## 对象的内存解析

|  |
| --- |
|  |
|  |
| 在new对象的时候，会首先在堆空间中开辟一块空间，然后赋给对象中的属性默认值，还会再栈空间中开辟一块与堆空间地址值(如下图中的0\*12ab)相同的空间来指向堆空间中的对象  对象的声明是在栈空间，赋值时在堆空间。 |
| 局部变量存在栈空间中，将new出来的对象存放在堆中，对象的属性（非static的）加载在堆空间中。  方法区：类的加载信息、常量池、静态域。 |

## 对象数组的内存解析

|  |
| --- |
| 对象数组的内存解析:  引用类型 的变量，只可能存储两类值：null，或者地址值(含有变量类型) |
| 匿名对象： |

## 理解“万物皆为对象”

|  |
| --- |
|  |

## 可变个数的形参

|  |
| --- |
|  |
|  |

## 递归方法的使用

|  |
| --- |
| 上面的那个递归方法，实际上相当于从100加到1 |

## Java中的参数传递机制：值传递

|  |
| --- |
|  |

## 权限修饰符

|  |
| --- |
|  |
|  |

## 构造器

|  |
| --- |
|  |
| 注意：构造器虽然也叫构造方法，但是它不是一个方法，它是用来创建对象的， |
| 构造器还可以初始化对象的某个属性，比如： |
|  |
|  |
|  |

## this关键字

|  |
| --- |
| 可以用来修饰属性、方法、构造器   1. this修饰属性、方法，this理解未当前对象，在类的方法中，可以使用”this.属性/方法”的方式，调用当前对象的属性/方法，但是，通常选择省略”this.”，特殊情况：如果方法的形参和类的属性同名时，我们必须显示的使用“this.变量”的方法，表名此变量是属性而非形参。 2. 修饰构造方法，this可以理解为当前对象或者当前正在创建的对象。 |
| this调用构造器：   1. 在构造方法中的写法“this(形参列表)”来调用本类中的其他构造器； 2. 构造器中，不能通过这种方式调用自己； 3. 如果一个类中有n个构造器，则最多有n-1个构造器中使用这种写法，否则会出现死循环； 4. 规定：这种写法必须写在调用方的首行； 5. 构造器的内部不能同时调用多个构造器（除非a调b，b调用c）；，即最多只能声明那个一个调用构造器的方法。 |

## 多态性

|  |
| --- |
| 何为多态性:  对象的多态性：父类的引用指向子类的对象（即左面是父类的声明，右面new的是子类的对象）； |
|  |
|  |
| **多态的使用：**  当调用子类与父类同名同参方法时，实际上执行的时子类重写父类的方法（又叫虚拟方法调用）；  不能调用子类特有的方法。  即：有了对象的多态性以后，我们在编译期只能调用父类声明的方法，但在运行期，实际运行的时子类重写父类的方法；  总结：编译看左边、运行看右边 |
| **多态的好处：**  当同一种重载的方法的形参有相同的父类的时候，能将多个方法写成一个（最好是方法体内的内容是一致的，当然了，不一样也可以） |
| 当main方法里面的function()里面些new dog 的时候，控制台打印的是下面的：    当写的是new Animal的时候，控制台打印的是下面的内容： |
|  |
| 多态是一个运行时行为 |

## 进一步理解重载和重写

|  |
| --- |
|  |

## Object类及其内部方法

|  |
| --- |
|  |

## static关键字

|  |
| --- |
|  |
|  |
|  |
|  |
| 如何确定一个属性或者方法是否需要声明为static： |
| 应用举例： |
| 备注：如果将代码中第13和14行代码互换位置，那么输出的制造个数都是2，原因是调了两次构造器，每次都会导致total加1. |

## 单例设计模式

|  |
| --- |
|  |
| 实现： |
| **饿汉式实现：** |
| **懒汉式实现**（这个是线程不安全的，后续需要改进）： |
| 区分单例模式的饿汉式和懒汉式： |
| 单例模式的**使用场景**： |
|  |
|  |

## 代码块(初始化块)

|  |
| --- |
| 1. 作用：用来初始化类、对象 2. 代码块如果有修饰的话，只能用static。 3. 分类：静态代码块和非静态代码块 4. 静态代码块： 5. 内部可以有输出语句 6. 随着类的加载而执行 7. 自始至终只执行一次 8. 作用：初始化当前类的信息 9. 如果一个类中定义了多个静态代码块，则按照声明的先后顺序执行 10. 静态代码块的执行要优先于非静态代码块的执行 11. 静态代码块内只能调用静态的属性或方法，不能调用非静态的 12. 非静态代码块： 13. 内部可以有输出语句 14. 随着对象的创建而执行 15. 每次创建对象都会被执行 16. 作用：可以在创建对象的时候对对象的属性等进行初始化 17. 如果一个类中定义了多个非静态代码块，则按照声明的先后顺序执行 18. 执行要晚于静态代码块 19. 非静态代码块内部既可以调用静态的属性和方法，也能调用非静态的属性和方法 |
|  |
|  |

## final关键字：

|  |
| --- |
|  |

## 抽象类和抽象方法abstract

|  |
| --- |
|  |
|  |
| 使用abstract的注意点： |
| 练习： |
| 新建Manager和CommonEmployee类（可以有自己独有的属性），继承Employee，然后在重写的work方法中重写属于自己的逻辑      写测试类，分别调用里面的方法    **灵感：**之前在做教育系统工资统计的时候，聂总给了一个思路是写一个接口，根据不同的部门来写不同的实现类来实现这个接口中的方法，可以参照上面的这种思路，写一个工资计算的抽象方法，然后用不同的子类来重写这个方法，在进行计算的时候只需要调用不同的子类的同一个方法即可 |

## 接口

|  |
| --- |
|  |

## 代理模式：

|  |
| --- |
| 两个类共同实现同一个接口，一个是代理类，一个是被代理类，然后在程序中调用代理类的方法，执行被代理类中的逻辑。  就像明星和经纪人，有谈业务、签合同、唱歌、收钱等动作，谈业务、签合同、收钱是由经纪人代理明星来完成的，相当于调用的是经纪人中的方法，明星自己唱歌(但是是在经纪人中调的唱歌的方法)，这样既有谈业务、签合同、收钱的操作(经纪人代替完成)，又有唱歌（明星自己完成） |
| 运行： |
|  |

## 内部类

|  |
| --- |
| 1580737393(1) |
|  |
|  |

## idea模板设置：

|  |
| --- |
| 加上组名：    比如要声明一个public static final String类型的常量，可以这样： |

## 多线程

**代码位置：**

**D:\work\work\shangxuetang**

|  |
| --- |
|  |
|  |
|  |
|  |
|  |
| **写多线程:** |
| 方式一： 继承Thread类：   1. 创建一个继承Thread类的子类 2. 重写Thread的run()方法，将该线程要进行的操作写到这个方法体里面 3. 创建Thread的子类的对象 4. 通过此对象调用start() |
| 注意：一个Thread对象只能start一次，重复调用会抛异常！    这个时候应该重新new一个对象，然后用新的对象调用start()方法 |
| 练习：  创建两个线程，其中一个线程遍历100以内的偶数，另一个线程遍历100以内的奇数  两个线程      结果：    也可以这样实现（用匿名对象的方式） |
| **线程的常用方法** |
|  |
| Thread.currentThread().getName():    设置线程名setName():    用构造器给线程命名    yield()方法:  当线程执行到截图中能被20整除的位置,该线程就将执行权让给其他线程执行 |
| **线程的调度(线程优先级的使用)** |
|  |
|  |
|  |
|  |
| 多线程实现方式二:实现Runnable接口 |
|  |
|  |
| 上面的买票需求： |
| 两种创建方式的对比： |
|  |
| **线程的生命周期** |
|  |
|  |
| **线程的同步**  主要解决的是线程安全问题 |
| 方式一：同步代码块  synchronized(同步监视器){  //需要被同步的代码(**包含的代码既不能多也不能少**)  }  说明：   1. 操作共享数据的代码，即为需要被同步的代码 2. 共享数据：多个线程共同操作的变量 3. 同步监视器：俗称“锁”，任何一个类的对象都可以充当锁。   对于同步监视器的要求：多个线程必须共用同一把锁  补充：  ①在用实现Runnable接口的方式创建多线程的过程中，可以考虑用this关键字充当监视器  ②在继承Thread类的方式中，慎用this关键字来充当监视器，可以考虑用当前类来充当同步监视器      上面的代码也可以这么写： |
| 局限性：在共享数据的部分，代码走的是串行的 |
| 方式二：同步方法  如果操作共享数据的代码正好完整地声明在一个方法中，可以将此方法声明成同步的   1. 解决实现Runnable接口方式创建多线程的安全问题      1. 解决继承Thread类的方式创建多线程的安全问题 |
| 关于同步方法的总结：   1. 同步方法仍然涉及到同步监视器，只是我们不需要显示的声明； 2. 非静态的同步方法，同步监视器时this，对于静态同步方法，他的监视器时当前类本身； 3. 不论是同步代码块还是同步方法，都有一个共同的局限性：在被共享数据的部分，代码走的时串行的方式 |
|  |
| 用同步机制解决单例的懒汉式实现中存在的线程不安全问题    过程：  第一步：address = allocate //申请内存空间  第二部：instance(Object): //实例化对象  第三步：myinstance = address: 对象指向申请到的内存空间的地址。 |
| **线程的死锁问题** |
|  |
| **Lock锁的方式解决线程安全问题** |
|  |
| **面试题：synchronized方式与lock方式的异同？**   1. 相同点：都是用来解决线程安全问题的； 2. 不同点： 3. synchronized方式是在执行完相应的同步代码后自动地释放代码监视器；而lock方式需要手动地启动同步（调用lock()方法）&&手动结束同步（调用unlock()方法）。 |
|  |
| **线程的通信：** |
| 例如：两个线程交替打印1-100  涉及到的线程通信的三个方法：   1. wait(): 一旦执行此方法，当前线程就会进入阻塞状态并释放同步监视器 2. notify()：一旦执行此方法，就会唤醒被wait()的一个线程，如果此时有多个线程被wait()，那么就唤醒优先级高的那个； 3. notifyAll(): 一旦执行此方法，就会唤醒所有被wait()的线程。   下面的线程执行过程是这样的，线程一进入，拿到锁，打印数字，然后进入等待状态，执行wait()方法会释放锁，这是线程二进来并拿到锁，用notify()方法将线程一唤醒，但是线程二现在拿着锁所有线程一进不来，线程二打印完数字&&进入wait()状态会释放锁，这时候线程一拿到锁进来，唤醒线程二。。。。。。。      结果： |
| **注意：**   1. **上面的三个方法只能出现在同步代码块或者同步方法中;** 2. **上面的三个方法的调用者必须与同步监视器一致，否则会出现IllegalMonitorStateException异常；** 3. **上面的三个方法不是定义在Thread类中了，而是定义在java.lang.Object类中了。** |
| sleep()方法与wait()方法的异同 |
| 1. 相同点：都会使当前线程进入阻塞状态； 2. 不同点： 3. 方法声明的位置不同，sleep()方法声明在Thread类中，wait()声明在Object中； 4. 调用场景不同：sleep()方法可以在任何场景下调用，而wait()必须使用在同步代码块或者同步方法中，用同步监视器进行调用； 5. 关于是否释放同步监视器：sleep()不会释放，wait()会释放锁。 |
| 一个多线程的经典问题  1581597262(1)        测试类： |
| 创建多线程的方式三：实现Callable接口(jdk5新增的) |
| 与实现Runnable相比，实现Callable的好处：   1. 相比于run()方法，可以有返回值； 2. 方法中可以抛异常； 3. 支持泛型的返回值； 4. 需要借助FutureTask类，比如获取返回结果。 |
| **线程池** |
|  |
| 线程池的代码实现 |
|  |

## 常用类：

|  |
| --- |
| **String** |
| 不可变性的体现：   1. 当对字符串重新赋值时，需要重新指定区域赋值，不能使用原有的value进行赋值； 2. 当对现有字符串进行拼接操作时，也需要重新制定内存区域赋值，不能使用原有的value进行赋值； 3. 当调用String类的replace()方法修改指定字符或字符串时，也需要重新指定内存区域赋值，也不同使用原有的value进行赋值。 |
|  |
|  |
|  |
|  |
| 一道面试题： |
| **String的常用方法** |
|  |
| **String与其他类型的数据、包装类之间的转换** |
|  |
| **String与字符数组char[] 之间的转换** |
| **StringBuffer && StringBuilder** |
|  |
| **StringBuffer的常用方法** |
|  |
| **关于日期和时间的类：** |
|  |
|  |
| String&&StringBuilder&&StringBuffer常见题目 |
| 1. /\*\*  \* 将字符串中的指定部分进行反转，如abcdefg反转为abfedcg  \* 除了下面的方法，还能进行拼接（这里就不写了）  \*/  @Test **public void** reverse() {  **int** startIndex = 2;  **int** endIndex = 5;  String str = **"abcdefg"**;  **char**[] array = str.toCharArray();  **for** (**int** i = startIndex, j = endIndex; i < j; i++, j--) {  **char** temp = array[i];  array[i] = array[j];  array[j] = temp;  }  System.***out***.println(array); } |
| 获取一个字符串在另一个字符串中出现的次数（比如"ab"在"abkkcadkabkebfkabkskab"中出现的次数）  **public int** getCount(String mainStr, String subStr) {  **int** mainLength = mainStr.length();  **int** subLength = subStr.length();  **int** count = 0;  **int** index;  **if** (mainLength >= subLength) { *//如果mainLength < subLength的话，count一定是0* **while** ((index = mainStr.indexOf(subStr)) != -1) {  count++;  mainStr = mainStr.substring(index + subStr.length());  }  }  **return** count; }  @Test **public void** testCount() {  String mainStr = **"abkkcadkabkebfkabkskab"**;  String subStr = **"ab"**;  **int** count = getCount(mainStr, subStr);  System.***out***.println(**"==========>得到的count的值是："** + count); } |
| */\*\*  \* 获取两个字符串中的相同的子字符串(只考虑两个字符串中只有一个相同子串)  \* str1 = "abcwerthelloyuiidef", str2 = "cvhellobnm"  \*/*  **public** String getSameStr(String str1, String str2) {  **if** (str1 != **null** && str2 != **null**) {  *//先判断str1和str2的长度，较大的叫maxStr，较小的叫minStr* String maxStr = (str1.length() >= str2.length()) ? str1 : str2;  String minStr = (str1.length() < str2.length()) ? str1 : str2;  *//声明一个length，让他是较小的字符串的长度* **int** length = minStr.length();  **for** (**int** i = 0; i < length; i++) { *//循环较小长度的字符串  //截取index=j 到k之间的部分和maxStr比较，如果maxStr包含截取的字符串，则将该截取的字符串返回* **for** (**int** j = 0, k = length - i; k <= length - j; j++, k++) {  String subStr = minStr.substring(j, k);  **if** (maxStr.contains(subStr)) {  **return** subStr;  }  }  }  }  **return null**; }  @Test **public void** testGetSameStr() {  String str1 = **"abcwerthelloyuiidef"**;  String str2 = **"cvhellobnm"**;  String resultStr = getSameStr(str1, str2);  System.***out***.println(**"=========>最大相同子字符串为："** + resultStr); } |

## 日期和时间的API(jdk1.8以前的)

|  |
| --- |
|  |
|  |
|  |

## 日期和时间的API(jdk1.8及以后的)

|  |
| --- |
| LocalDate && LocalTime && LocalDateTime    @Test **public void** LocalNowTest() {  *//now():获取当前的日期、时间、日期+时间* LocalDate date = LocalDate.*now*();  LocalTime time = LocalTime.*now*();  LocalDateTime dateTime = LocalDateTime.*now*();  System.***out***.println(**"=======>当前日期："** + date); *//=======>当前日期：2020-03-07* System.***out***.println(**"=========>当前时间："** + time); *//=========>当前时间：19:17:56.884* System.***out***.println(**"=========>当前日期+时间："** + dateTime);*//=========>当前日期+时间：2020-03-07T19:17:56.884    //of():设定指定的年月日时分秒，不存在时间的偏移量* LocalDateTime dateTime1 = LocalDateTime.*of*(2020, 3, 7, 19, 30, 22);  System.***out***.println(**"======>用of()设置的日期+时间"** + dateTime1);*//======>用of()设置的日期+时间2020-03-07T19:30:22    //getXxx():获取一些属性值的方法 (不只是下面的这些，还有很多getXxx方法)（今天是2020-03-07）* System.***out***.println(dateTime.getDayOfMonth()); *//7* System.***out***.println(dateTime.getDayOfWeek()); *//SATURDAY* System.***out***.println(dateTime.getMonth()); *//MARCH* System.***out***.println(dateTime.getMonthValue()); *//3* System.***out***.println(dateTime.getMinute()); *//39    //withXxx():设置相关属性  // 不可变性* LocalDate date1 = date.withDayOfMonth(22);  System.***out***.println(**"===>date.withDayOfMonth(22)中date的值："** + date); *//===>date.withDayOfMonth(22)中date的值：2020-03-07* System.***out***.println(**"===>返回值date1的值："** + date1); *//===>返回值date1的值：2020-03-22    //plusXxx()：加一定的时间（不可变性）* LocalDateTime dateTime2 = dateTime.plusDays(3);  System.***out***.println(**"dateTime.plusDays(3)中dateTime的值："** + dateTime);*//dateTime.plusDays(3)中dateTime的值：2020-03-07T19:58:41.206* System.***out***.println(**"返回的dateTime2的值："** + dateTime2); *//返回的dateTime2的值：2020-03-10T19:58:41.206    //minusXxx():减一定的时间（也具有不可变性）* LocalDateTime dateTime3 = dateTime.minusDays(3);  System.***out***.println(**"dateTime.minusDays(3)中dateTime的值："** + dateTime);*//dateTime.minusDays(3)中dateTime的值：2020-03-07T19:58:41.206* System.***out***.println(**"返回的dateTime3的值是："** + dateTime3); *//返回的dateTime3的值是：2020-03-04T19:58:41.206* } |
| Instant(有偏移量，个人感觉不实用) |
|  |
| **DateTimeFormatter** |

## java比较器

|  |
| --- |
| 自然排序： |
|  |
| */\*\*  \* Comparable接口(自然排序)  \*/* @Test **public void** comparableTest() {  String[] strArr = **new** String[]{**"PP"**, **"BB"**, **"DD"**, **"AA"**, **"FF"**, **"QQ"**, **"MM"**, **"SS"**};  Arrays.*sort*(strArr);  System.***out***.println(Arrays.*toString*(strArr)); *//[AA, BB, DD, FF, MM, PP, QQ, SS]* } |
| */\*\*  \** ***@description:*** *自定义类实现Comparable自然排序  \* 如果需要排序，我们可以让自定义类实现Comparable接口，重写其中的compareTo(obj)方法  \* 需要在compareTo(obj)方法中指明如何排序  \** ***@date:*** *2020/3/8 15:30  \** ***@author:*** *lizhenhong  \*/*  **public class** Goods **implements** Comparable{   *//商品名称* **private** String **name**;  *//价格* **private double price**;   **public** Goods() {  }   **public** Goods(String name, **double** price) {  **this**.**name** = name;  **this**.**price** = price;  }   **public** String getName() {  **return name**;  }   **public void** setName(String name) {  **this**.**name** = name;  }   **public double** getPrice() {  **return price**;  }   **public void** setPrice(**double** price) {  **this**.**price** = price;  }   @Override  **public** String toString() {  **return "Goods{"** +  **"name='"** + **name** + **'\''** +  **", price='"** + **price** + **'\''** +  **'}'**;  }  *//排序：先按照价格从低到高，再按照名称从高到低*  @Override  **public int** compareTo(Object o) {  **if** (o **instanceof** Goods) { *//判断传入的参数是否是Goods类型的* Goods goods = (Goods) o;  *//方式一：手写比较大小* **if** (**this**.**price** > goods.**price**) {  **return** 1;  } **else if** (**this**.**price** < goods.**price**) {  **return** -1;  } **else** {  *//如果两个price是相等的，那么根据name排序* **return -this**.**name**.compareTo(goods.**name**);  }  *//方式二：调用包装类Double中的compare()方法  //return Double.compare(this.price, goods.price);* } **else** {  *//如果入参不是Goods类型的，抛异常*  **throw new** RuntimeException(**"入参不是Goods类型的！"**);  }   } }  测试类：  @Test **public void** goodsComparableTest() {  Goods[] goods = **new** Goods[5];  goods[0] = **new** Goods(**"小米"**, 699);  goods[1] = **new** Goods(**"华为"**, 1099);  goods[2] = **new** Goods(**"联想"**, 6299);  goods[3] = **new** Goods(**"苹果"**, 1);  goods[4] = **new** Goods(**"戴尔"**, 1099);  Arrays.*sort*(goods);  System.***out***.println(Arrays.*toString*(goods));  *//[Goods{name='苹果', price='1.0'}, Goods{name='小米', price='699.0'}, Goods{name='戴尔', price='1099.0'}, Goods{name='华为', price='1099.0'}, Goods{name='联想', price='6299.0'}]* } |
| **定制排序：** |
| */\*\*  \* 定制排序(Comparator接口)  \* 实现对数组{"PP", "BB", "DD", "AA", "FF", "QQ", "MM", "SS"}内元素从大到小排列  \*/* @Test **public void** comparatorTest() {  String[] strArr = **new** String[]{**"PP"**, **"BB"**, **"DD"**, **"AA"**, **"FF"**, **"QQ"**, **"MM"**, **"SS"**};  Arrays.*sort*(strArr, **new** Comparator<String>() {  @Override  **public int** compare(String o1, String o2) {  **if** (o1 **instanceof** String && o2 **instanceof** String) {  **return** -o1.compareTo(o2);*//从大到小排列* }  **throw new** RuntimeException(**"输入的类型不一致"**);  }  });  System.***out***.println(Arrays.*toString*(strArr));*//[SS, QQ, PP, MM, FF, DD, BB, AA]* } |
| */\*\*  \* 自定义类实现Comparator接口定制排序  \* 先按照名称从低到高，再按照价格从高到低  \*/*  @Test **public void** goodsComparatorTest() {  Goods[] goods = **new** Goods[6];  goods[0] = **new** Goods(**"mi小米"**, 699);  goods[1] = **new** Goods(**"huawei华为"**, 1099);  goods[2] = **new** Goods(**"lenvol联想"**, 6299);  goods[3] = **new** Goods(**"iphone苹果"**, 1);  goods[4] = **new** Goods(**"dell戴尔"**, 1099);  goods[5] = **new** Goods(**"dell戴尔"**, 1);  Arrays.*sort*(goods, **new** Comparator<Goods>() {  @Override  **public int** compare(Goods o1, Goods o2) {  *//判断，如果name相同，则按价格从高到低* **if** (o1.getName().equals(o2.getName())) {  **return** -Double.*compare*(o1.getPrice(), o2.getPrice());  } **else** {  **return** o1.getName().compareTo(o2.getName());  }  }  });  System.***out***.println(Arrays.*toString*(goods));*//[Goods{name='dell戴尔', price='1099.0'}, Goods{name='dell戴尔', price='1.0'}, Goods{name='huawei华为', price='1099.0'}, Goods{name='iphone苹果', price='1.0'}, Goods{name='lenvol联想', price='6299.0'}, Goods{name='mi小米', price='699.0'}]* } |
| **Comparable与Comparator的对比：**   1. Comparable接口的方式一旦一定，保证了Comparable接口实现类的对象再任何位置都能比较大小； 2. Comparator接口属于临时性的比较大小。 |

## System、Math、BigInteger、BigDecimal的使用

|  |
| --- |
| System类： |
| **Math类：** |
| **BigInteger && BigDecimal** |
| BigInteger： |
| BigDecimal： |

## 枚举类：

|  |
| --- |
|  |
| 自定义方式创建枚举类(jdk5.0之前) |
| **class** Reason{   *//1. 声明Reason对象的属性(用private final修饰)* **private final** String **seasonName**;  **private final** String **seasonDesc**;   *//2. 私有化类的构造器，并给对象赋值* **private** Reason(String seasonName, String seasonDesc) {  **this**.**seasonName** = seasonName;  **this**.**seasonDesc** = seasonDesc;  }   *//3. 提供当前枚举类的多个对象(public static final 的)* **public static final** Reason ***SPRING*** = **new** Reason(**"春天"**, **"春天来了"**);  **public static final** Reason ***SUMMER*** = **new** Reason(**"夏天"**, **"夏至未至"**);  **public static final** Reason ***AUTUMN*** = **new** Reason(**"秋天"**, **"秋高气爽"**);  **public static final** Reason ***WINTER*** = **new** Reason(**"冬天"**, **"冰天雪地"**);    *//4. 其他诉求(比如获得枚举类的属性)* **public** String getSeasonName() {  **return seasonName**;  }   **public** String getSeasonDesc() {  **return seasonDesc**;  }    *//其他诉求2(比如提供toString()方法)* @Override  **public** String toString() {  **return "Reason{"** +  **"seasonName='"** + **seasonName** + **'\''** +  **", seasonDesc='"** + **seasonDesc** + **'\''** +  **'}'**;  } }  测试类：  */\*\*  \** ***@description:*** *自定义方式创建枚举类(jdk5.0之前)  \** ***@date:*** *2020/3/14 16:29  \** ***@author:*** *lizhenhong  \*/* **public class ZiDingYiEnumTest**{@Test   **public void** testZidingyi() {  Reason reason = Reason.***SPRING***;  System.***out***.println(reason); //Reason{seasonName='春天', seasonDesc='春天来了'}  } } |
| 使用enum关键字 |
| java.lang.Enum类的主要方法    最主要的方法：    */\*\*  \* 使用enum关键字定义枚举类(jdk5.0及以后)  \*/* **enum** Reason1{   *//1. 提供当前枚举类的对象(多个对象之间用","隔开，末尾的对象用";"结束)* ***SPRING***(**"春天"**, **"春天来了"**),  ***SUMMER***(**"夏天"**, **"夏至未至"**),  ***AUTUMN***(**"秋天"**, **"秋高气爽"**),  ***WINTER***(**"冬天"**, **"冰天雪地"**);   *//2. 声明Reason1对象的属性(用private final修饰)* **private final** String **seasonName**;  **private final** String **seasonDesc**;   *//2. 私有化类的构造器，并给对象赋值* **private** Reason1(String seasonName, String seasonDesc) {  **this**.**seasonName** = seasonName;  **this**.**seasonDesc** = seasonDesc;  }    *//4. 其他诉求(比如获得枚举类的属性)* **public** String getSeasonName() {  **return seasonName**;  }   **public** String getSeasonDesc() {  **return seasonDesc**;  }   *//其他诉求2(比如提供toString()方法)(enum不需要提供toString()方法) // @Override // public String toString() { // return "Reason1{" + // "seasonName='" + seasonName + '\'' + // ", seasonDesc='" + seasonDesc + '\'' + // '}'; // }* }  测试类：  */\*\*  \** ***@description:*** *使用enum关键字定义枚举类(定义的枚举类默认继承于java.lang.Enum类)  \** ***@date:*** *2020/3/14 17:13  \** ***@author:*** *lizhenhong  \*/* **public class** EnumTest {  @Test  **public void** testEnum() {  Reason1 autumn = Reason1.***AUTUMN***;  *//toString()方法* System.***out***.println(autumn); *//AUTUMN  //values()方法* Reason1[] values = Reason1.*values*();  **for** (Reason1 value : values) {  System.***out***.println(**"====>使用values()方法返回的数组对象："** + value);  *//====>使用values()方法返回的数组对象：SPRING  //====>使用values()方法返回的数组对象：SUMMER  //====>使用values()方法返回的数组对象：AUTUMN  //====>使用values()方法返回的数组对象：WINTER* }  *//valueOf(String str)方法：根据提供的str返回枚举类中对象名是str的对象  //如果没有str的枚举类对象，则会抛异常IllegalArgumentException: No enum constant enums.Reason1....* Reason1 winter = Reason1.*valueOf*(**"WINTER"**);  *//Reason1 winter = Reason1.valueOf("WINTER1");(没有str的枚举类对象，会报错)* System.***out***.println(**"=====>使用valueOf(String str)方法返回的对象："** + winter);*//====>使用valueOf(String str)方法返回的对象：WINTER* } } |

## 注解

|  |
| --- |
|  |
| 实例 |
| **如何自定义注解：**      备注：如果注解内部有成员且没有default，则需要给成员变量赋值。 |
| */\*\*  \** ***@description:*** *自定义注解  \** ***@date:*** *2020/3/15 16:22  \** ***@author:*** *lizhenhong  \*/* **public** @**interface** MyAnnotation {   *//没有属性值的* String value();   *//有属性值的* String value1() **default "hello"**;   } |
| **jdk中的元注解：** |

## java集合

|  |
| --- |
| java集合框架概述： |
| **集合框架涉及到的API:** |
| **Collection接口中的方法：**  **package** collection;  **import** org.junit.Test;  **import** java.util.\*;  */\*\*  \** ***@description:*** *Collection接口的常用方法  \** ***@date:*** *2020/3/18 20:44  \** ***@author:*** *lizhenhong  \*/* **public class** CollectionTest {   @Test  **public void** test() {  Collection coll = **new** ArrayList();  *//add(Object obj)方法:将元素obj添加到coll中* coll.add(**"zhangyunzhu"**);  coll.add(**"AA"**);  coll.add(234); *//自动装箱* coll.add(**new** Date());  *//size()方法：获取添加到coll中的元素的个数* System.***out***.println(coll.size()); *//4  //addAll(Collection coll1)方法：将coll1集合中的元素全部添加到coll中* Collection coll1 = **new** ArrayList();  coll1.add(**"yunzhu"**);  coll1.add(678);  coll.addAll(coll1);  System.***out***.println(coll.size()); *//6* System.***out***.println(**"==>coll是："** + coll); *//==>coll是：[zhangyunzhu, AA, 234, Wed Mar 18 20:54:25 CST 2020, lo, 678]* System.***out***.println(**"==>coll1是："** + coll1); *//==>coll1是：[yunzhu, 678]  //isEmpty(): 判断集合中是否有元素* System.***out***.println(coll.isEmpty()); *//false  //clear(): 清空集合元素* coll1.clear();  System.***out***.println(coll1.isEmpty()); *//true* }   @Test  **public void** test2() {  Collection coll = **new** ArrayList();  coll.add(123);  coll.add(**"zhangyunzhu"**);  coll.add(**new** String(**"JAVA"**));  coll.add(**false**);  coll.add(**new** Person(**"Tom"**, 18));  *//contains(Object obj): 判断集合中是否包含obj  //判断的时候，h会调用obj对象所在类的equals()方法* **boolean** contains = coll.contains(**"zhangyunzhu"**);  System.***out***.println(contains); *//true* System.***out***.println(coll.contains(**new** String(**"JAVA"**)));*//true(这里调的是equals()方法)* System.***out***.println(coll.contains(**new** Person(**"Tom"**, 18)));*//false(如果想让他也是true，自己需要在Person类里面重写一下equals()// 方法)    //containsAll(Collection coll): 判断coll集合中的元素是否都在当前(被调用的集合中)* Collection coll1 = Arrays.*asList*(123, 456);  Collection coll2 = Arrays.*asList*(123, **"zhangyunzhu"**);  System.***out***.println(coll.containsAll(coll1)); *//false* System.***out***.println(coll.containsAll(coll2)); *//true    //remove(Object obj)* System.***out***.println(coll.toString()); *//[123, zhangyunzhu, JAVA, false, Person{name='Tom', age=18}]* coll.remove(123);  System.***out***.println(coll.toString()); *//[zhangyunzhu, JAVA, false, Person{name='Tom', age=18}]    //removeAll(Collection coll): 将coll从当前集合中全部移除* coll.removeAll(Arrays.*asList*(123, **"zhangyunzhu"**));  System.***out***.println(coll.toString()); *//[JAVA, false, Person{name='Tom', age=18}]    //retainAll(Collection coll): 求两个集合的交集, 并返回给当前集合* Collection collection = Arrays.*asList*(123, 456, **"zhangyunzhu"**,**"JAVA"**);  System.***out***.println(coll); *//[JAVA, false, Person{name='Tom', age=18}]* coll.retainAll(collection);  System.***out***.println(**"==>containsAll后的coll："** + coll); *//==>containsAll后的coll：[JAVA]    //equals(Object obj): obj和当前集合是否一样（如果是ArrayList<>()，则需要满足顺序也相同，否则返回false）* }   @Test  **public void** test3() {  Collection coll = **new** ArrayList();  coll.add(123);  coll.add(**"zhangyunzhu"**);  coll.add(**new** String(**"JAVA"**));  coll.add(**false**);  coll.add(**new** Person(**"Tom"**, 18));  *//hashCode(): 返回当前对象的hash值(貌似是根据当前对象生成的一个随机数)* System.***out***.println(coll.hashCode()); *//-2090143225  //toArray(): 集合-->数组* Object[] array = coll.toArray();  **for** (**int** i = 0; i < array.**length**; i++) {  System.***out***.print(array[i] + **"，"**); *//123，zhangyunzhu，JAVA，false，Person{name='Tom', age=18}，* }  System.***out***.println();  *//数组-->集合：Arrays.asList(T t),  // 注意：这个方法会把T实别成一个元素，如果形参的为止是基本数据类型的数组，需要将其写成其包装类数组* List<String> asList = Arrays.*asList*(**new** String[]{**"AA"**, **"HH"**, **"yun"**});  System.***out***.println(asList); *//[AA, HH, yun]    //iterator()：返回Iterator接口的实例，写在下面的IteratorTest.java中* }   }  **class** Person{  **private** String **name**;  **private int age**;   **public** Person() {  }   **public** Person(String name, **int** age) {  **this**.**name** = name;  **this**.**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** String toString() {  **return "Person{"** +  **"name='"** + **name** + **'\''** +  **", age="** + **age** +  **'}'**;  } } |

## Iterator迭代器：

|  |
| --- |
|  |
| **package** collection;  **import** org.junit.Test;  **import** java.util.ArrayList; **import** java.util.Collection; **import** java.util.Iterator;  */\*\*  \** ***@description:*** *使用Iterator迭代器进行集合元素的遍历  \* 应用迭代器中的hasNext()方法和next()方法  \** ***@date:*** *2020/3/22 16:30  \** ***@author:*** *lizhenhong  \*/* **public class** IteratorTest {   @Test  **public void** test() {  Collection coll = **new** ArrayList();  coll.add(123);  coll.add(456);  coll.add(**new** Person(**"zhangyunzhu"**,18));  coll.add(**new** String(**"zhangyunzhu"**));  coll.add(**false**);  Iterator iterator = coll.iterator();  **while** (iterator.hasNext()) {  System.***out***.print(iterator.next()+**", "**);*//123, 456, Person{name='zhangyunzhu', age=18}, zhangyunzhu, false,* }  System.***out***.println();  }   } |
| 迭代器运行原理： |
| **迭代器常见的错误写法：** |
| @Test  **public void** test1() {  Collection coll = **new** ArrayList();  coll.add(123);  coll.add(456);  coll.add(**new** Person(**"zhangyunzhu"**,18));  coll.add(**new** String(**"zhangyunzhu"**));  coll.add(**false**);  Iterator iterator = coll.iterator();  *//错误写法一，跳跃输出，并且报错java.util.NoSuchElementException // while (iterator.next() != null) { // System.out.println(iterator.next()); // }    //错误写法二：会导致无限循环输出第一个元素，* **while** (coll.iterator().hasNext()) {  System.***out***.println(coll.iterator().next());  }      } |
| **迭代器中remove()方法的使用：** |
| *//remove()移除* @Test **public void** test2() {  Collection coll = **new** ArrayList();  coll.add(123);  coll.add(456);  coll.add(**new** Person(**"zhangyunzhu"**,18));  coll.add(**new** String(**"zhangyunzhu"**));  coll.add(**false**);  Iterator iterator = coll.iterator();  System.***out***.println(**"移除前的coll："** + coll.toString());*//移除前的coll：[123, 456, Person{name='zhangyunzhu', age=18}, zhangyunzhu, false]* **while** (iterator.hasNext()) {  Object o = iterator.next();  **if** (**"zhangyunzhu"**.equals(o)) {  iterator.remove();  }  }  *//这里必须重新声明一个，因为上面的那个iterator的指针已经到了最后了* Iterator iterator1 = coll.iterator();  System.***out***.println(**"移除后的coll:"**);  **while** (iterator1.hasNext()) {  System.***out***.print(iterator1.next() + **", "**); *//123, 456, Person{name='zhangyunzhu', age=18}, false,* }  System.***out***.println(); } |

## Collection的子接口

### List接口

|  |
| --- |
| ArrayList, LinkedList, Vector的异同： |

### ArrayList源码分析

|  |
| --- |
|  |
| **jdk1.7及以前：**  构造器：  当new ArrayList()的时候，相当于创建了一个Object[10]，      添加数据：    如果当前数组长度不够，则扩容：    扩容为原来的1.5倍： |
| **jdk8**：  构造器：  当new ArrayList()的时候，：  底层的Object[] elementData初始化为{}，并没有创建    add()： |

### LinkedList()源码分析：

|  |
| --- |
| LinkedList list = new LinkedList();  这里就是声名了一个对象，内部声明了Node类型的first和last属性默认值为null    Node定义为：(体现了LinkedList的双向链表的说法)    add()方法： |

### List接口中的常用方法

|  |
| --- |
|  |
| **package** collection;  **import** org.junit.Test;  **import** java.util.ArrayList; **import** java.util.Arrays; **import** java.util.List;  */\*\*  \** ***@description:*** *Collection的子接口之List接口  \** ***@date:*** *2020/3/28 16:05  \** ***@author:*** *lizhenhong  \*/* **public class** ListTest {    @Test  **public void** test1(){  ArrayList list = **new** ArrayList();  list.add(123);  list.add(**"zhangyunzhu"**);  list.add(12);  list.add(**new** Person(**"Tom"**, 11));  list.add(**false**);  System.***out***.println(list); *//[123, zhangyunzhu, 12, Person{name='Tom', age=11}, false]    //void add(int index, Object obj) : 在index位置插入obj元素* list.add(1, 3333);  System.***out***.println(list); *//[123, 3333, zhangyunzhu, 12, Person{name='Tom', age=11}, false]    //boolean addAll(int index, Collection coll) : 在index位置开始将coll中的所有元素添加到list中去* List<Integer> list1 = Arrays.*asList*(1, 2, 3, 4, 5);  list.addAll(1, list1);  System.***out***.println(list); *//[123, 1, 2, 3, 4, 5, 3333, zhangyunzhu, 12, Person{name='Tom', age=11}, false]* System.***out***.println(list.size()); *// 11* }    @Test  **public void** test2(){  ArrayList list = **new** ArrayList();  list.add(123);  list.add(**"zhangyunzhu"**);  list.add(12);  list.add(**new** Person(**"Tom"**, 11));  list.add(**false**);  System.***out***.println(list); *//[123, zhangyunzhu, 12, Person{name='Tom', age=11}, false]    //int indexOf(Object obj) : 返回obj元素在当前集合中首次出现的index位置(没有返回-1)* **int** index = list.indexOf(**"zhangyunzhu"**);  System.***out***.println(index); *//1* **int** index1 = list.indexOf(1111);  System.***out***.println(index1); *//-1    //int lastIndexOf(Object obj): 返回obj元素在当前集合中最后一次出现的index位置(没有返回-1)* **int** zhangyunzhu = list.lastIndexOf(**"zhangyunzhu"**);  System.***out***.println(zhangyunzhu); *//1    //Object remove(int index): 移除指定index位置的元素，并返回此元素* Object remove = list.remove(0);  System.***out***.println(remove); *//123* System.***out***.println(list); *//[zhangyunzhu, 12, Person{name='Tom', age=11}, false]    //Object set(int index, Object obj): 设置index位置的元素为obj，并返回设置之前的元素* Object shezhi = list.set(1, **"shezhi"**);  System.***out***.println(shezhi); *//12* System.***out***.println(list); *//[zhangyunzhu, shezhi, Person{name='Tom', age=11}, false]    //List subList(int fromIndex, int toIndex): 返回从fromIndex到toIndex位置左闭右开区间的子集合* List subList = list.subList(0, 2);  System.***out***.println(subList); *//[zhangyunzhu, shezhi]* System.***out***.println(list); *//[zhangyunzhu, shezhi, Person{name='Tom', age=11}, false]* } } |
| **常用方法：** |

### Collection子接口之-Set接口

|  |
| --- |
| HashSet是无续的、不可重复的，并不代表顺序是随机的  无序性：不等于随机性。存储的数据在底层数组种并不是按照数组索引的顺序添加，而是根据数据的哈希值进行添加 的；  不可重复性：保证添加的元素根据equals()判断时，不能返回true，即：相同的元素只能添加一个。 |

## map接口

|  |
| --- |
|  |
|  |

### HashMap的底层实现原理：

|  |
| --- |
|  |
|  |
|  |

### **map的常用方法**

|  |
| --- |
|  |
| **package** map;  **import** org.junit.Test;  **import** java.util.HashMap; **import** java.util.Map;  */\*\*  \** ***@description:*** *Map的常用方法  \** ***@date:*** *2020/4/3 15:55  \** ***@author:*** *lizhenhong  \*/* **public class** MapTest {   @Test  **public void** test1() {  Map<String, Object> map = **new** HashMap<>();  map.put(**"lizhenhong"**, **"LIZHENHONG"**);  map.put(**"zhangyunzhu"**, **"ZHANGYUNZHU"**);  map.put(**"123"**, **"abc"**);  map.put(**"456"**, **"def"**);  System.***out***.println(map); *//{zhangyunzhu=ZHANGYUNZHU, 123=abc, 456=def, lizhenhong=LIZHENHONG}    //putAll(Map map):将map中的全部元素放到新的map中* Map<String, Object> map1 = **new** HashMap<>();  map1.putAll(map);  System.***out***.println(map1); *//{lizhenhong=LIZHENHONG, zhangyunzhu=ZHANGYUNZHU, 123=abc, 456=def}    //remove(Object key): 按照指定的key将元素移除，返回的是要移除的value* Object remove = map.remove(**"123"**);  System.***out***.println(remove); *//abc* System.***out***.println(map); *//{zhangyunzhu=ZHANGYUNZHU, 456=def, lizhenhong=LIZHENHONG}    //clear(): 清空map数据(但是map对象仍然存在)* map.clear();  System.***out***.println(map.size()); *//0* System.***out***.println(map); *//{}* }   @Test  **public void** test2() {  Map<String, Object> map = **new** HashMap<>();  map.put(**"lizhenhong"**, **"LIZHENHONG"**);  map.put(**"zhangyunzhu"**, **"ZHANGYUNZHU"**);  map.put(**"123"**, **"abc"**);  map.put(**"456"**, **"def"**);    *//get(Object key): 获取key对应的value* Object zhangyunzhu = map.get(**"zhangyunzhu"**);  System.***out***.println(zhangyunzhu); *//ZHANGYUNZHU    //containsKey(Object key): 判断map中是否包含指定的key* **boolean** lizhenhong = map.containsKey(**"lizhenhong"**);  System.***out***.println(lizhenhong); *//true   //containsValue(Object value): 判断map中是否包含指定的value* **boolean** lizhenhong1 = map.containsValue(**"lizhenhong"**);  System.***out***.println(lizhenhong1); *//false    //int size():map的长度* System.***out***.println(map.size()); *//4   //boolean isEmpty():判断当前map是否为空* System.***out***.println(map.isEmpty()); *//false* map.clear();  System.***out***.println(map.isEmpty()); *//true   //boolean equals(Object obj): 判断当前的map和obj是否相等（略）* }   } |

### map的遍历

|  |
| --- |
| **package** map;  **import** org.junit.Test;  **import** java.util.\*;  */\*\*  \** ***@description:*** *map的遍历  \** ***@date:*** *2020/4/4 17:06  \** ***@author:*** *lizhenhong  \*/* **public class** MapForeachTest {   @Test  **public void** test1() {  Map<String, Object> map = **new** HashMap<>();  map.put(**"lizhenhong"**, **"LIZHENHONG"**);  map.put(**"zhangyunzhu"**, **"ZHANGYUNZHU"**);  map.put(**"123"**, **"abc"**);  map.put(**"456"**, **"def"**);   *//遍历所有的key集合:keySet()* Set<String> keySet = map.keySet();  Iterator<String> iterator = keySet.iterator();  **while** (iterator.hasNext()) {  System.***out***.print(iterator.next() + **","**); *//zhangyunzhu,123,456,lizhenhong,* }  System.***out***.println();    *//遍历所有的value集合：values()* Collection<Object> values = map.values();  **for** (Object value : values) {  System.***out***.print(value+**","**); *//ZHANGYUNZHU,abc,def,LIZHENHONG,* }  System.***out***.println();    *//遍历所有的key-value: entrySet()* Set<Map.Entry<String, Object>> entrySet = map.entrySet();  Iterator<Map.Entry<String, Object>> entry = entrySet.iterator();  **while** (entry.hasNext()) {  Map.Entry<String, Object> next = entry.next();  System.***out***.print(next+**","**); *//zhangyunzhu=ZHANGYUNZHU,123=abc,456=def,lizhenhong=LIZHENHONG,* }   }   } |

## Collections工具类

|  |
| --- |
| **另外，Collections中还提供了synchronizedXXX(XXX)方法，可以将原来线程不安全的ArrayList、HashMap等转换成线程安全的(看源码是给加了synchronized关键字)。** |
| **package** collections;  **import** org.junit.Test;  **import** java.util.ArrayList; **import** java.util.Arrays; **import** java.util.Collections; **import** java.util.List;  */\*\*  \** ***@description:*** *Collections工具类的常用方法  \** ***@date:*** *2020/4/4 18:25  \** ***@author:*** *lizhenhong  \*/* **public class** CollectionsTest {   @Test  **public void** test1() {  List<Integer> list = **new** ArrayList<>();  list.add(12);  list.add(23);  list.add(12);  list.add(5435);  list.add(-33);  list.add(0);  System.***out***.println(list); *//[12, 23, 12, 5435, -33, 0]  //reverse(List list): 将list中的元素反转* Collections.*reverse*(list);  System.***out***.println(list); *//[0, -33, 5435, 12, 23, 12]    //shuffle(List list): 将list进行随机排序* Collections.*shuffle*(list);  System.***out***.println(list); *//[5435, 23, 0, -33, 12, 12] (这个的话每次输出结果是不同的)    //sort(List list): 将list中的元素进行升序排序（从小到大）* Collections.*sort*(list);  System.***out***.println(list); *//[-33, 0, 12, 12, 23, 5435]    //swap(List list, int index1, int index2): 将list的index1和index2位置的元素进行交换* Collections.*swap*(list, 1, 2);  System.***out***.println(list); *//[-33, 12, 0, 12, 23, 5435]    //int frequency(Collection coll, Object obj): coll集合中obj出现的频率* **int** frequency = Collections.*frequency*(list, 12);  System.***out***.println(frequency); *//2* }   @Test  **public void** test2() {  List<Integer> list = **new** ArrayList<>();  list.add(12);  list.add(23);  list.add(12);  list.add(5435);  list.add(-33);  list.add(0);    *//copy(List dest, List src): 将src中的元素赋值到dest中* List<Integer> dest = Arrays.*asList*(**new** Integer[list.size()]);  System.***out***.println(dest.size()); *//6* Collections.*copy*(dest, list);  System.***out***.println(dest); *//[12, 23, 12, 5435, -33, 0]   //synchronizedXXX(XXX): 将list，map等转换成线程安全的，既可操作Collection，又可操作map*  *//比如下面返回的integerList就是线程安全的。* List<Integer> integerList = Collections.*synchronizedList*(list);    }   } |

## File类的使用

|  |
| --- |
|  |
| **常用构造器(实例化)** |
| **package** io;  **import** org.junit.Test;  **import** java.io.File;  */\*\*  \** ***@description:*** *IO流之File类的使用  \** ***@date:*** *2020/4/6 15:13  \** ***@author:*** *lizhenhong  \*/* **public class** FileTest {   *//实例化(此时的各个file只是内存中的几个对象，还并没有涉及到硬盘)* @Test  **public void** test1() {  *//1. File(String pathName):* File file1 = **new** File(**"hello.txt"**); *//相对路径* File file2 = **new** File(**"D:\\study\\day25\_泛型与File\\hello.txt"**); *//绝对路径* System.***out***.println(file1); *//hello.txt* System.***out***.println(file2); *//D:\study\day25\_泛型与File\hello.txt   //File(String parentPath, String childPath): (parentPath:上一层目录, chilePath: 本层目录)* File file3 = **new** File(**"D:\\study\\day25\_泛型与File"**, **"hello.txt"**);  System.***out***.println(file3); *//D:\study\day25\_泛型与File\hello.txt    //File(File file, String childPath):* File file4 = **new** File(file2, **"hello.txt"**);  System.***out***.println(file4); *//D:\study\day25\_泛型与File\hello.txt\hello.txt* } } |
| **常用方法：**    @Test **public void** test2() {  File file1 = **new** File(**"hello.txt"**);  File file2 = **new** File(**"D:\\study\\day25\_泛型与File"**);    *//获取绝对路径* System.***out***.println(file1.getAbsolutePath()); *//D:\work\work\shangxuetang\duoxiancheng\hello.txt  //获取路径* System.***out***.println(file1.getPath()); *//hello.txt  //获取名称* System.***out***.println(file1.getName()); *//hello.txt  //获取上层文件目录路径。如果没有则返回null* System.***out***.println(file1.getParent()); *//null  //获取文件长度(字节数)，不能获取目录的长度* System.***out***.println(file1.length()); *//0  //最后一次修改时间(毫秒值)* System.***out***.println(file1.lastModified()); *//0* System.***out***.println(**"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"**);  System.***out***.println(file2.getAbsolutePath()); *//D:\study\day25\_泛型与File* System.***out***.println(file2.getPath()); *//D:\study\day25\_泛型与File* System.***out***.println(file2.getName()); *//day25\_泛型与File* System.***out***.println(file2.getParent()); *//D:\study* System.***out***.println(file2.length()); *//4096* System.***out***.println(file2.lastModified()); *//1579306381776   //list(): 返回指定文件目录下的文件的文件名String类型数组(要求是这个路径必须存在，否则空指针)* File file3 = **new** File(**"D:\\work\\work\\shangxuetang"**);  String[] fileList = file3.list();  **for** (String s : fileList) {  System.***out***.print(s + **","**); *//.idea,duoxiancheng,out,shangxuetang.iml,src,* }  System.***out***.println();    *//listFiles(): 以文件的绝对路径返回文件数组* File[] files = file3.listFiles();  **for** (File file : files) {  System.***out***.print(file + **", "**); *//D:\work\work\shangxuetang\.idea, D:\work\work\shangxuetang\duoxiancheng, D:\work\work\shangxuetang\out, D:\work\work\shangxuetang\shangxuetang.iml, D:\work\work\shangxuetang\src,* }  System.***out***.println(); } |
| 返回true之后，当前路径的”hello.txt”文件消失，在目标路径出现hi.txt文件    @Test **public void** test3() {  *//renameTo(File file): 把文件重命名为指定的文件路径(要想保证返回true，必须保证file1在硬盘中存在，file2不能在硬盘中存在)* File file1 = **new** File(**"hello.txt"**);  File file2 = **new** File(**"D:\\study\\day25\_泛型与File\\hi.txt"**);  **boolean** renameTo = file2.renameTo(file1);  System.***out***.println(renameTo); *//true* } |
| @Test **public void** test4() {  File file1 = **new** File(**"hello.txt"**);  System.***out***.println(file1.isDirectory()); *//false* System.***out***.println(file1.isFile()); *//true* System.***out***.println(file1.exists()); *//true* System.***out***.println(file1.canRead()); *//true* System.***out***.println(file1.canWrite()); *//true* System.***out***.println(file1.isHidden()); *//false* } |
|  |

## IO流

|  |
| --- |
|  |
| **流的分类：**        **规律：**所有的API都是以基类作为结尾的  **重点关注：**上面的表格带深色背景的。  **结论：**   1. 对于文本文件(.txt, .java, .c, .cpp等)，使用字符流处理即可； 2. 对于非文本文件(.jpg, .mp3, .mp4, .avi, .doc, .ppt等)，需要使用字节流来处理。 |

### FileReader读数据操作

|  |
| --- |
| **将本model下面的hello.txt文件中的内容读取到程序中，并打印再控制台上**  hello.txt中的内容：    */\*\*  \* 将本model下面的hello.txt文件中的内容读取到程序中，并打印再控制台上  \* 说明：为了保证流资源一定执行关闭操作，要进行try/catch/finally操作  \* 要求读入的文件一定要存在，否则会抛FileNotFoundException  \** ***@throws*** *IOException  \*/* @Test **public void** test() **throws** IOException {  FileReader fr = **null**;  **try** {  *//1. 实例化File对象并指明要操作的文件* File file = **new** File(**"hello.txt"**); *//要求读入的文件一定要存在，否则会抛FileNotFoundException  //2. 提供具体的流* fr = **new** FileReader(file);  *//3. 数据的读入  //read():返回读入的一个字符，如果达到了文件的末尾，则返回-1* **int** read = fr.read();  **while** (read != -1) {  System.***out***.print((**char**) read);  read = fr.read();  }  } **catch** (IOException e) {  e.printStackTrace();  } **finally** {  *//4. 关闭流(这个一定要有)* **if** (fr != **null**) {  fr.close();  }  } }  控制台： |
| */\*\*  \* 对上面的read()方法的优化：用read(char[] cbuf)  \*/* @Test  **public void** test1(){  *//1. 实例化File对象* FileReader fr = **null**;  **try** {  File file = **new** File(**"hello.txt"**);  *//2. 实例化具体的流* fr = **new** FileReader(file);  *//3. 读入的操作* **char**[] cbuf = **new char**[4];  **int** len;  **while** ((len = fr.read(cbuf)) != -1) {  *//方式一： // for (int i = 0; i < len; i++) { // System.out.print(cbuf[i]); //lizhenhong // }  //错误的写法(原因：这里的数组是赋值操作，当最后的字符少于len时，会有没赋值的元素被输出) // for (int i = 0; i < cbuf.length; i++) { // System.out.print(cbuf[i]); //lizhenhongho // }  //方式二:用String()构造器* String str = **new** String(cbuf, 0, len);  System.***out***.print(str); *//lizhenhong  //错误的写法(原因和方式一的错误写法类似) // String str = new String(cbuf); // System.out.print(str); //lizhenhongho* }  } **catch** (IOException e) {  e.printStackTrace();  } **finally** {  *//4. 关闭流* **try** {  **if** (fr != **null**) {  fr.close();  }  } **catch** (IOException e) {  e.printStackTrace();  }  }  } |

### FileWriter写数据的操作

|  |
| --- |
| */\*\*  \* 从内存中向硬盘的文件中写数据  \* File对应的硬盘中的文件如果不存在，在输出的过程中会自动创建文件；  \* File对应的硬盘中的文件如果存在：  \* 如果流使用的构造器是FileWriter(file, false)/Filewriter(file),会对原有文件进行覆盖；  \* 如果流使用的构造器是FileWriter(file, true)，不会对原有文件进行覆盖，会在原有内容后面追加。  \*/* @Test **public void** testFileWriter() {  FileWriter fw = **null**;  **try** {  *//1. 提供File类的对象(这里如果指明的文件不存在的话，程序会在相应的位置新建一个)* File file = **new** File(**"hello1.txt"**);  *//2. 提供FileWriter对象，用于写数据* fw = **new** FileWriter(file);  *//3. 写数据的操作* fw.write(**"my name is lizhenhong!"**);  fw.write(**"hello!"**);  } **catch** (IOException e) {  e.printStackTrace();  } **finally** {  *//4. 关闭流的操作* **try** {  **if** (fw != **null**) {  fw.close();  }  } **catch** (IOException e) {  e.printStackTrace();  }  } }  **效果：** |

### 使用FileReader和FileWriter实现文本文件的复制

|  |
| --- |
| 开始时的hello.txt    开始时的hello1.txt    **代码：**  */\*\*  \* 使用FileReader和FileWriter实现文本文件的复制  \*/* @Test **public void** testReaderAndWriter() {  FileReader fr = **null**;  FileWriter fw = **null**;  **try** {  *//1. 创建File类的对象，指明读入文件和写出文件* File srcFile = **new** File(**"hello.txt"**);  File destFile = **new** File(**"hello1.txt"**);  *//2. 创建输入流和输出流的对象* fr = **new** FileReader(srcFile);  fw = **new** FileWriter(destFile);  *//3. 数据的读入和写出操作* **char**[] cbuf = **new char**[5];  **int** len; *//记录每次读入的数组中的字符的个数* **while** ((len = fr.read(cbuf)) != -1) {  *//每次写出len个字符* fw.write(cbuf, 0, len);  }  } **catch** (IOException e) {  e.printStackTrace();  } **finally** {  *//4. 关闭流的操作* **if** (fr != **null**) {  **try** {  fw.close();  } **catch** (IOException e) {  e.printStackTrace();  }  }  **if** (fw != **null**) {  **try** {  fr.close();  } **catch** (IOException e) {  e.printStackTrace();  }  }  } }  效果： |

### FileInputStream和FileOutputStream读写非文本文件

|  |
| --- |
| */\*\*  \* 实现对图片的复制操作  \* 读取本model下面的”截屏.jpg”并写入同目录的”截屏1.jpg”  \*/* @Test **public void** test() {  FileInputStream fis = **null**;  FileOutputStream fos = **null**;  **try** {  File srcFile = **new** File(**"截屏.jpg"**);  File destFile = **new** File(**"截屏1.jpg"**);  fis = **new** FileInputStream(srcFile);  fos = **new** FileOutputStream(destFile);  **byte**[] buffer = **new byte**[6];  **int** len;  **while** ((len = fis.read(buffer)) != -1) {  fos.write(buffer, 0, len);  }  System.***out***.println(**"图片复制成功！"**);  } **catch** (IOException e) {  e.printStackTrace();  } **finally** {  **if** (fis != **null**) {  **try** {  fis.close();  } **catch** (IOException e) {  e.printStackTrace();  }  }  **if** (fos != **null**) {  **try** {  fos.close();  } **catch** (IOException e) {  e.printStackTrace();  }  }  } } |

### 缓冲流(字节型)实现非文本文件的复制

**说明：**

**引用缓冲流的作用：**提高流的读取与写入的速度。 原因：缓冲流内部提供了一个缓冲区。

|  |
| --- |
| */\*\*  \* 用缓冲流实现图片的复制  \*/* @Test  **public void** bufferedStreamTest() {  BufferedInputStream bis = **null**;  BufferedOutputStream bos = **null**;  **try** {  *//1. 造文件* File srcFile = **new** File(**"截屏.jpg"**);  File destFile = **new** File(**"截屏2.jpg"**);  *//2. 造流的对象  //2.1 首先要造节点流* FileInputStream fis = **new** FileInputStream(srcFile);  FileOutputStream fos = **new** FileOutputStream(destFile);  *//2.2 造缓冲流* bis = **new** BufferedInputStream(fis);  bos = **new** BufferedOutputStream(fos);  *//3. 复制* **byte**[] buffer = **new byte**[5];  **int** len;  **while** ((len = bis.read(buffer)) != -1) {  bos.write(buffer, 0, len);  }  } **catch** (IOException e) {  e.printStackTrace();  } **finally** {  *//4. 关闭流操作(要求先关闭外层的流，再关闭内层的流)  //说明：当外层的流关闭的时候，实际上程序会将其内层的流也关闭* **if** (bos != **null**) {  **try** {  bos.close();  } **catch** (IOException e) {  e.printStackTrace();  }  }  **if** (bis != **null**) {  **try** {  bis.close();  } **catch** (IOException e) {  e.printStackTrace();  }  } *// fos.close(); // fis.close();* }  }  运行结果： |

### 对象流 && 序列化机制

|  |
| --- |
| **对象的序列化机制：**    **要满足对象是可序列化的，需要满足的要求：**   * + - 1. 该类必须实现如下两个接口之一：  1. Serializable（用的比较多） 2. Externalizable    * + 1. 需要当前类提供一个全局常量serialVersionUID；        2. 保证该类中所有的属性都是可序列化的（基本数据类型默认是可序列化的）；        3. 备注：被static或者transient关键字所修饰的属性是不能被序列化的； |

## NIO

|  |
| --- |
|  |

## 网络编程

### 概述

|  |
| --- |
| **IP：**    **端口号：** |

### 网络协议

|  |
| --- |
| **建立连接：**    **关闭连接：** |

### TCP网络编程

|  |
| --- |
| **package** tcp;  **import** org.junit.Test;  **import** java.io.ByteArrayOutputStream; **import** java.io.IOException; **import** java.io.InputStream; **import** java.io.OutputStream; **import** java.net.InetAddress; **import** java.net.ServerSocket; **import** java.net.Socket;  */\*\*  \** ***@author:*** *lizhenhong  \** ***@description: 网络编程之TCP协议，将客户端的请求打印到控制台*** *\** ***@date:*** *2020/4/19 16:05  \*/* **public class** TcpTest {   */\*\*  \* 客户端(后启动)  \*/* @Test  **public void** testCliect(){  Socket socket = **null**;  OutputStream stream = **null**;  **try** {  *//1. 创建socket对象，指明服务器端的ip和端口号* InetAddress address = InetAddress.*getByName*(**"127.0.0.1"**);  socket = **new** Socket(address, 8899);  *//2. 获取输出流用于输出数据* stream = socket.getOutputStream();  *//3. 写数据* stream.write(**"lizhenhong"**.getBytes());  } **catch** (IOException e) {  e.printStackTrace();  } **finally** {  *//4. 关闭outputStream && Socket资源* **if** (stream != **null**) {  **try** {  stream.close();  } **catch** (IOException e) {  e.printStackTrace();  }  }  **if** (socket != **null**) {  **try** {  socket.close();  } **catch** (IOException e) {  e.printStackTrace();  }  }  }   }  */\*\*  \* 服务端(要先启动)  \*/* @Test  **public void** testServer() {   ServerSocket serverSocket = **null**;  Socket socket = **null**;  InputStream inputStream = **null**;  ByteArrayOutputStream baos = **null**;  **try** {  *//1. 创建ServerSocket对象，指明自己的端口号* serverSocket = **new** ServerSocket(8899);  *//2. 接收来自客户端的请求* socket = serverSocket.accept();  *//3. 获取输入流* inputStream = socket.getInputStream();  *//显示到控制台  //不建议这么写，因为在utf8中，汉字占三个字节 // byte[] buffer = new byte[20]; // int len; // if ((len = inputStream.read(buffer)) != -1) { // String str = new String(buffer, 0, len); // System.out.print(str); // }  //可以用下面的这个类来做  //4. 读取输入流中的数据* baos = **new** ByteArrayOutputStream();  **byte**[] buffer = **new byte**[5];  **int** len;  **while** ((len = inputStream.read(buffer)) != -1) {  *//这样写的话，baos会将所有字符进行拼接，暂时写到ByteArrayOutputStream类中，* baos.write(buffer, 0, len);  }  System.***out***.print(baos.toString()); *//lizhenhong* System.***out***.println();  System.***out***.println(**"收到了来自于："** + socket.getInetAddress().getHostAddress() + **"的数据"**); *//收到了来自于：127.0.0.1的数据* } **catch** (IOException e) {  e.printStackTrace();  } **finally** {  *//5. 关闭资源* **if** (baos != **null**) {  **try** {  baos.close();  } **catch** (IOException e) {  e.printStackTrace();  }  }  **if** (inputStream != **null**) {  **try** {  inputStream.close();  } **catch** (IOException e) {  e.printStackTrace();  }  }  **if** (socket != **null**) {  **try** {  socket.close();  } **catch** (IOException e) {  e.printStackTrace();  }  }  **if** (serverSocket != **null**) {  **try** {  serverSocket.close();  } **catch** (IOException e) {  e.printStackTrace();  }  }  }  }  } |
| **从客户端发送文件给服务端，服务端保存到本地，然后返回"保存成功"给客户端，并关闭相应的连接**  **package** tcp;  **import** org.junit.Test;  **import** java.io.\*; **import** java.net.InetAddress; **import** java.net.ServerSocket; **import** java.net.Socket; **import** java.net.UnknownHostException;  */\*\*  \** ***@author:*** *lizhenhong  \** ***@description:*** *从客户端发送文件给服务端，服务端保存到本地，然后返回"保存成功"给客户端，并关闭相应的连接  \** ***@date:*** *2020/4/19 16:48  \*/* **public class** TcpTest2 {   */\*\*  \* 客户端  \*/* @Test  **public void** testCliect() {  Socket socket = **null**;  OutputStream outputStream = **null**;  FileInputStream fis = **null**;  InputStream is = **null**;  ByteArrayOutputStream baos = **null**;  **try** {  socket = **new** Socket(InetAddress.*getByName*(**"127.0.0.1"**), 9090);  outputStream = socket.getOutputStream();  fis = **new** FileInputStream(**"截屏.jpg"**);  **byte**[] buffer = **new byte**[1024];  **int** len;  *//这里的.read()方法是一个线程阻塞的方法，需要给它一个明确的指示才能停止，否则会一直在这个循环里* **while** ((len = fis.read(buffer)) != -1) {  outputStream.write(buffer, 0, len);  }  *//图片文件传输完成后，关闭图片的输出* socket.shutdownOutput();   *//客户端接收服务端你的反馈并显示到控制台上* is = socket.getInputStream();  baos = **new** ByteArrayOutputStream();  **byte**[] bytes = **new byte**[24];  **int** len1;  **while** ((len1 = is.read(bytes)) != -1) {  baos.write(bytes, 0, len1);  }  System.***out***.print(baos.toString()); *//图片传输完成* } **catch** (IOException e) {  e.printStackTrace();  } **finally** {  *//关闭资源* **if** (baos != **null**) {  **try** {  baos.close();  } **catch** (IOException e) {  e.printStackTrace();  }  }  **if** (is != **null**) {  **try** {  is.close();  } **catch** (IOException e) {  e.printStackTrace();  }  }  **if** (fis != **null**) {  **try** {  fis.close();  } **catch** (IOException e) {  e.printStackTrace();  }  }  **if** (outputStream != **null**) {  **try** {  outputStream.close();  } **catch** (IOException e) {  e.printStackTrace();  }  }  **if** (socket != **null**) {  **try** {  socket.close();  } **catch** (IOException e) {  e.printStackTrace();  }  }   }   }   */\*\*  \* 服务器端  \*/* @Test  **public void** testServer() {  ServerSocket serverSocket = **null**;  Socket socket = **null**;  OutputStream os = **null**;  InputStream is = **null**;  FileOutputStream fos = **null**;  **try** {  serverSocket = **new** ServerSocket(9090);  socket = serverSocket.accept();  is = socket.getInputStream();  fos = **new** FileOutputStream(**"截屏3.jpg"**);  **byte**[] buffer = **new byte**[1024];  **int** len;  **while** ((len = is.read(buffer)) != -1) {  fos.write(buffer, 0, len);  }  System.***out***.println(**"服务端图片传输完成"**); *//服务端图片传输完成  //服务端给予客户端反馈* os = socket.getOutputStream();  os.write(**"图片传输完成"**.getBytes());  } **catch** (IOException e) {  e.printStackTrace();  } **finally** {  **if** (fos != **null**) {  **try** {  fos.close();  } **catch** (IOException e) {  e.printStackTrace();  }  }  **if** (is != **null**) {  **try** {  is.close();  } **catch** (IOException e) {  e.printStackTrace();  }  }  **if** (os != **null**) {  **try** {  os.close();  } **catch** (IOException e) {  e.printStackTrace();  }  }  **if** (socket != **null**) {  **try** {  socket.close();  } **catch** (IOException e) {  e.printStackTrace();  }  }  **if** (serverSocket != **null**) {  **try** {  serverSocket.close();  } **catch** (IOException e) {  e.printStackTrace();  }  }  }  } } |

### UDP网络编程

|  |
| --- |
|  |

## 反射

### 概述

|  |
| --- |
|  |
| **静态语言 vs 动态语言** |
| **反射机制的作用：** |
| **反射中用到的主要API:** |

利用反射进行对象的操作

|  |
| --- |
| package reflection;  import org.junit.Test;  import java.lang.reflect.Constructor; import java.lang.reflect.Field; import java.lang.reflect.Method;  */\*\*  \* @author: lizhenhong  \* @description: java反射机制  \* @date: 2020/4/19 20:19  \*/* public class ReflectionTest {   */\*\*  \* 不用反射的方式  \*/* @Test  public void testNormal() {  Person person = new Person("zhangyunzhu", 23);  System.*out*.println(person.toString()); *//Person{name='zhangyunzhu', age=23}* person.age = 18;  System.*out*.println(person.toString()); *//Person{name='zhangyunzhu', age=18}* person.show(); *//this is a person class* }   */\*\*  \* 使用反射  \* @throws Exception  \*/* @Test  public void testReflection() throws Exception {  Class personClass = Person.class;  *//1. 通过反射，创建Person类的对象* Constructor constructor = personClass.getConstructor(String.class, int.class);  Person zhangyunzhu = (Person) constructor.newInstance("zhangyunzhu", 23);  System.*out*.println(zhangyunzhu.toString()); *//Person{name='zhangyunzhu', age=23}   //2. 通过反射，调用对象指定的属性、方法  //2.1 调用属性* Field age = personClass.getDeclaredField("age"); *//去到Person中的age属性* age.set(zhangyunzhu, 18); *//将age属性设置为18* System.*out*.println(zhangyunzhu.toString()); *//Person{name='zhangyunzhu', age=18}  //2.2 调用方法* Method show = personClass.getDeclaredMethod("show"); *//调用空参的show方法* show.invoke(zhangyunzhu);  *//这里输出的是: this is a person class* }  } |
| 调用私有构造：  */\*\*  \* 通过反射，调用类内部的私有结构，如：构造器、属性、方法  \*/* @Test public void testReflection1() throws Exception{  Class<Person> personClass = Person.class;  *//1. 调用私有的构造器* Constructor<Person> constructor = personClass.getDeclaredConstructor(String.class);  constructor.setAccessible(true);  Person person = constructor.newInstance("Tom");*//private Person(String name)* System.*out*.println(person); *//Person{name='person', age=0}  //2. 调用私有属性* Field name = personClass.getDeclaredField("name");  name.setAccessible(true);  name.set(person, "lizhenhong");  System.*out*.println(person); *//Person{name='lizhenhong', age=0}  //3. 调用私有方法* Method nation = personClass.getDeclaredMethod("showNation", String.class);  nation.setAccessible(true);  String china = (String) nation.invoke(person, "China");*//相当于person.showNation("中国")方法  //上面输出的是: 我的国籍是:China* System.*out*.println(china); *//China* } |

### Class类的理解

|  |
| --- |
|  |

### 获取Class实例的4种方式

|  |
| --- |
| */\*\*  \* Class实例的获取  \*/* @Test public void testForClass() throws ClassNotFoundException {  *//方式一：调用运行时类的属性: .class* Class<Person> personClass1 = Person.class;  System.*out*.println(personClass1); *//class reflection.Person   //方式二：通过运行时类的对象,调用getClass()方法* Person p1 = new Person();  Class<? extends Person> p1Class = p1.getClass();  System.*out*.println(p1Class); *//class reflection.Person   //方式三：通过调用Class的静态方法* Class<?> name = Class.*forName*("reflection.ReflectionTest");  System.*out*.println(name); *//class reflection.ReflectionTest   //方式四：使用类的加载器: ClassLoader* ClassLoader classLoader = ReflectionTest.class.getClassLoader();  Class<?> loadClass = classLoader.loadClass("reflection.ReflectionTest");  System.*out*.println(loadClass); *//class reflection.ReflectionTest* } |
|  |
| 类的加载过程： |
|  |
|  |
|  |
| */\*\*  \** ***@author*** *lizhenhong  \** ***@Description:*** *了解类的加载器  \** ***@date*** *2020/5/7 20:54  \*/*  @Test public void test() {   //对于自定义类，是使用系统类加载器进行加载  ClassLoader classLoader = ClassLoaderTest.class.getClassLoader();  System.*out*.println(classLoader); //sun.misc.Launcher$AppClassLoader@18b4aac2   //调用系统类加载器的getParent()方法，可以获取扩展类加载器  ClassLoader loaderParent = classLoader.getParent();  System.*out*.println(loaderParent); //sun.misc.Launcher$ExtClassLoader@a09ee92  //调用扩展类加载器的getParent()方法，是不能获取引导类加载器的，  // 引导类加载器主要加载Java核心类库，是无法加载自定义类的  ClassLoader classLoader1 = loaderParent.getParent();  System.*out*.println(classLoader1); //null } |

### 创建运行时类的对象

|  |
| --- |
| package reflection;  import org.junit.Test;  */\*\*  \** ***@author*** *lizhenhong  \** ***@Description:*** *通过反射创建对应的运行时类的对象  \** ***@date*** *2020/5/10 20:22  \*/* public class NewInstanceTest {   @Test  public void test() throws IllegalAccessException, InstantiationException {  Class<Person> personClass = Person.class;  */\*\*  \* 调用newInstance()方法，创建对应的运行时类的对象(本质上调用的时空参的构造器)  \* 要想此方法能正常的创建运行时类的对象，需要满足以下条件：  \* 1. 运行时类必须有空参构造器  \* 2. 空参构造器的访问权限足够被这里的方法访问到(通常设置成public)  \* 在javaBean中要求提供一个public的空参构造器，原因：  \* 1. 便于通过反射创建运行时类的对象  \* 2. 便于子类继承此运行时类时，默认调用super()时，保证父类有此构造器。  \*/* Person person = personClass.newInstance();  System.*out*.println(person); //Person{name='null', age=0}  }  } |

### 反射的动态性

|  |
| --- |
| @Test public void testDongtai() {  for (int i = 0; i < 100; i++) {  int num = new Random().nextInt(3); *//0, 1, 2* String classPath = "";  switch (num) {  case 0:  classPath = "java.util.Date";  break;  case 1:  classPath = "java.lang.Object";  break;  case 2:  classPath = "reflection.Person";  break;  }  try {  Object instance = getInstance(classPath);  *//每次都是随机输出的，输出的内容不同* System.*out*.println(instance);  } catch (Exception e) {  e.printStackTrace();  }  } }  */\*\*  \* 创建一个指定类的对象  \* @param classPath :指的是类的完全限定名  \*/* public Object getInstance(String classPath) throws Exception {  Class name = Class.*forName*(classPath);  Object newInstance = name.newInstance();  return newInstance; }  输出结果： |

### 获得运行时类内部结构

**先定义几个类、接口、注解**

|  |
| --- |
| package reflection.huodejiegou;  import java.io.Serializable;  */\*\*  \** ***@author*** *lizhenhong  \** ***@Description:*** *造一个Person1的父类  \** ***@date*** *2020/5/16 14:54  \*/* public class Creature<T> implements Serializable {   private char gender;  public double wight;   private void breath() {  System.*out*.println("animals all have to breath to stay alive");  }   public void eat() {  System.*out*.println("大家都需要吃东西");  }  } |
| package reflection.huodejiegou;  import java.lang.annotation.ElementType; import java.lang.annotation.Retention; import java.lang.annotation.RetentionPolicy; import java.lang.annotation.Target;  */\*\*  \** ***@author*** *lizhenhong  \** ***@Description: 建一个注解*** *\** ***@date*** *2020/5/16 15:07  \*/* @Target({ElementType.*TYPE*, ElementType.*FIELD*, ElementType.*METHOD*, ElementType.*PARAMETER*, ElementType.*CONSTRUCTOR*, ElementType.*LOCAL\_VARIABLE*}) @Retention(RetentionPolicy.*RUNTIME*) public @interface MyAnnotation {  String value() default "hello"; } |
| package reflection.huodejiegou;  */\*\*  \** ***@author*** *lizhenhong  \** ***@Description: 声明一个接口，里面有一个void抽象方法*** *\** ***@date*** *2020/5/16 14:58  \*/* public interface MyInterface {   void info();  } |
| package reflection.huodejiegou;  */\*\*  \** ***@author*** *lizhenhong  \** ***@Description:*** *被获得相关结构的运行时类  \** ***@date*** *2020/5/16 14:52  \*/* @MyAnnotation(value = "Person") public class Person extends Creature<String> implements Comparable<String>, MyInterface {   private String name;  int age;  public int id;   public Person() {  }   @MyAnnotation(value = "abc")  private Person(String name) {  this.name = name;  }   Person(String name, int age) {  this.name = name;  this.age = age;  }   @MyAnnotation  private String show(String nation) {  System.*out*.println("这里是Person1类的show()方法，入参:" + nation);  return nation;  }   public String display(String interest, int age) throws NullPointerException, ClassCastException {  return interest + age;  }   @Override  public int compareTo(String o) {  return 0;  }   @Override  public void info() {  System.*out*.println("这里是实现的MyInterface的info()方法");  } } |

#### 获得属性、属性的数据类型、修饰符

|  |
| --- |
| package reflection.huodejiegou;  import org.junit.Test;  import java.lang.reflect.Field; import java.lang.reflect.Modifier;  */\*\*  \** ***@author*** *lizhenhong  \** ***@Description:*** *通过反射获取当前运行时类的所有属性、以记属性的权限修饰符、数据类型  \** ***@date*** *2020/5/16 15:27  \*/* public class FieldTest {   @Test  public void test1() {  Class<Person> personClass = Person.class;  //获取属性结构  //1. getFields(): 获取当前运行时类及其父类中public的属性  Field[] fields = personClass.getFields();  //遍历获得的属性  for (Field field : fields) {  System.*out*.println(field);//public int reflection.huodejiegou.Person.id  //public double reflection.huodejiegou.Creature.wight  }   System.*out*.println("====================================");  //2. getDeclaredFields(): 获取当前运行时类中声明的所有属性(包括private的)(不包含父类中声明的属性)  Field[] declaredFields = personClass.getDeclaredFields();  for (Field declaredField : declaredFields) {  System.*out*.println(declaredField); //private java.lang.String reflection.huodejiegou.Person.name  //int reflection.huodejiegou.Person.age  //public int reflection.huodejiegou.Person.id  }  }   */\*\*  \* 获取具体类型的属性(包括具体的权限修饰符、数据类型、变量名)  \*/* @Test  public void test2() {  Class<Person> personClass = Person.class;  Field[] declaredFields = personClass.getDeclaredFields();  for (Field declaredField : declaredFields) {  //1. 权限修饰符(getModifiers(): 获取的是int类型数字，可以用Modifier.toString(modifier)方法转换成具体的修饰符)  int modifier = declaredField.getModifiers();  System.*out*.println(modifier); //2  //0  //1 // System.out.println("====================================");  String modifierString = Modifier.*toString*(modifier);  System.*out*.println(modifierString); //private  //  //public  System.*out*.println("====================================");  //2. 数据类型(getType(): 获得的是全限定名类的数据类型，可以加一个getName()方法)  Class type = declaredField.getType();  System.*out*.println(modifierString + "\t" + type.getName());//privateclass java.lang.String  //int  //public int   //3. 变量名(getName()方法：获得属性名)  String name = declaredField.getName();  System.*out*.println(name); //name  //age  //id  }  }  } |

#### 获取运行时类的方法结构

|  |
| --- |
| package reflection.huodejiegou;  import org.junit.Test;  import java.lang.reflect.Constructor; import java.lang.reflect.Method;  */\*\*  \** ***@author*** *lizhenhong  \** ***@Description:*** *获取运行时类的方法结构  \** ***@date*** *2020/5/16 16:34  \*/* public class MethodTest {   @Test  public void test1() {  Class<Person> personClass = Person.class;   //获取所有的方法(getMethods(): 获取当前运行时类及其所有父类的public的方法)  //也可以用getName()获得方法名称  Method[] methods = personClass.getMethods();  for (Method method : methods) {  */\*\*  \* public int reflection.huodejiegou.Person.compareTo(java.lang.String)  \* public int reflection.huodejiegou.Person.compareTo(java.lang.Object)  \* public void reflection.huodejiegou.Person.info()  \* public java.lang.String reflection.huodejiegou.Person.display(java.lang.String)  \* public void reflection.huodejiegou.Creature.eat()  \* public final void java.lang.Object.wait() throws java.lang.InterruptedException  \* public final void java.lang.Object.wait(long,int) throws java.lang.InterruptedException  \* public final native void java.lang.Object.wait(long) throws java.lang.InterruptedException  \* public boolean java.lang.Object.equals(java.lang.Object)  \* public java.lang.String java.lang.Object.toString()  \* public native int java.lang.Object.hashCode()  \* public final native java.lang.Class java.lang.Object.getClass()  \* public final native void java.lang.Object.notify()  \* public final native void java.lang.Object.notifyAll()  \*/* System.*out*.println(method); //太长了，写到上面了，下同  }   //getDeclaredMethods(): 获取当前运行时类中声明的所有方法(不包含父类中的)  //也可以用getName()获得方法名称  Method[] declaredMethods = personClass.getDeclaredMethods();  for (Method declaredMethod : declaredMethods) {  */\*\*  \* public int reflection.huodejiegou.Person.compareTo(java.lang.String)  \* public int reflection.huodejiegou.Person.compareTo(java.lang.Object)  \* public void reflection.huodejiegou.Person.info()  \* private java.lang.String reflection.huodejiegou.Person.show(java.lang.String)  \* public java.lang.String reflection.huodejiegou.Person.display(java.lang.String)  \*/* System.*out*.println(declaredMethod);  */\*\*  \* compareTo  \* compareTo  \* info  \* display  \* show  \*/* System.*out*.println(declaredMethod.getName());   }  //getConstructors(): 获取运行时类自己的构造器(public的)  Constructor<?>[] constructors = personClass.getConstructors();  for (Constructor<?> constructor : constructors) {  System.*out*.println(constructor); //public reflection.huodejiegou.Person()  }   //getDeclaredConstructors(): 获取当前运行时类自己的所有构造方法(包括private的)  Constructor<?>[] declaredConstructor = personClass.getDeclaredConstructors();  for (Constructor<?> constructor : declaredConstructor) {  */\*\*  \* reflection.huodejiegou.Person(java.lang.String,int)  \* private reflection.huodejiegou.Person(java.lang.String)  \* public reflection.huodejiegou.Person()  \*/* System.*out*.println(constructor);  }   }  } |

#### 获取运行时类的方法的内部结构(方法)

|  |
| --- |
| */\*\*  \* 获取方法内部的结构  \* @ 注解  \* 权限修饰符、返回值类型、方法名(参数列表) throws xxxException{}  \* 输出结果：  \* public int compareTo(java.lang.String args\_0)  \* public volatile int compareTo(java.lang.Object args\_0)  \* public void info()  \* public java.lang.String display(java.lang.String args\_0, int args\_1) throws java.lang.NullPointerException, java.lang.ClassCastException  \*  \** ***@reflection.huodejiegou.MyAnnotation(value=hello)*** *\* private java.lang.String show(java.lang.String args\_0)  \*/*  @Test  public void test2() {   Class<Person> personClass = Person.class;  Method[] declaredMethods = personClass.getDeclaredMethods();  for (Method method : declaredMethods) {  //1. 获取方法的注解(@Retention(RetentionPolicy.RUNTIME)的才能获取到，CLASS的和SOURCE的都不能)  Annotation[] annotations = method.getAnnotations();  for (Annotation annotation : annotations) {  System.*out*.println(annotation); //@reflection.huodejiegou.MyAnnotation(value=hello)  }  //2. 权限修饰符  System.*out*.print(Modifier.*toString*(method.getModifiers()) + " "); //public public volatile public private public   //3. 返回值类型  */\*\*  \* int  \* int  \* void  \* class java.lang.String  \* class java.lang.String  \*/* Class<?> returnType = method.getReturnType(); // System.out.print(returnType + "\t");  */\*\*  \* int  \* int  \* void  \* java.lang.String  \* java.lang.String  \*/* System.*out*.print(returnType.getName() + " ");   //4. 方法名  */\*\*  \* compareTo  \* compareTo  \* info  \* display  \* show  \*/* String name = method.getName();  System.*out*.print(name);  System.*out*.print("(");  //5. 形参列表  Class<?>[] parameterTypes = method.getParameterTypes();  int i = 0;  if (parameterTypes != null && parameterTypes.length > 0) {  for (Class<?> type : parameterTypes) {  if (i < parameterTypes.length - 1) {  System.*out*.print(type.getName() + " " + "args\_" + i + ", ");  } else {  System.*out*.print(type.getName() + " " + "args\_" + i);  }  i++;  }  }  System.*out*.print(")");   //6. 抛出的异常  Class<?>[] exceptionTypes = method.getExceptionTypes();  if (exceptionTypes != null && exceptionTypes.length > 0) {  System.*out*.print(" throws ");  int j = 0;  for (Class<?> type : exceptionTypes) {  if (j < exceptionTypes.length - 1) {  System.*out*.print(type.getName() + ", ");  } else {  System.*out*.println(type.getName());  }  j++;  }  }   System.*out*.println();  }    } |

#### 获取运行时类的构造器

|  |
| --- |
| */\*\*  \* 构造器  \*/* @Test public void test3() {  Class<Person> personClass = Person.class;  //getConstructors(): 获取当前运行时类中声明为public的构造器  Constructor<?>[] constructors = personClass.getConstructors();  for (Constructor<?> constructor : constructors) {  System.*out*.println(constructor); //public reflection.huodejiegou.Person()  }   //getDeclaredConstructors(): 获取当前运行时类中声明的所有的构造器(包括private的)  Constructor<?>[] declaredConstructors = personClass.getDeclaredConstructors();  for (Constructor<?> declaredConstructor : declaredConstructors) {  */\*\*  \* reflection.huodejiegou.Person(java.lang.String,int)  \* private reflection.huodejiegou.Person(java.lang.String)  \* public reflection.huodejiegou.Person()  \*/* System.*out*.println(declaredConstructor);  } } |

#### 获取运行时类的其他结构

|  |
| --- |
| package reflection.huodejiegou;  import org.junit.Test;  import java.lang.annotation.Annotation; import java.lang.reflect.ParameterizedType; import java.lang.reflect.Type;  */\*\*  \** ***@author*** *lizhenhong  \** ***@Description:*** *获取运行时类的其他结构  \** ***@date*** *2020/5/16 20:30  \*/* public class OtherTest {   */\*\*  \* 获取运行时类的父类  \*/* @Test  public void test1() {  Class<Person> personClass = Person.class;  Class<? super Person> superclass = personClass.getSuperclass();  System.*out*.println(superclass); //class reflection.huodejiegou.Creature  }   */\*\*  \* 获取带泛型的父类  \*/* @Test  public void test2() {  Class<Person> personClass = Person.class;  Type superclass = personClass.getGenericSuperclass();  System.*out*.println(superclass); //reflection.huodejiegou.Creature<java.lang.String>   //获取运行时类的父类的泛型  Type genericSuperclass = personClass.getGenericSuperclass();  ParameterizedType paramType = (ParameterizedType) genericSuperclass;  Type[] arguments = paramType.getActualTypeArguments();  System.*out*.println(arguments[0].getTypeName()); //java.lang.String  for (Object argument : arguments) {  System.*out*.println(argument); //class java.lang.String  }  }   */\*\*  \* 获取运行时类实现的接口  \*/* @Test  public void test3() {  Class<Person> personClass = Person.class;  Class<?>[] interfaces = personClass.getInterfaces();  for (Class<?> anInterface : interfaces) {  */\*\*  \* interface java.lang.Comparable  \* interface reflection.huodejiegou.MyInterface  \*/* System.*out*.println(anInterface);   }  Class<?>[] interfaces1 = personClass.getSuperclass().getInterfaces();  for (Class<?> aClass : interfaces1) {  System.*out*.println(aClass); //interface java.io.Serializable  }  }   */\*\*  \* 获取运行时类所在的包  \*/* @Test  public void test5() {  Class<Person> personClass = Person.class;  Package aPackage = personClass.getPackage();  System.*out*.println(aPackage); //package reflection.huodejiegou  System.*out*.println(aPackage.getName()); //reflection.huodejiegou  }   */\*\*  \* 获取运行时类声明的注解  \*/* @Test  public void test6() {  Class<Person> personClass = Person.class;  Annotation[] annotations = personClass.getAnnotations();  for (Annotation annotation : annotations) {  System.*out*.println(annotation); //@reflection.huodejiegou.MyAnnotation(value=Person)  }  }  } |

#### 调用运行时类中的指定属性

|  |
| --- |
| package reflection;  import org.junit.Test; import reflection.huodejiegou.Person;  import java.lang.reflect.Field;  */\*\*  \** ***@author*** *lizhenhong  \** ***@Description:*** *调用运行时类中指定的结构(属性*、*方法*、*构造器)  \** ***@date*** *2020/5/16 22:12  \*/* public class ReflectionsTest {   */\*\*  \* 获取属性(这种方法由于只能获得public的属性，故使用频率不高)  \*/* @Test  public void getField() throws Exception {  Class<Person> personClass = Person.class;  //创建运行时类的对象  Person person = personClass.newInstance();  //获取指定的属性(要求此属性时public的)  Field id = personClass.getField("id");  */\*\*  \* 设置当前属性的值  \* set(Object obj, Object value):  \* 参数一：指明被设置对象  \* 参数二：设置的值  \*/* id.set(person, 18);  //获取当前对象的当前属性值  Object o = id.get(person);  System.*out*.println(o); //18   }   */\*\*  \* 这个比较重要，需要掌握  \* 获取属性(这个是常用的方法)  \*/* @Test  public void testGetField() throws Exception {  Class<Person> personClass = Person.class;  //1. 创建运行时类的对象  Person person = personClass.newInstance();  //2。 getDeclaredField(String fieldName): 获取运行时类中指定名称的属性  Field name = personClass.getDeclaredField("name");  //3. etAccessible(true): 保证当前属性是可访问的  name.setAccessible(true);  //4. 根据实际需求设置指定对象的属性值  name.set(person,"zhangyunzhu");  System.*out*.println(name.get(person)); //zhangyunzhu   }   */\*\*  \* 调用运行时类的指定方法  \*/* @Test  public void testMethod() {   }  } |

#### 调用运行时类的指定方法

|  |
| --- |
| */\*\*  \* 调用运行时类的指定方法  \*/* @Test public void testMethod() throws Exception {   Class<Person> personClass = Person.class;  //创建运行时类的对象  Person person = personClass.newInstance();  */\*\*  \* 1. 获取指定的某个方法getDeclaredMethod(String name, Class*<?> *class):  \* 参数一: 要获取的方法的名称  \* 参数二: 方法的形参列表  \*/* Method show = personClass.getDeclaredMethod("show", String.class);  //2. 保证当前方法是可用的  show.setAccessible(true);  */\*\*  \* 3. 调用invoke(Object obj, Object args)执行  \* 参数一: 方法的调用者  \* 参数二: 被调用方法的入参(参数列表)  \* invoke方法是有返回值的，得到的返回值即为运行时类中被调用的方法的返回值  \*/* Object china = show.invoke(person, "China");  System.*out*.println(china); //China   System.*out*.println("===============静态方法的调用==================");  //调用静态方法private static void showDesc  Method showDesc = personClass.getDeclaredMethod("showDesc");  showDesc.setAccessible(true);  Object invoke = showDesc.invoke(person); //调用静态方法的时候后加的  System.*out*.println(invoke); //null } |

#### 调用运行时类的指定构造器

|  |
| --- |
| */\*\*  \* 调用运行时类的指定构造器  \*/* @Test  public void testConstrctor() throws Exception {  Class<Person> personClass = Person.class; // Person person = personClass.newInstance(); //最常用的方法，方便，实惠  //private Person(String name) {  */\*\*  \* 1. 获取指定构造器  \* getDeclaredConstructor(Class*<?> *... paramType)：  \* 入参: 构造器的参数列表  \*/* Constructor<Person> constructor = personClass.getDeclaredConstructor(String.class);  //2. 保证此构造器时可访问的  constructor.setAccessible(true);  //3. 调用此构造器(创建运行时类的对象)  Person person = constructor.newInstance("zhangyunzhu");  System.*out*.println(person.toString()); //Person{name='zhangyunzhu', age=0, id=0}   } |

声明四个学生类，一个老师类

学生一：姓名，年龄，年级

学生二：姓名，年龄，年级

学生三：姓名，年龄，性别

学生四：姓名，年龄，性别，show()方法

教师一：姓名，年龄，性别，年级

教师二：姓名，年龄，性别，年级

首先取任意两个对象判断是否相等，然后用反射的方式获取每个对象中年龄的值，根据年龄值倒叙排序

## java8新特性

|  |
| --- |
|  |
| **好处：** |

### Lambda表达式

#### 初识lambda

|  |
| --- |
|  |
| **举例：**  package lambda;  import org.junit.Test;  import java.util.Comparator;  */\*\*  \* @description: lambda表达式的实用举例  \* @date: 2020/5/12 19:06  \* @author: lizhenhong  \*/* public class LambdaTest {   @Test  public void test1() {  Runnable r1 = new Runnable() {  @Override  public void run() {  System.*out*.println("用普通的方式写的runnable"); *//用普通的方式写的runnable* }  };  r1.run();   System.*out*.println("===========================================");    Runnable r2 = () -> System.*out*.println("用lambda表达式些的runnable"); *//用lambda表达式些的runnable* r2.run();  }   @Test  public void test2() {  Comparator<Integer> com1 = new Comparator<Integer>() {  @Override  public int compare(Integer o1, Integer o2) {  return Integer.*compare*(o1, o2);  }  };  int compare = com1.compare(12, 13); *//-1 //表明前面的数小* System.*out*.println(compare);   System.*out*.println("===========================================");  *//lambda表达式的写法* Comparator<Integer> com2 = (o1, o2) -> Integer.*compare*(o1, o2);  int compare2 = com2.compare(222, 132); *//1 //表明前面的数大* System.*out*.println(compare2);   System.*out*.println("===========================================");  *//方法引用* Comparator<Integer> com3 = Integer::*compare*;  int compare3 = com3.compare(2232, 1332); *//1 //表明前面的数大* System.*out*.println(compare3);   }   } |

#### lambda表达式的语法和使用

|  |
| --- |
|  |
|  |
| package lambda;  import org.junit.Test;  import java.util.Comparator; import java.util.function.Consumer;  */\*\*  \* @description: lambda的语法和使用  \* @date: 2020/5/12 19:25  \* @author: lizhenhong  \*/* public class LambdaToUse {   *//1. 无参、无返回值* @Test  public void testNoParamNoReturn() {  Runnable r1 = new Runnable() {  @Override  public void run() {  System.*out*.println("用普通的方式写的runnable"); *//用普通的方式写的runnable* }  };  r1.run();   System.*out*.println("===========================================");   Runnable r2 = () -> System.*out*.println("用lambda表达式些的runnable"); *//用lambda表达式些的runnable* r2.run();  }   *//2. 一个参数、无返回值* @Test  public void testOneParamNoReturn() {  Consumer<String> consumer = new Consumer<String>() {  @Override  public void accept(String s) {  System.*out*.println(s); *//这个类之前真的没有用过* }  };  consumer.accept("这个类之前真的没有用过");   System.*out*.println("=============================================");    Consumer<String> consumer1 = (String s)-> {  System.*out*.println(s); *//用lambda写一下试试* };  consumer1.accept("用lambda写一下试试");  }   *//3. 数据类型可以省略，因为可由编译器推断得出，称为"类型推断"* @Test  public void test3() {  Consumer<String> consumer1 = (String s)-> {  System.*out*.println(s); *//用lambda写一下试试* };  consumer1.accept("用lambda写一下试试");   System.*out*.println("=============================================");   Consumer<String> consumer2 = (s)-> {  System.*out*.println(s); *//用lambda写一下试试* };  consumer2.accept("用lambda写一下试试");    }   *//4. 如果lambda只需要一个参数时，参数的小括号可以省略* @Test  public void test4() {  Consumer<String> consumer1 = (s)-> {  System.*out*.println(s); *//用lambda写一下试试* };  consumer1.accept("用lambda写一下试试");   System.*out*.println("=============================================");  Consumer<String> consumer2 = s-> {  System.*out*.println(s); *//用lambda写一下试试* };  consumer2.accept("用lambda写一下试试");    }   *//5. 如果lambda需要两个或两个以上的参数，多条执行语句，并且可以有返回值(直接把方法体中的内容搬过去)* @Test  public void test5() {  Comparator<Integer> com = new Comparator<Integer>() {  @Override  public int compare(Integer o1, Integer o2) {  System.*out*.println("o1是: " + o1); *//o1是: 11* System.*out*.println("o2是: " + o2); *//o2是: 22* return o1.compareTo(o2); *//-1* }  };  System.*out*.println(com.compare(11, 22));   System.*out*.println("=============================");  Comparator<Integer> com2 = (o1,o2)->{  System.*out*.println("o1是: " + o1); *//o1是: 111* System.*out*.println("o2是: " + o2); *//o2是: 2* return o1.compareTo(o2); *//1* };  System.*out*.println(com2.compare(111, 2));  }   *//6. 当lambda体只有一条语句时，return与大括号如果有的话，大括号和return都可以省略* @Test  public void test6() {  Comparator<Integer> com1 = (o1,o2)->{  return o1.compareTo(o2); *//1* };  System.*out*.println(com1.compare(111, 2));   System.*out*.println("=============================");   Comparator<Integer> com2 = (o1, o2) -> o1.compareTo(o2); *//-1* System.*out*.println(com2.compare(111, 223));  }   @Test  public void test7() {  Consumer<String> consumer = s -> {  System.*out*.println(s); *//用Consumer试试第六种语法* };  consumer.accept("用Consumer试试第六种语法");   System.*out*.println("=============================");   Consumer<String> consumer1 = s -> System.*out*.println(s); *//再试试* consumer1.accept("再试试");    }   } |
| **总结：**   1. ->左边: lambda形参列表的参数类型可以省略(类型更新)；   如果形参列表只有一个参数，参数列表的小括号可以省略。   1. ->右边：lambda体: 一般应该使用一对{}进行包裹；   如果lambda体只有一条执行语句(可能时return语句)，这时可以省略{}和return关键字(这里必须同步省略，即要么都省略，要么都写上)。 |

#### 函数式(Function)接口

|  |
| --- |
| **如果一个接口中，只声明了一个抽象方法，则此接口就称之为函数时接口**  **看到day29，13** |