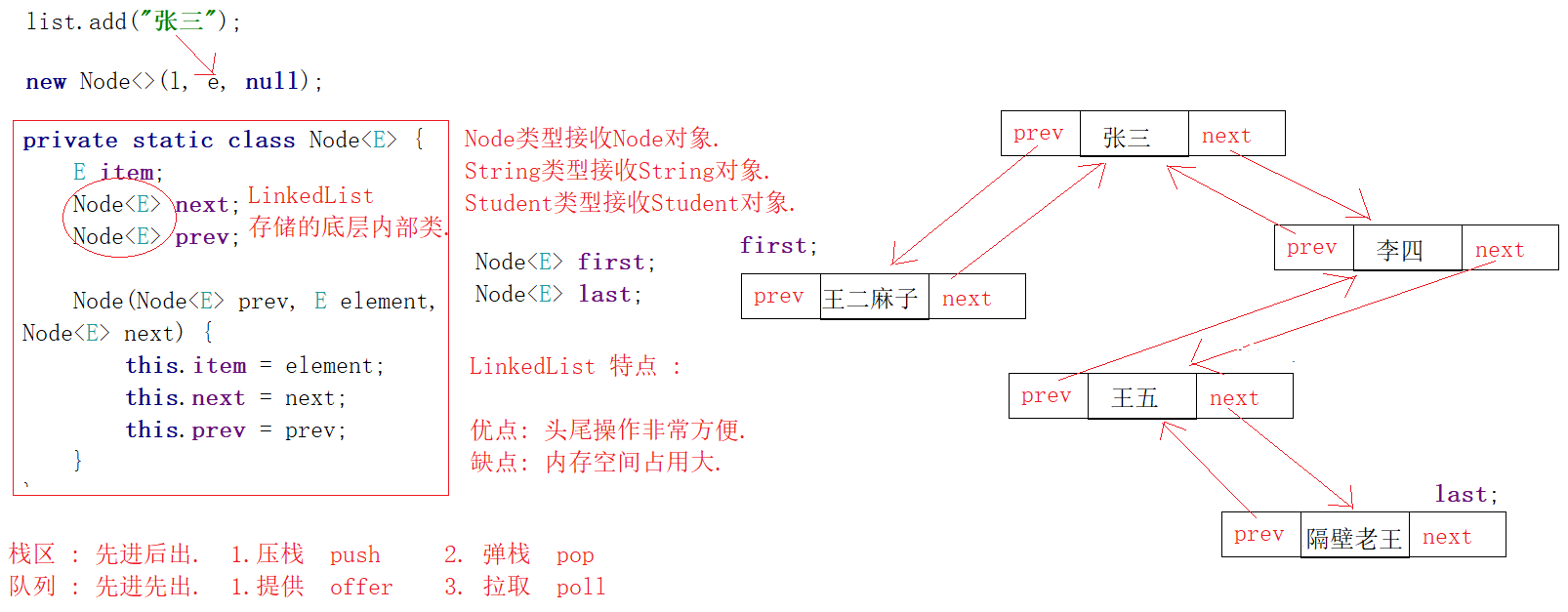
public class Test1 {  
 public static void main(String[] args) {  
  
 *// 1. 创建一个 List 集合  
 // ArrayList<String> list = new ArrayList<>();  
  
 // 通用性 :* List<String> list = new ArrayList<>();  
  
 *// 增加 :* list.add("刘德华");  
 list.add("郭富城");  
 list.add("张学友");  
 list.add("黎明");  
 list.add("郭德纲");  
  
 *// 需求1 : 郭富城的老婆, 是 `方媛`* list.add(2, "方媛");  
  
 System.*out*.println(list);  
  
 *// 需求2 : 删除 `郭德纲`* list.remove(5);  
  
 *// 需求3 : 将 `方媛` 修改为 `范冰冰`* list.set(2, "范冰冰");  
  
 System.*out*.println(list);  
  
 *// 需求4 : 遍历 : size() + get();* for (int i = 0; i < list.size(); i++) {  
 String name = list.get(i);  
 System.*out*.println("name = " + name);  
 }  
 }  
}



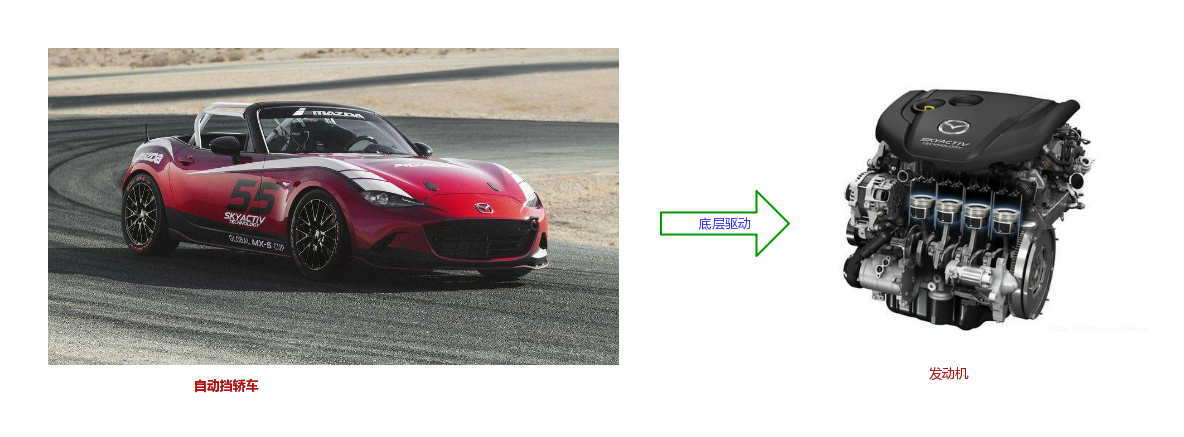
## 9.2 LinkedList 链表 : 头尾操作

public class Test2 {  
 public static void main(String[] args) {  
  
 *// LinedList 系统底层代码大量使用, 但是, 我们开发者很少使用.  
 // LinkedList 操作特点 : 头尾操作特别方便* LinkedList<String> list = new LinkedList<>();  
  
 list.add("张三");  
 list.add("李四");  
 list.add("王五");  
  
 System.*out*.println(list);  
  
 *// 需求 : 将 `王二麻子` 添加到集合的头部* list.addFirst("王二麻子");  
  
 *// 需求 : 将 `隔壁老王` 添加到集合的尾部* list.addLast("隔壁老王");  
  
 System.*out*.println(list);  
  
 *// 获取头尾元素* String first = list.getFirst();  
 String last = list.getLast();  
 System.*out*.println(first + " : " + last);  
  
 *// 删除头尾元素* list.removeFirst();  
 list.removeLast();  
 System.*out*.println(list);  
 }  
}





# 10. Java中的数据结构 : (了解)



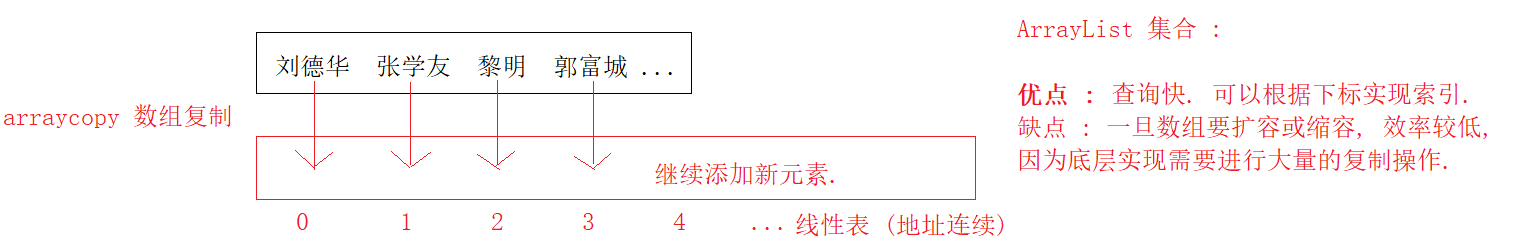
## 10.1 数组结构 : (线性表)

ArrayList 集合类底层是数组结构. Object[] elementData;

请问 : Object[] 数组长度是固定的, 但是 ArrayList 是一个可变数组, 那底层到底是怎么实现的呢 ???

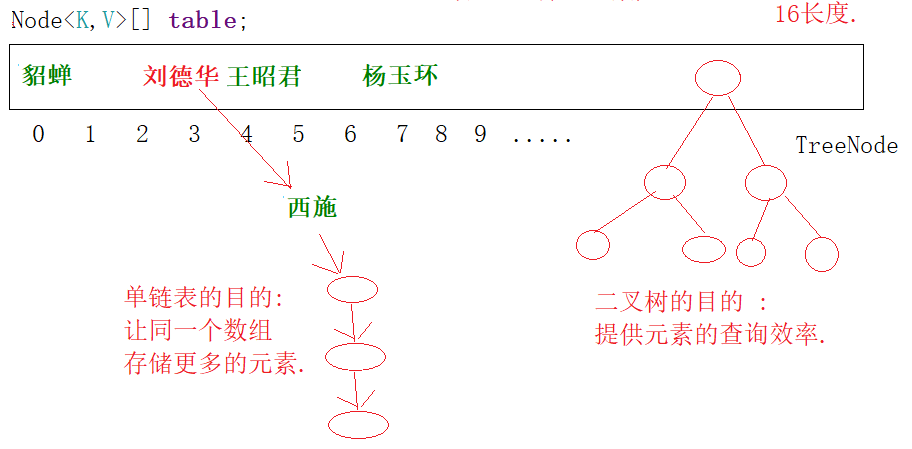
底层实现 : System.arraycopy(original, 0, copy, 0, Math.min(original.length, newLength)); 就是数组复制.

线性表的含义 : 内存空间, 地址连续.

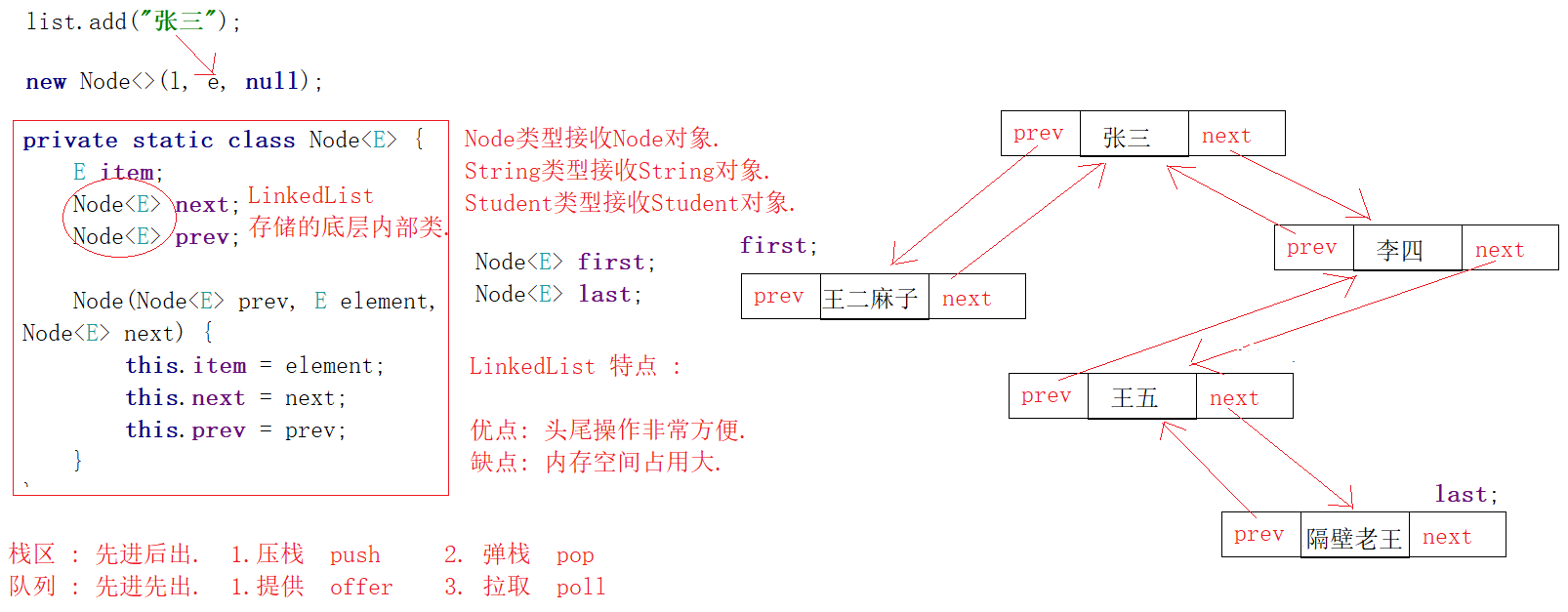


## 10.2 单链表结构 :

**HashSet : 单链表** (底层其实是由 HashMap 支持的.)



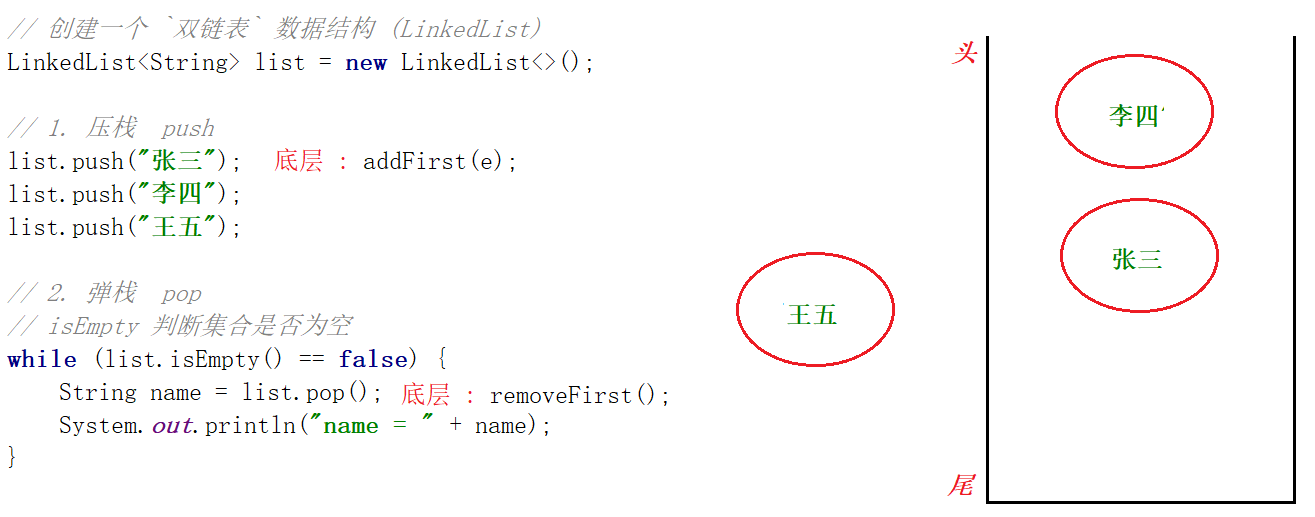
## 10.3 双链表结构 :



## 10.4 栈结构 : push -> pop

栈结构 : 先进后出.

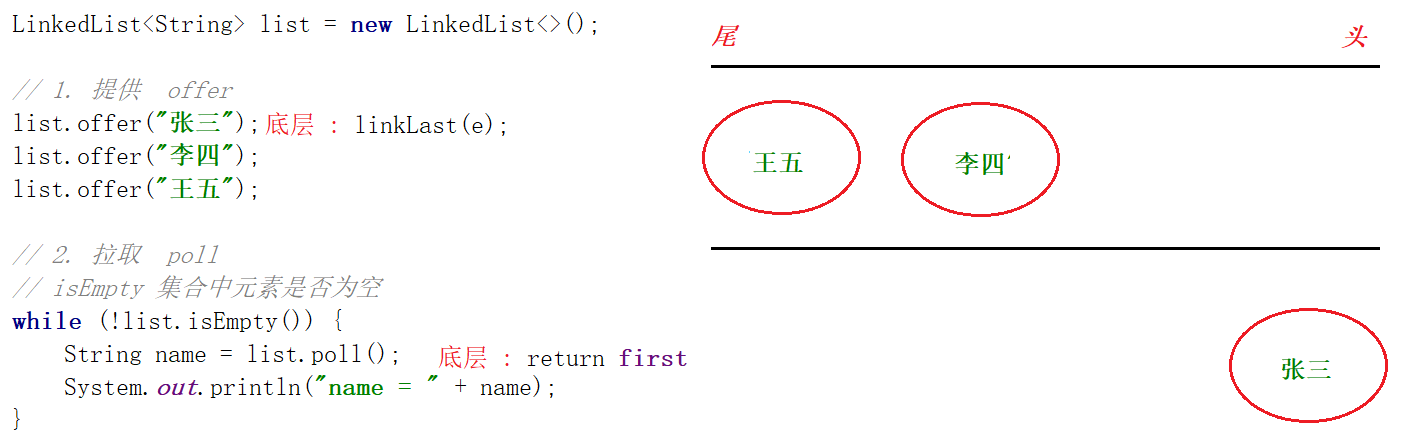
public class Test3 {  
 public static void main(String[] args) {  
  
 *// 栈结构 :  
  
 // 创建一个 `双链表` 数据结构 (LinkedList)* LinkedList<String> list = new LinkedList<>();  
  
 *// 1. 压栈 push* list.push("张三");  
 list.push("李四");  
 list.push("王五");  
  
 *// 2. 弹栈 pop  
 // isEmpty 判断集合是否为空* while (list.isEmpty() == false) {  
 String name = list.pop();  
 System.*out*.println("name = " + name);  
 }  
 }  
}



## 10.5 队列结构 : offer -> poll

队列结构 : 先进先出

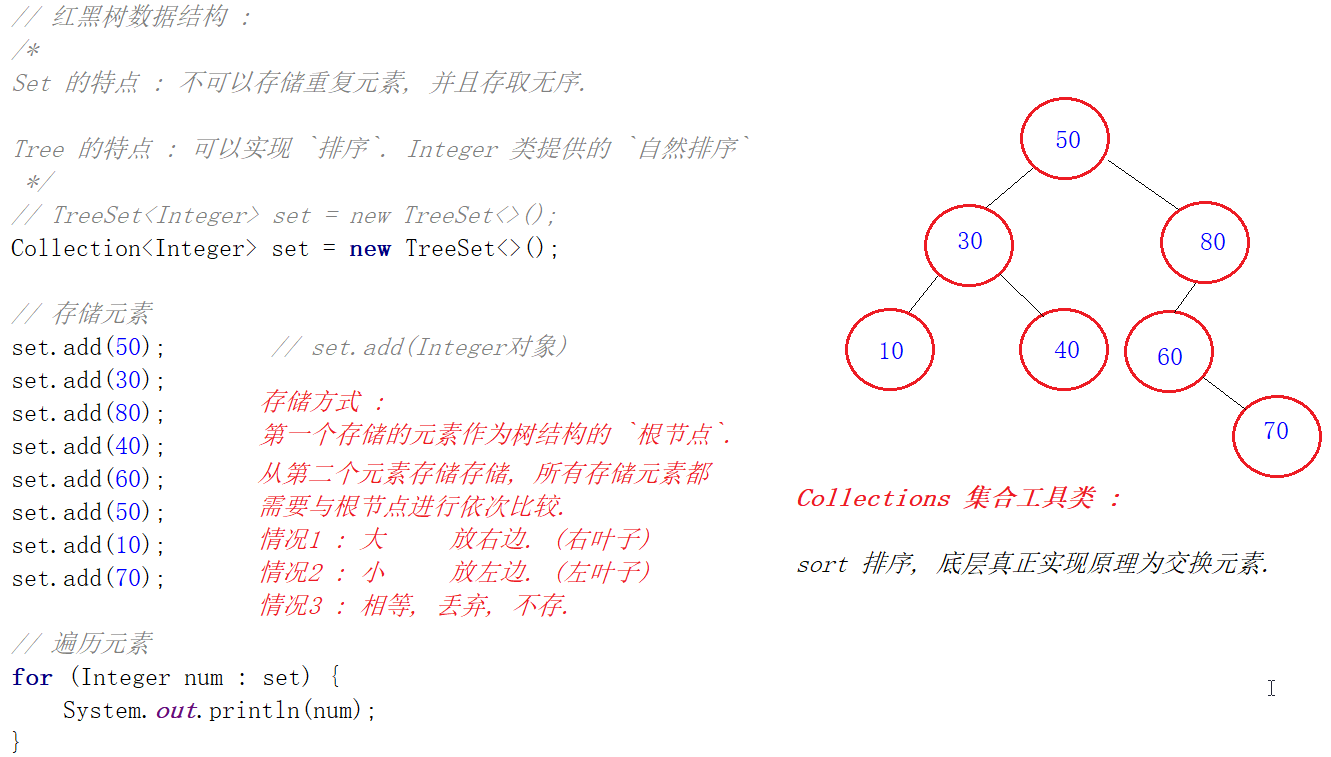
public class Test4 {  
 public static void main(String[] args) {  
  
 LinkedList<String> list = new LinkedList<>();  
  
 *// 1. 提供 offer* list.offer("张三");  
 list.offer("李四");  
 list.offer("王五");  
  
 *// 2. 拉取 poll  
 // isEmpty 集合中元素是否为空* while (!list.isEmpty()) {  
 String name = list.poll();  
 System.*out*.println("name = " + name);  
 }  
 }  
}



## 10.6 红黑树结构 : 排序

TreeSet : 底层就是 `红黑树 / 二叉树` 数据结构.

// 红黑树数据结构 :  
/\*  
Set 的特点 : 不可以存储重复元素, 并且存取无序.  
  
Tree 的特点 : 可以实现 `排序`. Integer 类提供的 `自然排序`  
 \*/  
// TreeSet<Integer> set = new TreeSet<>();  
Collection<Integer> set = new TreeSet<>();  
  
// 存储元素  
set.add(50); // set.add(Integer对象)  
set.add(30);  
set.add(80);  
set.add(40);  
set.add(60);  
set.add(50);  
set.add(10);  
set.add(70);  
  
// 遍历元素  
for (Integer num : set) {  
 System.out.println(num);  
}



# 11. Set 接口 :

Set 接口的特点 :

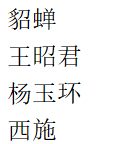
1. 存取无序. (迭代无序) hashCode 方法.

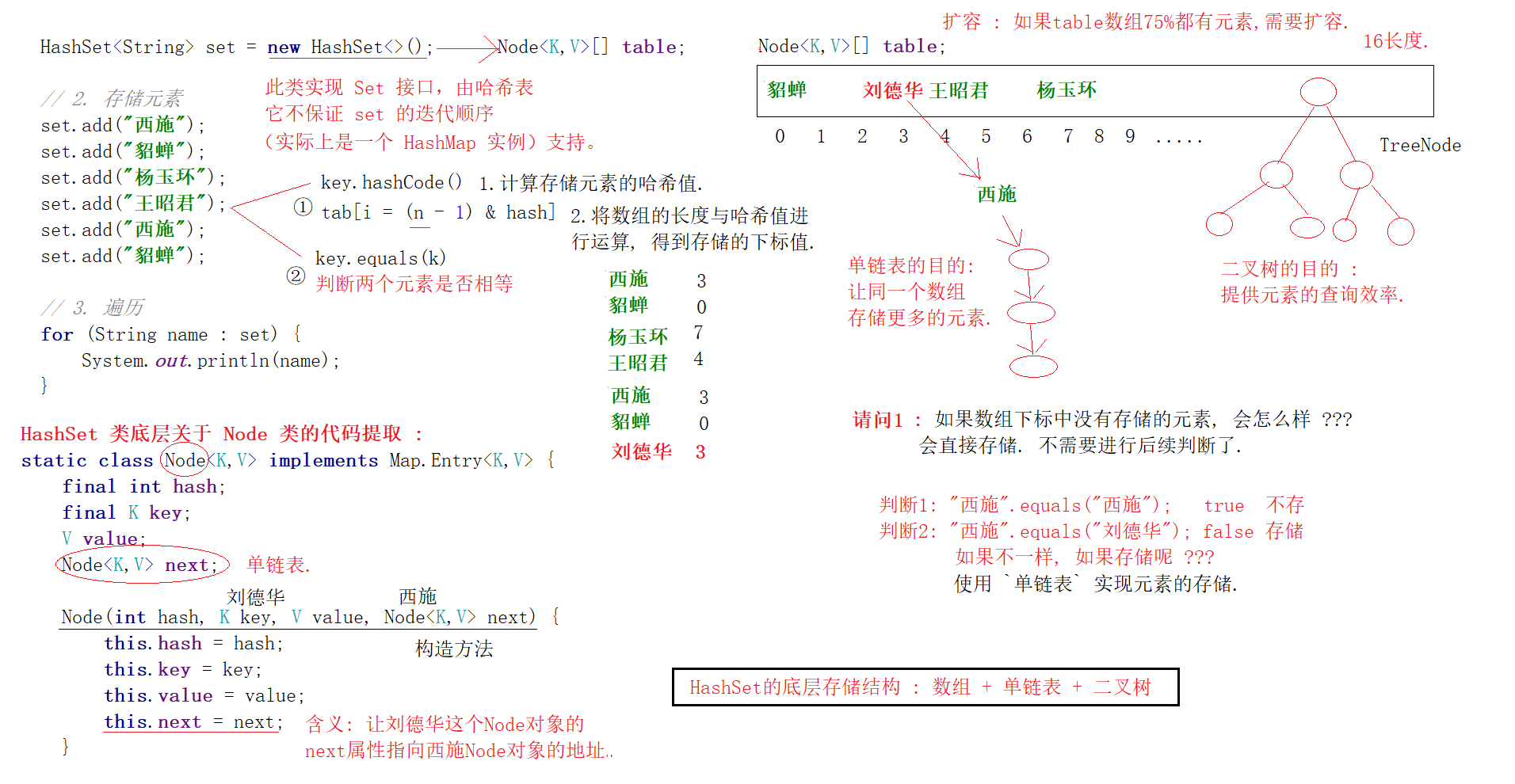
2. 元素不可重复. equals 方法

## 11.1 HashSet 实现类 : (重点)



public class Test2 {  
 public static void main(String[] args) {  
  
 // 1. 创建一个 `HashSet` 哈希无序集合对象  
 /\*  
 Set 特点 :  
 1. 存储无序. 底层是需要根据存储元素的哈希值 hashCode 方法进行下标计算的.  
 2. 元素不可重复. 底层是通过两个元素的 equals 方法判断实现的.  
 \*/  
 HashSet<String> set = new HashSet<>();  
  
 // 2. 存储元素  
 set.add("西施");  
 set.add("貂蝉");  
 set.add("杨玉环");  
 set.add("王昭君");  
 set.add("西施");  
 set.add("貂蝉");  
  
 // 3. 遍历  
 for (String name : set) {  
 System.out.println(name);  
 }  
 }  
}





## 11.2 需求 : 存储自定义对象 (Student对象)

**小结 :** 如果往 `哈希表` 中存储自定义对象, 那么该集合中的自定义对象必须重写 Object 类的 `hashCode + equals` 方法.

问题1 : HashSet 是无序不可重复集合. 既然是这样, 为什么存储 Student 对象出现了元素重复 ???

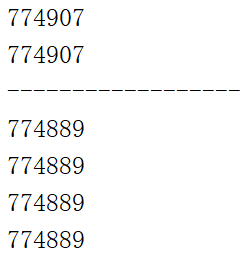
验证1 : 两个完全相同数据的 Student 对象, 计算出来的哈希值是否相等 ???

**Student 类重写 Object 类的 hashCode 方法 :**

// 重写 Object 类的 hashCode 方法.  
// 原因 : 如果两个对象的数据属性相同, 那么这两个对象计算出来的哈希值就应该相同.  
@Override  
public int hashCode() {  
 // 根据对象的属性数据计算 (name, age)  
 // 18 + 774889 -> 774907  
 return age + name.hashCode();  
}

**测试类 :**

public class Test4 {  
 public static void main(String[] args) {  
  
 // Student 对象数据相同, 哈希值不相同. 这样的话, 存储下标就非常可能计算出来是不相等.  
 // Object 类的 hashCode 方法 :  
 // 实际上，由 Object 类定义的 hashCode 方法确实会针对不同的对象返回不同的整数。  
 // （这一般是通过将该对象的内部地址转换成一个整数来实现的)  
  
 // 需求 : 根据 Student 对象的数据内容来计算哈希值  
 Student stu1 = new Student("张三", 18);  
 Student stu2 = new Student("张三", 18);  
  
 System.out.println(stu1.hashCode());  
 System.out.println(stu2.hashCode());  
  
 System.out.println("------------------");  
  
 // 原因 : String 重写 hashCode 方法, 根据字符串内容来计算哈希值.  
 int hashCode1 = "张三".hashCode();  
 int hashCode2 = "张三".hashCode();  
 int hashCode3 = new String("张三").hashCode();  
 int hashCode4 = new String("张三").hashCode();  
 System.out.println(hashCode1);  
 System.out.println(hashCode2);  
 System.out.println(hashCode3);  
 System.out.println(hashCode4);  
 }  
}

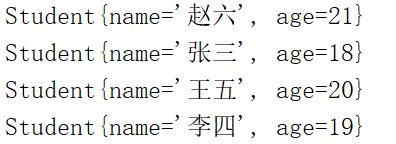


**Student 类重写 Object 类的 hashCode 和 equals 方法 :**

import java.util.Objects;  
  
public class Student {  
 private String name;  
 private int age;  
  
 public Student(String name, int age) {  
 this.name = name;  
 this.age = age;  
 }  
  
 public Student() {  
 }  
  
 @Override  
 public String toString() {  
 return "Student{" +  
 "name='" + name + '\'' +  
 ", age=" + age +  
 '}';  
 }  
  
 *// 重写 Object 类的 hashCode 方法.  
 // 原因 : 如果两个对象的数据属性相同, 那么这两个对象计算出来的哈希值就应该相同.* @Override  
 public int hashCode() {  
 *// 根据对象的属性数据计算 (name, age)  
 // 18 + 774889 -> 774907  
 // return age + name.hashCode();* return Objects.*hash*(name, age);  
 }  
  
 *// 重写 Object 类的 equals 方法.  
 // 原因 : 如果两个对象的下标存储一致, 需要判断两个对象是否为同一个对象.* @Override  
 public boolean equals(Object o) {  
 if (this == o) return true;  
 if (o == null || o.getClass() != getClass()) return false;  
 Student stu = (Student) o;  
 return stu.age == age && Objects.*equals*(name, stu.name);  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 public int getAge() {  
 return age;  
 }  
  
 public void setAge(int age) {  
 this.age = age;  
 }  
}

**测试类 :**

public class Test3 {  
 public static void main(String[] args) {  
  
 *// 需求 : 向 Set 集合中存储 Student 自定义对象, 要保证 `学生对象` 不重复.  
 // 1. 创建一个 `HashSet` 哈希无序集合对象* HashSet<Student> set = new HashSet<>();  
  
 *// 2. 添加元素* set.add(new Student("张三", 18));  
 set.add(new Student("李四", 19));  
 set.add(new Student("王五", 20));  
 set.add(new Student("赵六", 21));  
 set.add(new Student("张三", 18));  
 set.add(new Student("李四", 19));  
  
 *// 3. 遍历* for (Student stu : set) {  
 System.*out*.println(stu);  
 }  
 }  
}



## 11.2 LinkedHashSet 实现类 : (了解)

**说明 :** 具有**可预知迭代顺序**的 Set 接口的**哈希表**和**链接列表**实现。

特点 :

1. 迭代有序.

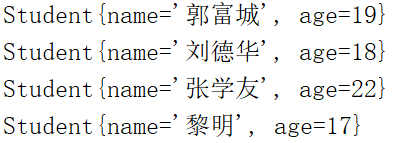
2. 元素唯一.

**Student 类定义 :**

public class Student {  
 private String name;  
 private int age;  
  
 public Student(String name, int age) {  
 this.name = name;  
 this.age = age;  
 }  
  
 public Student() {  
 }  
  
 *// 重写 hashCode 方法, 保证对象属性数据一致, 哈希值就一致* @Override  
 public int hashCode() {  
 *// return age + name.hashCode();* return Objects.*hash*(age, name);  
 }  
  
 *// 重写 equals 方法, 保证对象属性一致, 这两个对象就相等.* @Override  
 public boolean equals(Object o) {  
 if (this == o) return true;  
 if (o == null || o.getClass() != getClass()) return false;  
 Student stu = (Student) o;  
 return age == stu.age && Objects.*equals*(name, stu.name);  
 }  
}

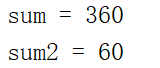
**测试类 :**

import java.util.HashSet;  
import java.util.LinkedHashSet;  
  
public class Test1 {  
 public static void main(String[] args) {  
  
 *// `哈希表` 保证元素唯一的实现方案 : 底层会自动调用元素的 `hashCode` 计算下标, 如果下标一致.  
 // 紧接着继续调用元素的 equals 方法判断两个元素是否相等.  
  
 // HashSet 集合迭代无序.  
 // HashSet<Student> set = new HashSet<Student>();  
  
 // LinkedHashSet 集合迭代有序.* HashSet<Student> set = new LinkedHashSet<Student>();  
  
 set.add(new Student("郭富城", 19));  
 set.add(new Student("刘德华", 18));  
 set.add(new Student("刘德华", 18));  
 set.add(new Student("张学友", 22));  
 set.add(new Student("黎明", 17));  
 set.add(new Student("张学友", 22));  
  
 for (Student stu : set) {  
 System.*out*.println(stu);  
 }  
 }  
}



# 12. 可变参数 : (了解)

public class Test2 {  
 public static void main(String[] args) {  
  
 *// 可变参数 : ...* int[] arr = {10, 20, 30, 40, 50, 60, 70, 80};  
 int sum = *getSum*(arr);  
 System.*out*.println("sum = " + sum);  
  
 int sum2 = *getSum*(10, 20, 30);  
 System.*out*.println("sum2 = " + sum2);  
 }  
  
 *// 需求 : 计算多个元素的累加和  
 // 理解 : 可变参数的底层就是数组. 如果传入的参数是单个单个存在, 底层自动将参数转换为数组.  
 // 要求 : 如果参数列表存在多个参数, 那么可变参数必须是参数列表的最后一个参数.  
 // vararg (Variable argument) must be the last in the list* public static int getSum(int... array) {  
 int sum = 0;  
 for (int i = 0; i < array.length; i++) {  
 sum += array[i];  
 }  
 return sum;  
 }  
  
 */\*public static int getSum(int[] array) {  
 int sum = 0;  
 for (int i = 0; i < array.length; i++) {  
 sum += array[i];  
 }  
 return sum;  
 }\*/*}



# 13. Collections 集合工具类 : (重点)

Arrays.**toString**(数组); 将数组中的元素逐个输出显示.

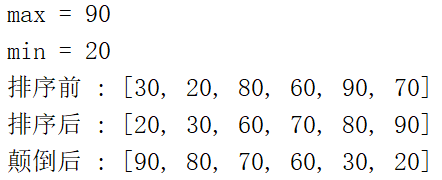
## 13.1 常用方法 :

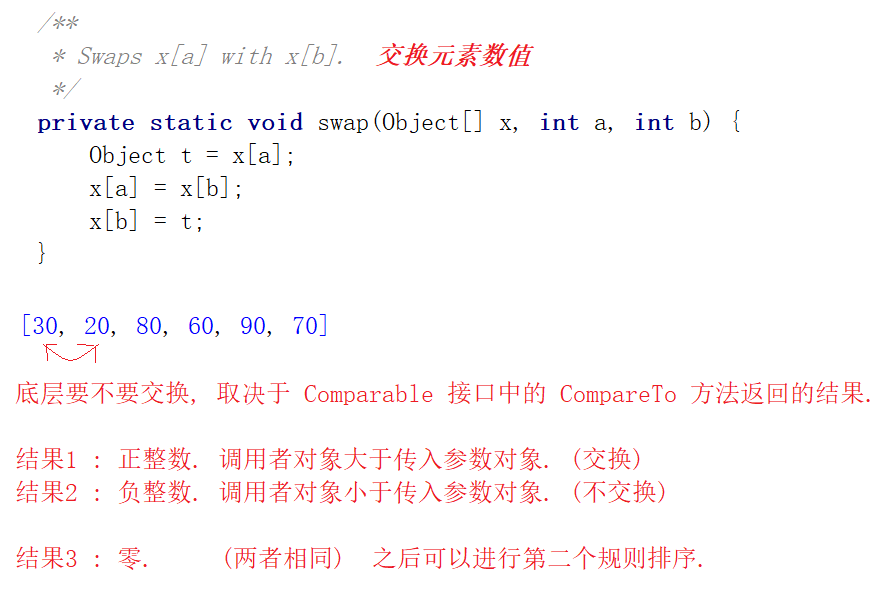
**addAll 添加所有 :**

public class Test3 {  
 public static void main(String[] args) {  
  
 // Collections 集合工具类 :  
  
 // ArrayList 与 Collection 是什么关系呢 ??? 实现关系.  
 // ArrayList implements List extends Collection.  
 ArrayList<String> list = new ArrayList<>();  
  
 // boolean addAll(Collection<? super T> c, T... elements) 第一个参数为 Collection 接口类型.  
 // 如果参数为接口类型, 其真正需要的是什么 ??? 其真正需要的是该接口的实现类对象.  
 Collections.addAll(list, "Jack", "Rose", "Peter", "Lucy", "Lily");  
  
 System.out.println(list);  
 }  
}



public class Test4 {  
 public static void main(String[] args) {  
  
 ArrayList<Integer> numbers = new ArrayList<>();  
  
 *// addAll 添加所有* Collections.*addAll*(numbers, 30, 20, 80, 60, 90, 70);  
  
 *// max 最大值* Integer max = Collections.*max*(numbers);  
 System.*out*.println("max = " + max);  
  
 *// min 最小值* Integer min = Collections.*min*(numbers);  
 System.*out*.println("min = " + min);  
  
 *// sort 排序* System.*out*.println("排序前 : " + numbers);  
 Collections.*sort*(numbers); *// Integer 整型数值的排序, 从小到大.* System.*out*.println("排序后 : " + numbers);  
  
 *// reverse* Collections.*reverse*(numbers);  
 System.*out*.println("颠倒后 : " + numbers);  
 }  
}





## 13.2 Comparable 自然排序 :

**说明 : 此接口强行对实现它的每个类的对象进行整体排序。这种排序被称为类的自然排序，类的 compareTo 方法被称为它的自然比较方法。**

**字符串的自然排序 : 字典顺序 (a ~ z)**

public class Test5 {  
 public static void main(String[] args) {  
  
 *// 请问 : 字符串能够排序吗 ???* ArrayList<String> list = new ArrayList<>();  
 list.add("England");  
 list.add("America");  
 list.add("India");  
 list.add("China");  
 list.add("French");  
 list.add("Canada");  
  
 Collections.*sort*(list);  
  
 *// 结果 ??? 可以, 根据 `字典` 排序.* for (String s : list) {  
 System.*out*.println(s);  
 }  
 }  
}



Comparable 是**自然排序.**

public final class Integer extends Number implements Comparable<Integer>

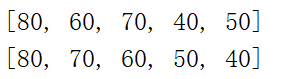
public final class String implements java.io.Serializable, Comparable<String>, CharSequence

**结论 :** 不能自定义类继承 Integer, String, 因为这两个类是 final 修饰的类, 最终类, 因此就不能重写 compareTo 方法.

## 13.3 Comparator 比较器排序 :

**演示一 :** Integer 类的自然排序为 `从小到大`. 希望重新定义为 `**从大到小**` ???

import java.util.ArrayList;  
import java.util.Collections;  
import java.util.Comparator;  
  
public class Test10 {  
 public static void main(String[] args) {  
  
 *// 需求 : 对 Integer 元素进行 `从大到小` 的排序.* ArrayList<Integer> list = new ArrayList<>();  
 list.add(80);  
 list.add(60);  
 list.add(70);  
 list.add(40);  
 list.add(50);  
 System.*out*.println(list);  
  
 *// sort : Comparable 自然排序  
 // Collections.sort(list);  
  
 // sort : Comparator 第二个参数是一个接口* Collections.*sort*(list, new Comparator<Integer>() {  
 @Override  
 public int compare(Integer o1, Integer o2) {  
 *// return -o1.compareTo(o2);* return o2.compareTo(o1);  
 }  
 });  
  
 System.*out*.println(list);  
  
 }  
}



**演示二 :** String 类的自定排序为 `从a~z`, 希望从 `z~a`, 或者根据字符串的长度来排序 ???

public class Test11 {  
 public static void main(String[] args) {  
  
 ArrayList<String> list = new ArrayList<>();  
 list.add("England");  
 list.add("America");  
 list.add("India");  
 list.add("China");  
 list.add("French");  
 list.add("Canada");

*// sort : Comparator 第二个参数是一个接口*  
 Collections.*sort*(list, new Comparator<String>() {  
 @Override  
 public int compare(String o1, String o2) {  
 *// 方式一 : 从 z 到 a  
 // return o2.compareTo(o1);  
  
 // 方式二 : 按照长度来排序* int result = o1.length() - o2.length();  
  
 *// 说明 : 如果长度一致, 按照字符串的自然排序* if (result == 0) {  
 result = o1.compareTo(o2);  
 }  
  
 return result;  
 }  
 });  
  
 for (String name : list) {  
 System.*out*.println(name);  
 }  
 }  
}



## 13.4 练习 - 自定义对象的排序 :

**说明 :** 创建一个学生类，存储到ArrayList集合中完成指定排序操作。

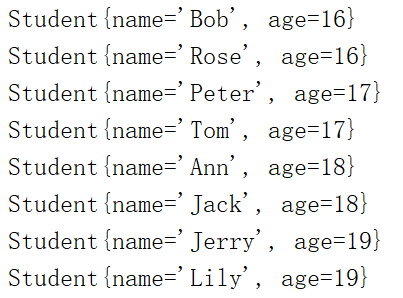
**实现方式一 : 自然排序 (Comparable)**

**Student 类 :**

public class Student implements Comparable<Student> {  
 private String name;  
 private int age;  
  
 public Student(String name, int age) {  
 this.name = name;  
 this.age = age;  
 }  
  
 public Student() {  
 }  
  
 *// 重写 hashCode 方法, 保证对象属性数据一致, 哈希值就一致* @Override  
 public int hashCode() {  
 *// return age + name.hashCode();* return Objects.*hash*(age, name);  
 }  
  
 *// 重写 equals 方法, 保证对象属性一致, 这两个对象就相等.* @Override  
 public boolean equals(Object o) {  
 if (this == o) return true;  
 if (o == null || o.getClass() != getClass()) return false;  
 Student stu = (Student) o;  
 return age == stu.age && Objects.*equals*(name, stu.name);  
 }  
  
 @Override  
 public String toString() {  
 return "Student{" +  
 "name='" + name + '\'' +  
 ", age=" + age +  
 '}';  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 public int getAge() {  
 return age;  
 }  
  
 public void setAge(int age) {  
 this.age = age;  
 }  
  
 @Override  
 public int compareTo(Student o) {  
 *// 1. 根据年龄比较* int result = this.age - o.age; *// 18 - 18  
 // 2. 如果年龄相同, 按照对象的姓名进行自然排序* if (result == 0) {  
 result = this.name.compareTo(o.name);  
 }  
 return result;  
 }  
}

**测试类 :**

public class Test6 {  
 public static void main(String[] args) {  
  
 *// 定义一个集合, 存储自定义对象* ArrayList<Student> list = new ArrayList<>();  
  
 list.add(new Student("Jack", 18));  
 list.add(new Student("Rose", 16));  
 list.add(new Student("Peter", 17));  
 list.add(new Student("Lily", 19));  
 list.add(new Student("Ann", 18));  
 list.add(new Student("Bob", 16));  
 list.add(new Student("Tom", 17));  
 list.add(new Student("Jerry", 19));  
  
 *// 自然排序 ??? 要求集合中的自定义类实现 Comparable 接口.  
 // Student conforms to Comparable, 学生类要遵守 Comparable 接口.* Collections.*sort*(list);  
 for (Student stu : list) {  
 System.*out*.println(stu);  
 }  
  
 }  
}



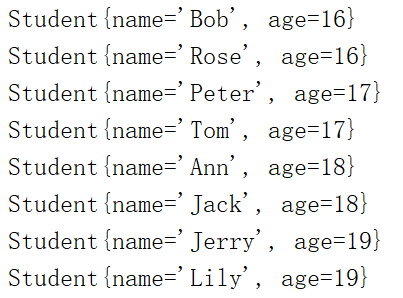
**实现方式二 : 比较器排序**

**Student 类 :**

public class Student {  
 private int stuId;  
 private int score;  
 private int age;  
  
 public Student(int stuId, int score, int age) {  
 this.stuId = stuId;  
 this.score = score;  
 this.age = age;  
 }  
  
 @Override  
 public String toString() {  
 return "Student{" +  
 "stuId='" + stuId + '\'' +  
 ", score=" + score +  
 ", age=" + age +  
 '}';  
 }  
  
 public int getStuId() {  
 return stuId;  
 }  
  
 public void setStuId(int stuId) {  
 this.stuId = stuId;  
 }  
  
 public int getScore() {  
 return score;  
 }  
  
 public void setScore(int score) {  
 this.score = score;  
 }  
  
 public int getAge() {  
 return age;  
 }  
  
 public void setAge(int age) {  
 this.age = age;  
 }  
}

**测试类 :**

public class Test6 {  
 public static void main(String[] args) {  
  
 *// 定义一个集合, 存储自定义对象* ArrayList<Student> list = new ArrayList<>();  
  
 list.add(new Student("Jack", 18));  
 list.add(new Student("Rose", 16));  
 list.add(new Student("Peter", 17));  
 list.add(new Student("Lily", 19));  
 list.add(new Student("Ann", 18));  
 list.add(new Student("Bob", 16));  
 list.add(new Student("Tom", 17));  
 list.add(new Student("Jerry", 19));  
  
 *// 比较器排序 ??? 要求调用 sort 方法时, 传入第二个 Comparator 接口类型的参数* Collections.*sort*(list, new Comparator<Student>() {  
 @Override  
 public int compare(Student o1, Student o2) {  
 *// 年龄* int result = o1.getAge() - o2.getAge();  
 *// 姓名* if (result == 0) {  
 result = o1.getName().compareTo(o2.getName());  
 }  
 return result;  
 }  
 });  
 for (Student stu : list) {  
 System.*out*.println(stu);  
 }  
 }  
}



**请问 :** 集合中存储自定义对象, 两种比较都可以实现, 如何选择呢 ????

**需求 :** 如果自定义对象的**比较规则基本不变**的话, 应该选择 `自然排序`, 因为自然是自定义类实现 Comparable 接口. 调用 sort 方法无需进行第二个参数传递.

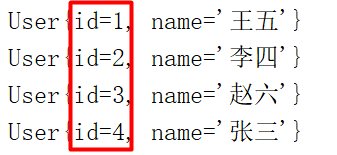
**需求 :** 如果自定义对象**比较规则灵活多变**, 建议使用 `比较器排序 Compartor 接口`, 每次排序, 通过 sort 方法的第二个参数来确定规则.

**第一个类 :** User 用户类, 该用户类排序规则基本工具, 根据用户的**编号排序**.

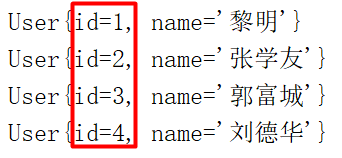
public class User implements Comparable<User> {  
 private int id;  
 private String name;  
  
 @Override  
 public int compareTo(User o) {  
 return this.id - o.id;  
 }  
  
 public User(int id, String name) {  
 this.id = id;  
 this.name = name;  
 }  
  
 @Override  
 public String toString() {  
 return "User{" +  
 "id=" + id +  
 ", name='" + name + '\'' +  
 '}';  
 }  
  
 public int getId() {  
 return id;  
 }  
  
 public void setId(int id) {  
 this.id = id;  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
}

**测试类编写 :**

public class Test1 {  
 public static void main(String[] args) {  
  
 *// 需求 : 为 User 排序* ArrayList<User> list = new ArrayList<>();  
  
 list.add(new User(4, "张三"));  
 list.add(new User(2, "李四"));  
 list.add(new User(1, "王五"));  
 list.add(new User(3, "赵六"));  
  
 Collections.*sort*(list);  
 */\*Collections.sort(list, new Comparator<User>() {  
 @Override  
 public int compare(User o1, User o2) {  
 return o1.getId() - o2.getId();  
 }  
 });\*/* for (User user : list) {  
 System.*out*.println(user);  
 }  
 }  
}



public class Test2 {  
 public static void main(String[] args) {  
  
 *// 需求 : 为 User 排序* ArrayList<User> list = new ArrayList<>();  
  
 list.add(new User(4, "刘德华"));  
 list.add(new User(2, "张学友"));  
 list.add(new User(1, "黎明"));  
 list.add(new User(3, "郭富城"));  
  
 Collections.*sort*(list);  
 */\*Collections.sort(list, new Comparator<User>() {  
 @Override  
 public int compare(User o1, User o2) {  
 return o1.getId() - o2.getId();  
 }  
 });\*/* for (User user : list) {  
 System.*out*.println(user);  
 }  
 }  
}

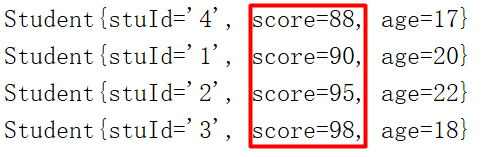


**第二个类 :** Student 类, 该类拥有 (stuId, Score, age …), 有时, 该学生需要根据**编号排序**, 也有可能需要根据**成绩排序**, 还有可能根据**年龄排序**)

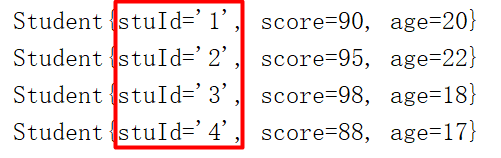
public class Student {  
 private int stuId;  
 private int score;  
 private int age;  
  
 public Student(int stuId, int score, int age) {  
 this.stuId = stuId;  
 this.score = score;  
 this.age = age;  
 }  
  
 @Override  
 public String toString() {  
 return "Student{" +  
 "stuId='" + stuId + '\'' +  
 ", score=" + score +  
 ", age=" + age +  
 '}';  
 }  
  
 public int getStuId() {  
 return stuId;  
 }  
  
 public void setStuId(int stuId) {  
 this.stuId = stuId;  
 }  
  
 public int getScore() {  
 return score;  
 }  
  
 public void setScore(int score) {  
 this.score = score;  
 }  
  
 public int getAge() {  
 return age;  
 }  
  
 public void setAge(int age) {  
 this.age = age;  
 }  
}

**测试类编写 :**

public class Test3 {  
 public static void main(String[] args) {  
  
 ArrayList<Student> list = new ArrayList<>();  
  
 list.add(new Student(3, 98, 18));  
 list.add(new Student(4, 88, 17));  
 list.add(new Student(1, 90, 20));  
 list.add(new Student(2, 95, 22));  
  
 *// 根据成绩 ???* Collections.*sort*(list, new Comparator<Student>() {  
 @Override  
 public int compare(Student o1, Student o2) {  
 return o1.getScore() - o2.getScore();  
 }  
 });  
  
 for (Student stu : list) {  
 System.*out*.println(stu);  
 }  
 }  
}

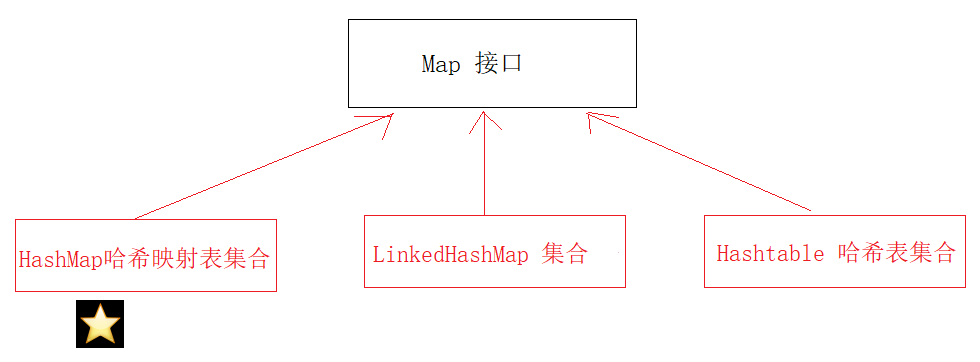


public class Test4 {  
 public static void main(String[] args) {  
  
 ArrayList<Student> list = new ArrayList<>();  
  
 list.add(new Student(3, 98, 18));  
 list.add(new Student(4, 88, 17));  
 list.add(new Student(1, 90, 20));  
 list.add(new Student(2, 95, 22));  
  
 *// 根据学号 ???* Collections.*sort*(list, new Comparator<Student>() {  
 @Override  
 public int compare(Student o1, Student o2) {  
 return o1.getStuId() - o2.getStuId();  
 }  
 });  
   
 for (Student stu : list) {  
 System.*out*.println(stu);  
 }  
 }  
}



# 14. Map 映射表集合 : (重要)

**作用 : 通过 key 找 value.**



**接口 Map<K,V> 映射表 / 键值对**

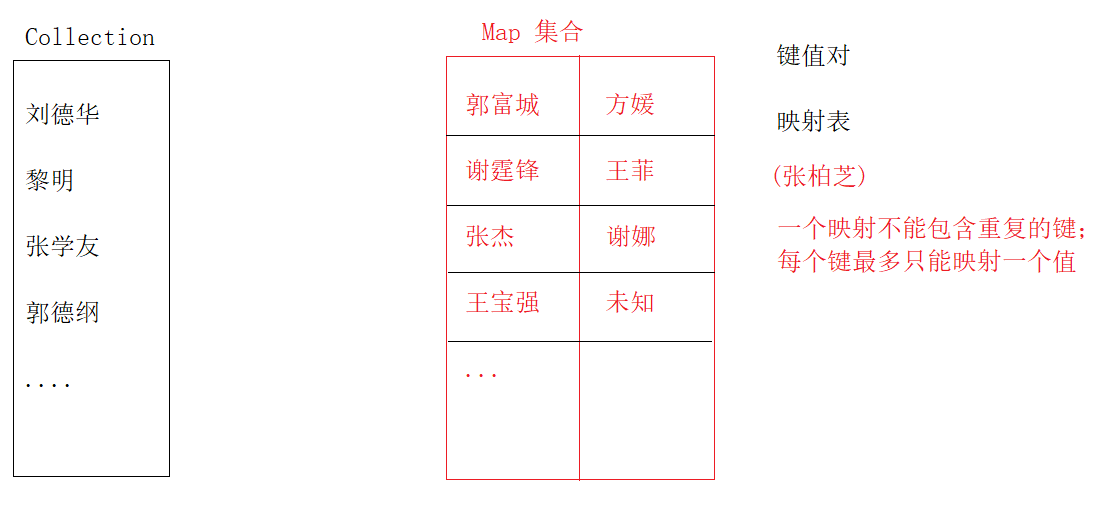
**类型参数：**

K - 此映射所维护的键的类型 key

V - 映射值的类型 value

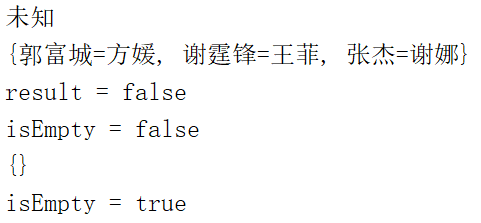
**说明 :** 将键映射到值的对象。一个映射不能包含重复的键；每个键最多只能映射一个值。此接口取代 Dictionary 类





## 14.1 Map 集合中的常用方法 :

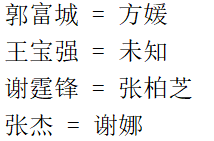
public class Test1 {  
 public static void main(String[] args) {  
  
 *// 1. 定义一个 Map 集合.* HashMap<String, String> map = new HashMap<>();  
  
 *// 增加 : put* map.put("郭富城", "方媛");  
 map.put("谢霆锋", "张柏芝");  
 map.put("张杰", "谢娜");  
 map.put("王宝强", "未知");  
  
 *// 删除 : remove* String value = map.remove("王宝强");  
 System.*out*.println(value);  
  
 *// 修改 : put* map.put("谢霆锋", "王菲"); *// 如果 key 重复, 新值就会覆盖旧值* System.*out*.println(map); *// 重写 toString();  
  
 // containsKey 判断是否存在指定 key* boolean result = map.containsKey("王宝强");  
 System.*out*.println("result = " + result); *// false  
  
 // isEmpty* boolean isEmpty = map.isEmpty();  
 System.*out*.println("isEmpty = " + isEmpty);  
  
 *// clear* map.clear();  
 System.*out*.println(map);  
  
 isEmpty = map.isEmpty();  
 System.*out*.println("isEmpty = " + isEmpty);  
 }  
}



## 14.2 Map 集合遍历 – 键找值方式

**keySet + get 方式 :**

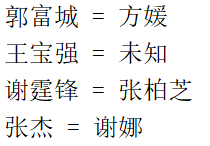
public class Test2 {  
 public static void main(String[] args) {  
  
 *// 1. 定义一个 Map 集合.* HashMap<String, String> map = new HashMap<>();  
  
 map.put("郭富城", "方媛");  
 map.put("谢霆锋", "张柏芝");  
 map.put("张杰", "谢娜");  
 map.put("王宝强", "未知");  
  
 *// keySet 获取 map 映射表中所有 key 的集合, 返回 Set 集合, 因为 key 不能重复* Set<String> keySet = map.keySet();  
 for (String key : keySet) {  
 *// get 全部查询* String value = map.get(key);  
 System.*out*.println(key + " = " + value);  
 }  
 }  
}

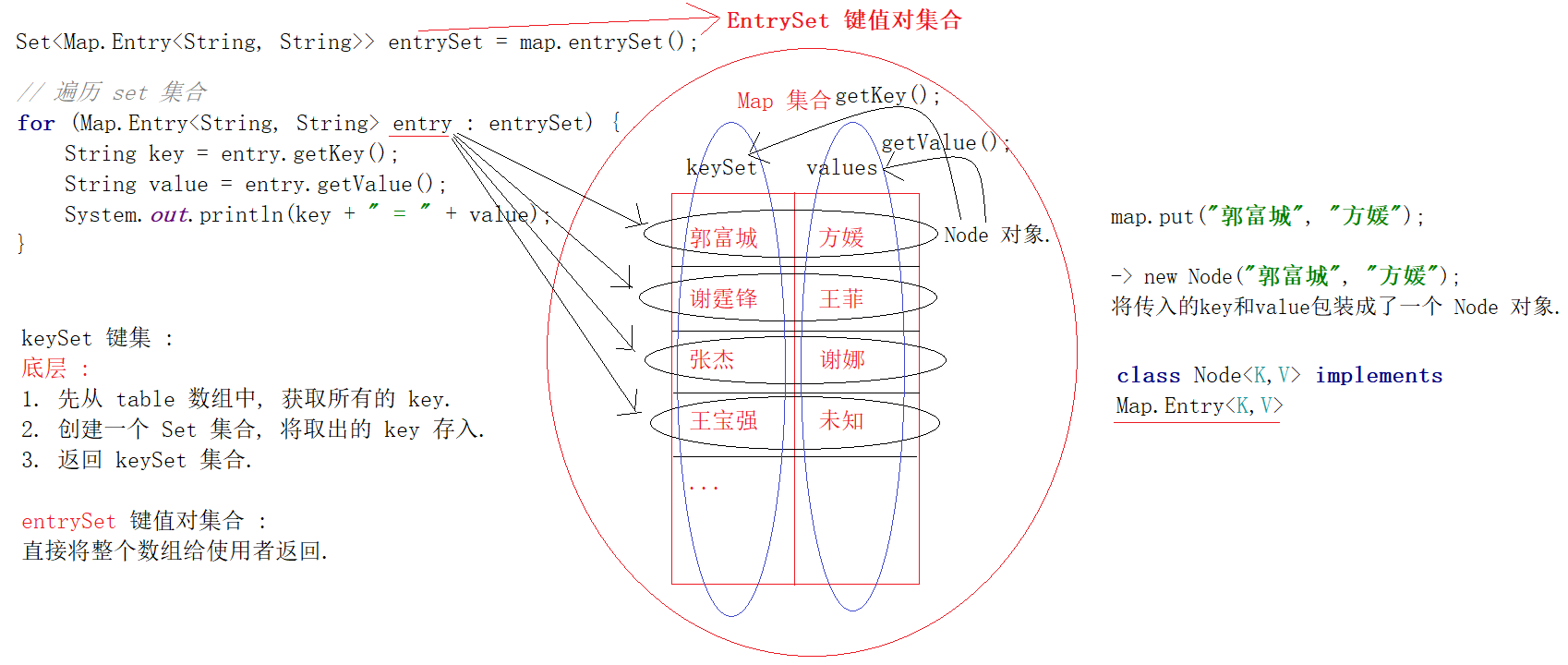


## 14.3 Map 集合遍历 – 键值对方式

**entrySet 方法获取 :**

public class Test3 {  
 public static void main(String[] args) {  
  
 *// 1. 定义一个 Map 集合.* HashMap<String, String> map = new HashMap<>();  
  
 map.put("郭富城", "方媛");  
 map.put("谢霆锋", "张柏芝");  
 map.put("张杰", "谢娜");  
 map.put("王宝强", "未知");  
  
 *// Set<Map.Entry<K,V>> entrySet() 返回此映射中包含的映射关系的 Set 视图。  
 // 思考1 : entrySet 返回的是一个 set 集合. 因为键值对对象不可以重复.  
 // 思考2 : Set 集合中存储的是什么对象 ??? Node 对象, 而Node对象在外部的名称为 Map.Entry<K, V>.  
 // Set<Map.Entry<String, String>> entrySet = map.entrySet();  
 // 遍历 set 集合* for (Map.Entry<String, String> entry : map.entrySet()) {  
 String key = entry.getKey();  
 String value = entry.getValue();  
 System.*out*.println(key + " = " + value);  
 }  
 }  
}





## 14.4 HashMap存储自定义类型键:

new Student(“张三”, 18); -> 上海 说明: key 的类型为 Student 类型, Value 的类型为 String 类型.

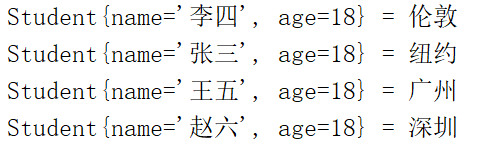
**小结 :** 如果自定义对象作为 `Map` 的 key, 那么自定义对象所属的类必须要重写 `hashCode + equals` 方法.

**Student 类 :**

public class Student {  
 private String name;  
 private int age;  
  
 public Student(String name, int age) {  
 this.name = name;  
 this.age = age;  
 }  
  
 public Student() {  
 }  
  
 *// 重写 hashCode 方法* @Override  
 public int hashCode() {  
 return Objects.*hash*(age, name);  
 }  
  
 *// 重写 equals 方法* @Override  
 public boolean equals(Object o) {  
 if (this == o) return true;  
 if (o == null || o.getClass() != getClass()) return false;  
 Student stu = (Student) o;  
 return stu.age == age && Objects.*equals*(name, stu.name);  
 }  
  
 @Override  
 public String toString() {  
 return "Student{" +  
 "name='" + name + '\'' +  
 ", age=" + age +  
 '}';  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 public int getAge() {  
 return age;  
 }  
  
 public void setAge(int age) {  
 this.age = age;  
 }  
}

**测试类 :**

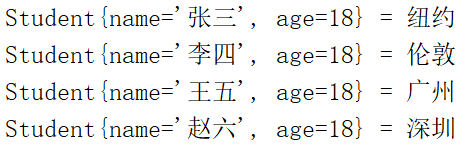
public class Test1 {  
 public static void main(String[] args) {  
  
 *// 1. 创建一个 `映射表` 集合对象  
 // Map 集合仅根据 key 来计算存储下标. 如果 key 为自定义对象, 那么自定义对象必须重写 `hashCode + equals` 方法.* HashMap<Student, String> map = new HashMap<>();  
  
 *// 2. 存储数据* map.put(new Student("张三", 18), "上海");  
 map.put(new Student("李四", 18), "北京");  
 map.put(new Student("王五", 18), "广州");  
 map.put(new Student("赵六", 18), "深圳");  
 map.put(new Student("张三", 18), "纽约");  
 map.put(new Student("李四", 18), "伦敦");  
  
 *// 3. 遍历* Set<Map.Entry<Student, String>> entrySet = map.entrySet();  
 for (Map.Entry<Student, String> entry : entrySet) {  
 Student key = entry.getKey();  
 String value = entry.getValue();  
 System.*out*.println(key + " = " + value);  
 }  
 }



## 14.5 LinkedHashMap 链接哈希映射表集合 (了解) :

**说明 :** Map 接口的哈希表和**链接**列表实现，具有**可预知的迭代顺序**。

public class Test3 {  
 public static void main(String[] args) {  
  
 *// 1. 创建一个 `链接映射表` 集合对象  
 // 特点 : 可预知的迭代顺序  
 // Map 主要的功能为 `根据键, 寻找对应的数据`* HashMap<Student, String> map = new LinkedHashMap<>();  
  
 *// 2. 存储数据* map.put(new Student("张三", 18), "上海");  
 map.put(new Student("李四", 18), "北京");  
 map.put(new Student("王五", 18), "广州");  
 map.put(new Student("赵六", 18), "深圳");  
 map.put(new Student("张三", 18), "纽约");  
 map.put(new Student("李四", 18), "伦敦");  
  
 *// 3. 遍历* Set<Map.Entry<Student, String>> entrySet = map.entrySet();  
 for (Map.Entry<Student, String> entry : entrySet) {  
 Student key = entry.getKey();  
 String value = entry.getValue();  
 System.*out*.println(key + " = " + value);  
 }  
 }  
}



## 14.6 Map 集合练习 :

**需求：计算一个字符串中每个字符出现次数。**

分析：

举例 : abcabaz

a -> 3 key (Character) value (Integer)

b -> 2

c -> 1

z -> 1

步骤一 : 提示用户输入

步骤二 : 定义一个 `映射表` 集合, <Character, Integer>

步骤三 : 遍历字符串, 取出的每一个字符就对应 `映射表` 中的 key.

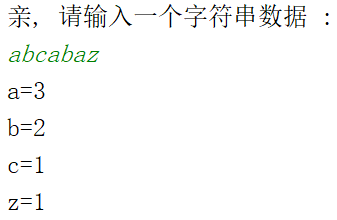
步骤四 : 判断取出的 key 是否在 `映射表` 集合中已经存在.

条件一 : 如果已经存在, 取出对应的值, 然后自增, 最后重写放入. (新值覆盖旧值)

条件二 : 如果不存在, 表示该 key 为第一次出现, 将 value 设置为 1 即可.

**代码实现 :**

public class Test4 {  
 public static void main(String[] args) {  
  
 *// 1. 创建一个键盘录入对象* Scanner sc = new Scanner(System.*in*);  
 System.*out*.println("亲, 请输入一个字符串数据 :");  
 String line = sc.nextLine();  
 sc.close();  
  
 *// 2. 定义一个 `映射表` 集合* HashMap<Character, Integer> map = new HashMap<>();  
  
 *// 3. 遍历字符串 (length + charAt)* for (int i = 0; i < line.length(); i++) {  
  
 *// 4. 取出当前遍历的字符* char key = line.charAt(i);  
  
 *// 5. 判断当前取出的 key 在映射表中是否已经存在* if (map.containsKey(key)) {  
 *// 5.1 已经存在, 需要取出原来的值, 自增, 最后修改* Integer value = map.get(key);  
 value++;  
 map.put(key, value);  
 } else {  
 *// 5.2 第一次出现* map.put(key, 1);  
 }  
 }  
  
 *// 6. entrySet 遍历  
 // Set<Map.Entry<Character, Integer>> entrySet = map.entrySet();* for (Map.Entry<Character, Integer> entry : map.entrySet()) {  
 System.*out*.println(entry);  
 }  
 }  
}

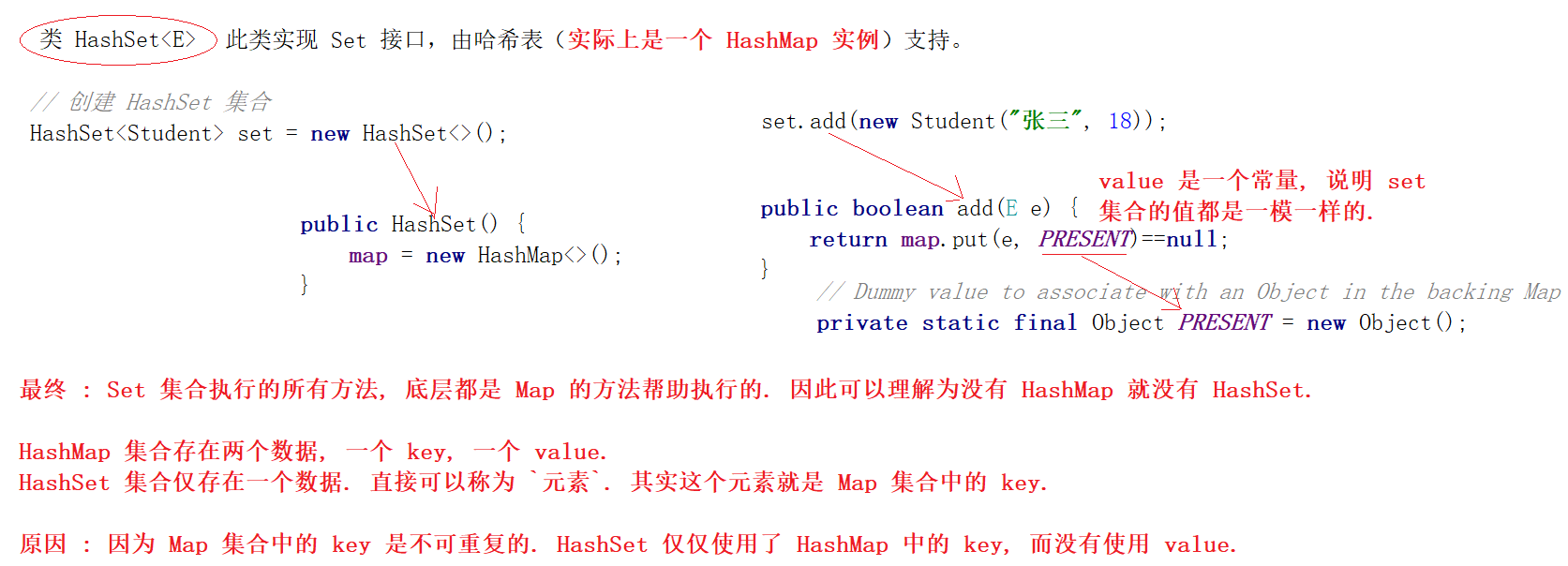


## 14.7 HashSet 与 HashMap 的关系 :

**关系 :** HashSet 类实现 Set 接口，由哈希表（实际上是一个 HashMap 实例）支持。

HashMap 映射表集合, key 是不能重复的, value 是可以重复的.

HashSet 哈希集合, 可以保证元素不重复.



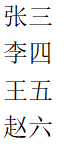
## 14.8 Vector与 Hashtable : (两个古老的类)

Vector 类可以实现可增长的对象数组。与数组一样，它包含可以使用整数索引进行访问的组件。但是，Vector 的大小可以根据需要增大或缩小，以适应创建 Vector 后进行添加或移除项的操作。

从 Java 2 平台 v1.2 开始，此类改进为可以实现 [List](mk:@MSITStore:C:\Users\xieco\Desktop\JDK9%20API文档\_JDK6API中文参考%5b沈东良%5d(070114).chm::/java/util/List.html) 接口，

Enumeration : 此接口的功能与 Iterator 接口的功能是重复的。此外，Iterator 接口添加了一个可选的移除操作，并使用较短的方法名。新的实现应该优先考虑使用 Iterator 接口而不是 Enumeration 接口。

public class Test4 {  
 public static void main(String[] args) {  
  
 *// 创建一个 Vector 对象 :  
 // ArrayList 集合取代了 Vector 类. 使用时, 优先使用 ArrayList 类.* Vector<String> vector = new Vector<>();  
  
 vector.addElement("张三");  
 vector.addElement("李四");  
 vector.addElement("王五");  
 vector.addElement("赵六");  
  
 *// 遍历* Enumeration<String> en = vector.elements();  
 while (en.hasMoreElements()) {  
 String element = en.nextElement();  
 System.*out*.println(element);  
 }  
 }  
}



**请问 :** Hashtable 和 HashMap 有什么区别 ???

1. Hashtable 可以存储任何非 null 的元素.

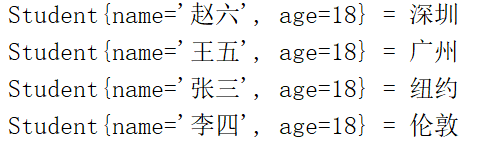
2. HashMap 可以存储 null 元素.

3. Hashtable 类是同步安全的.

4. HashMap 类是非同步安全的. (多线程) [**synchronizedMap**](mk:@MSITStore:C:\Users\xieco\Desktop\JDK9%20API文档\_JDK6API中文参考%5b沈东良%5d(070114).chm::/java/util/Collections.html#synchronizedMap(java.util.Map))([Map](mk:@MSITStore:C:\Users\xieco\Desktop\JDK9%20API文档\_JDK6API中文参考%5b沈东良%5d(070114).chm::/java/util/Map.html)<K,V> m) 该方法可以将非同步安全的 Map 集合转换为同步安全的 Map 的集合.

Hashtable : 此类实现一个哈希表，该哈希表将键映射到相应的值。任何非 null 对象都可以用作键或值。从Java 2 平台 v1.2起，此类就被改进以实现 [Map](mk:@MSITStore:C:\Users\xieco\Desktop\JDK9%20API文档\_JDK6API中文参考%5b沈东良%5d(070114).chm::/java/util/Map.html) 接口，

public class Test5 {  
 public static void main(String[] args) {  
  
 // 1. 创建一个 `映射表` 集合对象  
 // Map 集合仅根据 key 来计算存储下标. 如果 key 为自定义对象, 那么自定义对象必须重写 `hashCode + equals` 方法.  
 // HashMap 集合取代了 Hashtable 类. 因此, 建议使用 HashMap 集合类.  
 Hashtable<Student, String> table = new Hashtable<>();  
  
 // 2. 存储数据  
 table.put(new Student("张三", 18), "上海");  
 table.put(new Student("李四", 18), "北京");  
 table.put(new Student("王五", 18), "广州");  
 table.put(new Student("赵六", 18), "深圳");  
 table.put(new Student("张三", 18), "纽约");  
 table.put(new Student("李四", 18), "伦敦");  
  
 // 3. 遍历  
 Set<Map.Entry<Student, String>> entrySet = table.entrySet();  
 for (Map.Entry<Student, String> entry : entrySet) {  
 Student key = entry.getKey();  
 String value = entry.getValue();  
 System.out.println(key + " = " + value);  
 }  
 }  
}



## 14.9 集合小结 :

集合共分类两类 :

1. 单列集合 Collection (add, remove, iterator)

List extends Collection. 特点: 有序,可重复. (下标) (add, remove, set, get)

实现类 : ArrayList 数组列表, 查询快, 效率高. 底层是线性表.(可变数组)

Set extends Collection. 特点: 无序 (hashCode), 不可重复 (equals).

实现类 : HashSet 哈希集合. 元素唯一性. 底层依赖于 HashMap 的实现, 仅仅使用了 HashMap 的 key 来保证 Set 元素的唯一性.

2. 双列集合 Map (put, get, keySet, entrySet)

实现类 : HashMap implements Map. 哈希映射表

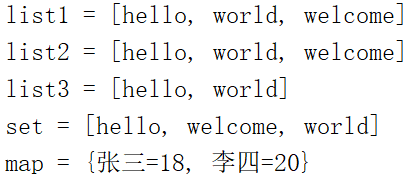
底层 : 数组(可变) + 单链表(同一个下标存储多个元素) + 二叉树(查询效率提升)

# 15. JDK9对集合添加元素的优化 : (了解)

List, Set, Map 接口的特有静态方法 : of(E …);

**注意点 :** 使用 of 创建完成的集合对象, 后期不可以对元素进行 `增删改` 操作, 并且 Set 和 Map 接口的 of 方法, 必须要手动保证 key 的不重复性.

public class Test1 {  
 public static void main(String[] args) {  
  
 *// 需求 : 添加元素  
  
 // 方式一 : add 方法* ArrayList<String> list1 = new ArrayList<>();  
 list1.add("hello");  
 list1.add("world");  
 list1.add("welcome");  
  
 *// 方式二 : Collections.addAll(集合对象, 可变参数);* ArrayList<String> list2 = new ArrayList<>();  
 Collections.*addAll*(list2, "hello", "world");  
 list2.add("welcome");  
  
 *// 方式三 : of 静态方法  
 // 缺点 : 添加完毕后, 不可以对集合中的元素进行 `增删改` 操作.  
 // 前提条件 : 保证集合中的元素无需进行 `增删改` 操作.* List<String> list3 = List.*of*("hello", "world");  
 *// list3.add("welcome"); // UnsupportedOperationException 不支持的操作异常.* System.*out*.println("list1 = " + list1);  
 System.*out*.println("list2 = " + list2);  
 System.*out*.println("list3 = " + list3);  
  
 Set<String> set = Set.*of*("hello", "world", "welcome"); *// 需要自己保证元素不重复.* System.*out*.println("set = " + set);  
  
 Map<String, Integer> map = Map.*of*("张三", 18, "李四", 20);  
 System.*out*.println("map = " + map);  
 }  
}



# 16. 案例 – 斗地主 (版本二)

3♥, 3♦, 3♣, 3♠, 4♥, 4♦, 4♣, 4♠, 5♥, 5♦, 5♣, 5♠, 6♥, 6♦, 6♣, 6♠, 7♥, 7♦, 7♣, 7♠, 8♥, 8♦, 8♣, 8♠, 9♥, 9♦, 9♣, 9♠, 10♥, 10♦, 10♣, 10♠, J♥, J♦, J♣, J♠, Q♥, Q♦, Q♣, Q♠, K♥, K♦, K♣, K♠, A♥, A♦, A♣, A♠, 2♥, 2♦, 2♣, 2♠, 小☺, 大☠

**映射表集合 :**

3♥, 3♦, 3♣, 3♠, 4♥, 4♦, 4♣, 4♠, 5♥, 5♦, 5♣, 5♠, 6♥, 6♦, 6♣, 6♠, …… 小☺, 大☠ value

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 …… 52 53 key

分析 :

步骤一 : 定义一个 `映射表` 集合, <Integer, String> 准备一副扑克牌

步骤二 : 准备一副纯数字牌. (0~53)

步骤三 : 准备三个玩家和一个底牌.

步骤四 : 将纯数字牌实现洗牌. (乱序的数字牌)

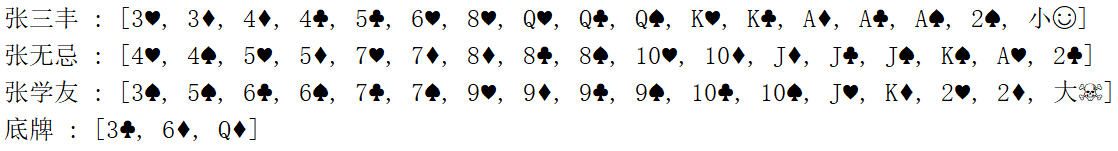
步骤五 : 将洗完的乱序数字牌发给三个玩家和底牌.

步骤六 : 对玩家手中的数字牌实现排序.

步骤七 : 根据玩家手中有序的数字牌从 `映射表` 中取出对应的值.

**代码实现 :**

public class Test2 {  
 public static void main(String[] args) {  
  
 *// 1.1 创建一个 `映射表` 集合. (准备一个扑克牌)  
 // 0 -> 3♥, 1 -> 3♦, 2 -> 3♣, 3 -> 3♠, ...... 52 -> 小☺, 53 -> 大☠* HashMap<Integer, String> pokers = new HashMap<>();  
  
 *// 1.2 准备数字 [3,4,5,6,7,8,9,10,J,Q,K,A,2]* List<String> numbers = List.*of*("3", "4", "5", "6", "7", "8", "9", "10", "J", "Q", "K", "A", "2");  
 *// 1.3 准备花色 [♥,♦,♣,♠]* List<String> colors = List.*of*("♥", "♦", "♣", "♠");  
  
 *// 1.4 遍历* int map\_key = 0;  
 for (String number : numbers) {  
 for (String color : colors) {  
 String poker = number + color;  
 *// 1.5 将拼接完成 poker 存储到 pokers 映射表中.* pokers.put(map\_key++, poker);  
 }  
 }  
  
 *// 1.6 最后添加 `大小王`* pokers.put(map\_key++, "小☺");  
 pokers.put(map\_key, "大☠");  
 System.*out*.println("pokers = " + pokers);  
  
 *// 2. 准备一副数字牌, 专门发给玩家的牌* ArrayList<Integer> numberPokers = new ArrayList<>();  
 for (int i = 0; i < 54; i++) {  
 numberPokers.add(i);  
 }  
  
 *// 3. 洗牌* Collections.*shuffle*(numberPokers);  
  
 *// 4. 准备三个玩家和一个底牌* ArrayList<Integer> numberPlayer1 = new ArrayList<>();  
 ArrayList<Integer> numberPlayer2 = new ArrayList<>();  
 ArrayList<Integer> numberPlayer3 = new ArrayList<>();  
 ArrayList<Integer> numberBase = new ArrayList<>();  
  
 *// 5. 发牌* for (int i = 0; i < numberPokers.size(); i++) {  
  
 *// 5.1 取牌* Integer numberPoker = numberPokers.get(i);  
  
 *// 5.2 判断* if (i >= numberPokers.size() - 3) {  
 *// 底牌* numberBase.add(numberPoker);  
 } else {  
 *// 玩家* if (i % 3 == 0) {  
 numberPlayer1.add(numberPoker);  
 } else if (i % 3 == 1) {  
 numberPlayer2.add(numberPoker);  
 } else {  
 numberPlayer3.add(numberPoker);  
 }  
 }  
 }  
  
 *// 6. 排序* Collections.*sort*(numberPlayer1);  
 Collections.*sort*(numberPlayer2);  
 Collections.*sort*(numberPlayer3);  
 Collections.*sort*(numberBase);  
  
  
 *// 7.1 创建玩家和底牌* ArrayList<String> player1 = new ArrayList<>();  
 ArrayList<String> player2 = new ArrayList<>();  
 ArrayList<String> player3 = new ArrayList<>();  
 ArrayList<String> base = new ArrayList<>();  
  
 *// 7.2 根据玩家手中有序的 key 从映射表扑克牌集合中取出对应 value.* for (Integer key : numberPlayer1) {  
 String value = pokers.get(key);  
 player1.add(value);  
 }  
 for (Integer key : numberPlayer2) {  
 String value = pokers.get(key);  
 player2.add(value);  
 }  
 for (Integer key : numberPlayer3) {  
 String value = pokers.get(key);  
 player3.add(value);  
 }  
 for (Integer key : numberBase) {  
 String value = pokers.get(key);  
 base.add(value);  
 }  
  
 System.*out*.println("张三丰 : " + player1);  
 System.*out*.println("张无忌 : " + player2);  
 System.*out*.println("张学友 : " + player3);  
 System.*out*.println("底牌 : " + base);  
 }  
}

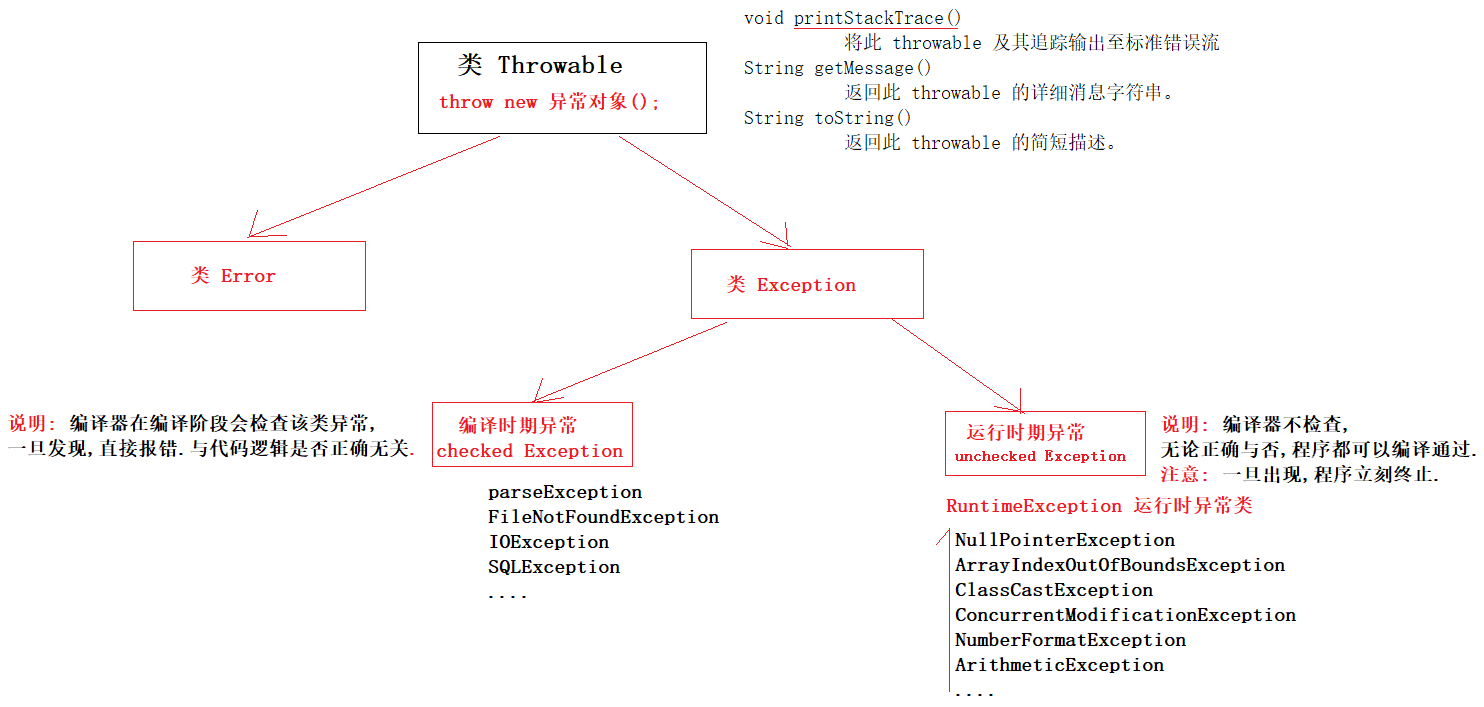


# 17. 异常: Exception (了解)

**异常 :** 非正常. 类似于生活中的 `感冒, 发烧`. (**提醒机制**) 可以实现提前处理, 从而可以避免程序异常终止. (提前终止程序)

**错误 :** 类似与 `癌症…`. 程序开发中不需要处理, 应该尽量避免.

## 17.1 异常分类 : Throwable

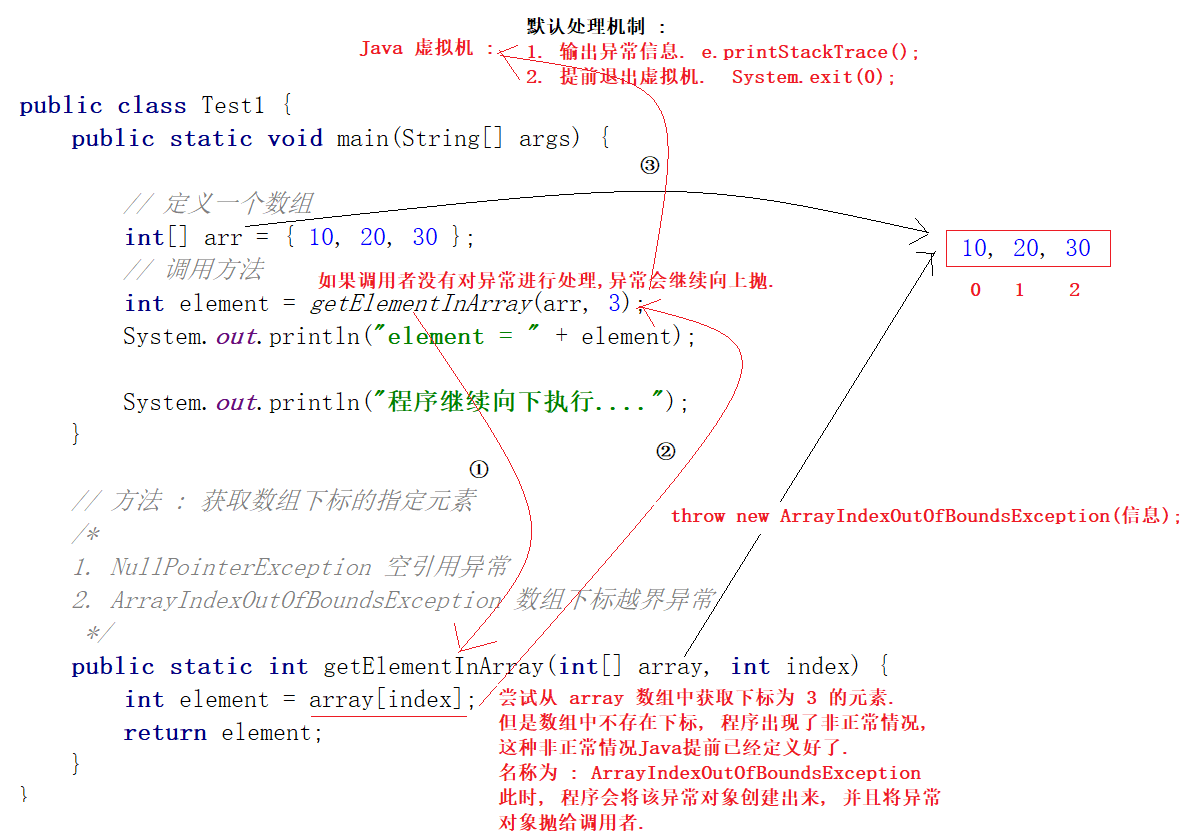


## 17.2 异常的产生过程图解 :

**小结 :** 异常一旦发生, 如果不处理, 程序就会提前终止.

运行时期异常, 不需要 throws 关键字, 其底层会实现自动向上抛. 但是, 编译时期异常必须手动实现向上抛.

public class Test1 {  
 public static void main(String[] args) {  
  
 *// 定义一个数组* int[] arr = { 10, 20, 30 };  
 *// 调用方法* int element = *getElementInArray*(arr, 3);  
 System.*out*.println("element = " + element);  
  
 System.*out*.println("程序继续向下执行....");  
 }  
  
 *// 方法 : 获取数组下标的指定元素  
 /\*  
 1. NullPointerException 空引用异常  
 2. ArrayIndexOutOfBoundsException 数组下标越界异常  
 \*/* public static int getElementInArray(int[] array, int index) {  
 int element = array[index];  
 return element;  
 }  
}

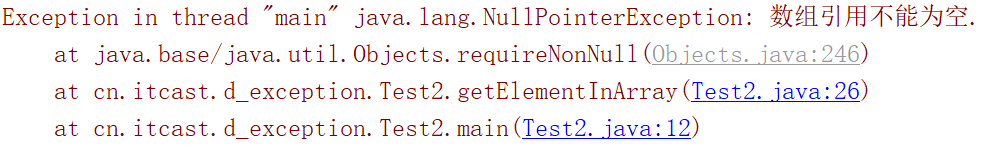


## 17.3 手动实现异常的抛出 : throw

格式 : throw new 异常类型(信息);

**代码演示 :**

public class Test2 {  
 public static void main(String[] args) {  
  
 *// 定义一个数组* int[] arr = { 10, 20, 30 };  
 *// 调用方法* int element = *getElementInArray*(null, 3);  
 System.*out*.println("element = " + element);  
  
 System.*out*.println("程序继续向下执行....");  
 }  
  
 *// 方法 : 获取数组下标的指定元素  
 /\*  
 1. NullPointerException 空引用异常  
 2. ArrayIndexOutOfBoundsException 数组下标越界异常  
 \*/* public static int getElementInArray(int[] array, int index) {  
  
 *// 参数进行逻辑判断* if (array == null) {  
 *// 手动抛出异常对象 ...* throw new NullPointerException("数组引用不能为空.");  
 }  
  
 if (index < 0 || index >= array.length) {  
 throw new ArrayIndexOutOfBoundsException("数组下标越界异常.");  
 }  
  
 *// 正常执行逻辑 ...* int element = array[index];  
 return element;  
 }  
}



## 17.4 Objects工具类的非空判断 :

// 方法 : 获取数组下标的指定元素  
/\*  
1. NullPointerException 空引用异常  
2. ArrayIndexOutOfBoundsException 数组下标越界异常  
 \*/  
public static int getElementInArray(int[] array, int index) {  
  
 // 参数进行逻辑判断  
 Objects.requireNonNull(array, "数组引用不能为空.");  
 // Objects.requireNonNull(array);  
 /\*if (array == null) {  
 // 手动抛出异常对象 ...  
 throw new NullPointerException("数组引用不能为空.");  
 }\*/  
  
 if (index < 0 || index >= array.length) {  
 throw new ArrayIndexOutOfBoundsException("数组下标越界异常.");  
 }  
  
 // 正常执行逻辑 ...  
 int element = array[index];  
 return element;  
}

**底层代码 :**

public static <T> T requireNonNull(T obj, String message) {  
 if (obj == null)  
 throw new NullPointerException(message);  
 return obj;  
}

## 17.5 异常的处理方案一 : throws

**实现 :** 定义一个方法, 接收字符串参数, 并将字符串解析为一个日期对象.

public Date parse(String source) throws ParseException 含义说明 : 调用 parse 方法会发生解析异常, 如果发生了, 请问调用者如何处理 ???

**格式 : 方法的修饰符 返回值类型 方法名(参数列表) throws 异常类型 { 方法体 … }**

public class Test3 {  
 *// 说明 : main 方法如果向上抛异常, 就是 `虚拟机` 处理. 虚拟机会输出异常信息, 并且提前终止程序.  
 // 学习过程 : 一般都会直接抛, 不会处理.* public static void main(String[] args) throws ParseException {  
  
 Scanner sc = new Scanner(System.*in*);  
 System.*out*.println("亲, 请输入出生年月日, 格式(yyyy-MM-dd)");  
 String source = sc.nextLine();  
 sc.close();  
  
 *// 调用方法* boolean result = *parseStringToDate*(source);  
 System.*out*.println("result = " + result);  
 }  
  
 *// 定义一个方法, 接收字符串参数, 并将字符串解析为一个日期对象.  
 // throws ParseException 调用 parseStringToDate 方法会发生解析异常, 如果发生了, 请问调用者如何处理 ???* public static boolean parseStringToDate(String source) throws ParseException {  
  
 *// 1. 创建一个日期格式化对象* SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");  
 *// 2. 执行解析* Date date = dateFormat.parse(source);  
 System.*out*.println("date = " + date);  
 *// 3. 解析成功* return true;  
 }  
}



## 17.6 异常的处理方案二 : try-catch (没有资源)

格式 :

try {

编写可能会发生异常的代码…

} catch (异常类型 e) {

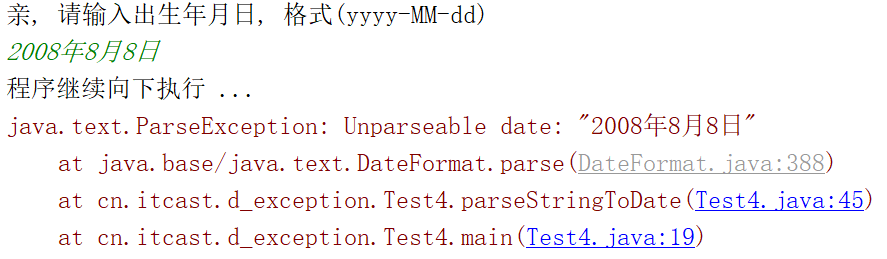
处理代码 …

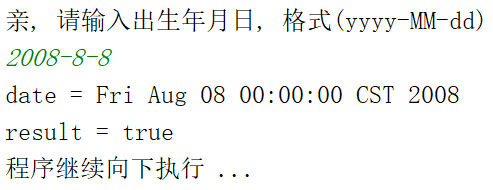
}

理解 : 只要程序 catch 到了对应的异常对象, 程序就不会提前终止, 反而可以继续向下执行.

**测试代码 :**

public class Test4 {  
 public static void main(String[] args) {  
  
 Scanner sc = new Scanner(System.*in*);  
 System.*out*.println("亲, 请输入出生年月日, 格式(yyyy-MM-dd)");  
 String source = sc.nextLine();  
 sc.close();  
  
 *// 调用方法 :* try {  
 *// 编写可能会发生异常的代码…* boolean result = *parseStringToDate*(source);  
 System.*out*.println("result = " + result);  
 } catch (ParseException e) {  
 *// 处理代码 …  
 // System.out.println("程序发生了解析异常...");  
 // 将错误信息写入到 `文件中`. (日志)  
 /\*  
 String message = e.getMessage();  
 System.out.println("message = " + message);  
 String s = e.toString();  
 System.out.println("s = " + s);  
 \*/* e.printStackTrace(); *// 查看信息, 程序是可以向后执行的.* }  
  
 System.*out*.println("程序继续向下执行 ...");  
 }  
  
 *// 定义一个方法, 接收字符串参数, 并将字符串解析为一个日期对象.  
 // throws ParseException 调用 parseStringToDate 方法会发生解析异常, 如果发生了, 请问调用者如何处理 ???* public static boolean parseStringToDate(String source) throws ParseException {  
  
 *// 1. 创建一个日期格式化对象* SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");  
 *// 2. 执行解析* Date date = dateFormat.parse(source);  
 System.*out*.println("date = " + date);  
 *// 3. 解析成功* return true;  
 }  
}





## 17.7 finally 代码块的使用 : try – catch- finally (资源)

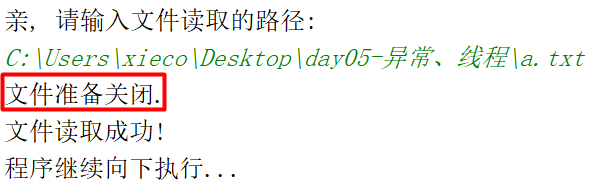
**使用场景 :** IO流读写操作中.

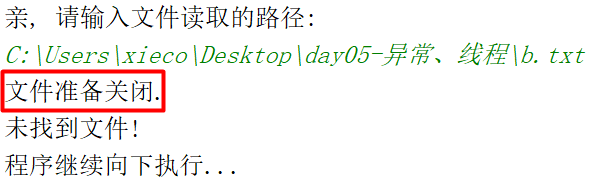
**说明 :** new FileReader(“**文件资源**”); 该方法会抛出一个异常, FileNotFoundException 文件找不到, 最终使用完毕文件后, 程序应该关闭该文件资源.

**finally 会在返回的 return 之前被执行. 无论程序是否发生异常, finally 代码块都可以正常被执行. 因为程序中使用 finally 进行资源的关闭.**

**IO 流异常的标准处理写法 :**

public class Test6 {  
 public static void main(String[] args) {  
  
 Scanner sc = new Scanner(System.*in*);  
 System.*out*.println("亲, 请输入文件读取的路径:");  
 String path = sc.nextLine();  
 sc.close();  
  
 String result = *readFile*(path);  
 System.*out*.println(result);  
  
 System.*out*.println("程序继续向下执行...");  
 }  
  
 *// 方法 : 读取文件* public static String readFile(String path) {  
 *// 模拟 : 读取文件信息* FileReader reader = null; *// 定义* try {  
 reader = new FileReader(path); *// 初始化* } catch (FileNotFoundException e) {  
 *// System.out.println("未找到文件.");* return "未找到文件!";  
 } catch (IOException e) {  
 return "文件读取失败!.";  
 } finally {  
 System.*out*.println("文件准备关闭.");  
 if (reader != null) {  
 try {  
 reader.close(); *// null.close();* } catch (IOException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
  
 return "文件读取成功!";  
 }  
}





## 17.8 异常的注意事项 :

注意一 : 如果子类重写父类定义的方法, 父类该方法没有声明 `编译时期` 异常, 子类重写时也不可以声明.

注意二 : 如果实现类重写接口定义的方法, 该接口的方法没有声明 `编译时期` 异常, 实现类重写时也不可以声明.

如果在方法重写中遇见了 `异常`, 重写方法仅能实现 `捕获`.

**Person 父类 :**

public class Person {  
  
 *// 行为 :* public void introduce() {  
 System.*out*.println("我是 Person 类.");  
 }  
}

**MyInterface 接口 :**

public interface MyInterface {  
 void run();  
}

**Student 子类 :**

import java.text.ParseException;  
import java.text.SimpleDateFormat;  
  
public class Student extends Person implements MyInterface {  
  
 @Override  
 public void introduce() {  
  
 *// 解析日期 :* SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");  
 *// 问题 : 重写的方法中能够将异常进行声明吗 ??? 不可以.方法中只能进行捕获.* try {  
 dateFormat.parse("2008-08-08");  
 } catch (ParseException e) {  
 }  
 }  
  
 @Override  
 public void run() {  
 SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");  
 try {  
 dateFormat.parse("2011-11-11");  
 } catch (ParseException e) {  
 e.printStackTrace();  
 }  
 }  
}

## 17.9 异常小结 :

异常总共分类两类 :

1. 编译时期异常. 该类异常编译器会检查, 必须做出处理.

处理方案一 : throws 异常声明. (我不处理, 让我的调用者处理)

处理方案二 : try – catch – finally (资源关闭) (一旦发生异常, 就可以进入对应catch语句中处理, 程序可以继续向下执行)

2. 运行时期异常. 该类异常编译器不检查, 因此可以不处理, 也可以处理.

处理方案 : try - catch 异常捕获, 说明: 运行时期的异常基本都与资源没有关系. 因此也不需要finally关闭资源.

# 18. 自定义异常 : (了解)

**说明 :** 程序开发中, 遇到一些特殊的业务逻辑, Java语言没有提供该逻辑对应的异常类, 此时, 就需要程序员自定义异常类实现.

**场景 :** 注册异常, 如果用户名已经被注册过了. 就不能再使用该用户名注册了, 此时, 需要程序抛出一个注册异常进行代码逻辑的提醒. (**提前处理**)

RegisterException 注册异常, 系统没有提供, 因此我们程序中需要自定义该异常类.

思考1 : 该异常继承自哪个类吗 ??? Exception

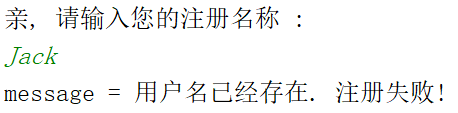
思考2 : 该异常需要调用者提前处理吗 ??? 前提处理 : Exception 不需要 : RuntimeException

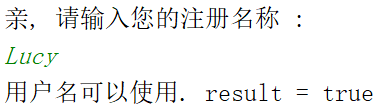
**自定义异常类 RegisterException :**

public class RegisterException extends Exception {  
  
 *// 提供两个构造方法 :* public RegisterException() {  
 super();  
 }  
  
 public RegisterException(String message) {  
 super(message);  
 }  
}

**测试自定义异常类 :**

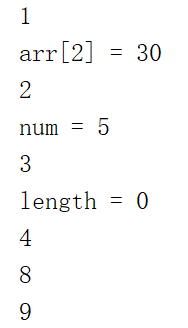
public class Test2 {  
 *// 数组模拟数据 :* private static String[] *names* = { "Jack", "Rose", "Peter", "Tome" };  
  
 public static void main(String[] args) {  
  
 *// 需求 : 模拟用户注册, 如果用户名重复, 就抛出 `注册异常`* Scanner sc = new Scanner(System.*in*);  
 System.*out*.println("亲, 请输入您的注册名称 :");  
 String name = sc.nextLine();  
 sc.close();  
  
 *// 判断该用户名是否已经被注册. (调用方法)* try {  
 boolean result = *checkUsername*(name);  
 System.*out*.println("用户名可以使用. result = " + result);  
 } catch (RegisterException e) {  
 String message = e.getMessage();  
 System.*out*.println("message = " + message);  
 }  
  
 }  
  
 *// 方法 : 判断用户名是否已经被注册* public static boolean checkUsername(String username) throws RegisterException {  
 *// 遍历 names 数组, 判断 username 是否已经包含* for (String name : *names*) {  
 *// 判断* if (name.equals(username)) {  
 *// 说明用户名已经存在, 需求, 抛出注册异常* throw new RegisterException("用户名已经存在. 注册失败!");  
 }  
 }  
 *// 用户名可以使用* return true;  
 }  
}





# 19. 异常的流程思考 :

public class Test7 {  
 public static void main(String[] args) {  
  
 try {  
 System.*out*.println(1);  
 int[] arr = { 10, 20, 30 };  
 System.*out*.println("arr[2] = " + arr[2]); *// ArrayIndexOutOfBoundsException: 3* System.*out*.println(2);  
 int num = 10 / 2; *// ArithmeticException: / by zero* System.*out*.println("num = " + num);  
 System.*out*.println(3);  
 *// String str = null;* String str = "";  
 int length = str.length(); *// NullPointerException* System.*out*.println("length = " + length);  
 System.*out*.println(4);  
 } catch (ArrayIndexOutOfBoundsException e) {  
 System.*out*.println(5);  
 *// return; // catch 中如果有 return 语句, 也会先执行 finally 代码块, 最后才返回.* } catch (ArithmeticException e) {  
 System.*out*.println(6);  
 } catch (NullPointerException e) {  
 System.*out*.println(7);  
 } finally {  
 *// finally 作用为关闭资源, 不是返回数据.* System.*out*.println(8);  
 }  
 System.*out*.println(9);  
 }  
}



# 20. 多线程 : MultiThread

多线程 : 线程就是程序的 `执行路径`. 多线程就可以理解为程序的多条执行路径.

请问 : 同一个Java程序, 能够同时执行多个 `死循环 / 耗时操作` ??? 可以, 需要使用 `多线程` 技术.

public class Test1 {  
 // 主方法又名为 `主线程`. 是 Java虚拟机启动时自动分配的执行路径. 主线程的入口就是主方法.  
 public static void main(String[] args) {  
  
 // 请问 : 同一个Java程序, 能够同时执行多个 `死循环` ???  
 // 说明 : 如果条件中直接书写 `常量`. 编译器能够理解该常量.  
 // 如果是变量, 编译器则无法理解. 因为变量只有在运行阶段才会被分配空间.  
  
 // 多线程可以实现多个 `耗时操作` 同时被执行. 单线程是没有办法让多个耗时操作同时被执行.  
 boolean flag = true;  
 while (flag) {  
 System.out.println("下载电影 <<西红柿首富>>."); // 耗时操作  
 }  
  
 while (true) {  
 System.out.println("下载电影 <<活着>>.");  
 }  
 }  
}

## 20.1 并发与并行 : (了解)

并发 : Concurrent

CPU 而言 : 在同一个 `时间段` 内, 多条线程同时被执行的. 但是, 这个过程是交替被执行的.

并行 : Parallel

CPU 而言 : 在同一个 `时间点` 内, 多条线程同时被执行的. 多核 CPU 可以执行并行执行.

## 20.2 线程与进程 : (了解)

进程 : 是由操作系统实现空间分配. 表示一个正在运行的程序.

线程 : 同一个程序中的多条执行路径. 线程可以由程序中的代码实现分配.

## 20.3 线程执行初体验 : (注意输出效果)

**线程子类 1 :**

public class DownloadMovie1 extends Thread {  
  
 @Override  
 public void run() {  
 while (true) {  
 System.*out*.println("下载电影 <<西红柿首富>>."); *// 耗时操作* }  
 }  
}

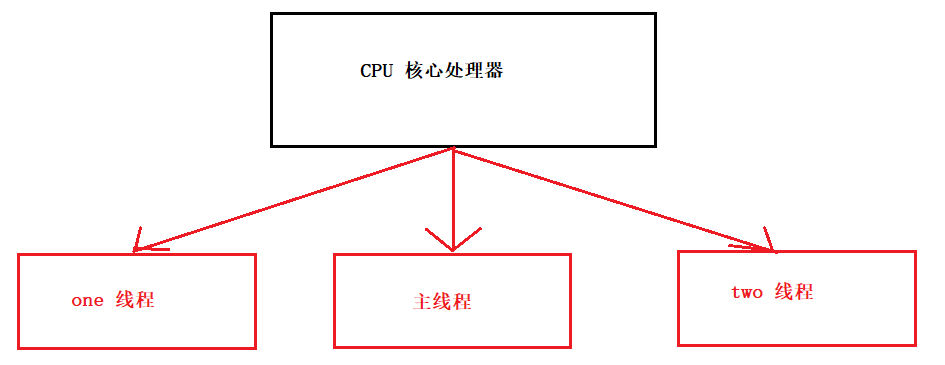
**线程子类 2 :**

public class DownloadMovie2 extends Thread {  
  
 @Override  
 public void run() {  
 while (true) {  
 System.*out*.println("下载电影 <<活着>>."); *// 耗时操作* }  
 }  
}

**测试类 :**

public class Test2 {  
 public static void main(String[] args) {  
  
 *// 多线程可以实现多个 `耗时操作` 同时被执行. 单线程是没有办法让多个耗时操作同时被执行.  
 // 多线程可以提高 CPU 的使用率.  
  
 // 创建一个子线程对象* DownloadMovie1 movie1 = new DownloadMovie1();  
 movie1.start();  
  
 *// 再创建一个子线程对象* DownloadMovie2 movie2 = new DownloadMovie2();  
 movie2.start();  
  
 *// 主线程执行代码 ...* while (true) {  
 System.*out*.println("main 方法在继续执行 ...");  
 }  
 }  
}

**执行效果为 : 多个死循环可以交替被执行.**



## 20.4 多线程的第一种实现方案 : 继承

**类 Thread :** *线程* 是程序中的执行线程。Java 虚拟机允许应用程序并发地运行多个执行线程。

**方案一 :**  一种方法是将类声明为 Thread 的子类。该子类应重写 Thread 类的 run 方法。接下来可以分配并启动该子类的实例。

步骤一 : 自定义一个类, 该类继承自 Thread. 那么该类就可以被称为 `线程类`.

步骤二 : 子类重写 Thread 类的 run() 方法.

步骤三 : 创建一个线程子类对象.

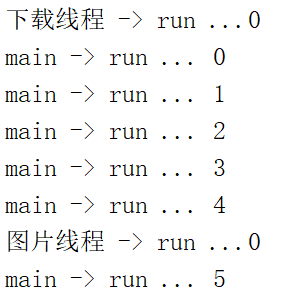
步骤四 : 启动子类实例.

**线程类 :**

// 1. 自定义类, 继承 Thread 类.  
public class MyThread extends Thread {  
 /\*  
 private String name; 说明: 线程 Thread 类本身就拥有 `名称`  
  
 public MyThread(String name) {  
 this.name = name;  
 }  
 \*/  
  
 // 2. 重写 run 方法.  
 @Override  
 public void run() {  
 for (int i = 0; i < 10; i++) {  
 System.out.println(getName() + " -> run ..." + i);  
 }  
 }  
}

**测试类 :**

public class Test1 {  
 public static void main(String[] args) {  
  
 *// 3. 创建一个线程子类对象* MyThread one = new MyThread();  
 *// 设置线程名称* one.setName("下载线程");  
 *// 4. 启动子线线程实例 (子线程的入口是 run 方法)* one.start();  
  
 MyThread two = new MyThread();  
 two.setName("图片线程");  
 two.start();  
  
 *// 主线程* for (int i = 0; i < 10; i++) {  
 System.*out*.println(Thread.*currentThread*().getName() + " -> run ... " + i);  
 }  
 }  
}



## 20.5 多线程的第二种实现方案 : 接口 (重点)

**方案二 :** 创建线程的另一种方法是声明实现 Runnable 接口的类。该类然后实现 run 方法。然后可以分配该类的实例，在创建 Thread 时作为一个参数来传递并启动。

步骤一 : 自定义类实现 Runnable 接口. 那么该类就可以被称为 `任务类`.

步骤二 : 自定义类实现接口的 run 方法.

步骤三 : 创建一个 `任务类` 对象.

步骤四 : 创建一个 `线程类` 对象, 并同时将 `任务类` 传入, 与该线程类对象进行绑定. 一旦该线程启动, 就会执行任务类代码.

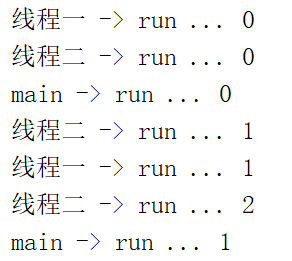
步骤五 : 启动线程类对象.

**任务类 :**

*// 1. 自定义类实现 Runnable 接口.*public class MyTask implements Runnable {  
  
 *// 2. 重写 run 方法.* @Override  
 public void run() {  
 for (int i = 0; i < 10; i++) {  
 System.*out*.println(Thread.*currentThread*().getName() + " -> run ... " + i);  
 }  
 }  
}

**测试类 :**

public class Test1 {  
 public static void main(String[] args) {  
  
 *// 3. 创建一个任务类对象* MyTask task = new MyTask();  
  
 *// 4. 创建一个线程类对象, 并同时将任务类对象传入* Thread t1 = new Thread(task, "线程一");  
  
 *// 5. 启动子线程,* t1.start();  
  
 *// 多个子线程* Thread t2 = new Thread(task, "线程二");  
 t2.start();  
  
 for (int i = 0; i < 10; i++) {  
 System.*out*.println(Thread.*currentThread*().getName() + " -> run ... " + i);  
 }  
 }  
}



## 20.6 匿名内部类实现多线程 :

public class Test1 {  
 public static void main(String[] args) {  
  
 /\*  
 Object obj = new Object() {  
 public void show() {  
 System.out.println("调用了 show 方法 ...");  
 }  
 };  
  
 // 强转类型. 匿名对象没有名称, 无法实现强转.  
 // 子类类型 子类名称 = (子类类型) obj;  
  
 new Object() {  
 public void show() {  
 System.out.println("调用了 show 方法 ...");  
 }  
 }.show();  
 \*/  
  
 // 匿名子类 :  
 // Thread t = new Thread(); // 创建了一个 Thread 类对象. new Object();  
 // 父类引用指向了子类对象.  
 // Thread t = new Thread(){}; // 创建了一个 Thread 类的匿名子类对象. new Object(){};  
 Thread t = new Thread() {  
 @Override  
 public void run() {  
 for (int i = 0; i < 10; i++) {  
 System.out.println(getName() + " -> run ... " + i);  
 }  
 }  
 };  
 t.start();  
  
 // 匿名实现类 :  
 // new Thread(Runnable target, String name);  
 // 请问1 : new Runnable(); 错误! 接口不能实例化对象.  
 // 请问2 : new Runnable(){}; 正确! 创建了一个 Runnable 接口的匿名实现类对象.  
 new Thread(new Runnable(){  
 @Override  
 public void run() {  
 for (int i = 0; i < 10; i++) {  
 System.out.println(Thread.currentThread().getName() + " -> run ... " + i);  
 }  
 }  
 }, "匿名接口实现类线程").start();  
  
 // 主线程执行代码...  
 for (int i = 0; i < 10; i++) {  
 System.out.println(Thread.currentThread().getName() + " -> run ... " + i);  
 }  
 }  
}



## 20.7 Thread 和 Runnable的对比与区别 :

继承方案 : Java语言只能 `单继承`. 继承有局限性. Student extends Person 行为: study, doHomework … chasingGirls

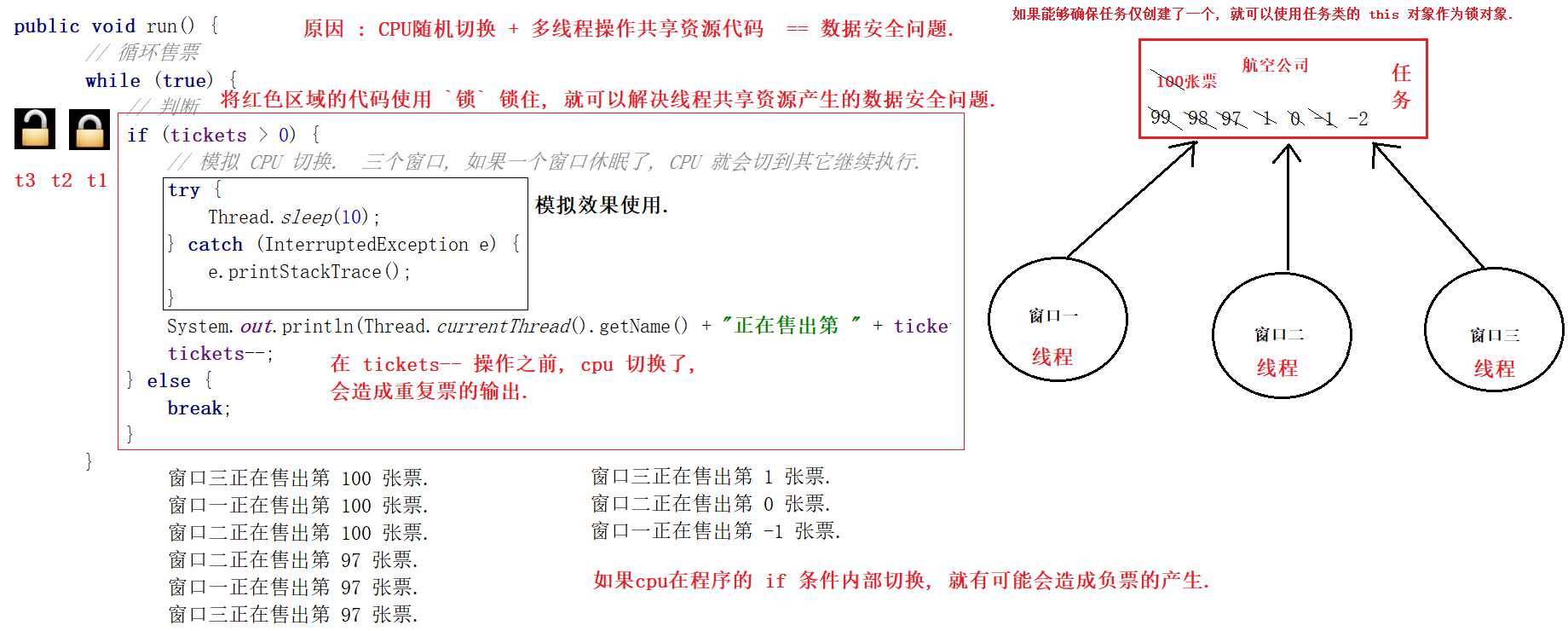
接口方案 : Java语言可以 `多实现`. 解决了继承中的局限性. Student extends Person implements Runnable

耦合性问题 : (类和类之间的关联度)

继承 : 线程和任务紧密关联. 不能分离.

接口 : 接口更加满足程序的解耦特性. (接口解耦)

# 21. 线程安全 : (了解)



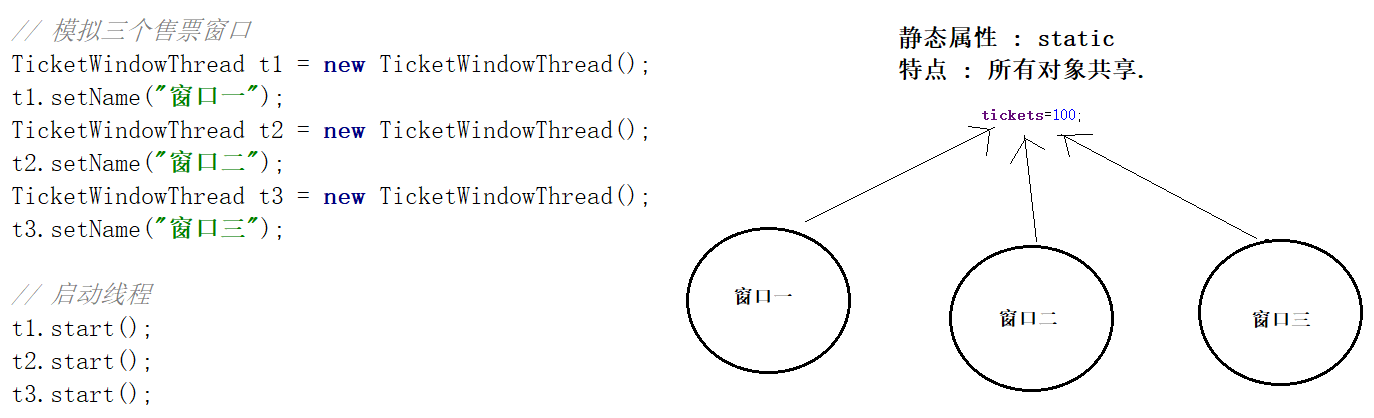
## 21.1 继承方案实现售票案例 :

**需求 :** 有一个航空公司, 售卖 100 张票. 有三个售票窗口同时卖同一家航空公司的 100 张票. 编写程序来模拟该行为.

**注意 :** 数据是每个对象共享的, 还是每个对象独有的. 如果是共享数据, 需要将属性数据定义为 `静态属性`. 因为静态属性可以被该类的所有对象所共享, 如果是非静态属性, 就是每个对象独有的.

**售票窗口线程类定义 :**

*// 1. 自定义类继承 Thread 类*public class TicketWindowThread extends Thread {  
 *// 属性* private static int *tickets* = 100;  
  
 *// 2. 重写 Thread 类的 run 方法. run 方法执行完毕就意味着子线程结束了.* @Override  
 public void run() {  
 *// 循环售卖航空公司的 100 张票* while (true) {  
 *// 判断* if (*tickets* > 0) {  
 *// 卖票* System.*out*.println(getName() + "正在售出第 " + *tickets* + " 张票.");  
 *tickets*--;  
 } else {  
 break; *// 票买完了, 跳出循环, 结束子线程* }  
 }  
 }  
}



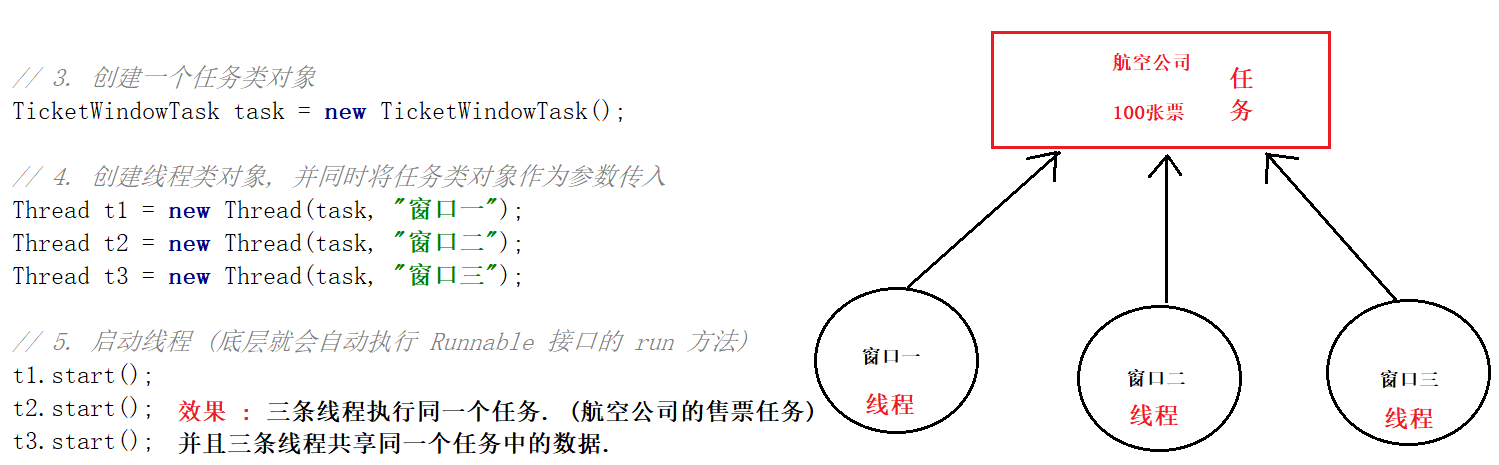
## 21.2 接口方案实现售票案例 :

**任务类 :**

*// 1. 自定义类, 实现 Runnable 接口. (售票任务类)*public class TicketWindowTask implements Runnable {  
 *// 属性* private int tickets = 100;  
  
 *// 2. 重写 Runnable 接口的 run() 方法.* @Override  
 public void run() {  
 *// 循环售票* while (true) {  
 *// 判断* if (tickets > 0) {  
  
 *// 模拟 CPU 切换. 三个窗口, 如果一个窗口休眠了, CPU 就会切到其它继续执行.* try {  
 Thread.*sleep*(10);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
  
 System.*out*.println(Thread.*currentThread*().getName() + "正在售出第 " + tickets + " 张票.");  
 tickets--;  
 } else {  
 break;  
 }  
 }  
 }  
}

**测试类 :**

public class Test2 {  
 public static void main(String[] args) {  
  
 *// 3. 创建一个任务类对象* TicketWindowTask task = new TicketWindowTask();  
  
 *// 4. 创建线程类对象, 并同时将任务类对象作为参数传入* Thread t1 = new Thread(task, "窗口一");  
 Thread t2 = new Thread(task, "窗口二");  
 Thread t3 = new Thread(task, "窗口三");  
  
 *// 5. 启动线程 (底层就会自动执行 Runnable 接口的 run 方法)* t1.start();  
 t2.start();  
 t3.start();  
 }  
}



## 21.3 线程同步代码块 synchronized :

synchronized (锁对象) {

锁住的代码. (只能由单条线程依次执行)

}

**任务类 :**

*// 1. 自定义类, 实现 Runnable 接口. (售票任务类)*public class TicketWindowTask implements Runnable {  
 *// 属性* private int tickets = 1000;  
  
 *// 定义一把锁* private Object lock = new Object();  
  
 *// 2. 重写 Runnable 接口的 run() 方法.* @Override  
 public void run() {  
 *// Object lock = new Object(); 不能将锁对象定义为 run 方法内部, 因为 run 方法会被每条子线程都执行.  
 // 循环售票* while (true) {  
 *// 判断  
 // 锁对象可以是任意对象, 但必须保证线程间锁对象的唯一性.* synchronized (this) {  
 if (tickets > 0) {  
  
 *// 模拟 CPU 切换. 三个窗口, 如果一个窗口休眠了, CPU 就会切到其它继续执行.  
 /\*try {  
 Thread.sleep(10);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }\*/* System.*out*.println(Thread.*currentThread*().getName() + "正在售出第 " + tickets + " 张票.");  
 tickets--;  
 } else {  
 break;  
 }  
 }  
 }  
 }  
}

**测试类 :**

public class Test1 {  
 public static void main(String[] args) {  
  
 *// 3. 创建一个任务类对象* TicketWindowTask task = new TicketWindowTask();  
  
 *// 4. 创建线程类对象, 并同时将任务类对象作为参数传入* Thread t1 = new Thread(task, "窗口一");  
 Thread t2 = new Thread(task, "窗口二");  
 Thread t3 = new Thread(task, "窗口三");  
  
 *// 5. 启动线程 (底层就会自动执行 Runnable 接口的 run 方法)* t1.start();  
 t2.start();  
 t3.start();  
 }  
}



## 21.4同步方法 : synchronized :

**任务类 :**

*// 1. 自定义类, 实现 Runnable 接口. (售票任务类)*public class TicketWindowTask2 implements Runnable {  
 *// 属性* private int tickets = 1000;  
  
 *// 定义一把锁* private Object lock = new Object();  
  
 *// 2. 重写 Runnable 接口的 run() 方法.* @Override  
 public void run() { *// 循环售票* while (true) {  
  
 *// this.sellTickets(); this 可不可以保证锁对象的唯一性 ??? this 表达当前对象, 而当前任务类对象仅有一个.* sellTickets();  
  
 *// 判断* if (tickets <= 0) {  
 break;  
 }  
 }  
 }  
  
 *// 请问 : 同步方法有没有锁对象 ??? 当前的 this 对象就是同步方法的锁对象* private synchronized void sellTickets() {  
 if (tickets > 0) {  
 *// 模拟 CPU 切换. 三个窗口, 如果一个窗口休眠了, CPU 就会切到其它继续执行.  
 /\*try {  
 Thread.sleep(10);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }\*/* System.*out*.println(Thread.*currentThread*().getName() + "正在售出第 " + tickets + " 张票.");  
 tickets--;  
 }  
 }  
}

**测试类 :**

public class Test1 {  
 public static void main(String[] args) {  
  
 *// 3. 创建一个任务类对象* TicketWindowTask2 task = new TicketWindowTask2();  
  
 *// 4. 创建线程类对象, 并同时将任务类对象作为参数传入* Thread t1 = new Thread(task, "窗口一");  
 Thread t2 = new Thread(task, "窗口二");  
 Thread t3 = new Thread(task, "窗口三");  
  
 *// 5. 启动线程 (底层就会自动执行 Runnable 接口的 run 方法)* t1.start();  
 t2.start();  
 t3.start();  
 }  
}

## 21.5 Lock与ReentrantLock 锁 :

**类 ReentrantLock :** 一个可重入的互斥锁 [Lock](mk:@MSITStore:C:\Users\xieco\Desktop\JDK9%20API文档\_JDK6API中文参考%5b沈东良%5d(070114).chm::/java/util/concurrent/locks/Lock.html)，它具有与使用 synchronized 方法和语句所访问的**隐式监视器锁**相同的一些基本行为和语义，但功能更强大。

接口 Lock :

|  |  |
| --- | --- |
| void | [**lock**](mk:@MSITStore:C:\Users\xieco\Desktop\JDK9%20API文档\_JDK6API中文参考%5b沈东良%5d(070114).chm::/java/util/concurrent/locks/Lock.html#lock())()            获取锁。 |
| void | [**unlock**](mk:@MSITStore:C:\Users\xieco\Desktop\JDK9%20API文档\_JDK6API中文参考%5b沈东良%5d(070114).chm::/java/util/concurrent/locks/Lock.html#unlock())**()            释放锁。** |

**任务类 :**

import java.util.concurrent.locks.Lock;  
import java.util.concurrent.locks.ReentrantLock;  
  
*// 1. 自定义类, 实现 Runnable 接口. (售票任务类)*public class TicketWindowTask3 implements Runnable {  
 *// 属性* private int tickets = 1000;  
  
 *// 定义一把锁* private Lock lock = new ReentrantLock();  
  
 *// 2. 重写 Runnable 接口的 run() 方法.* @Override  
 public void run() {  
 *// Object lock = new Object(); 不能将锁对象定义为 run 方法内部, 因为 run 方法会被每条子线程都执行.  
 // 循环售票* while (true) {  
sellTickets();  
  
 *// 判断* if (tickets <= 0) {  
 break;  
 }  
 }  
 }  
  
 private void sellTickets() {  
 *// 加锁 (互斥锁)* lock.lock();  
 if (tickets > 0) {  
 *// 模拟 CPU 切换. 三个窗口, 如果一个窗口休眠了, CPU 就会切到其它继续执行.  
 /\*try {  
 Thread.sleep(10);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }\*/* System.*out*.println(Thread.*currentThread*().getName() + "正在售出第 " + tickets + " 张票.");  
 tickets--;  
 }  
 *// 解锁* lock.unlock();  
 }  
}

## 21.6 静态同步方法 : (了解)

**任务类 :**

*// 1. 自定义类, 实现 Runnable 接口. (售票任务类)*public class TicketWindowTask4 implements Runnable {  
 *// 属性* private static int *tickets* = 1000;  
  
 *// 定义一把锁* private Object lock = new Object();  
  
 *// 2. 重写 Runnable 接口的 run() 方法.* @Override  
 public void run() {  
 *// Object lock = new Object(); 不能将锁对象定义为 run 方法内部, 因为 run 方法会被每条子线程都执行.  
 // 循环售票* while (true) {  
TicketWindowTask4.*sellTickets*();  
 *// sellTickets();  
  
 // 判断* if (*tickets* <= 0) {  
 break;  
 }  
 }  
 }  
  
 *// 请问 : 静态同步方法有没有锁对象 ??? TicketWindowTask4.class 对象,(字节码文件) 就是静态同步方法的锁对象.* private static synchronized void sellTickets() {  
 if (*tickets* > 0) {  
 *// 模拟 CPU 切换. 三个窗口, 如果一个窗口休眠了, CPU 就会切到其它继续执行.  
 /\*try {  
 Thread.sleep(10);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }\*/* System.*out*.println(Thread.*currentThread*().getName() + "正在售出第 " + *tickets* + " 张票.");  
 *tickets*--;  
 }  
 }  
}

## 21.7 小结 :

多线程 : 同一个程序的多条执行路径.

主线程的入口是 main 方法, 主线程有虚拟机创建, 并实现调用.

子线程的入口是 Runnable 接口中定义的 run 方法. 子线程必须使用 Thread 对象的 start 方法启动.

多线程的实现方案有两种 :

1. 继承. Thread 类.

2. 接口. Runnable 接口. (推荐使用)

多线程操作共享数据会出现数据安全问题. 请问, 如果解决 ??? 上锁.

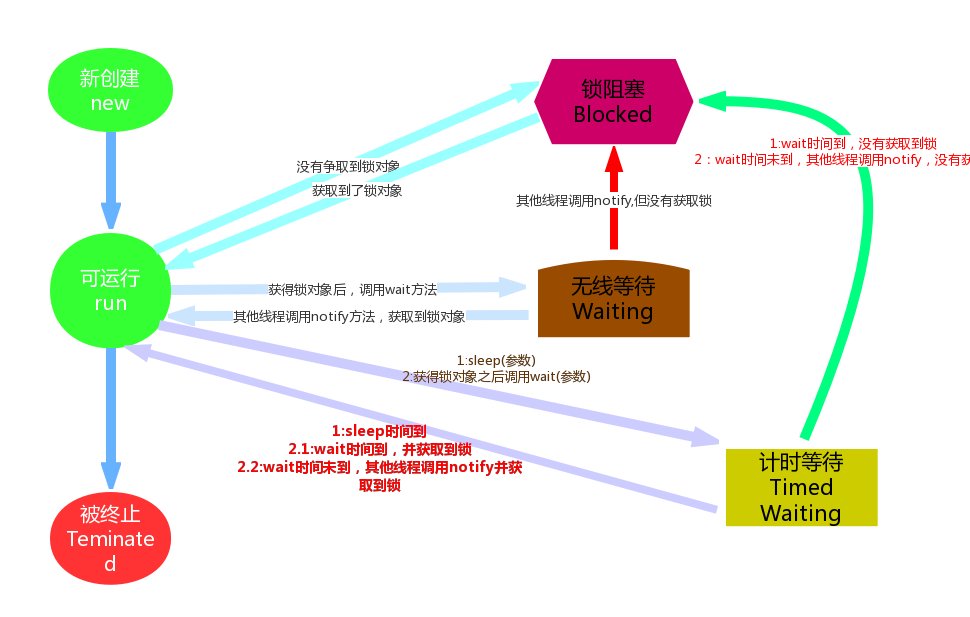
1. 同步锁

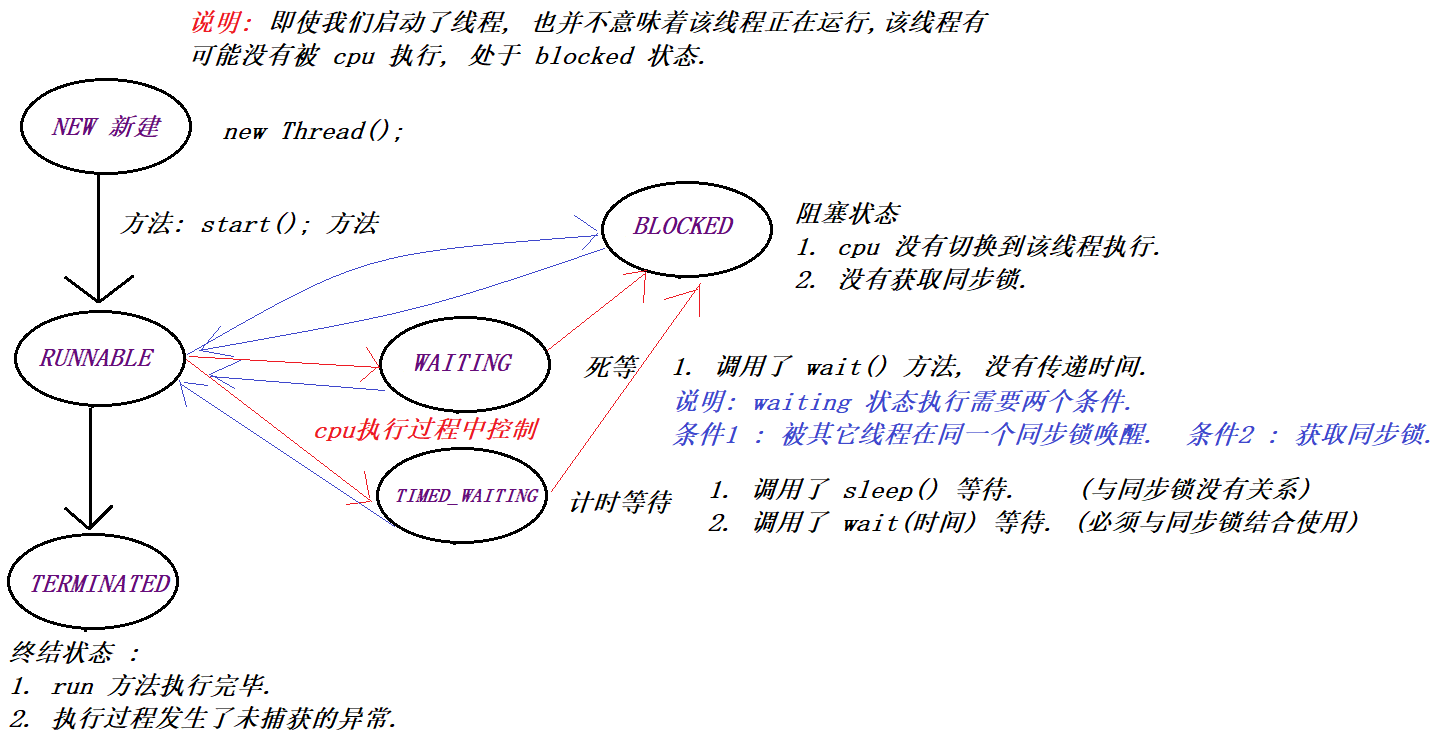
1.1 同步代码块. synchronized(锁对象) { }

1.2 同步方法. synchronized 同步方法的锁对象为 this, 当前对象.

2. 互斥锁 Lock接口 (加锁lock, 解锁 unlock); 实现类 : ReentrantLock

# 22. 线程状态 (了解)





## 22.1 Timed Waiting (计时等待)

实现一 : Thread.sleep(时间毫秒); 意味着当前线程就会进入到计时等待状态.

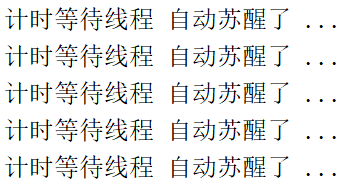
实现二 : 同步锁环境 + 同步锁对象.wait(时间毫秒); 意味着当前线程就会进入到计时等待状态.

演示一 :

public class Test2 {  
 public static void main(String[] args) {  
  
 *// 实现一 : Thread.sleep(时间毫秒); 意味着当前线程就会进入到计时等待状态.* new Thread(new Runnable() {  
 @Override  
 public void run() {  
 while (true) {  
  
 try {  
 Thread.*sleep*(1000);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
  
 System.*out*.println(Thread.*currentThread*().getName() + " 自动苏醒了 ...");  
 }  
 }  
 }, "计时等待线程").start();  
 }  
}

**演示二 :**

public class Test3 {  
 public static void main(String[] args) {  
  
 *// 实现二 : 同步锁环境 + 同步锁对象.wait(时间毫秒); 意味着当前线程就会进入到计时等待状态.* Object lock = new Object();  
  
 new Thread(new Runnable() {  
 @Override  
 public void run() {  
 while (true) {  
  
 synchronized (lock) {  
 try {  
 lock.wait(1000);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
  
 System.*out*.println(Thread.*currentThread*().getName() + " 自动苏醒了 ...");  
 }  
 }  
 }, "计时等待线程").start();  
 }  
}



## 22.2 Blocked 锁阻塞 :

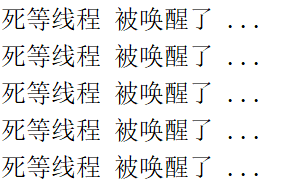
情况1 : 同时开启了多个线程, 如果线程没有被 cpu 切换到, 那么该线程就处于阻塞状态.

情况2 : 如果有多条线程, 并代码中存在 `同步锁` 环境, cpu 切换到该线程, 意味着该线程就能够被执行吗 ??? 取决于该线程是否拥有同步锁, 如果没有同步锁, 那么该线程就处理阻塞状态, 因此不能被执行.

举例 : 售票案例, 拥有 t1, t2, t3 线程, 如果 t1 线程进入到同步代码中, 那么 t2, t3 会自动进入到阻塞状态. 如果 t1 线程从同步代码中出来, 此时 t2, t3 线程就会自动解除阻塞状态.

## 22.3 Waiting (无限等待) :

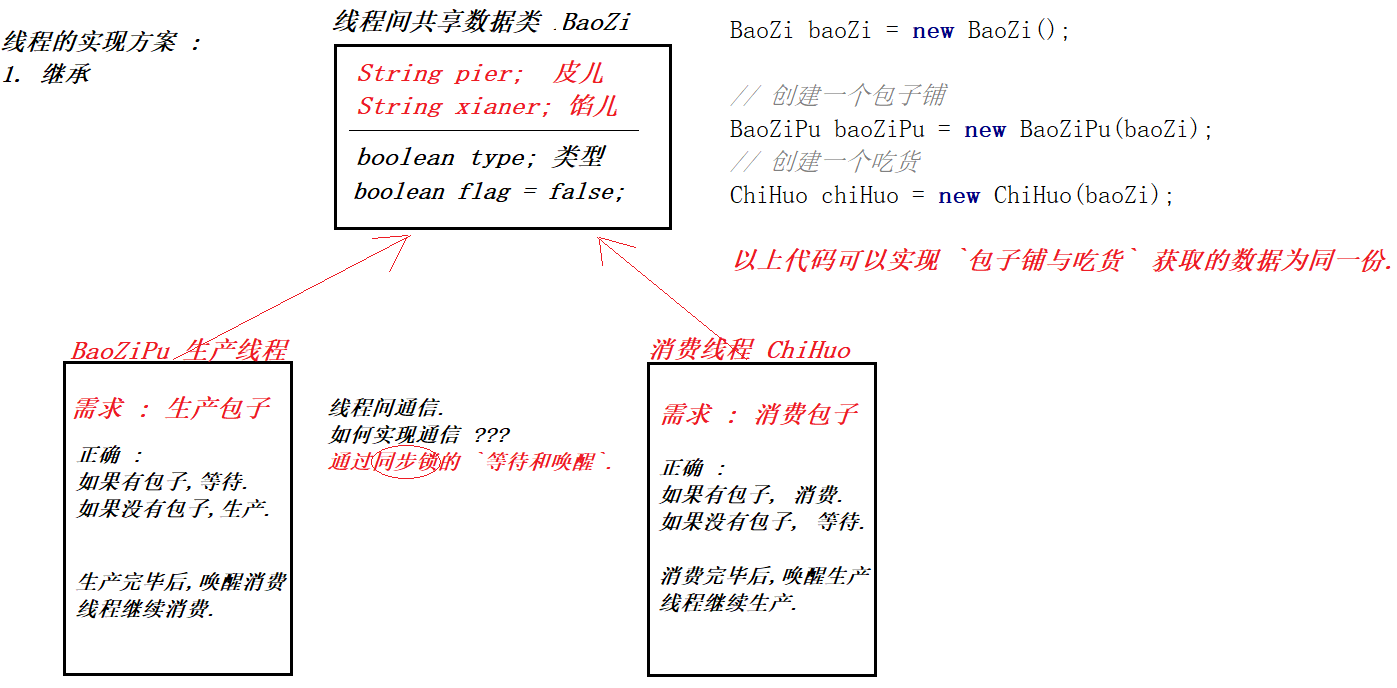
public class Test4 {  
 public static void main(String[] args) {  
  
 *// 实现 : 同步环境 + wait() + notify();* Object lock = new Object();  
  
 new Thread(new Runnable() {  
 @Override  
 public void run() {  
 while (true) {  
  
 synchronized (lock) {  
 try {  
 lock.wait();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
  
 System.*out*.println(Thread.*currentThread*().getName() + " 被唤醒了 ...");  
 }  
 }  
 }, "死等线程").start();  
  
 *// 唤醒线程* new Thread(new Runnable() {  
 @Override  
 public void run() {  
 while (true) {  
  
 try {  
 Thread.*sleep*(1000);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
  
 synchronized (lock) {  
 lock.notify();  
 }  
 }  
 }  
 }, "唤醒线程").start();  
 }  
}



# 23. 等待唤醒机制 :

说明 : 等待唤醒机制也被称为 `产生消费模型`.

## 案例一 : BaoZi 资源



**共享资源类 : BaoZi**

*// 线程间共享数据的类*public class BaoZi {  
 *// 属性* private String pier;  
 private String xianer;  
  
 *// 判断生产类型* boolean type = false;  
 *// 判断是否有包子* boolean flag = false; *// 默认没有包子* public String getPier() {  
 return pier;  
 }  
  
 public void setPier(String pier) {  
 this.pier = pier;  
 }  
  
 public String getXianer() {  
 return xianer;  
 }  
  
 public void setXianer(String xianer) {  
 this.xianer = xianer;  
 }  
}

**生产线程 : BaoZiPu**

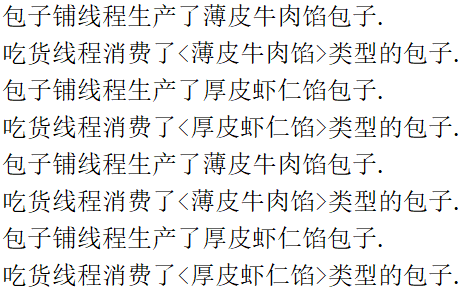
*// 包子铺负责不断生产包子*public class BaoZiPu extends Thread {  
 *// 属性* private BaoZi baoZi;  
  
 *// 构造方法* public BaoZiPu(BaoZi baoZi) {  
 this.baoZi = baoZi;  
 }  
  
 @Override  
 public void run() {  
 *// 不断生产包子* while (true) {  
 synchronized (baoZi) {  
 *// 开始生产* produce();  
 }  
 }  
 }  
  
 private void produce() {  
  
 *// 区间值  
 /\*try {  
 long time = (long) (Math.random() \* 1000);  
 Thread.sleep(time);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }\*/  
  
 // 1. 判断, 如果有包子, 生产线程就需要等待.* if (baoZi.flag == true) {  
 *// 等待* try {  
 baoZi.wait();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
  
 *// 2. 没有包子, 生产线程就开始生产* if (baoZi.type == false) {  
 baoZi.setPier("薄皮");  
 baoZi.setXianer("牛肉馅");  
 } else {  
 baoZi.setPier("厚皮");  
 baoZi.setXianer("虾仁馅");  
 }  
  
 System.*out*.println(getName() + "生产了" + baoZi.getPier() + baoZi.getXianer() + "包子.");  
  
 *// 3. 修改类型 (取反)* baoZi.type = !baoZi.type;  
  
 *// 4. 修改包子的 `标记` flag* baoZi.flag = true;  
  
 *// 5. 唤醒另一个线程开始消费* baoZi.notify();  
 }  
}

**消费线程 : ChiHuo**

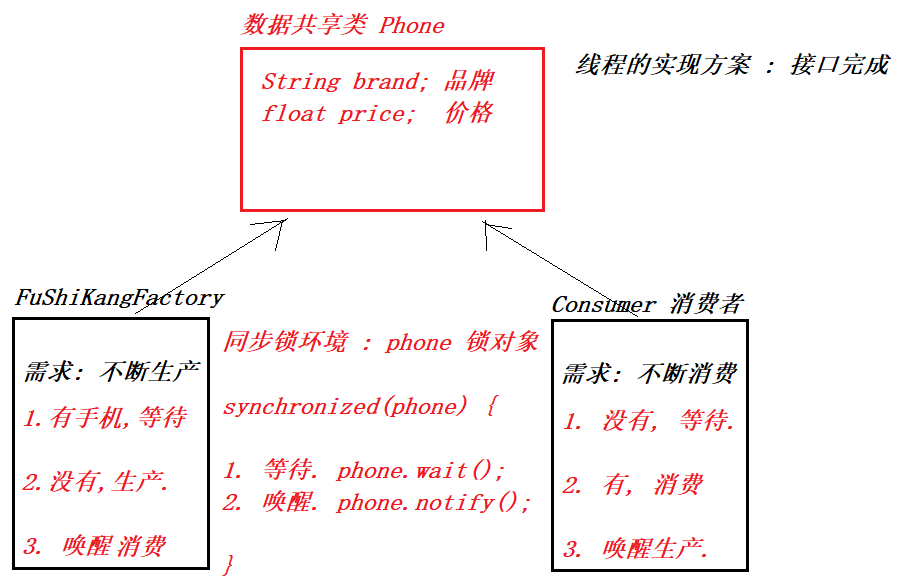
public class ChiHuo extends Thread {  
 *// 属性* private BaoZi baoZi;  
  
 *// 构造方法* public ChiHuo(BaoZi baoZi) {  
 this.baoZi = baoZi;  
 }  
  
 @Override  
 public void run() {  
 *// 不断消费* while (true) {  
 synchronized (baoZi) {  
 *// 开始消费* consume();  
 }  
 }  
 }  
  
 private void consume() {  
  
 *// 区间值  
 /\*try {  
 long time = (long) (Math.random() \* 1000);  
 Thread.sleep(time);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }\*/  
  
 // 1. 判断, 如果没有包子, 消费线程就等待* if (baoZi.flag == false) {  
 try {  
 baoZi.wait();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
  
 *// 2. 如果有包子, 消费线程就开始消费* String pier = baoZi.getPier();  
 String xianer = baoZi.getXianer();  
 System.*out*.println(getName() + "消费了<" + pier + xianer + ">类型的包子.");  
  
 *// 3. 给数据进行清空* baoZi.setPier(null);  
 baoZi.setXianer(null);  
  
 *// 4. 修改包子的标记.* baoZi.flag = false;  
  
 *// 5. 唤醒生产线程开始生产包子* baoZi.notify();  
 }  
}

**测试类 :**

public class Test1 {  
 public static void main(String[] args) {  
  
 BaoZi baoZi = new BaoZi();  
  
 *// 创建一个包子铺* BaoZiPu baoZiPu = new BaoZiPu(baoZi);  
 baoZiPu.setName("包子铺线程");  
 *// 创建一个吃货* ChiHuo chiHuo = new ChiHuo(baoZi);  
 chiHuo.setName("吃货线程");  
  
 *// 启动线程* baoZiPu.start();  
 chiHuo.start();  
 }  
}



## 案例二 : Phone 资源



**Phone 共享资源类 :**

*// 共享资源*public class Phone {  
 *// 属性* private String brand;  
 private float price;  
  
 *// 标记* boolean flag = false;  
 *// 类型判断* boolean type = false;  
  
 public String getBrand() {  
 return brand;  
 }  
  
 public void setBrand(String brand) {  
 this.brand = brand;  
 }  
  
 public float getPrice() {  
 return price;  
 }  
  
 public void setPrice(float price) {  
 this.price = price;  
 }  
}

**FuShiKangFactory 生产线程 :**

*// 富士康工厂 : 生产手机*public class FuShiKangFactory implements Runnable {  
 *// 属性* private Phone phone;  
  
 *// 构造方法* public FuShiKangFactory(Phone phone) {  
 this.phone = phone;  
 }  
  
 @Override  
 public void run() {  
 *// 不断生产* while (true) {  
 *// 同步锁环境, 因为要执行 `等待和唤醒`* synchronized (phone) {  
  
 *// 1. 如果有 phone, 生产线程就进入到等待状态* if (phone.flag == true) {  
 try {  
 phone.wait();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
  
 *// 2. 没有 phone, 需要生产* if (phone.type == false) {  
 phone.setBrand("iPhoneXXX");  
 phone.setPrice(18888.88f);  
 } else {  
 phone.setBrand("Nokia王者归来");  
 phone.setPrice(88.88f);  
 }  
 *// 2.2 切换类型 (反转)* phone.type = !phone.type;  
 System.*out*.println(Thread.*currentThread*().getName() + "生产了 " + phone.getBrand() + " : " + phone.getPrice() + " 手机.");  
  
 *// 3. 修改手机的标记* phone.flag = true;  
  
 *// 4. 唤醒消费线程来消费* phone.notify();  
 }  
 }  
 }  
}

**Consumer 消费线程 :**

public class Consumer implements Runnable {  
 *// 属性 (数据)* private Phone phone;  
  
 *// 构造方法* public Consumer(Phone phone) {  
 this.phone = phone;  
 }  
  
 @Override  
 public void run() {  
 *// 不断消费* while (true) {  
 *// 同步锁环境, 因为要使用线程的等待和唤醒.* synchronized (phone) {  
  
 *// 1. 判断, 如果没有手机, 就需要等待* if (phone.flag == false) {  
 try {  
 phone.wait();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
  
 *// 2. 如果有手机, 开始消费* String brand = phone.getBrand();  
 float price = phone.getPrice();  
 System.*out*.println(Thread.*currentThread*().getName() + "消费了<" + brand + " : " + price + ">手机.");  
  
 *// 2.2 清空数据* phone.setBrand(null);  
 phone.setPrice(0.0f);  
  
 *// 3. 修改手机的标记* phone.flag = false;  
  
 *// 4. 唤醒生产线程开始生产* phone.notify();  
 }  
 }  
 }  
}

**测试类 :**

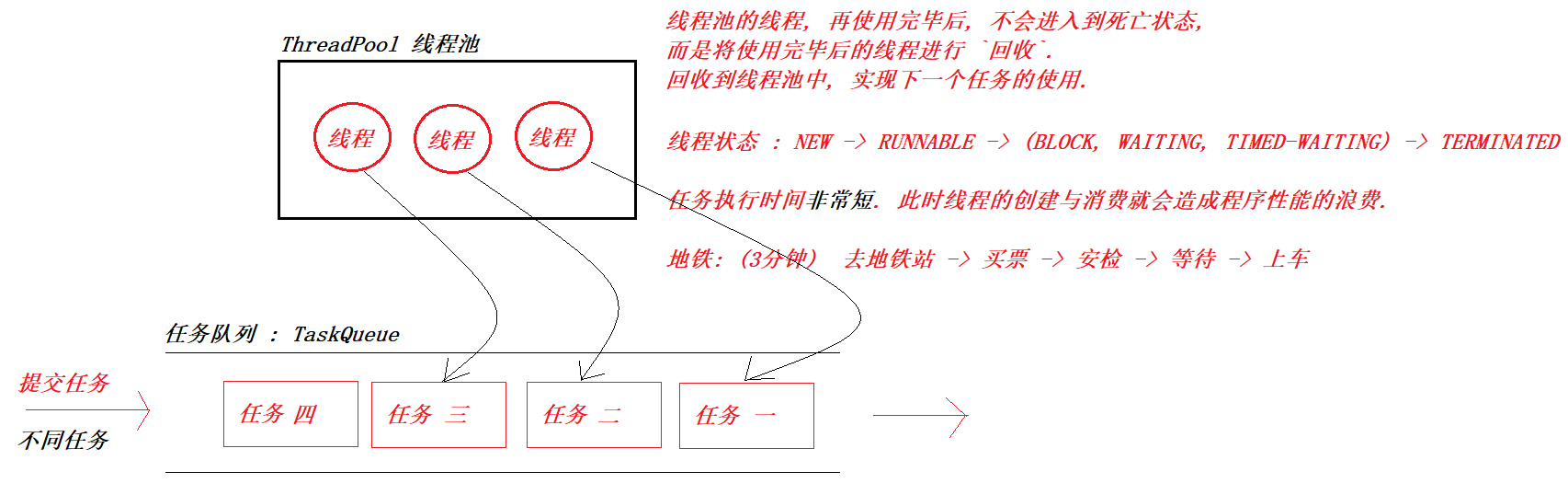
public class Test1 {  
 public static void main(String[] args) {  
  
 *// 共享资源类对象* Phone phone = new Phone();  
  
 *// 1. 生产任务* FuShiKangFactory factory = new FuShiKangFactory(phone);  
 *// 2. 消费任务* Consumer consumer = new Consumer(phone);  
  
 *// 3. 让任务与线程实现关联* Thread t1 = new Thread(factory, "生产线程");  
 Thread t2 = new Thread(consumer, "消费线程");  
  
 *// 4. 启动线程* t1.start();  
 t2.start();  
 }  
}



# 24. 线程池 : ThreadPool (了解)

类 ThreadPoolExecutor :

强烈建议程序员使用较为方便的 [Executors](mk:@MSITStore:C:\Users\xieco\Desktop\JDK9%20API文档\_JDK6API中文参考%5b沈东良%5d(070114).chm::/java/util/concurrent/Executors.html) 工厂方法 [Executors.newCachedThreadPool()](mk:@MSITStore:C:\Users\xieco\Desktop\JDK9%20API文档\_JDK6API中文参考%5b沈东良%5d(070114).chm::/java/util/concurrent/Executors.html#newCachedThreadPool())（无界线程池，可以进行自动线程回收）、[Executors.newFixedThreadPool(int)](mk:@MSITStore:C:\Users\xieco\Desktop\JDK9%20API文档\_JDK6API中文参考%5b沈东良%5d(070114).chm::/java/util/concurrent/Executors.html#newFixedThreadPool(int))（固定大小线程池）和 [Executors.newSingleThreadExecutor()](mk:@MSITStore:C:\Users\xieco\Desktop\JDK9%20API文档\_JDK6API中文参考%5b沈东良%5d(070114).chm::/java/util/concurrent/Executors.html#newSingleThreadExecutor())（单个后台线程），它们均为大多数使用场景预定义了设置。

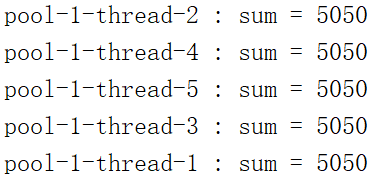


**方式一 : 手动创建线程, 并手动启动线程方式.**

public class Test1 {  
 public static void main(String[] args) {  
  
 *// 任务对象* for (int i = 0; i < 50; i++) {  
 Runnable task = new Runnable() {  
 @Override  
 public void run() {  
 int sum = 0;  
 for (int i = 1; i <= 100; i++) {  
 sum += i;  
 }  
 System.*out*.println(Thread.*currentThread*().getName() + " : sum = " + sum);  
 }  
 };  
  
 *// 需求 : 需要以上 50 个任务同时被执行* Thread t = new Thread(task);  
 t.start();  
 }  
 }  
}

**方式二 :创建线程池, 直接提交50个不同的任务对象.**

public class Test2 {  
 public static void main(String[] args) {  
  
 // 1. 创建一个线程池对象  
 // ExecutorService threadPool = Executors.newCachedThreadPool();  
 ExecutorService threadPool = Executors.newFixedThreadPool(5);  
 // ExecutorService threadPool = Executors.newSingleThreadExecutor();  
  
 // 任务对象  
 for (int i = 0; i < 50; i++) {  
 Runnable task = new Runnable() {  
 @Override  
 public void run() {  
 int sum = 0;  
 for (int i = 1; i <= 100; i++) {  
 sum += i;  
 }  
 System.out.println(Thread.currentThread().getName() + " : sum = " + sum);  
 }  
 };  
  
 // 需求 : 需要以上 50 个任务同时被执行  
 threadPool.submit(task);  
 }  
  
 // 关闭线程池  
 threadPool.shutdown();  
 }  
}



**模拟 : 使用线程池模拟卖票任务**

public class Test4 {  
 public static void main(String[] args) {  
  
 *// 1. 创建一个线程池对象* ExecutorService threadPool = Executors.*newFixedThreadPool*(3);  
  
 *// 2. 创建任务对象 (多个不同类型的任务对象)* TicketWindowTask task = new TicketWindowTask();  
  
 *// 3. 提交任务 (将各个类型的任务对象提交给线程池执行)* threadPool.submit(task); *// 卖票任务* threadPool.submit(task); *// 下载任务* threadPool.submit(task); *// 上传任务  
  
 // 4. 关闭线程池* threadPool.shutdown();  
 }  
}

# 25. Lambda 表达式 :

Lambda 表达式就是匿名对象的 `简写形式`.

语法 :

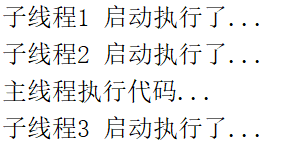
1. 一个小括号. 接口中抽象方法的参数列表

2. 一个箭头 分隔符. 左边是参数列表, 右边是方法实现体.

3. 一个大括号 接口中抽象方法的实现体代码.

## 25.1 Runnable 接口的优化写法 :

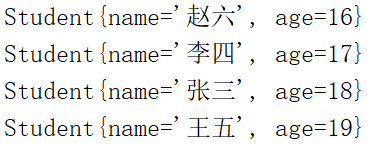
public class Test1 {  
 public static void main(String[] args) {  
  
 *// new Thread(Runnable target); 传参时, 传递了一个匿名实现类对象.* new Thread(new Runnable() {  
 @Override  
 public void run() {  
 System.*out*.println("子线程1 启动执行了...");  
 }  
 }).start();  
  
 *// 问题 : 匿名实现类对象能否再次简化.  
 // 可以, 使用 Lambda 表达式.  
 /\*  
 1. 一个小括号. 含义: 抽象方法的参数列表  
 2. 一个箭头. 含义: 分隔符  
 3. 一个大括号. 含义: 抽象方法的实现体代码.  
 \*/* new Thread(() -> {System.*out*.println("子线程2 启动执行了...");}).start();  
  
 *// Lambda 表达式的省略写法.  
 /\*  
 规则1 : 如果参数列表仅有一个参数, 可以省略小括号. 无论是否拥有参数, 都可以省略参数类型.  
 规则2 : 如果方法体仅有一条语句, 可以直接省略大括号和分号.  
 规则3 : 方法如果有返回值, 并且仅有一条语句. 同时也可以省略 return 关键字.  
 \*/* new Thread(() -> System.*out*.println("子线程3 启动执行了...")).start();  
  
 System.*out*.println("主线程执行代码...");  
 }  
}



## 25.2 Comparator 接口的优化写法 :

需求 : 对集合中的 Student 对象进行排序. 规则为: 按照年龄从小到大排列.

public class Test2 {  
 public static void main(String[] args) {  
  
 ArrayList<Student> list = new ArrayList<>();  
  
 list.add(new Student("张三", 18));  
 list.add(new Student("李四", 17));  
 list.add(new Student("王五", 19));  
 list.add(new Student("赵六", 16));  
  
 *// 需求 : 将集合中的元素实现 `从小到大` 的年龄排序.  
 // 实现方式一 : 匿名实现类. new Comparator<Student>() { ... }  
 /\*Collections.sort(list, new Comparator<Student>() {  
 @Override  
 public int compare(Student o1, Student o2) {  
 return o1.getAge() - o2.getAge();  
 }  
 });\*/  
  
 // 实现方式二 : Lambda 表达式. (简化匿名实现类书写)  
 /\*Collections.sort(list, (Student o1, Student o2) -> {  
 return o1.getAge() - o2.getAge();  
 });\*/  
  
 // 实现方式三 : 省略形式* Collections.*sort*(list, (o1, o2) -> o1.getAge() - o2.getAge());  
  
 *// 遍历 list 集合* for (Student stu : list) {  
 System.*out*.println(stu);  
 }  
 }  
}



## 25.3 函数式接口的语法检查说明 :

Runnable 接口 和 Comparator 接口. 这两个接口都是 `函数式` 接口.

**请问 :** 什么是函数式接口 ??? **有, 且仅有一个抽象方法**的接口被称为 `函数式` 接口. (自动推导)

接口 Runnable :

|  |  |
| --- | --- |
| void | [**run**](mk:@MSITStore:C:\Users\xieco\Desktop\JDK9%20API文档\_JDK6API中文参考%5b沈东良%5d(070114).chm::/java/lang/Runnable.html#run())()            使用实现接口 Runnable 的对象创建一个线程时，启动该线程将导致在独立执行的线程中调用对象的 run 方法。 |

接口 Comparator :

|  |  |
| --- | --- |
| int | [**compare**](mk:@MSITStore:C:\Users\xieco\Desktop\JDK9%20API文档\_JDK6API中文参考%5b沈东良%5d(070114).chm::/java/util/Comparator.html#compare(T,%20T))([T](mk:@MSITStore:C:\Users\xieco\Desktop\JDK9%20API文档\_JDK6API中文参考%5b沈东良%5d(070114).chm::/java/util/Comparator.html) o1, [T](mk:@MSITStore:C:\Users\xieco\Desktop\JDK9%20API文档\_JDK6API中文参考%5b沈东良%5d(070114).chm::/java/util/Comparator.html) o2)            比较用来排序的两个参数。 |
| boolean | [**equals**](mk:@MSITStore:C:\Users\xieco\Desktop\JDK9%20API文档\_JDK6API中文参考%5b沈东良%5d(070114).chm::/java/util/Comparator.html#equals(java.lang.Object))([Object](mk:@MSITStore:C:\Users\xieco\Desktop\JDK9%20API文档\_JDK6API中文参考%5b沈东良%5d(070114).chm::/java/lang/Object.html) obj)            指示某个其他对象是否“等于”此 Comparator。 |

**请问 :** 为什么 Comparator 也被称为一个函数式接口呢 ???

因为与Object类型相同的抽象方法, 函数式接口默认是不会推导的.因为该方法可以被认为非抽象方法.

语法检查关键字 : **@FunctionalInterface 函数式接口定义**

// 接口的作用 : 定义行为规范. 如果接口中的行为规范与 Object 类定义的方法一致, 那么该行为规则就可以认为是 `通用行为规则`  
// 说明 : 因为任何一个类的对象都自动拥有 Object 类中的所有方法. (行为)  
// Lambda 表达式不会去推导与 Object 类相同的行为规范.  
  
// 语法检查 :  
@FunctionalInterface // 目标: 唯一的那个抽象方法.  
public interface MyComparator {  
  
 int max(int a, int b);  
 // double min(int a, int b); 函数式接口中不允许存在多个抽象方法.  
  
 // 这些与 Object 类相同的方法, 可以理解为 `非抽象方法`.  
 boolean equals(Object obj);  
 int hashCode();  
}

**测试类 :**

public class Test1 {  
 public static void main(String[] args) {  
  
 /\*  
 int max = testMethod(10, 20, new MyComparator() {  
 @Override  
 public int max(int a, int b) {  
 return a > b ? a : b;  
 }  
 });  
 \*/  
  
 // 如何实现推导呢 ??? 根据抽象方法参数类型和返回值实现推导.  
 int max = testMethod(10, 20, (a, b) -> { return a > b ? a : b; }); // 两个参数,一个返回值.  
 System.out.println("max = " + max);  
 }  
  
 public static int testMethod(int n1, int n2, MyComparator comparator) {  
 int max = comparator.max(n1, n2);  
 return max;  
 }  
}



## 25.4 Lambda 练习 : (无参无返回值)

Lambda 表达式, 仅可以作为参数进行传递.

条件1 : 必须有方法, 将接口作为方法的参数列表.

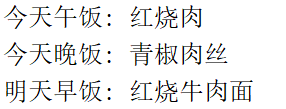
条件2 : 必须有接口, 而且该接口有, 且仅有一个抽象方法.

**接口定义 :**

public interface Cook {  
  
 *// 抽象方法* void makeFood(String food);  
}

**测试类 :**

public class Test3 {  
 public static void main(String[] args) {  
  
 *// 调用 :  
 // 方式一 : 匿名实现类  
 keepAlive*("红烧肉", new Cook() {  
 @Override  
 public void makeFood(String food) {  
 System.*out*.println("今天午饭: " + food);  
 }  
 });  
  
 *// 方式二 : Lambda 表达式  
 keepAlive*("青椒肉丝", (food) -> { System.*out*.println("今天晚饭: " + food); });  
  
 *// 方式三 : 省略写法  
 keepAlive*("红烧牛肉面", food -> System.*out*.println("明天早饭: " + food));  
 }  
  
 *// 方法 : 将接口作为方法的参数列表, 为了能使用 Lambda 表达式作为参数传递  
 // Cook 是一个接口, 可以传递匿名实现类对象.* public static void keepAlive(String food, Cook cook) {  
 cook.makeFood(food);  
 }  
}



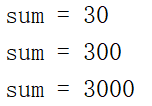
## 25.5 Lambda 练习 : (有参有返回值)

**接口定义 :**

public interface Calculator {  
  
 *// 抽象方法* int sum(int num1, int num2);  
}

**测试类 :**

public class Test4 {  
 public static void main(String[] args) {  
  
 *// 调用 :  
 // 方式一 : 匿名实现类  
 calucatorMethod*(10, 20, new Calculator() {  
 @Override  
 public int sum(int num1, int num2) {  
 return num1 + num2;  
 }  
 });  
  
 *// 方式二 : Lambda 写法  
 calucatorMethod*(100, 200, (int a, int b) -> {return a + b;});  
  
 *// 方式三 : 省略写法  
 calucatorMethod*(1000, 2000, (a, b) -> a + b);  
 }  
  
 *// 方法, 将接口作为方法的参数类型, 为了调用方法时, 能够传递 Lambda 表达式* public static void calucatorMethod(int n1, int n2, Calculator calculator) {  
 int sum = calculator.sum(n1, n2);  
 System.*out*.println("sum = " + sum);  
 }  
}



## 25.6 Lambda 表达式的使用前提条件 :

条件一 : 必须有函数式接口. (有, 且仅有一个抽象方法的接口)

条件二 : 必须有方法将函数式接口作为参数列表.

## 25.7 Lambda 表达式的省略规则 :

规则1 : 如果仅有一个参数, 可以将小括号省略.

规则2 : 参数列表中的所有参数类型都可以省略.

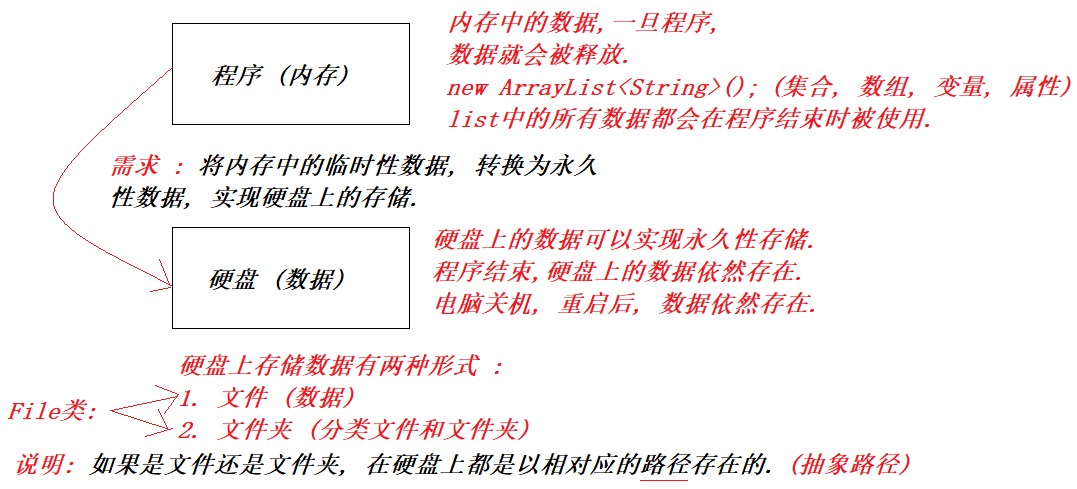
规则3 : 如果方法体仅有一条输出语句, 可以直接省略大括号和分号. 无论是否有返回值, return 关键字都可以省略.

# 26. File 文件类 : (重点)

## 26.1 File 类的概述 :

File类 : 文件和目录路径名的抽象表示形式。

**File 类其实就是永久性数据存在于硬盘上的一种形式.**



## 26.2 File 类的构造方法 :

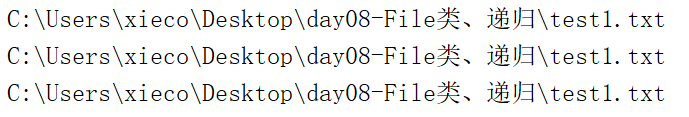
public File(String pathname) ：通过将给定的**路径名字符串**转换为抽象路径名来创建新的 File实例。

public File(String parent, String child) ：从**父路径名字符串和子路径名字符串**创建新的 File实例。

public File(File parent, String child) ：从**父抽象路径名和子路径名字符串**创建新的 File实例。

**请问 :** 为什么创建 File 对象, 没有无参构造方法呢 ??? 因为文件对象必须与硬盘上的路径相对应.

public class Test1 {  
 public static void main(String[] args) throws IOException {  
  
 *// public File(String pathname) ：通过将给定的路径名字符串转换为抽象路径名来创建新的 File实例。* File file1 = new File("C:\\Users\\xieco\\Desktop\\day08-File类、递归\\test1.txt"); *// 绝对路径 (从硬盘的根目标)  
 // File file1 = new File("test2.txt"); // 相对路径 (针对相当项目而言的路径)* System.*out*.println(file1);  
 *// file1.createNewFile();  
  
 // public File(String parent, String child) ：从父路径名字符串和子路径名字符串创建新的 File实例。* File file2 = new File("C:\\Users\\xieco\\Desktop\\day08-File类、递归", "test1.txt");  
 System.*out*.println(file2);  
  
 *// public File(File parent, String child) ：从父抽象路径名和子路径名字符串创建新的 File实例。* File parent = new File("C:\\Users\\xieco\\Desktop\\day08-File类、递归");  
 File file3 = new File(parent, "test1.txt");  
 System.*out*.println(file3);  
 }  
}



## 26.3 File类常用获取功能的方法 :

public String getAbsolutePath() ：返回此File的绝对路径名字符串。 获取文件对象的绝对路径(从根盘符开始)

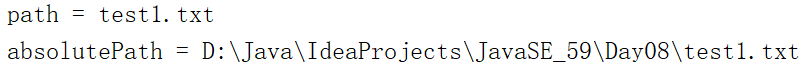
public String getPath() ：将此File转换为路径名字符串。获取构造路径

public String getName() ：返回由此File表示的文件或目录的名称。 文件名称

public long length() ：返回由此File表示的文件的长度。 文件中的数据长度.

**演示一 :**

public class Test2 {  
 public static void main(String[] args) {  
  
 File file = new File("test1.txt"); *// 相对路径 (当前项目的根目录)* String path = file.getPath(); *// 构建路径 (相对路径)* System.*out*.println("path = " + path);  
  
 String absolutePath = file.getAbsolutePath(); *// 绝对路径* System.*out*.println("absolutePath = " + absolutePath);  
 }  
}



**演示二 :**

public class Test3 {  
 public static void main(String[] args) {  
  
 File file1 = new File("test1.txt");  
 File file2 = new File("C:\\Users\\xieco\\Desktop\\day08-File类、递归\\test1.txt");  
  
 *// getName(); 获取文件名称* String name1 = file1.getName();  
 String name2 = file2.getName();  
 System.*out*.println("name1 = " + name1);  
 System.*out*.println("name2 = " + name2);  
  
 *// length(); 获取文件中的数据长度. (字节单位 byte)  
 // 说明 : 不可以直接获取文件夹的长度. 只能获取文件的长度.* long length = file1.length();  
 System.*out*.println("length = " + length);  
 }  
}

## 26.5 File 类的常用判断功能方法 :

public boolean exists() ：此File表示的文件或目录是否实际存在。 是否存在

public boolean isDirectory() ：此File表示的是否为目录。 是否为目录/文件夹

public boolean isFile() ：此File表示的是否为文件。 是否为文件

public class Test4 {  
 public static void main(String[] args) {  
  
 File file1 = new File("test1.txt");  
 File file2 = new File("C:\\Users\\xieco\\Desktop\\day08-File类、递归\\test1.txt");  
  
 *// exists(); 是否存在* boolean file1Exists = file1.exists();  
 System.*out*.println("file1Exists = " + file1Exists);  
  
 boolean file2Exists = file2.exists();  
 System.*out*.println("file2Exists = " + file2Exists);  
  
 *// isDirectory(); 判断是否为文件夹 (文件夹是不应该有后缀名的)* boolean directory = file1.isDirectory();  
 System.*out*.println("directory = " + directory);  
  
 *// isFile(); 判断是否为文件 (文件应该存在后缀名)* boolean file = file1.isFile();  
 System.*out*.println("file = " + file);  
 }  
}

## 26.6 File类常用创建与删除功能方法 :

public boolean createNewFile() ：当且仅当具有该名称的文件尚不存在时，创建一个新的空文件。

public boolean delete() ：删除由此File表示的文件或目录。 删除文件/文件夹

public boolean mkdir() ：创建由此File表示的目录。(make directory 只能创建 `单级` 目录)

public boolean mkdirs() ：创建由此File表示的目录，包括任何必需但不存在的父目录。(make directories 可以创建 `多级` 目录)

public File getParentFile() ：返回次抽象路径名父目录的抽象路劲名; 如果此路径名没有指定父目录, 则返回 null.

**创建文件 :**

后缀名 : 给相对应的软件查看的. 记事本 (txt) 照片 (jpg, png) 视频 (avi, mp4, flv)

**演示一 :**

public class Test5 {  
 public static void main(String[] args) throws IOException {  
  
 File file = new File("C:\\Users\\xieco\\Desktop\\day08-File类、递归\\test1.txt");  
 *// File file = new File("C:\\Users\\xieco\\Desktop\\day09\\test1.txt");  
  
 // public boolean createNewFile() ：当且仅当具有该名称的文件尚不存在时，创建一个新的空文件。  
 /\*  
 说明1 : 如果文件对象的父目录存在, 不会抛出异常. 如果父目录不存在, 就会抛出一个 IOException. 输入输出异常.  
 说明2 : 如果父目录存在, 文件不存在, 会创建成功, 如果父目录文件已经存在, 就会创建失败.  
 \*/* boolean result = file.createNewFile();  
 System.*out*.println("result = " + result);  
 }  
}

**演示二 :**

public class Test6 {  
 public static void main(String[] args) throws IOException {  
  
 *// 如果父目录不存在, 优先创建父目录, 然后再创建文件对象. (因为父目录不存在, 是不可以直接创建文件对象的)* File file = new File("C:\\Users\\xieco\\Desktop\\day09\\test1.txt");  
  
 *// public boolean mkdir() ：创建由此File表示的目录。(make directory 只能创建 `单级` 目录)  
 // 需求 : 判断 file 对象的父目录是否存在.  
 // 1. 首先获取父目录文件对象* File parentFile = file.getParentFile();  
 *// 2. 然后判断父目录是否已经存在* if (parentFile.exists() == false) {  
 *// 3. 创建父目录* parentFile.mkdir();  
 }  
 boolean result = file.createNewFile();  
 System.*out*.println("result = " + result);  
 }  
}

**创建文件夹 :**

public class Test8 {  
 public static void main(String[] args) {  
  
 File file = new File("C:\\Users\\xieco\\Desktop\\day09\\folder4");  
  
 *// mkdir(); 仅能创建单级文件夹  
 // boolean result = file.mkdir();  
  
 // mkdirs(); 可以创建多级文件夹, 同样也可以创建单级文件夹* boolean result = file.mkdirs();  
 System.*out*.println("result = " + result);  
 }  
}

**删除 :**

public class Test9 {  
 public static void main(String[] args) {  
  
 /\*  
 说明1 : 如果是文件, 可以直接删除成功.  
 说明2 : 如果是文件夹, 可以直接删除空文件夹, 但是不能删除非空文件夹.  
  
 解释 : 非空文件夹需要使用 `递归` 从内向外依次删除.  
 \*/  
  
 File file = new File("C:\\Users\\xieco\\Desktop\\day09");  
  
 boolean result = file.delete();  
 System.out.println("result = " + result);  
 }  
}

## 26.7 File 类的目录遍历 :

public String[] list() ：返回一个String数组，表示该File目录中的所有子文件或目录。 显示文件的清单

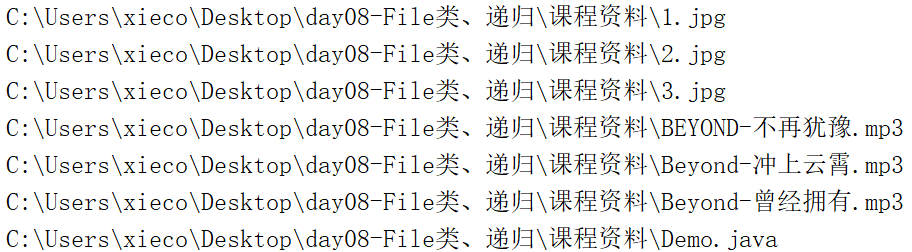
public File[] listFiles() ：返回一个File数组，表示该File目录中的所有的子文件或目录。 所有文件对象的清单.

**list 方法 :**

public class Test10 {  
 public static void main(String[] args) {  
  
 File file = new File("C:\\Users\\xieco\\Desktop\\day08-File类、递归\\课程资料");  
  
 *// 需求1 : 获取 file 对象中的所有 `文件名称清单`* String[] names = file.list();  
 for (String name : names) {  
  
 *// 需求2 : 根据文件名称和父目录对象, 得到一个绝对路径的文件对象* File f = new File(file, name);  
 System.*out*.println(f);  
 }  
 }  
}

**listFiles 方法 :**

public class Test11 {  
 public static void main(String[] args) {  
  
 File file = new File("C:\\Users\\xieco\\Desktop\\day08-File类、递归\\课程资料");  
  
 *// 需求 : 获取 file 文件夹中的所有文件对象清单* File[] files = file.listFiles();  
 for (File f : files) {  
 System.*out*.println(f);  
 }  
 }  
}



# 27. 递归 : recursion (了解)

## 27.1 概述 :

**请问 :** 什么是递归 ??? 方法自己调用自己就是递归. **(内存: 压栈, 弹栈)**

public class Test1 {  
 public static void main(String[] args) {  
  
 *// StackOverflowError, 错误应该尽量避免. (栈溢出错误)  
 main*(args);  
 }  
}

## 27.2 递归求阶乘 :

需求 : 求 5 的阶乘.

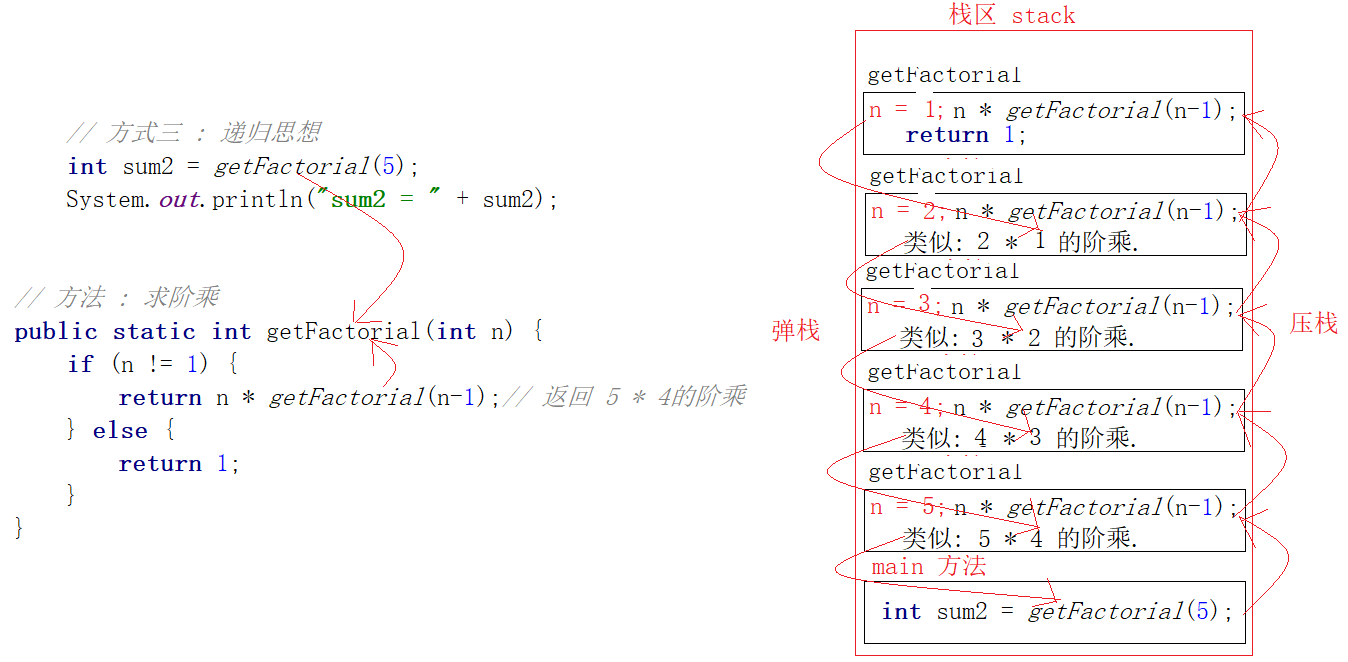
请问 : 5 的阶乘是多少 ??? 我不知道, 但是我知道它等于 5 \* 4的阶乘.

请问 : 4 的阶乘是多少 ??? 我不知道, 但是我知道它等于 4 \* 3的阶乘.

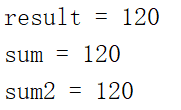
请问 : 3 的阶乘是多少 ??? 我不知道, 但是我知道它等于 3 \* 2的阶乘.

请问 : 2 的阶乘是多少 ??? 我不知道, 但是我知道它等于 2 \* 1的阶乘.

请问 : 1 的阶乘是多少 ??? 等于1. (最后一定要有结果, 没有结果就是 `死递归`, 这样就会造成栈溢出错误.)



public class Test2 {  
 public static void main(String[] args) {  
  
 *// 方式一 : 运算实现* int result = 5 \* 4 \* 3 \* 2 \* 1;  
 System.*out*.println("result = " + result);  
  
 *// 方式二 : 循环思考* int sum = 1;  
 for (int i = 5; i >= 1; i--) {  
 sum \*= i;  
 }  
 System.*out*.println("sum = " + sum);  
  
 *// 方式三 : 递归思想 (非常消耗栈区内存)* int sum2 = *getFactorial*(5);  
 System.*out*.println("sum2 = " + sum2);  
 }  
  
 *// 方法 : 求阶乘* public static int getFactorial(int n) {  
 if (n != 1) {  
 return n \* *getFactorial*(n-1); *// 返回 5 \* 4的阶乘* } else {  
 return 1;  
 }  
 }  
}



## 27.3 递归求年龄 :

有5个人, 围成一圈.

请问 : 第5个人多大 ??? 我比第 4 个人大 2 岁.

请问 : 第4个人多大 ??? 我比第 3 个人大 2 岁.

请问 : 第3个人多大 ??? 我比第 2 个人大 2 岁.

请问 : 第2个人多大 ??? 我比第 1 个人大 2 岁.

请问 : 第1个人多大 ??? 我10 岁.

public class Test3 {  
 public static void main(String[] args) {  
  
 // 求年龄 :  
 /\*  
 步骤一 : 自定义一个方法, 实现递归调用.  
 步骤二 : 思考如何结束递归. (结束条件: 如果问第一个人, 直接返回 10, 不再执行递归调用)  
 步骤三 : 如何继续递归. (只要不是第一个人, 就继续)  
 \*/  
 int age = getAge(5);  
 System.out.println("age = " + age);  
 }  
  
 // 方法 : 求年龄  
 public static int getAge(int n) {  
 // return 将结果返回给上一层的调用者.  
 if (n != 1) {  
 return 2 + getAge(n - 1);  
 } else {  
 return 10;  
 }  
 }  
}



## 27.4 递归打印多级目录 :

public class Test1 {  
 public static void main(String[] args) {  
  
 *// 1. 使用 file 对象关联硬盘上的文件夹* File file = new File("C:\\Users\\xieco\\Desktop\\day08-File类、递归\\课程资料");  
  
 *// 2. 调用方法  
 scanFolder*(file);  
 }  
  
 *// 方法 : 扫描所有文件夹中的所有文件* public static void scanFolder(File file) {  
 *// 2. 列出 file 对象中的所有文件对象清单 (listFiles)* File[] files = file.listFiles();  
  
 *// 3. 遍历* for (File f : files) {  
  
 *// 4. 判断 (文件夹)* if (f.isDirectory()) {  
 *// 4.1 文件夹 (再次进入到文件夹中, 获取所有文件对象清单)  
 scanFolder*(f);  
 } else {  
 *// 4.2 文件* System.*out*.println(f);  
 }  
 }  
 }  
}



## 27.5 删除非空文件夹 :

public class Test2 {  
 public static void main(String[] args) {  
  
 *// 需求 : 删除非空文件夹* File file = new File("C:\\Users\\xieco\\Desktop\\day08-File类、递归\\课程资料");  
  
 *// 思路 : 删除 (清单, 循环进入, 删文件, 循环出来, 删文件夹)* boolean result = *deleteFolder*(file);  
 System.*out*.println("result = " + result);  
 }  
  
 *// 方法 : 删除传入的文件夹* public static boolean deleteFolder(File folder) {  
 *// 1. 获取文件夹中所有文件清单 (包含文件和文件夹)* File[] files = folder.listFiles();  
  
 *// 2. 遍历文件数组* for (File f : files) {  
 *// 3. 判断* if (f.isDirectory()) {  
 *// 3.1 文件夹 (递归进入, 再次获取清单, 继续判断)  
 deleteFolder*(f);  
 } else {  
 *// 3.2 文件* f.delete();  
 }  
 }  
  
 *// 4. 循环结束, 就可以删除空文件夹了.* boolean result = folder.delete();  
 return result;  
 }  
}

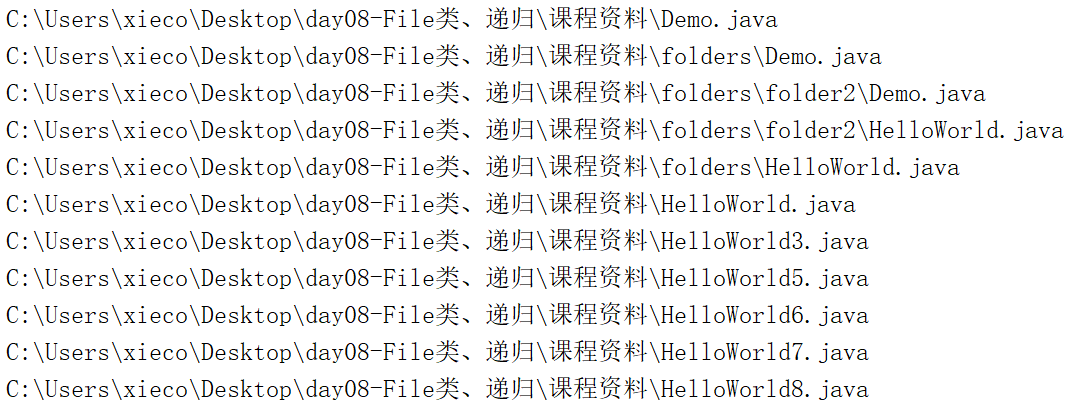
# 28. File类综合案例 :

## 28.1 文件搜索 :

需求 : 搜索指定后缀名的文件. .jpg .avi .java …

**思考 :** 在扫描的基础上, 获取文件, 然后进行文件的后缀名判断即可.

public class Test3 {  
 public static void main(String[] args) {  
  
 *// 需求 : 文件搜索 (jpg, avi, java)  
  
 // 1. 关联需要搜索的文件夹* File file = new File("C:\\Users\\xieco\\Desktop\\day08-File类、递归\\课程资料");  
  
 *// 2. 文件扫描 + 后缀名  
 searchFile*(file, ".java");  
 }  
  
 *// 方法 :* public static void searchFile(File folder, String suffix) {  
  
 *// 1. 获取文件夹中所有文件清单* File[] files = folder.listFiles();  
  
 *// 2. 遍历* for (File f : files) {  
 *// 3. 判断* if (f.isDirectory()) {  
 *// 3.1 文件夹 (递归调用)  
 searchFile*(f, suffix);  
 } else {  
 *// 3.2 文件 (先获取文件对象名称, 然后调用 endsWith 字符串的后缀名判断方法* if (f.getName().endsWith(suffix)) {  
 System.*out*.println(f);  
 }  
 }  
 }  
 }  
}



**关于 String 类的 `前缀与后缀` 判断方法 :**

public class Test4 {  
 public static void main(String[] args) {  
  
 // https: (协议) //www.baidu.com (后缀名)  
 // com(company 商业公司) cn (中国) edu (education) org (organization)  
 // String path = "http://www.itcast.cn";  
 String path = "https://www.baidu.com";  
  
 // 判断1 : path 是否以 .com 结尾. (后缀 suffix)  
 boolean result = path.endsWith(".com");  
 System.out.println("result = " + result);  
  
 // 判断2 : path 是否以 https 开头 (前缀 prefix)  
 boolean result2 = path.startsWith("https");  
 System.out.println("result2 = " + result2);  
 }  
}



## 28.2 文件过滤器优化 :

|  |  |
| --- | --- |
| [File](mk:@MSITStore:C:\Users\xieco\Desktop\JDK9%20API文档\_JDK6API中文参考%5b沈东良%5d(070114).chm::/java/io/File.html)[] | [**listFiles**](mk:@MSITStore:C:\Users\xieco\Desktop\JDK9%20API文档\_JDK6API中文参考%5b沈东良%5d(070114).chm::/java/io/File.html#listFiles(java.io.FileFilter))([FileFilter](mk:@MSITStore:C:\Users\xieco\Desktop\JDK9%20API文档\_JDK6API中文参考%5b沈东良%5d(070114).chm::/java/io/FileFilter.html) filter)            返回抽象路径名数组，这些路径名表示此抽象路径名表示的目录中满足指定过滤器的文件和目录。 |



**MyFileFilter 文件过滤器实现类 :**

import java.io.File;  
import java.io.FileFilter;  
  
public class MyFileFilter implements FileFilter {  
 *// 属性 (数据)* private String suffix;  
  
 *// 构造方法* public MyFileFilter(String suffix) {  
 this.suffix = suffix;  
 }  
  
 @Override  
 public boolean accept(File pathname) {  
  
 */\*  
 返回 true, 意味着当前文件对象, 会被存储到返回的 files 数组中.  
 返回 false, 意味着当前文件对象, 就不会被存储到返回的 files 数组中.  
 \*/  
 // System.out.println(pathname);  
  
 // 条件1 : 如果是文件夹, 应该返回 true.  
 // 条件2 : 如果是文件, 就需要判断后缀名, 然后返回对应的结果.* if (pathname.isDirectory() || pathname.getName().endsWith(suffix)) {  
 return true;  
 }  
  
 return false;  
 }  
}

**测试类 :**

public class Test5 {  
 public static void main(String[] args) {  
  
 *// 需求 : 文件搜索 (jpg, avi, java)  
  
 // 1. 关联需要搜索的文件夹* File file = new File("C:\\Users\\xieco\\Desktop\\day08-File类、递归\\课程资料");  
  
 *// 2. 文件扫描 + 后缀名  
 searchFile*(file, ".jpg");  
 }  
  
 *// 方法 :* public static void searchFile(File folder, String suffix) {  
  
 *// 1. 获取文件夹中所有文件清单  
 // 方法 : listFiles(FileFilter filter); 文件过滤器* File[] files = folder.listFiles(new MyFileFilter(suffix));  
  
 *// System.out.println("files = " + Arrays.toString(files));  
  
 // 2. 遍历* for (File f : files) {  
 *// 3. 判断* if (f.isDirectory()) {  
 *// 3.1 文件夹 (递归调用)  
 searchFile*(f, suffix);  
 } else {  
 System.*out*.println(f);  
 }  
 }  
 }  
}



## 28.3 匿名实现类完成 :

public class Test6 {  
 public static void main(String[] args) {  
  
 *// 需求 : 文件搜索 (jpg, avi, java)  
  
 // 1. 关联需要搜索的文件夹* File file = new File("C:\\Users\\xieco\\Desktop\\day08-File类、递归\\课程资料");  
  
 *// 2. 文件扫描 + 后缀名  
 searchFile*(file, ".jpg");  
 }  
  
 *// 方法 :* public static void searchFile(File folder, String suffix) {  
  
 *// 1. 获取文件夹中所有文件清单  
 // 方法 : listFiles(FileFilter filter); 文件过滤器* File[] files = folder.listFiles(new FileFilter() {  
 @Override  
 public boolean accept(File pathname) {  
 return pathname.isDirectory() || pathname.getName().endsWith(suffix);  
 }  
 });  
  
 *// 2. 遍历* for (File f : files) {  
 *// 3. 判断* if (f.isDirectory()) {  
 *// 3.1 文件夹 (递归调用)  
 searchFile*(f, suffix);  
 } else {  
 System.*out*.println(f);  
 }  
 }  
 }  
}

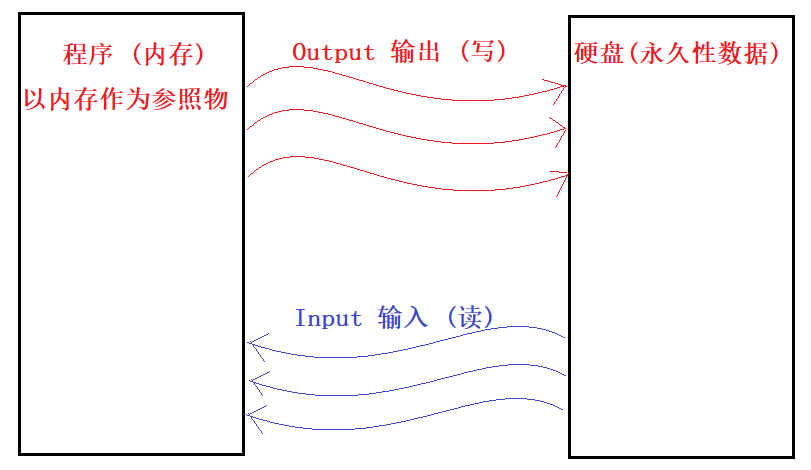


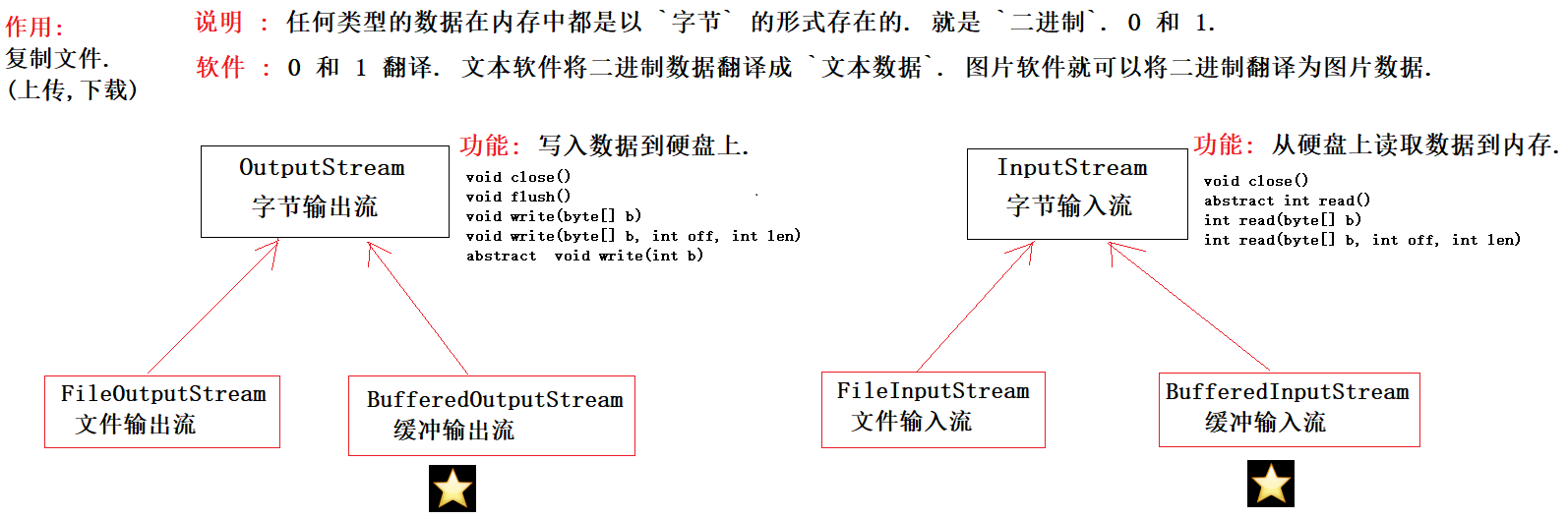
## 28.3 Lambda 优化 :

public class Test7 {  
 public static void main(String[] args) {  
  
 *// 需求 : 文件搜索 (jpg, avi, java)  
  
 // 1. 关联需要搜索的文件夹* File file = new File("C:\\Users\\xieco\\Desktop\\day08-File类、递归\\课程资料");  
  
 *// 2. 文件扫描 + 后缀名  
 searchFile*(file, ".jpg");  
 }  
  
 *// 方法 :* public static void searchFile(File folder, String suffix) {  
  
 *// 1. 获取文件夹中所有文件清单  
 // 方法 : listFiles(FileFilter filter); 文件过滤器  
 // File[] files = folder.listFiles(pathname -> pathname.isDirectory() || pathname.getName().endsWith(suffix));* File[] files = folder.listFiles(p -> p.isDirectory() || p.getName().endsWith(suffix));  
  
 *// 2. 遍历* for (File f : files) {  
 *// 3. 判断* if (f.isDirectory()) {  
 *// 3.1 文件夹 (递归调用)  
 searchFile*(f, suffix);  
 } else {  
 System.*out*.println(f);  
 }  
 }  
 }  
}



# 29. IO流概述 : (重点)





## 29.1 FileOutputStream 文件字节输出流 :



### 29.1.1 write 方法的演示 :

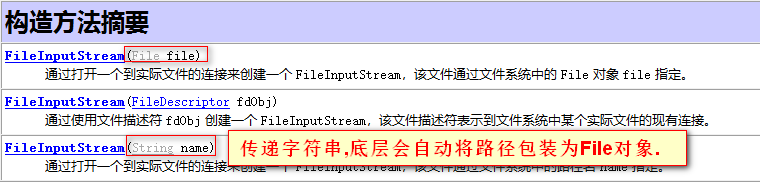
public class Test1 {  
 public static void main(String[] args) throws IOException {  
  
 // 1. 创建一个文件输出流对象 (FileOutputStream)  
 // new FileOutputStream(new File("路径"));  
 // 文件输出流对象写入数据到文件中, 如果文件不存在, 该对象会自动创建该文件.  
 FileOutputStream out = new FileOutputStream("test1.txt");  
  
 // 2. 向文件中输出数据  
 // 2.1 write(int)  
 out.write('7');  
 out.write('8');  
 out.write('9');  
  
 out.write(97);  
 out.write(98);  
 out.write(99);  
  
 // 2.2 write(byte[])  
 byte[] bytes = new byte[]{65, 66, 67, 68, 69};  
 out.write(bytes);  
  
 // 2.3 write(byte[] b, int off, int len); offset 偏移量 length 长度  
 // 含义 : 从 b 字节数组中, 哪个 off 偏移位置开始, 取 len 个长度.  
 out.write(bytes, 0, 3); // {65, 66, 67}  
 out.write(bytes, 3, 2); // {68, 69}  
  
 // 3. 关闭文件输出流对象  
 out.close();  
  
 System.out.println("执行完成...");  
 }  
}



### 29.1.2 数据追加与换行 :

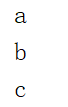
public class Test2 {  
 public static void main(String[] args) throws IOException {  
  
 *// 1. 创建了一个文件输出流对象, 实现文件数据的拼接.* FileOutputStream out = new FileOutputStream("test1.txt", true);  
  
 *// 需求 : 写入字符串到文件中. 使用 String 类的 getBytes() 方法实现字节数组的转换  
 // out.write("\r\n大家好, 我们都是好人.".getBytes());  
  
 /\*  
 需求 : 实现数据的换行.  
 方式一 : \r\n (了解)  
 方式二 : System.lineSeparator(); 获取系统的换行分隔符.  
 \*/* String lineSeparator = System.*lineSeparator*();  
 out.write((lineSeparator + "大家好, 我们都是大大大好人.").getBytes());  
  
 *// 关闭资源* out.close();  
 }  
}

## 29.2 FileInputStream 文件字节输入流 :



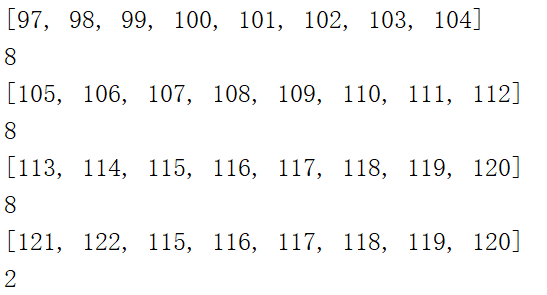
### 29.2.1 read 一个一个字节读取数据 :

public class Test1 {  
 public static void main(String[] args) throws IOException {  
  
 *// 1. 创建一个文件输入流对象 (FileInputStream)* FileInputStream in = new FileInputStream("test1.txt");  
  
 *// 2. 读取数据* int read = -1;  
 *// in.read(); 一个字节一个字节实现数据的读取.* while ((read = in.read()) != -1) {  
 System.*out*.println((char) read); *// 输出的数据内容是根据类型来判断.* }  
  
 */\*  
 int read = in.read();  
 System.out.println("read = " + read); // 97  
 read = in.read();  
 System.out.println("read = " + read); // 98  
 read = in.read();  
 System.out.println("read = " + read); // 99  
 read = in.read();  
 System.out.println("read = " + read); // ??? 1.null 2.数值(-1) 3. 报异常  
 \*/  
  
 // 3. 关闭资源* in.close();  
 }  
}



### 29.2.2 read 一个数组一个数组读取数据 :

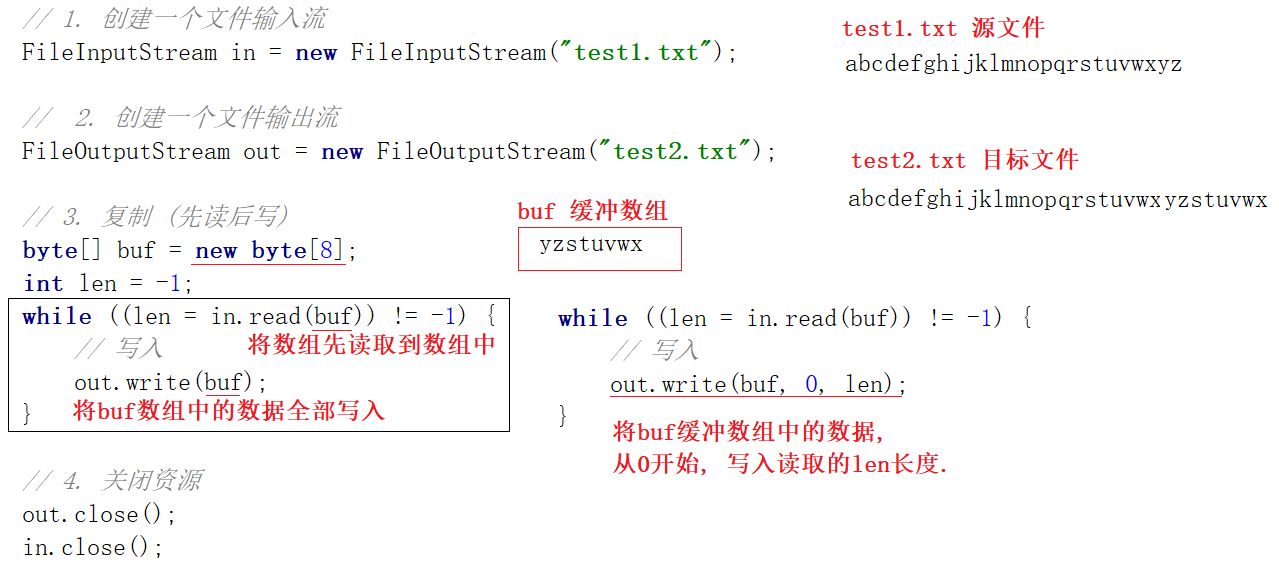
public class Test2 {  
 public static void main(String[] args) throws IOException {  
  
 *// 1. 创建一个文件输入流对象 (FileInputStream)* FileInputStream in = new FileInputStream("test1.txt");  
  
 *// 2. read(byte[] b);* byte[] buf = new byte[8];  
 int len = -1;  
 *// in.read(buf); 一个字节数组一个字节数组实现数据读取.* while ((len = in.read(buf)) != -1) {  
 System.*out*.println(Arrays.*toString*(buf));  
 System.*out*.println(len);  
 }  
  
 *// 3. 关闭资源* in.close();  
 }  
}



## 29.3 read 和 write 的结合使用模式 :

数据: abcdefghijklmnopqrstuvwxyz

public class TestCopy2 {  
 public static void main(String[] args) throws IOException {  
  
 *// 实现文件复制 : a ~ z 的数据.  
  
 // 1. 创建一个文件输入流* FileInputStream in = new FileInputStream("test1.txt");  
  
 *// 2. 创建一个文件输出流* FileOutputStream out = new FileOutputStream("test2.txt");  
  
 *// 3. 复制 (先读后写)* byte[] buf = new byte[8];  
 int len = -1;  
 while ((len = in.read(buf)) != -1) {  
 *// 写入* out.write(buf, 0, len);  
 }  
  
 *// 4. 关闭资源* out.close();  
 in.close();  
 }  
}



## 29.4 复制文件 (文本, 图片, 视频, 音频 …)

### 29.4.1 实现一 : 字节方式

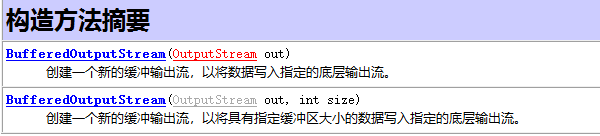
public class TestCopy1 {  
 public static void main(String[] args) throws IOException {  
  
 *// 开始* long start = System.*currentTimeMillis*();  
  
 *// 复制文件 : 先读后写  
 // 1. 创建一个字节文件输入流* FileInputStream in = new FileInputStream("C:\\Users\\xieco\\Desktop\\day08-File类、递归\\课程资料\\福利赠送.flv");  
 *// 2. 创建一个字节文件输出流* FileOutputStream out = new FileOutputStream("C:\\Users\\xieco\\Desktop\\flzs1.flv");  
  
 *// 3. 读写操作* int read = -1;  
 *// in.read() 实现一个字节一个字节读取数据. 1M -> 1024KB  
 // 内存速度快, 硬盘执行的速度是非常慢. 每次与文件交互, 都是与硬盘打交道.* while ((read = in.read()) != -1) {  
 out.write(read);  
 out.flush();  
 }  
  
 *// 4. 关闭资源* out.close();  
 in.close();  
  
 *// 结束* long end = System.*currentTimeMillis*();  
 *// 共耗时: 55438 毫秒.* System.*out*.println("共耗时: " + (end - start) + " 毫秒.");  
 }  
}

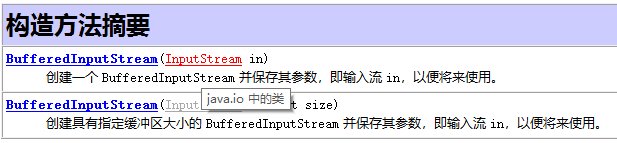
### 29.4.2 实现二 : 字节数组方式

public class TestCopy3 {  
 public static void main(String[] args) throws IOException {  
  
 *// 开始* long start = System.*currentTimeMillis*();  
  
 *// 复制文件 : 先读后写  
 // 1. 创建一个字节文件输入流 (北京)* FileInputStream in = new FileInputStream("C:\\Users\\xieco\\Desktop\\day08-File类、递归\\课程资料\\福利赠送.flv");  
 *// 2. 创建一个字节文件输出流 (上海)* FileOutputStream out = new FileOutputStream("C:\\Users\\xieco\\Desktop\\flzs2.flv");  
  
 *// 3. 读写操作* byte[] buf = new byte[1024]; *// 1024 \* 8 (卡车)* int len = -1;  
 *// in.read(buf); 一个字节数组一个字节数组实现数据读取.* while ((len = in.read(buf)) != -1) {  
 *// 一个字节数组一个字节数组实现数据写入* out.write(buf, 0, len);  
 }  
  
 *// 4. 关闭资源* out.close();  
 in.close();  
  
 *// 结束* long end = System.*currentTimeMillis*();  
 *// 共耗时: 86 毫秒.* System.*out*.println("共耗时: " + (end - start) + " 毫秒.");  
 }  
}

### 29.4.3 实现三 : 字节缓冲流方式

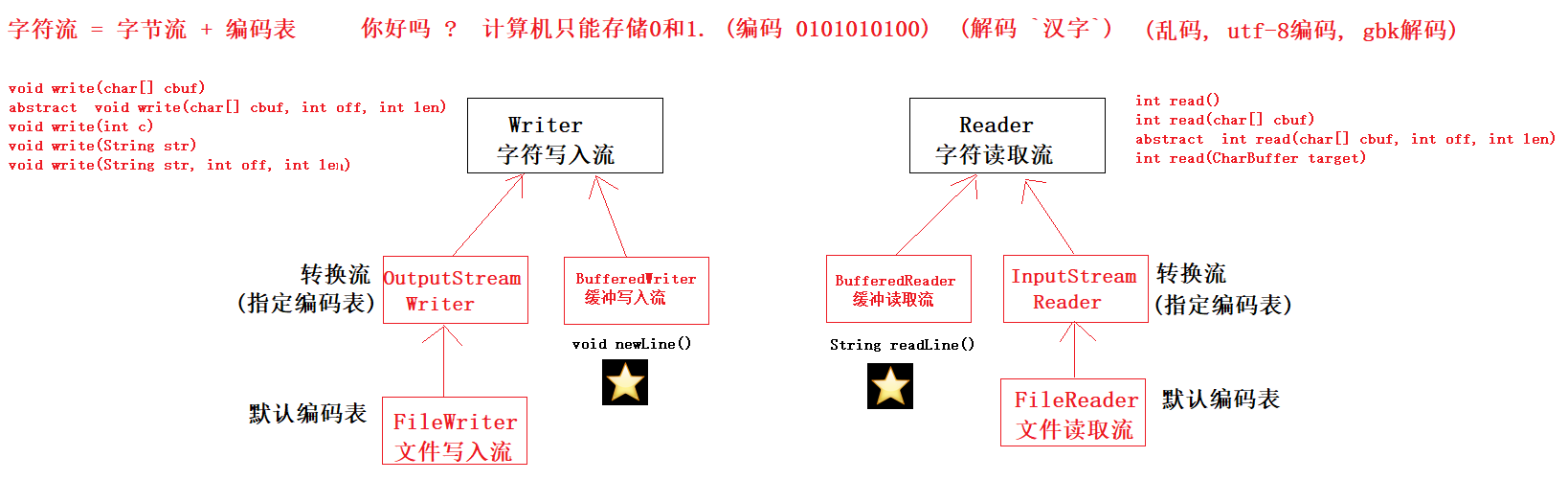
**BufferedInputStream & BufferedOutputStream 字节缓冲高效输入 / 输出流**





public class TestCopy4 {  
 public static void main(String[] args) throws IOException {  
  
 *// 开始* long start = System.*currentTimeMillis*();  
  
 *// 复制文件 : 先读后写  
 // 1. 创建一个字节文件输入流 (北京)  
 // FileInputStream in = new FileInputStream("C:\\Users\\xieco\\Desktop\\day08-File类、递归\\课程资料\\福利赠送.flv");* BufferedInputStream bis = new BufferedInputStream(new FileInputStream("C:\\Users\\xieco\\Desktop\\day08-File类、递归\\课程资料\\福利赠送.flv"));  
  
 *// 2. 创建一个字节文件输出流 (上海)  
 // FileOutputStream out = new FileOutputStream("C:\\Users\\xieco\\Desktop\\flzs3.flv");* BufferedOutputStream bos = new BufferedOutputStream(new FileOutputStream("C:\\Users\\xieco\\Desktop\\flzs4.flv"));  
  
 *// 3. 读写操作* byte[] buf = new byte[1024]; *// 1024 \* 8 (卡车)* int len = -1;  
 *// in.read(buf); 一个字节数组一个字节数组实现数据读取.* while ((len = bis.read(buf)) != -1) {  
 *// 一个字节数组一个字节数组实现数据写入* bos.write(buf, 0, len);  
 }  
  
 *// 4. 关闭资源* bos.close();  
 bis.close();  
  
 *// 结束* long end = System.*currentTimeMillis*();  
 *// 共耗时: 33 毫秒.* System.*out*.println("共耗时: " + (end - start) + " 毫秒.");  
 }  
}

## 29.5 FileReader 文件字符读取流 :



**字节流读取中文汉字的问题 :**

public class Test1 {  
 public static void main(String[] args) throws IOException {  
  
 *// 1. 创建一个字节输入流 (读取)* FileInputStream in = new FileInputStream("test1.txt");  
  
 *// 2. 读取* int read = -1;  
 */\*  
 汉字 : 一个汉字不止一个字节. 字符集. (GBK 2个字节组成一个汉字, UTF-8 3个字节组成一个汉字)  
 \*/  
  
 // in.read(); 字节的 read 方法就是一个字节一个字节读取数据.  
 // 英文字符 : a ~ z 每个字符都是一个字节组成的. 读取就是正确的.  
 // 一个汉字三个字节, 因此如果一个字节一个字节读取, 就会出现数据错误!  
 // 结论 : 字节流无法解决汉字读取的问题.* while ((read = in.read()) != -1) {  
 System.*out*.print((char) read);  
 }  
  
 *// 3. 关闭资源* in.close();  
 }  
}



**字符流读取中文汉字 :**

public class Test2 {  
 public static void main(String[] args) throws IOException {  
  
 *// 1. 创建一个字符读取流 (读取)* FileReader reader = new FileReader("test1.txt");  
  
 *// 2. 读取* int read = -1;  
 *// reader.read(); 字符流的 read 是根据底层编码表的规则实现读取的. (项目编码 UTF-8)* while ((read = reader.read()) != -1) {  
 System.*out*.print((char) read);  
 }  
  
 *// 3. 关闭资源* reader.close();  
 }  
}



## 29.6 FileWriter 文件字符写出流 :

public class Test1 {  
 public static void main(String[] args) throws IOException {  
  
 *// 1. 创建一个字符文件写入流对象* FileWriter writer = new FileWriter("test3.txt");  
  
 *// 2. 写入数据 (字符串)  
 // 字符流写入字符数据, 无需实现字节转换, 因为字符流底层拥有默认编码表.* writer.write("How are you doing ?"); writer.write("你好吗?"); *// 字节数据  
  
 // 3. 关闭资源* writer.close();  
 }  
}

## 29.7 BufferedReader与BuffereWriter 文件字符缓冲流 :

读取 : 一行一行读. readLine();

写入 : 一行一行写. newLine();

**缓冲字符流读取 :**

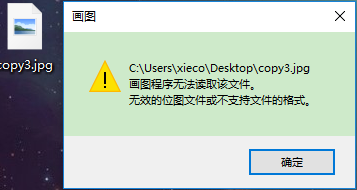
public class Test3 {  
 public static void main(String[] args) throws IOException {  
  
 *// 1. 创建一个高效的缓冲字符读取流* BufferedReader reader = new BufferedReader(new FileReader("test3.txt"));  
  
 *// 2. 一行一行读* String line = null;  
 while ((line = reader.readLine()) != null) {  
 System.*out*.println(line);  
 }  
  
 */\*  
 String line = reader.readLine();  
 System.out.println("line = " + line);  
  
 line = reader.readLine();  
 System.out.println("line = " + line);  
  
 line = reader.readLine();  
 System.out.println("line = " + line); // null  
 \*/  
  
 // 3. 关闭资源* reader.close();  
 }  
}

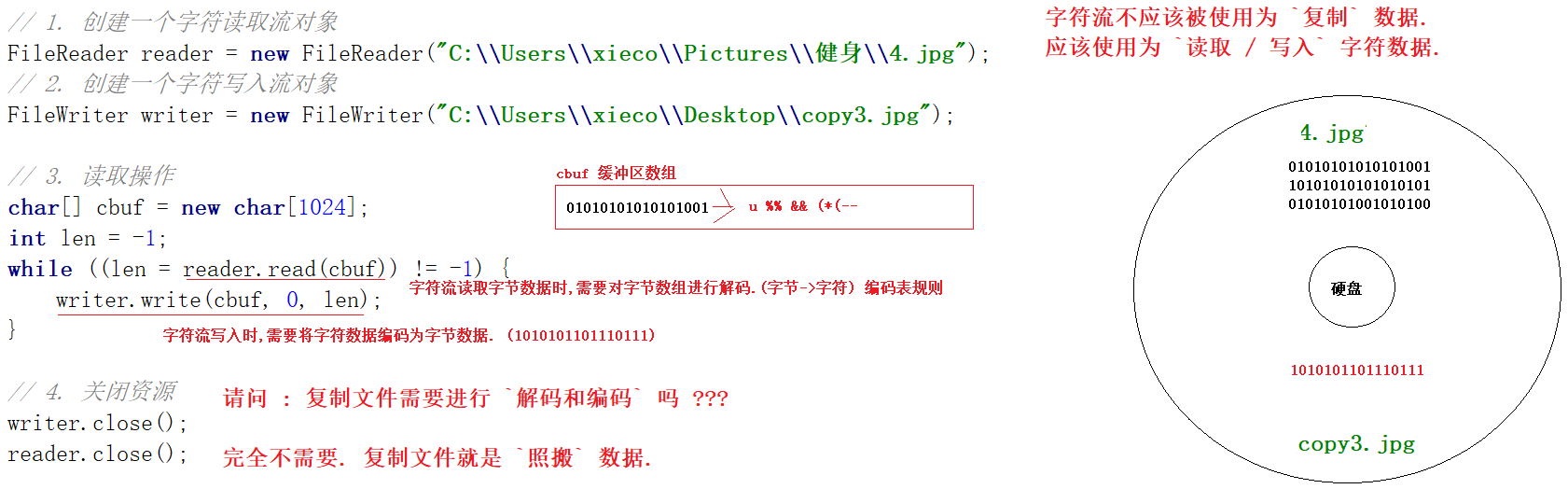
**缓冲字符流写入 :**

public class Test2 {  
 public static void main(String[] args) throws IOException {  
  
 *// 1. 创建一个高效的缓冲字符写入流* BufferedWriter writer = new BufferedWriter(new FileWriter("test4.txt"));  
  
 *// 2. 写入, 换行, 刷新* writer.write("I want to use the bathroom ?");  
 writer.newLine();  
 writer.flush();  
  
 writer.write("go a head.");  
 writer.newLine();  
 writer.flush();  
  
 *// 3. 关闭资源* writer.close();  
 }  
}

## 29.8 思考 : 字符流可以复制文件吗 ??? (了解)

public class TestCopy5 {  
 public static void main(String[] args) throws IOException {  
  
 *// 1. 创建一个字符读取流对象* FileReader reader = new FileReader("C:\\Users\\xieco\\Pictures\\健身\\4.jpg");  
 *// 2. 创建一个字符写入流对象* FileWriter writer = new FileWriter("C:\\Users\\xieco\\Desktop\\copy3.jpg");  
  
 *// 3. 读写操作* char[] cbuf = new char[1024];  
 int len = -1;  
 while ((len = reader.read(cbuf)) != -1) {  
 writer.write(cbuf, 0, len);  
 }  
  
 *// 4. 关闭资源* writer.close();  
 reader.close();  
 }  
}





## 29.9 综合练习实现 :

### 1. 文本排序 :

3.侍中、侍郎郭攸之、费祎、董允等，此皆良实，志虑忠纯，是以先帝简拔以遗陛下。愚以为宫中之事，事无大小，悉以咨之，然后施行，必得裨补阙漏，有所广益。

8.愿陛下托臣以讨贼兴复之效，不效，则治臣之罪，以告先帝之灵。若无兴德之言，则责攸之、祎、允等之慢，以彰其咎；陛下亦宜自谋，以咨诹善道，察纳雅言，深追先帝遗诏，臣不胜受恩感激。

4.将军向宠，性行淑均，晓畅军事，试用之于昔日，先帝称之曰能，是以众议举宠为督。愚以为营中之事，悉以咨之，必能使行阵和睦，优劣得所。

2.宫中府中，俱为一体，陟罚臧否，不宜异同。若有作奸犯科及为忠善者，宜付有司论其刑赏，以昭陛下平明之理，不宜偏私，使内外异法也。

1.先帝创业未半而中道崩殂，今天下三分，益州疲弊，此诚危急存亡之秋也。然侍卫之臣不懈于内，忠志之士忘身于外者，盖追先帝之殊遇，欲报之于陛下也。诚宜开张圣听，以光先帝遗德，恢弘志士之气，不宜妄自菲薄，引喻失义，以塞忠谏之路也。

9.今当远离，临表涕零，不知所言。

6.臣本布衣，躬耕于南阳，苟全性命于乱世，不求闻达于诸侯。先帝不以臣卑鄙，猥自枉屈，三顾臣于草庐之中，咨臣以当世之事，由是感激，遂许先帝以驱驰。后值倾覆，受任于败军之际，奉命于危难之间，尔来二十有一年矣。

7.先帝知臣谨慎，故临崩寄臣以大事也。受命以来，夙夜忧叹，恐付托不效，以伤先帝之明，故五月渡泸，深入不毛。今南方已定，兵甲已足，当奖率三军，北定中原，庶竭驽钝，攘除奸凶，兴复汉室，还于旧都。此臣所以报先帝而忠陛下之职分也。至于斟酌损益，进尽忠言，则攸之、祎、允之任也。

5.亲贤臣，远小人，此先汉所以兴隆也；亲小人，远贤臣，此后汉所以倾颓也。先帝在时，每与臣论此事，未尝不叹息痛恨于桓、灵也。侍中、尚书、长史、参军，此悉贞良死节之臣，愿陛下亲之信之，则汉室之隆，可计日而待也。

**代码实现 :**

public class Practice1 {  
 public static void main(String[] args) throws IOException {  
  
 *// 1. 创建一个高效的缓冲字符读取流 (BufferedReader)* BufferedReader reader = new BufferedReader(new FileReader("in.txt"));  
  
 *// 4.1 创建一个 `映射表` 集合* HashMap<Integer, String> map = new HashMap<>();  
  
 *// 2. 一行一行读取数据* String line = null;  
 while ((line = reader.readLine()) != null) {  
  
 *// key : 3 value : 侍中、侍郎郭攸之、费祎、董允等  
 // 3. 使用 `点` 作为切割符, 切割 line 字符串* String[] split = line.split("\\.");  
  
 *// 4.2 将 split 数组中元素作为 key 和 value 存储到映射表中* int key = Integer.*parseInt*(split[0]);  
 String value = split[1];  
 map.put(key, value);  
 }  
  
 *// 6.1 创建一个高效的缓冲字符写出流* BufferedWriter writer = new BufferedWriter(new FileWriter("out.txt"));  
  
 *// 5. 遍历 map 集合  
 /\*Set<Map.Entry<Integer, String>> entrySet = map.entrySet();  
 for (Map.Entry<Integer, String> entry : entrySet) {  
 System.out.println(entry);  
 }\*/* for (int i = 1; i <= map.size(); i++) {  
 *// 说明, 此处的 i 变量就是 map 中的 key.* String value = map.get(i);  
  
 *// 6.2 写入数据到文件中* writer.write(i + "." + value);  
 writer.newLine();  
 writer.flush();  
 *// System.out.println(i + " = " + value);* }  
  
 *// 6.3 关闭资源* writer.close();  
 reader.close();  
 }  
}

**文件内容 : 有序**

1.先帝创业未半而中道崩殂，今天下三分，益州疲弊，此诚危急存亡之秋也。然侍卫之臣不懈于内，忠志之士忘身于外者，盖追先帝之殊遇，欲报之于陛下也。诚宜开张圣听，以光先帝遗德，恢弘志士之气，不宜妄自菲薄，引喻失义，以塞忠谏之路也。  
2.宫中府中，俱为一体，陟罚臧否，不宜异同。若有作奸犯科及为忠善者，宜付有司论其刑赏，以昭陛下平明之理，不宜偏私，使内外异法也。  
3.侍中、侍郎郭攸之、费祎、董允等，此皆良实，志虑忠纯，是以先帝简拔以遗陛下。愚以为宫中之事，事无大小，悉以咨之，然后施行，必得裨补阙漏，有所广益。  
4.将军向宠，性行淑均，晓畅军事，试用之于昔日，先帝称之曰能，是以众议举宠为督。愚以为营中之事，悉以咨之，必能使行阵和睦，优劣得所。  
5.亲贤臣，远小人，此先汉所以兴隆也；亲小人，远贤臣，此后汉所以倾颓也。先帝在时，每与臣论此事，未尝不叹息痛恨于桓、灵也。侍中、尚书、长史、参军，此悉贞良死节之臣，愿陛下亲之信之，则汉室之隆，可计日而待也。  
6.臣本布衣，躬耕于南阳，苟全性命于乱世，不求闻达于诸侯。先帝不以臣卑鄙，猥自枉屈，三顾臣于草庐之中，咨臣以当世之事，由是感激，遂许先帝以驱驰。后值倾覆，受任于败军之际，奉命于危难之间，尔来二十有一年矣。  
7.先帝知臣谨慎，故临崩寄臣以大事也。受命以来，夙夜忧叹，恐付托不效，以伤先帝之明，故五月渡泸，深入不毛。今南方已定，兵甲已足，当奖率三军，北定中原，庶竭驽钝，攘除奸凶，兴复汉室，还于旧都。此臣所以报先帝而忠陛下之职分也。至于斟酌损益，进尽忠言，则攸之、祎、允之任也。  
8.愿陛下托臣以讨贼兴复之效，不效，则治臣之罪，以告先帝之灵。若无兴德之言，则责攸之、祎、允等之慢，以彰其咎；陛下亦宜自谋，以咨诹善道，察纳雅言，深追先帝遗诏，臣不胜受恩感激。  
9.今当远离，临表涕零，不知所言。

### 2. 存储对象数据到文件 :

编程题 : 编写一个程序, 完成键盘录入员工信息，并计算员工到手薪资，将所有信息写入到文本文件中存储.

需求：键盘录入3个员工信息(姓名,薪资,扣税,到手薪资)

求出每个员工的到手薪资. 到手薪资 = 薪资 - 扣税.

并且将员工的信息写入employee.txt文件中

效果下所示 :

姓名 薪资 扣税 到手薪资

张三 15000 3000 12000

李四 16000 3200 12800

王五 17000 3500 13500

**Employee 类定义 :**

public class Employee {  
 private String name;  
 private int salary;  
 private int tax;  
 private int money;  
  
 public Employee(String name, int salary, int tax, int money) {  
 this.name = name;  
 this.salary = salary;  
 this.tax = tax;  
 this.money = money;  
 }  
  
 public Employee() {  
 }  
  
 @Override  
 public String toString() {  
 return "Employee{" +  
 "name='" + name + '\'' +  
 ", salary=" + salary +  
 ", tax=" + tax +  
 ", money=" + money +  
 '}';  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 public int getSalary() {  
 return salary;  
 }  
  
 public void setSalary(int salary) {  
 this.salary = salary;  
 }  
  
 public int getTax() {  
 return tax;  
 }  
  
 public void setTax(int tax) {  
 this.tax = tax;  
 }  
  
 public int getMoney() {  
 return money;  
 }  
  
 public void setMoney(int money) {  
 this.money = money;  
 }  
}

**代码实现 :**

public class Practice2 {  
 public static void main(String[] args) throws IOException {  
  
 *// 1. 创建一个键盘录入对象* Scanner sc = new Scanner(System.*in*);  
  
 *// 5.1 创建一个集合, 存储 Employee 类型的对象* ArrayList<Employee> emps = new ArrayList<>();  
  
 *// 2. 循环3次, 获取3个员工对象信息* for (int i = 1; i <= 3; i++) {  
  
 *// 3. 提示, 并接收用户输入  
 // 信息(姓名,薪资,扣税,到手薪资)* System.*out*.println("亲, 请输入第" + i + "个员工的姓名 :");  
 *// 张三 回车* String name = sc.nextLine(); *// nextLine(); 方法会处理回车问题. 并将 `回车` 作为该方法的结束符  
  
 // 说明 : 使用键盘录入接收用户 int 类型数据, 不要使用 nextInt(); 方法, 请使用 nextLine(); 方法 (自己解析)  
 // 12000* System.*out*.println("亲, 请输入第" + i + "个员工的薪资 :");  
 *// int salary = sc.nextInt(); // nextInt(); 不处理回车问题. 12000* int salary = Integer.*parseInt*(sc.nextLine());  
  
 *// 3000* System.*out*.println("亲, 请输入第" + i + "个员工的税收 :");  
 *// int tax = sc.nextInt(); // nextInt(); 不处理回车问题. 3000* int tax = Integer.*parseInt*(sc.nextLine());  
  
 *// 计算到手薪资* int money = salary - tax;  
  
 *// 4.2 将输入的数据封装为 `Employee 对象`* Employee emp = new Employee(name, salary, tax, money);  
  
 *// 5.2 将封装完毕后的员工对象存储到集合中* emps.add(emp);  
 }  
  
 sc.close();  
  
 *// 6. 将 emps 集合中的所有员工数据写入到指定文件中  
 writeListToFile*(emps, "employee.txt");  
 }  
  
 *// 方法, 将集合中的员工数据写入到指定文件中* public static void writeListToFile(ArrayList<Employee> list, String fileName) throws IOException {  
  
 *// 1. 创建一个文件字符缓冲输出流对象* BufferedWriter writer = new BufferedWriter(new FileWriter(fileName));  
  
 *// 2. 遍历集合* String title = "姓名\t\t薪资\t\t扣税\t\t到手薪资";  
 writer.write(title);  
 writer.newLine();  
 writer.flush();  
  
 for (Employee emp : list) {  
  
 *// 3. 取出员工数据, 拼接为一个字符串* StringBuilder sb = new StringBuilder();  
 sb.append(emp.getName()).append("\t\t").append(emp.getSalary()).append("\t")  
 .append(emp.getTax()).append("\t").append(emp.getMoney());  
 *// 写入, 换行, 刷新* writer.write(sb.toString());  
 writer.newLine();  
 writer.flush();  
 }  
  
 writer.close();  
 }  
}



### 3. 读取数据并查询 :

编程题 : 在项目根目录下已存在文件student.txt，文件中存放学生姓名与总成绩.

编写程序, 将数据读取到内存中, 并根据学生的名称, 输出该学生对应的总成绩信息.

学员,总成绩（中间用,分隔）

小明,180

小花,190

小丽,175

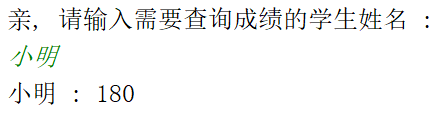
小草,120

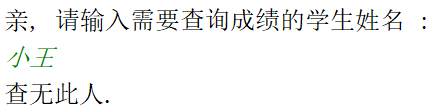
小罗,210

小蒋,200

**代码实现 :**

public class Test5 {  
 public static void main(String[] args) throws IOException {  
  
 *// 1. 创建一个高效的缓冲字符读取流* BufferedReader reader = new BufferedReader(new FileReader("student.txt"));  
  
 *// 4.1 定义一个 Map 映射表集合* HashMap<String, Integer> map = new HashMap<>();  
  
 *// 2. 一行一行读取文件数据* String line = null;  
 while ((line = reader.readLine()) != null) {  
  
 *// line -> 小明,180  
 // 3. 切割 split* String[] split = line.split(",");  
  
 *// 4.2 取出数据, 将数据封装到 Map 映射表集合中, 好处: 可以根据 key 查找对应的 value* String key = split[0];  
 int score = Integer.*parseInt*(split[1]);  
 map.put(key, score);  
 }  
  
 *// 5. 创建一个键盘录入对象* Scanner sc = new Scanner(System.*in*);  
  
 *// 6. 提示用户输入, 并接收* System.*out*.println("亲, 请输入需要查询成绩的学生姓名 :");  
 String name = sc.nextLine();  
  
 */\*for (Map.Entry<String, Integer> entry : map.entrySet()) {  
 String key = entry.getKey();  
 Integer value = entry.getValue();  
 System.out.println(key + " = " + value);  
 }\*/  
  
 // 7. 直接根据姓名从映射表中取值* Integer score = map.get(name);  
  
 *// 8. 根据 score 进行判断* if (score != null) {  
 System.*out*.println(name + " : " + score);  
 } else {  
 System.*out*.println("查无此人.");  
 }  
  
 *// 9. 关闭资源* sc.close();  
 reader.close();  
 }  
}





# 30. IO异常的处理方案 : (了解)

## 1. JDK7之前的处理 :

try – catch – finally 格式完成异常的捕获.

public class Test1 {  
 public static void main(String[] args) {  
  
 *// 复制文件 (字节流)* BufferedInputStream bis = null;  
 BufferedOutputStream bos = null;  
  
 try {  
 *// 1. 创建一个高效的字节输入流* bis = new BufferedInputStream(new FileInputStream("C:\\Users\\xieco\\Desktop\\颈椎操.flv"));  
 *// 2. 创建一个高效的字节输出流* bos = new BufferedOutputStream(new FileOutputStream("C:\\Users\\xieco\\Desktop\\copy1.flv"));  
  
 *// 3. 读写操作* byte[] buf = new byte[1024];  
 int len = -1;  
 while ((len = bis.read(buf)) != -1) {  
 bos.write(buf, 0, len);  
 bos.flush(); *// 刷新* }  
  
 } catch (IOException e) {  
 e.printStackTrace();  
 } finally {  
 if (bos != null) {  
 try {  
 bos.close();  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 }  
 if (bis != null) {  
 try {  
 bis.close();  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
 }  
}

## 2. JDK7之后的处理 :

try – with – resources 格式完成异常的捕获. 作用: 完成资源对象的自动关闭.

public class Test2 {  
 public static void main(String[] args) {  
  
 *// 复制文件 (字节流)  
 // 说明 : try-with-resources 在小括号中的资源可以实现自动关闭. 对象必须要实现 Closeable 接口.* try (  
 BufferedInputStream bis = new BufferedInputStream(new FileInputStream("C:\\Users\\xieco\\Desktop\\颈椎操.flv"));  
 BufferedOutputStream bos = new BufferedOutputStream(new FileOutputStream("C:\\Users\\xieco\\Desktop\\copy2.flv"))  
 ) {  
 *// 3. 读写操作* byte[] buf = new byte[1024];  
 int len = -1;  
  
 while ((len = bis.read(buf)) != -1) {  
 bos.write(buf, 0, len);  
 bos.flush(); *// 刷新* }  
  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
  
 }  
}

## 3. JDK9的改进 :

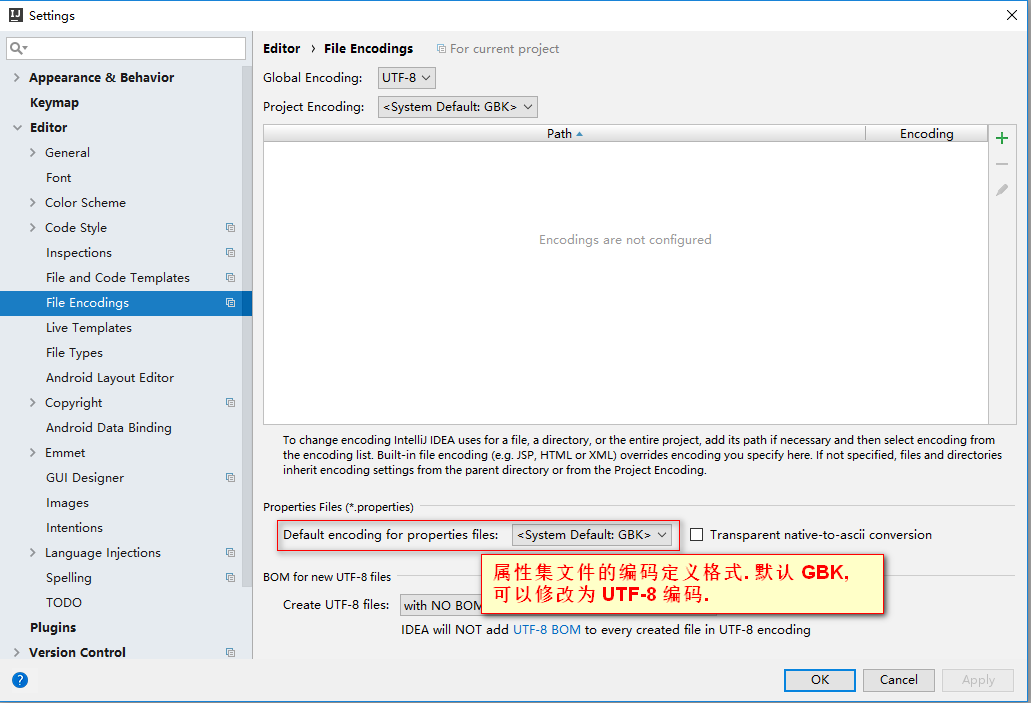
public class Test3 {  
 public static void main(String[] args) throws FileNotFoundException {  
  
 *// 复制文件 (字节流)  
 // 说明 : try-with-resources 在小括号中的资源可以实现自动关闭. 对象必须要实现 Closeable 接口.* BufferedInputStream bis = new BufferedInputStream(new FileInputStream("C:\\Users\\xieco\\Desktop\\颈椎操.flv"));  
 BufferedOutputStream bos = new BufferedOutputStream(new FileOutputStream("C:\\Users\\xieco\\Desktop\\copy3.flv"));  
  
 try (bis; bos) {  
 *// 3. 读写操作* byte[] buf = new byte[1024];  
 int len = -1;  
  
 while ((len = bis.read(buf)) != -1) {  
 bos.write(buf, 0, len);  
 bos.flush(); *// 刷新* }  
  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 }  
}

# 31. Properties 属性集类 : (重点)

**作用 :** 充当**简单**的配置文件.

**格式 :** key = value (**等号**, 冒号, 空格)

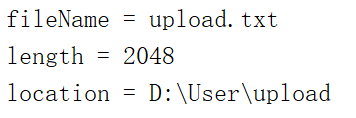
**后缀名 :** properties 属性集配置文件后缀名.





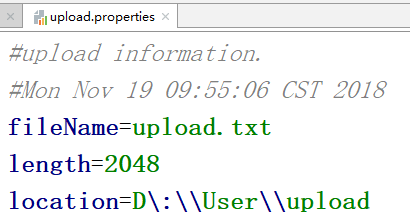
## 1. 设置数据与遍历 : (了解)

public class Test1 {  
 public static void main(String[] args) {  
  
 *// 1. 创建一个 Properties 属性集对象* Properties prop = new Properties();  
  
 *// 2. 设置数据到 prop 对象中* prop.setProperty("fileName", "upload.txt");  
 prop.setProperty("length", "2048");  
 prop.setProperty("location", "D:\\User\\upload");  
  
 *// 3. 遍历 prop 属性集对象中的数据* Set<String> names = prop.stringPropertyNames();  
 for (String name : names) {  
 String value = prop.getProperty(name);  
 System.*out*.println(name + " = " + value);  
 }  
 }  
}



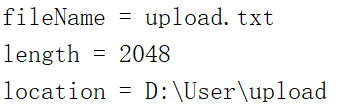
## 2. 存储数据到文件中 : (了解)

public class Test2 {  
 public static void main(String[] args) throws IOException {  
  
 *// 1. 创建一个 Properties 属性集对象* Properties prop = new Properties();  
  
 *// 2. 设置数据到 prop 对象中* prop.setProperty("fileName", "upload.txt");  
 prop.setProperty("length", "2048");  
 prop.setProperty("location", "D:\\User\\upload");  
  
 *// 3. 将 prop 对象中的数据存储到指定文件中. (建议: 文件后缀名为 .properites)  
 // store 很少使用, 原因为配置文件的信息基本都是通过手动直接输入的.* prop.store(new FileWriter("upload.properties"), "upload information.");  
 }  
}



## 3. 读取文件中的数据到属性集对象中 : (重点)

public class Test3 {  
 public static void main(String[] args) throws IOException {  
  
 *// 1. 创建一个属性集对象* Properties prop = new Properties();  
  
 *// 2. 将文件中的数据加载到 prop 对象中实现存储* prop.load(new FileReader("upload.properties"));  
  
 *// 3. 遍历* Set<String> names = prop.stringPropertyNames();  
 for (String name : names) {  
 String value = prop.getProperty(name);  
 System.*out*.println(name + " = " + value);  
 }  
 }  
}



## 4. Properties练习存与取 : (了解)

现在有一个大中华区胡润富豪榜,里面存放了大中华区十位富豪及其资金

该文件叫做fuhao.properties 我们知道该文件里的数据是键值对形式 (properties对象的键值对都是 String 类型的)

比如 马云=2000亿 这种形式存储,但是里面具体数据并不知道是什么

自顺丰速递上市以来,王卫身价飙升,现在已经是1860亿,现在有个需求

判断该富豪榜中是否有王卫,如果有则将其身价修改为1860亿.

将新的数据存到新的文件newfuhao.properties中

提示:采用Properties读取 和 存储

fuhao.txt 文件内容(注意:这个文件里内容我们是未知的!!!!!!)

王健林=2050亿

马云=2000亿

李嘉诚=1750亿

李兆基=1660亿

马化腾=1520亿

丁磊=1250亿

宗庆后=1120亿

姚振华=1050亿

郑家纯=1050亿

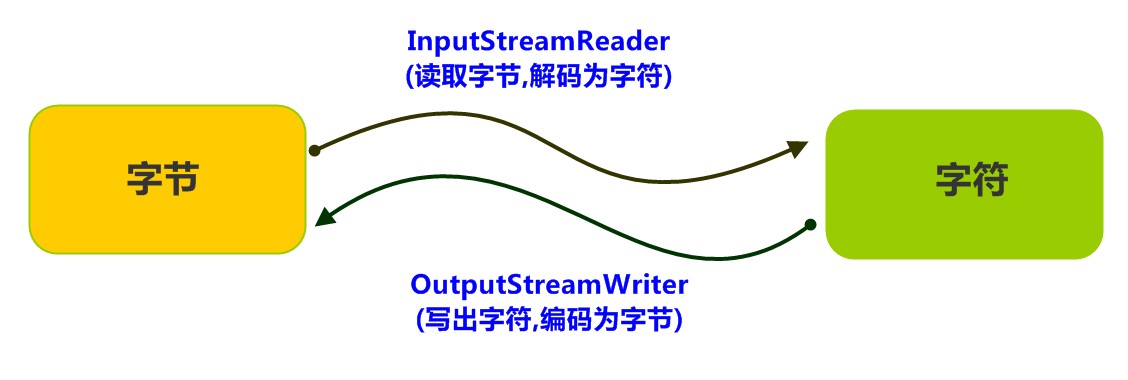
王卫=900亿

**代码实现 :**

public class Test4 {  
 public static void main(String[] args) throws IOException {  
  
 *// 需求 : 读取 fuhao.properties 文件, 判断是否存在王卫, 并修改数值, 最后实现重新存储  
  
 // 1. 创建一个 Properties 属性集对象* Properties prop = new Properties();  
  
 *// 2. 使用 load 方法将文件数据加载到 prop 对象中存储* prop.load(new FileReader("fuhao.properties"));  
  
 *// 3. 遍历属性集对象* Set<String> names = prop.stringPropertyNames();  
 for (String name : names) {  
  
 *// 4. 判断* if ("王卫".equals(name)) {  
 *// 5. 修改数值* prop.setProperty(name, "1860亿");  
 }  
 String value = prop.getProperty(name);  
 System.*out*.println(name + " = " + value);  
 }  
  
 *// 6. 将 prop 对象中的数据存储到一个新文件中* prop.store(new FileWriter("newFuHao.properties"), "rich people information.");  
 }  
}



# 32. 转换流 :



请问 : 转换流 (InputStreamReader 和 OutputStreamWriter) 和 便捷流 (FileReader 和 FileWriter) 是什么关系 ??? 继承关系.

公式 : 字符流 = 字节流 + 编码表

FileReader = InputStreamReader + 默认编码表

FileWriter = OutputStreamWriter + 默认编码表

小结 :

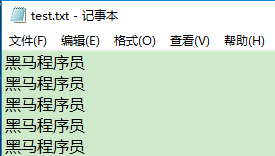
1. 如果文件数据编码与项目默认编码一致, 直接使用 `便捷流` 实现读取. 方便, 快捷.

2. 如果文件数据编码与项目默认编码不一致, 需要使用 `转换流` 指定编码表实现数据读取.

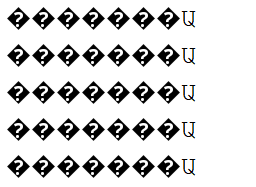
**乱码的产生说明 : 编解码不一致. (文件: GBK 默认编码: UTF-8)**

Windows 系统文件, 默认 GBK 编码.

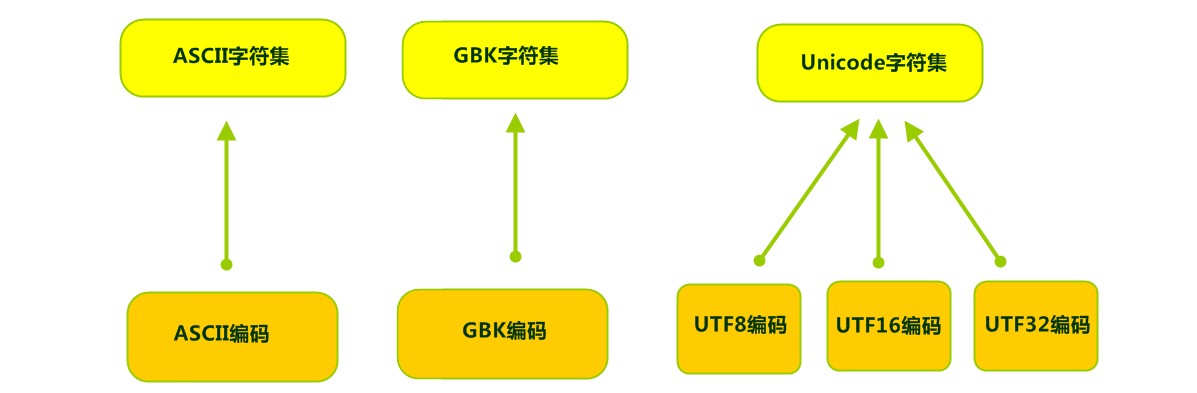
IDEA + FileWriter 读取数据, 会产生乱码.



public class Test1 {  
 public static void main(String[] args) throws IOException {  
BufferedReader reader = new BufferedReader(new FileReader("C:\\Users\\xieco\\Desktop\\test.txt"));  
  
 String line = null;  
 while ((line = reader.readLine()) != null) {  
 System.*out*.println(line);  
 }  
  
 reader.close();  
 }  
}



## 1. 字符集 : (了解) charsetName



UTF-8 :

如果一个字节能够存储, 就使用一个字节存储. (ASCII)

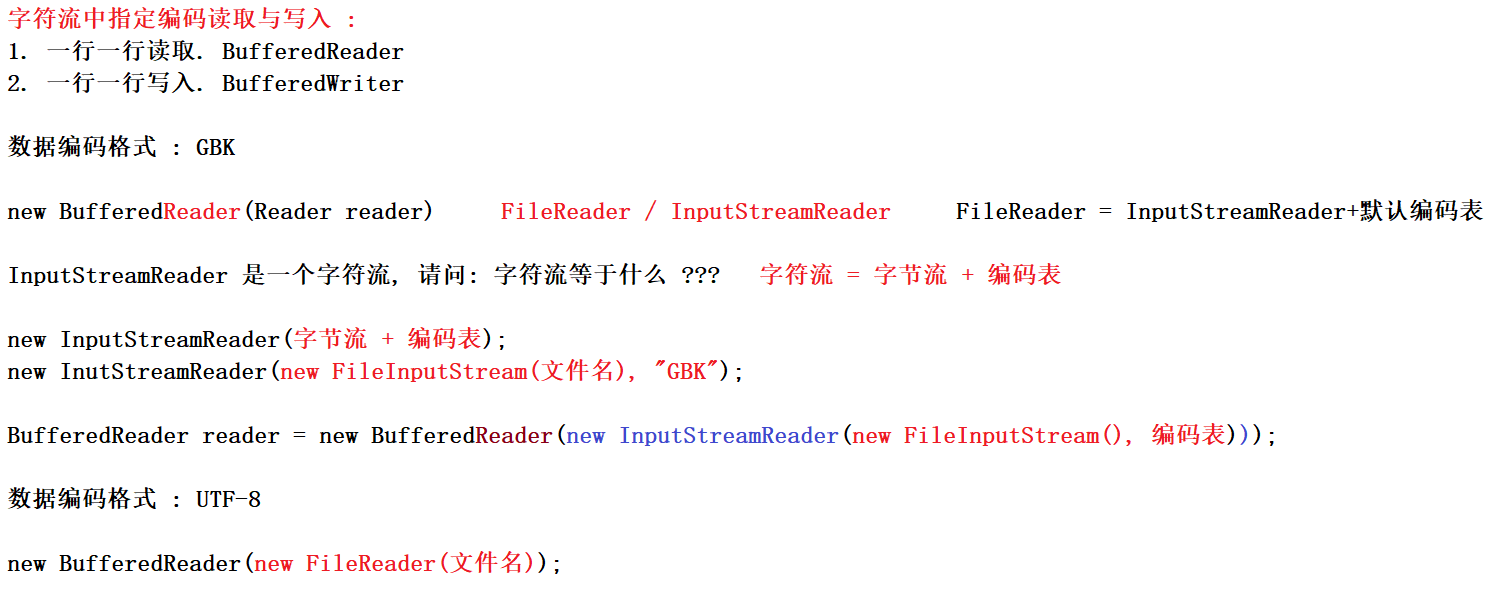
如果一个字节使用完毕, 就使用二个字节实现存储.

汉字在 UTF-8 编码中编码为了3个字节一个汉字.

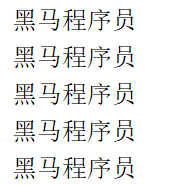
GBK : 国标王

一个汉字二个字节.

## 2. 指定编码读取 : GBK 方式



public class Test2 {  
 public static void main(String[] args) throws IOException {  
  
 // FileReader = InputStreamReader + 默认编码表  
 // 构造方法 : InputStreamReader(字节流, 编码表)  
 /\*  
 InputStreamReader in = new InputStreamReader(new FileInputStream("C:\\Users\\xieco\\Desktop\\test.txt"), "utf-8");  
 BufferedReader reader = new BufferedReader(in);  
 \*/  
  
 BufferedReader reader = new BufferedReader(new InputStreamReader(  
 new FileInputStream("C:\\Users\\xieco\\Desktop\\test.txt"), "gbk"));  
  
 String line = null;  
 while ((line = reader.readLine()) != null) {  
 System.out.println(line);  
 }  
  
 reader.close();  
 }  
}



## 3. 指定编码写出 : GBK 方式

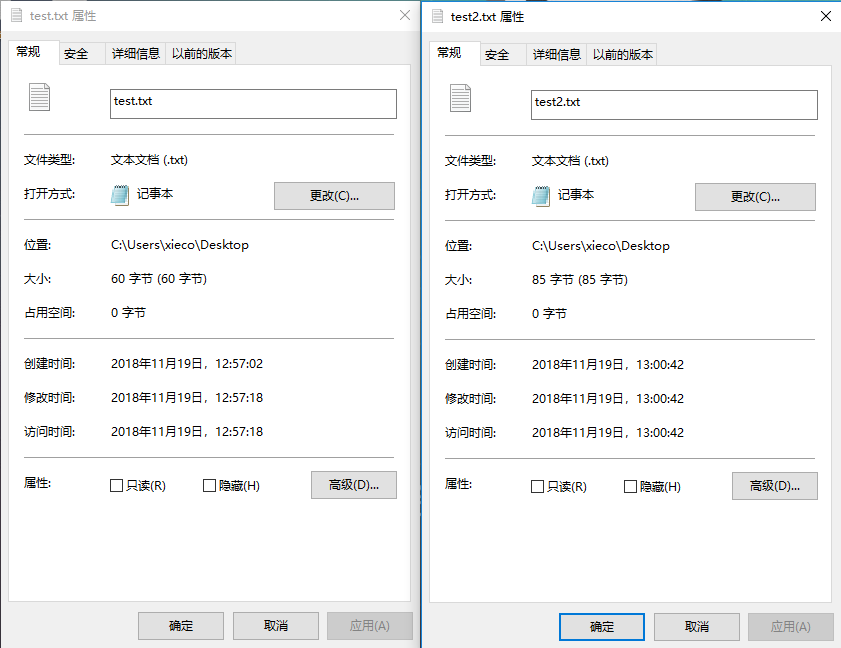
public class Test1 {  
 public static void main(String[] args) throws IOException {  
  
 // 硬盘的作用是存储数据, 数据在硬盘上是以字节的方式存储的. 字节就是 0 1 二进制数据.  
 // 编码 : 将字符数据转换为字节数据. (内存 -> 硬盘) 硬盘上的二进制.  
 // 解码 : 将字节数据转换为字符数据. (硬盘 -> 内存) 能读取的字符  
  
 // 默认编码表 (UTF-8)  
 // BufferedWriter writer = new BufferedWriter(new FileWriter("test2.txt"));  
  
 // 指定编码表 (GBK)  
 // new OutputStreamWriter(new FileOutputStream("test2.txt"), "GBK");  
 BufferedWriter writer = new BufferedWriter(new OutputStreamWriter(  
 new FileOutputStream("test2.txt"), "GBK"));  
  
 for (int i = 0; i < 5; i++) {  
 writer.write("黑马程序员");  
 writer.newLine();  
 writer.flush();  
 }  
  
 writer.close();  
 }  
}

## 4. 转换文件编码 : (重点)

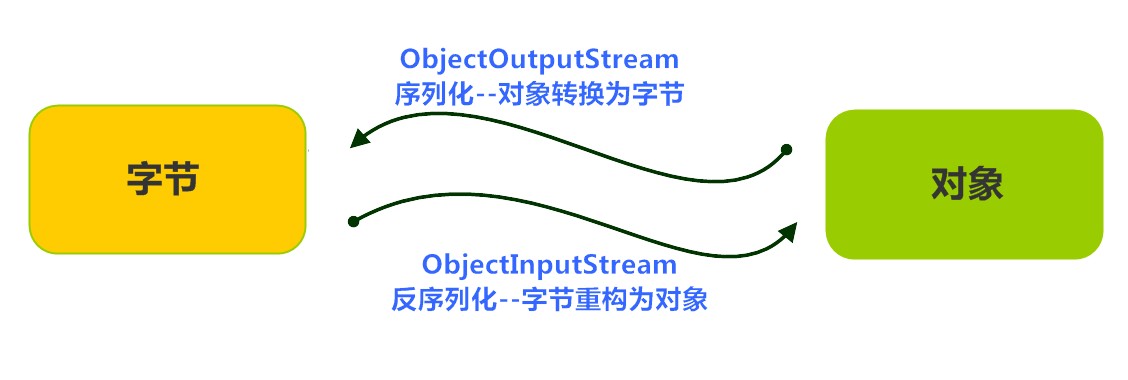
需求 : 读取 Windows 平台的 GBK 编码数据, 并使用 UTF-8 编码写入数据到硬盘的抽象路径中.

public class Test2 {  
 public static void main(String[] args) throws IOException {  
  
 // 需求说明 :  
 // 1. 硬盘上有一个 test.txt 文件, 编码格式为 : GBK (Windows 系统创建的)  
 // 2. 需要将 test.txt 文件读取到程序中. 一定要正确读取.  
 // 3. 将文件中的数据重新使用 utf-8 格式写入到 windows 硬盘中存储.  
  
 // BufferedWriter(Writer writer)  
 // new FileWriter() // 默认编码表 utf-8  
 // new OutputStreamWriter(new FileOutputStream(), "编码表名称"); 字符流对象 = 字节流 + 编码表  
  
 // 步骤一 : BufferedReader + (转换流指定GBK编码表)  
 // new InputStreamReader(new FileInputStream("C:\\Users\\xieco\\Desktop\\test.txt"), "GBK")  
 BufferedReader reader = new BufferedReader(new InputStreamReader(  
 new FileInputStream("C:\\Users\\xieco\\Desktop\\test.txt"), "GBK"));  
  
 // 步骤三 : 创建一个使用 utf-8 格式的写入流 (FileWriter, OutputStreamWriter+utf-8)  
 BufferedWriter writer = new BufferedWriter(new OutputStreamWriter(  
 new FileOutputStream("C:\\Users\\xieco\\Desktop\\test2.txt"), "UTF-8"));  
  
  
 // 步骤二 : 读取  
 String line = null;  
 while ((line = reader.readLine()) != null) {  
  
 // 步骤四 : 使用 UTF-8 格式将数据重新写入到硬盘  
 writer.write(line);  
 writer.newLine();  
 writer.flush();  
 System.out.println(line);  
 }

// 步骤五 : 释放资源  
 writer.close();  
 reader.close();  
 }  
}



# 33. 序列化与反序列化 : (了解)



序列化 : 将Java对象转换成为字节数据存储到文件中.

反序列化 : 将文件中的字节数据重构为一个Java对象.

## 1. 序列化一个自定义对象 :

**接口 Serializable :** 类通过实现 java.io.Serializable 接口以**启用其序列化功能**。

说明 : 该接口不存在任何抽象方法, 这样的接口被称为 `标识性` 接口. 仅仅使用该接口来启用序列化.

Student 类定义 :

/\*  
要求1 : 被序列化的所属类, 必须要实现 Serializable 接口.  
要求2 : 为了防止反序列化读取对象失败, 序列化所属的类必须要提供一个序列版本号.  
  
格式 : ANY-ACCESS-MODIFIER static final long serialVersionUID = 42L;  
  
序列化数据的关键字 : 被 transient 修饰的属性, 不参与序列化.  
 \*/  
  
import java.io.Serializable;  
  
public class Student implements Serializable {  
 // 属性  
 private String name;  
 private transient int age;  
  
 // 提供指定的序列版本号  
 private static final long serialVersionUID = 0L;  
  
 public Student(String name, int age) {  
 this.name = name;  
 this.age = age;  
 }  
  
 @Override  
 public String toString() {  
 return "Student{" +  
 "name='" + name + '\'' +  
 ", age=" + age +  
 '}';  
 }  
  
 public void doHomework() {  
 System.out.println("认真完成作业...");  
 }  
  
 public void study() {  
 System.out.println("好好学习, 天天向上.");  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 public int getAge() {  
 return age;  
 }  
  
 public void setAge(int age) {  
 this.age = age;  
 }  
}

**序列化 :**

public class Test1 {  
 public static void main(String[] args) throws IOException {  
  
 *// 序列化 : 将 Java 对象写入到文件中.  
 // 未来使用 : 将 Java 对象存储到数据库中, 并不是文件中.* Student stu = new Student("柳岩", 18);  
  
 *// 请问 : 如何将 stu 对象存储到文件中 ??? ObjectOutputStream -> writeObject  
 // 1. 创建一个序列化对象* ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("stu.txt"));  
  
 *// 2. 调用 writeObject 方法将对象写入到文件中* oos.writeObject(stu);  
  
 *// 3. 关闭资源* oos.close();  
 }  
}

**反序列化 :**

public class Test2 {  
 public static void main(String[] args) throws IOException, ClassNotFoundException {  
  
 *// 反序列化 : 将文件中的数据读取为程序的 Java 对象.  
 // ObjectInputStream -> readObject();  
  
 // 1. 创建一个反序列化对象* ObjectInputStream ois = new ObjectInputStream(new FileInputStream("stu.txt"));  
  
 *// 2. 读取文件中的数据* Object obj = ois.readObject();  
 System.*out*.println("obj = " + obj);  
  
 *// 3. 关闭资源* ois.close();  
 }  
}



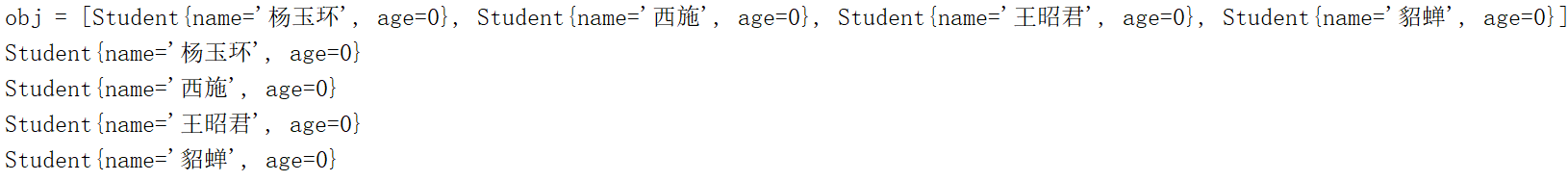
## 2. 序列化集合 : (了解)

**序列化 :**

public class Test3 {  
 public static void main(String[] args) throws IOException {  
  
 *// 集合序列化 :  
  
 // 1. 创建一个集合* ArrayList<Student> stus = new ArrayList<>();  
  
 stus.add(new Student("杨玉环", 16));  
 stus.add(new Student("西施", 18));  
 stus.add(new Student("王昭君", 17));  
 stus.add(new Student("貂蝉", 19));  
  
 *// 需求 : 将集合中的所有对象存储到文件中  
 // 2. 创建一个序列化对象* ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("stus.data"));  
  
 *// 3. 写入集合到文件* oos.writeObject(stus);  
  
 *// 4. 关闭资源* oos.close();  
 }  
}

**反序列化 :**

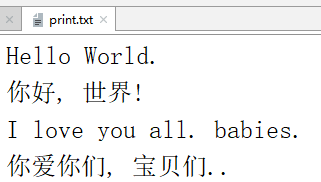
public class Test4 {  
 public static void main(String[] args) throws IOException, ClassNotFoundException {  
  
 *// 1. 创建一个反序列化对象* ObjectInputStream ois = new ObjectInputStream(new FileInputStream("stus.data"));  
  
 *// 2. 读取文件中的数据为 Java 对象.* Object obj = ois.readObject();  
 System.*out*.println("obj = " + obj);  
  
 ArrayList<Student> stus = (ArrayList<Student>) obj;  
 for (Student stu : stus) {  
 System.*out*.println(stu);  
 }  
  
 *// 3. 关闭资源* ois.close();  
 }  
}



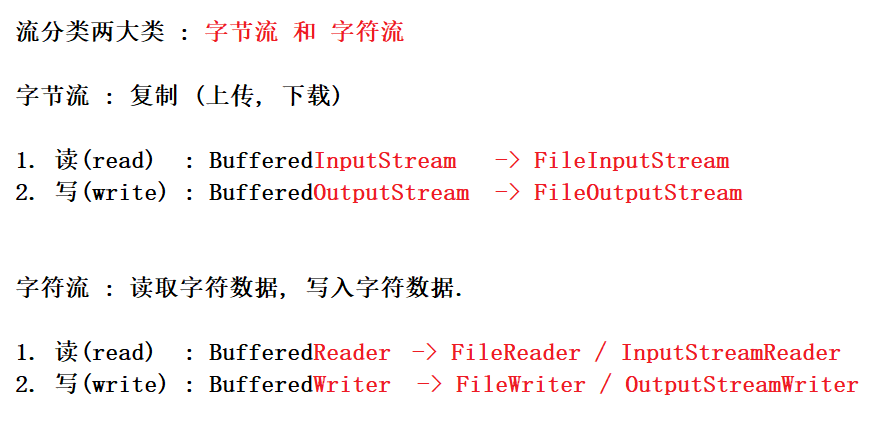
# 34. PrintStream打印流 : (了解)

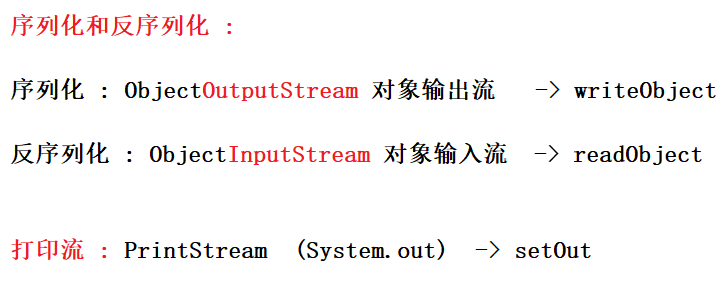
**System.out 就是一个打印流 .**

public class Test1 {  
 public static void main(String[] args) throws FileNotFoundException {  
  
 *// 技巧 : 更改系统的打印流方向.* PrintStream ps = new PrintStream("print.txt");  
 System.*setOut*(ps);  
  
 *// PrintStream 打印流提供了 print / println 方法.* System.*out*.println("Hello World.");  
 System.*out*.println("你好, 世界!");  
 System.*out*.println("I love you all. babies.");  
 System.*out*.println("你爱你们, 宝贝们..");  
 }  
}



# 35. IO流小结 :





# 36. Scanner 使用注意点 :

**演示一 :**

public class Test {  
  
 // 属性, 如果多个方法中都需要使用 Scanner 对象, 将 Scanner 对象定义为属性数据即可.  
 // private static Scanner sc = new Scanner(System.in);  
  
 public static void main(String[] args) {  
  
 // Scanner(InputStream source) System.in 系统默认输出流为键盘输入流  
  
 Scanner sc = new Scanner(System.in); // 系统的输入流, 默认从键盘输入. (文本框)  
  
 // next(); 以空格作为接收结束符.  
 String str = sc.next();  
 System.out.println("str = " + str);  
  
 // nextLine(); 以回车作为结束符. (处理回车)  
 String str2 = sc.nextLine();  
 System.out.println("str2 = " + str2);  
  
 sc.close(); // 如果 Scanner 是一个成员变量 / 静态变量, 就不需要关闭了.  
 }  
  
 // 方法 :  
 public static void testScanner() {  
  
 }  
}

**演示二 :**

public class Test2 {  
 public static void main(String[] args) {  
  
 Scanner sc = new Scanner(System.*in*);  
  
 String name = sc.nextLine();  
 System.*out*.println("name = " + name);  
  
 *// 18 回车  
 // int age = sc.nextInt(); // 只能接收 int 值, 但是不处理回车.  
 // sc.nextLine(); // 专门用户处理回车  
  
 // Integer 包装将字符串解析为 `整型数值`.* int age = Integer.*parseInt*(sc.nextLine());  
 System.*out*.println("age = " + age);  
 *//* String address = sc.nextLine(); *// 使用回车作为结束符, 接收数据完毕后, 会处理回车.* System.*out*.println("address = " + address);  
  
 sc.close();  
 }  
}