# 尚马教育 JAVA 基础课程

# 集合

文档编号：A14

创建日期： 2017-04-12

最后修改日期：2019-09-23

版 本 号：V3.0

电子版文件名：尚马教育-第一阶段-14.集合专题课程.docx

**文档修改记录：**

|  |  |  |  |
| --- | --- | --- | --- |
| 更新日期 | 更新作者 | 更新说明 | 版本号 |
| 2017-07-30 | 张元林 | 初始版本 | V1.0 |
| 2018-08-01 | 王绍成 | Java基础版本更新 | V2.0 |
| 2019-08-09 | 徐丽莎 | Java基础版本更新 | V3.0 |

**主讲人：**

**徐丽莎**

目录

[尚马教育 JAVA 基础课程 1](#_Toc26175)

[集合 1](#_Toc26462)

[1. 集合 3](#_Toc2589)

[1.1. 集合优势 3](#_Toc23480)

[1.2. 集合分类 3](#_Toc29903)

[1.3. 集合分类特点 4](#_Toc3461)

[2. Collection<T> 4](#_Toc11880)

[2.1. 常用方法 5](#_Toc1504)

[2.2. 案例 5](#_Toc8069)

[3. List<E> 5](#_Toc18204)

[3.1. 常用实现类 6](#_Toc13488)

[3.2. ArrayList<E> 6](#_Toc14617)

[3.2.1. 常用构造 6](#_Toc5599)

[3.2.2. 常用方法 7](#_Toc3390)

[3.2.3. 案例 7](#_Toc23289)

[3.3. Linkedist<E> 7](#_Toc32622)

[3.3.1. 常用构造 7](#_Toc11660)

[3.3.2. 常用方法 8](#_Toc17260)

[3.3.3. 案例 8](#_Toc2321)

[3.4. Vector 8](#_Toc10884)

[3.4.1. 常用构造 9](#_Toc5748)

[3.4.2. 常用方法 9](#_Toc11135)

[3.4.3. 案例 10](#_Toc17593)

[3.4.4. 实现类区别 10](#_Toc1832)

[4. Set<E> 10](#_Toc4088)

[4.1. 常用实现类 10](#_Toc21985)

[4.2. HashSet<E> 10](#_Toc12549)

[4.2.1. 常用构造 11](#_Toc28458)

[4.2.2. 常用方法 11](#_Toc891)

[4.2.3. 案例 11](#_Toc25085)

[4.3. LinkedHashSet<E> 11](#_Toc7430)

[4.3.1. 案例 11](#_Toc11630)

[4.4. TreeSet<E> 12](#_Toc10273)

[4.4.1. 常用构造 12](#_Toc8948)

[4.4.2. 案例 12](#_Toc26022)

[4.5. 注意 12](#_Toc26688)

[4.5.1. 实现类区别 12](#_Toc5078)

[5. Map<K,V> 12](#_Toc31890)

[5.1. 常用实现类 13](#_Toc11986)

[5.2. HashMap<K,V> 13](#_Toc1512)

[5.2.1. 常用构造 13](#_Toc23955)

[5.2.2. 常用方法 14](#_Toc3200)

[5.2.3. 案例 14](#_Toc14303)

[5.3. LinkedHashMap<K,V> 14](#_Toc16254)

[5.3.1. 案例 14](#_Toc13532)

[5.4. TreeMap<K,V> 14](#_Toc29627)

[5.4.1. 案例 14](#_Toc20262)

[5.4.2. 实现类区别 15](#_Toc22016)

[6. 集合元素排序 15](#_Toc4750)

[6.1. List集合元素排序 15](#_Toc21572)

[6.1.1. 常用方法(Collections) 15](#_Toc10482)

[6.1.2. 案例 15](#_Toc7586)

[6.2. Set集合元素排序 15](#_Toc26327)

[6.2.1. 案例 15](#_Toc22298)

[6.3. Map集合元素排序 16](#_Toc7581)

[6.3.1. 案例 16](#_Toc21349)

[7. 集合并行化Stream 16](#_Toc26758)

[7.1. 案例 16](#_Toc20069)

[8. 作业 16](#_Toc4853)

## 集合

* 集合是将多个元素组成一个单元的对象；
* 类似于数组，但数组最大的缺点是：长度受到限制（一经创建，就不可再改变），并且只能存放相同数据类型的元素；
* 集合的长度没有限制，可以存放任意多的元素，而且元素的数据类型也可以不同；
* 集合还提供一系列操纵数据的方法，如存储、检索等等。

### 集合优势

* 提供有用的数据结构和算法，从而减少编程工作；
* 提高了程序速度和质量,因为它提供了高性能的数据结构和算法；
* 允许不同 API 之间的相互操作，API之间可以来回传递集合；可以方便地扩展或改写集合。

### 集合分类

* 为了满足不同场合的需要，java.util包中包含有一系列集合类；
* 如：ArrayList类、LinkedList类、Vector类、HashMap类等等，接下来将逐一进行介绍；
* 集合类中只能存放对象，而不能存放原始数据类型的元素，所以当有原始数据类型需要存放时，只能将其转换成相应的包装类对象。

|  |
| --- |
|  |

### 集合分类特点

|  |
| --- |
|  |
|  |

## Collection<T>

* 集合层次结构中的根界面

|  |
| --- |
|  |

### 常用方法

|  |
| --- |
|  |

### 案例

|  |
| --- |
| **private** **static** **void** test1() {  // 创建集合对象  Collection collection = **new** ArrayList();  // 新增元素  collection.add(100);// 类类型 Interger  collection.add(**null**);  collection.add("hello");  collection.add("hello");  collection.add(1.0);  collection.add(1.0);  collection.add(**new** Object());  System.***out***.println(collection);  System.***out***.println(collection.size());  // 删除一个 元素  collection.remove("hello");  // collection.clear();  System.***out***.println(collection.isEmpty());  // 判断集合里面是否包含某个元素  System.***out***.println(collection.contains("hello"));  Object[] objs = collection.toArray();  System.***out***.println(Arrays.*toString*(objs));  // Object[] objs = new Object[collection.size()];  // collection.toArray(objs);  // System.out.println(Arrays.toString(objs));  // 遍历集合元素  System.***out***.println("-----------------------");  **for** (Object obj : collection) {  System.***out***.println(obj);  }  System.***out***.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  // Iterator<E> iterator() 获得集合对象的迭代器对象 (将集合的所有元素 都存储至迭代器对象 )  Iterator it = collection.iterator();// 集合的所有的元素都在it对象里面了  **while** (it.hasNext()) {  Object obj = it.next();  System.***out***.println("obj:" + obj);  }  System.***out***.println("++++++++++++++++++++++++++++++++++++");  // default void forEach(Consumer<? super T> action) 遍历集合所有元素  // forEach+lambda表达式  // lambda表达式: 替代接口的匿名内部类对象(要求： 接口里面 有且只有一个抽象方法)  // 语法: (数据类型 形参1,数据类型 形参2)->{  // //逻辑  // return;  // }  System.***out***.println("====================================");  collection.forEach((Object obj)->{  System.***out***.println(obj);  });  }} |

## List<E>

* 有序集合（也称为序列 ）。 该界面的用户可以精确控制列表中每个元素的插入位置。 用户可以通过整数索引（列表中的位置）访问元素，并搜索列表中的元素。

### 常用实现类

|  |
| --- |
|  |

### ArrayList<E>

* ArrayList是长度可变的对象引用数组，称为动态数组；
* 随着元素的添加，元素数目的增大，数组容量也会随之自动扩展；访问和遍历数组元素时，ArrayList的性能优越；
* ArrayList类继承了AbstractList类并实现了List接口

#### 常用构造

* rrayList类的构造方法有3种重载方式

|  |
| --- |
|  |

#### 常用方法

|  |
| --- |
|  |

#### 案例

|  |
| --- |
| **private** **static** **void** testArrayList() {  List<String> strList = **new** ArrayList<>();//底层创建了一个Object[] 10  strList.add("a");  strList.add("b");  strList.add("c");  strList.add("d");  strList.add(0, "aa");//0 size()-1  System.***out***.println(strList);  //查看单个元素  String element = strList.get(3);  System.***out***.println(element);  //修改  strList.set(0, "hello");  System.***out***.println(strList);  //删除  strList.remove(0);  System.***out***.println(strList);  strList.forEach((str)->{  System.***out***.println(str);  });  List<String> subStr = strList.subList(0, 3);//[a, b, c]  System.***out***.println("subStr:"+subStr);  // subStr.add("123");  // strList.remove(0);  strList.add(0,"222");  System.***out***.println(subStr);  System.***out***.println(strList);  } |

### Linkedist<E>

* LinkedList类用于创建链表数据结构；
* 链表中元素的数量不受任何限制，可以随意地添加和删除；
* 与ArrayList相比，如果需要频繁地添加和删除元素，LinkedList的性能更加优越；
* LinkedList类继承了AbstractSequentialList类，并实现了List接口；

#### 常用构造

|  |
| --- |
|  |

#### 常用方法

|  |
| --- |
|  |
|  |

#### 案例

|  |
| --- |
| **private** **static** **void** testLinkedList() {  //消息队列 LinkedList Queue 队列  LinkedList<Integer> linkedList = **new** LinkedList<>();  linkedList.add(10);  linkedList.add(100);  linkedList.add(1000);  linkedList.add(10000);  //把元素10删除  linkedList.remove(**new** Integer(10));  // linkedList.add(0, 100);  // linkedList.addFirst(1);  // linkedList.addLast(1);  // linkedList.offer(1);  // linkedList.offerFirst(2);  // linkedList.offerLast(3);  System.***out***.println(linkedList);  // System.out.println(linkedList.get(2));  // System.out.println(linkedList.getFirst());  // System.out.println(linkedList.getLast());  // System.out.println(linkedList.peek());  // System.out.println(linkedList.peekFirst());  // System.out.println(linkedList.peekLast());  // System.out.println(linkedList.poll());  // System.out.println(linkedList.pollFirst());  // System.out.println(linkedList.pollLast());  // System.out.println(linkedList.pop());  // linkedList.push(1);  // System.out.println(linkedList.pop());  // System.out.println(linkedList);  } |

### Vector

* Vector类与ArrayList类和LinkedList类很相似，最大的区别在于Vector是线程同步的；
* 如果在多线程的程序中要使用到集合框架，并且不希望线程与线程之间相互干扰，那么Vector是不错的选择；
* Vector类继承于AbstractList类，并实现了List接口。

#### 常用构造

|  |
| --- |
|  |

#### 常用方法

|  |
| --- |
|  |
|  |

#### 实现类区别

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| | **常用实现类** | **新增/查询/删除** | **底层数据结构** | **线程安全** | | --- | --- | --- | --- | | ArrayList<E>(常用) | 查询最快 | 动态数组 | 不安全 | | LinkedList<E> | 新增/删除最快 查询慢 | 双向链表 | 不安全 | | Vector<E> | 三者都慢 | 动态数组 | 安全 | |

## Set<E>

* 不包含重复元素的集合。

### 常用实现类

|  |
| --- |
|  |

### HashSet<E>

* 此类实现Set接口，由哈希表（实际为HashMap实例）支持。 对集合的迭代次序不作任何保证; 特别是，它不能保证订单在一段时间内保持不变。 这个类允许null元素。

#### 常用构造

|  |
| --- |
|  |

#### 常用方法

|  |
| --- |
|  |

#### 案例

|  |
| --- |
| **private** **static** **void** testHashSet() {  //不能重复  HashSet<String> set = **new** HashSet<>();  set.add("a");  set.add("a");  set.add("abc");  set.add("g");  set.add(**null**);  set.add("123");  System.***out***.println(set);  set.remove("a");  System.***out***.println(set);  //遍历方式  //增强for  //forEach  //iterator  } |

### LinkedHashSet<E>

* 是HashSet的子类，方法几乎与父类一致
* 元素有序。(元素插入顺序与遍历顺序是一致的)。

#### 案例

|  |
| --- |
| **private** **static** **void** testLinkedHashSet() {  LinkedHashSet<Integer> hashSet = **new** LinkedHashSet<>();  hashSet.add(1);  hashSet.add(10);  hashSet.add(11);  hashSet.add(1);  hashSet.add(**null**);  System.***out***.println(hashSet);  LinkedHashSet<User> userSet = **new** LinkedHashSet<>();  userSet.add(**new** User(1, "jim1"));  userSet.add(**new** User(1, "jim1"));  userSet.add(**new** User(3, "jim3"));  userSet.add(**new** User(4, "jim4"));  System.***out***.println(userSet);  } |

### TreeSet<E>

* 元素有序. （会按照自然顺序进行排列）,规定元素数据类型必须实现Comparable接口

#### 常用构造

|  |
| --- |
|  |

#### 案例

|  |
| --- |
| **private** **static** **void** testTreeSet() {  // 不能重复  TreeSet<Integer> set = **new** TreeSet<>();  set.add(0);  set.add(1);  set.add(10);  set.add(30);  set.add(100);  // set.add(null);  System.***out***.println(set);  // System.out.println(set.first());  // System.out.println(set.last());  // System.out.println(set.pollFirst());  // System.out.println(set.pollLast());  System.***out***.println(set);  } |

### 注意

* 由于set集合存储不重复的元素，如果要存储自定义类对象的时候，一定要重写重写equals(Object o) 和 hashCode() 方法
* 要求集合中存放的元素对应的类型必须重写equals(Object o) 和 hashCode() 方法
* 一般情况下，可以使用IDE提供的工具进行自动生成equals(Object o) 和 hashCode()

#### 实现类区别

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| | **常用实现类** | **底层数据结构** | **线程安全** | **元素是否可以为null** | | --- | --- | --- | --- | | HashSet<E> (推荐) | Hash表 | 不安全 | 可以为null | | LinkedHashSet<E> | 链表+hash表 | 不安全 | 可以为null | | TreeSet<E> | 红黑树 | 不安全 | 不能为null | |

## Map<K,V>

* 将键映射到值的对象。 Map不能包含重复的键; 每个键可以映射到最多一个值.

### 常用实现类

|  |
| --- |
|  |

### HashMap<K,V>

#### 常用构造

|  |
| --- |
|  |

#### 常用方法

|  |
| --- |
|  |

#### 案例

|  |
| --- |
| **private** **static** **void** testHashMap() {  Map<Integer, String> map = **new** HashMap<>();  // 存储元素  map.put(1, "a");// Node<K,V>  map.put(1, "b");  map.put(2, "a");  map.put(3, "a");  map.put(**null**, **null**);  System.***out***.println(map);  // 查询一组元素  String value = map.get(1);  System.***out***.println(value);  System.***out***.println(map.getOrDefault(3, "hello"));  // 修改  // String oldValue = map.replace(4, "hello");  // System.out.println(map.replace(3, "a1", "hello"));  // 删除  // System.out.println(map.remove(1));//b  // System.out.println(map.remove(1, "b1"));  // 判断  // System.out.println(map.containsKey(1));  // System.out.println(map.containsValue("a"));  // System.out.println(map.size());  // 遍历  // map.forEach(new BiConsumer<Integer, String>() {  // @Override  // public void accept(Integer key, String value) {  // System.out.println("key:"+key+"=="+value);  // }  // });  // map.forEach((key, value1)->{  // System.out.println("key:"+key+"=="+value1);  // });  // jdk1.8-  // Set<Map.Entry<Integer, String>> set = map.entrySet(); //视图  // set.forEach(entry->{  // System.out.println("key:"+entry.getKey()+"=="+entry.getValue());  // });  Set<Integer> keySet = map.keySet();//  keySet.forEach(key -> {  System.***out***.println("key:" + key + "==" + map.get(key));  });  // map.put(100, "abc");  Collection<String> values = map.values();  System.***out***.println(values);  // java.util.HashMap$Values  System.***out***.println(map);  } |

### LinkedHashMap<K,V>

* HashMap的子类，功能方法几乎一致。
* LinkedHashMap元素可以预测的 key 插入的顺序与遍历顺序一致

#### 案例

|  |
| --- |
| **private** **static** **void** testLinkedHashMap() {  Map<Integer, String> map = **new** LinkedHashMap<>();  // 存储元素  map.put(1, "a");// Node<K,V>  map.put(300, "a");  map.put(100, "b");  map.put(20, "a");  System.***out***.println(map);  } |

### TreeMap<K,V>

* TreeMap key值有序 按照自然顺序对key进行排列
* 常用的功能方法与以上实现类一致。

#### 案例

|  |
| --- |
| **private** **static** **void** testTreeMap() {  // key 数据 类型一定实现Comparable  TreeMap<Integer, String> map = **new** TreeMap<>();  // 存储元素  map.put(300, "a");  map.put(1, "a");  map.put(100, "b");  map.put(20, "a");  System.***out***.println(map);  System.***out***.println(map.firstKey());  System.***out***.println(map.firstEntry());  System.***out***.println(map.lastEntry());  } |

#### 实现类区别

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| | **实现类** | **底层数据结构** | **线程是否安全** | **K与V是否可以为null** | | --- | --- | --- | --- | | HashMap<K,V>(推荐) | hash表 | 否 | K和V 都可以为null | | LinkedHashMap<K,V> | 链表+hash表 | 否 | K和V 都可以为null | | TreeMap<K,V> | 红黑树 | 否 | K不可以为null , V可以 | | CouncurrentHashMap<K,V>(推荐) | 1.7 锁分段 1.8 CAS | 是 | K和V都不可以为null | | HashTable<K,V> | hash表 | 是(同步) | K和V都不可以为null | |

## 集合元素排序

### List集合元素排序

* Collections:此类仅由静态方法组合或返回集合。 它包含对集合进行操作的多种功能。

#### 常用方法(Collections)

|  |
| --- |
| static <T extends Comparable<? super T>> void sort(List<T> list)  **static** <T> **void** sort(List<T> list, Comparator<? **super** T> c)  **static** **void** shuffle(List<?> list)  **static** <T **extends** Object & Comparable<? **super** T>> T max(Collection<? **extends** T> coll)  **static** <T> T max(Collection<? **extends** T> coll, Comparator<? **super** T> comp)  **static** <T **extends** Object & Comparable<? **super** T>> T min(Collection<? **extends** T> coll)  **static** <T> T min(Collection<? **extends** T> coll, Comparator<? **super** T> comp) |

#### 案例

|  |
| --- |
| **private** **static** **void** testSortList() {  List<Integer> numList = **new** ArrayList<Integer>();  numList.add(10);  numList.add(0);  numList.add(100);  numList.add(20);  System.***out***.println(numList);// [10, 0, 100, 20]  // 集合转换成数组  // Integer[] nums = new Integer[numList.size()];  // numList.toArray(nums);  // Arrays.sort(nums);  // System.out.println(Arrays.toString(nums));  // 直接sort()  // Collections.sort(numList);//集合元素类型实现Coomparable接口  System.***out***.println("max:" + Collections.*max*(numList));  System.***out***.println("min:" + Collections.*min*(numList));  System.***out***.println(numList);// String +包装类  } |
| **private** **static** **void** testSortList2() {  List<User> userList = **new** ArrayList<>();  userList.add(**new** User(0, "tom"));  userList.add(**new** User(10, "jim1"));  userList.add(**new** User(10, "jam"));  userList.add(**new** User(21, "abc"));  userList.forEach(user -> {  System.***out***.println(user);  });  System.***out***.println("---------------------------");  // 元素必须实现Comparable接口  Collections.*sort*(userList);  userList.forEach(user -> {  System.***out***.println(user);  });  } |
| **public** **class** User **implements** Comparable<User>{  **private** Integer id;//唯一的 不可重复  **private** String name;//  @Override  **public** **int** compareTo(User user1) {  // return this.name.compareTo(user1.getName());//升序  // return user1.getId().compareTo(this.id);  //先根据id比较 id值一样 再根据name比  **int** result = **this**.getId().compareTo(user1.getId());  **if**(result==0) {  result = **this**.name.compareTo(user1.getName());  }  **return** result;  }  } |
| **private** **static** **void** testSortList3() {  List<Person> personList = **new** ArrayList<>();  personList.add(**new** Person(0, "tom"));  personList.add(**new** Person(10, "jim1"));  personList.add(**new** Person(10, "jam"));  personList.add(**new** Person(21, "abc"));  personList.forEach(person -> {  System.***out***.println(person);  });  System.***out***.println("---------------------------");  // Collections.sort(personList, new Comparator<Person>() {  // @Override  // public int compare(Person person1, Person person2) {  // //id 升序  // //name 降序  // int result = person1.getId().compareTo(person2.getId());  // if(result==0) {  // result = person1.getName().compareTo(person2.getName());  // }  // return result;  // }  // });  Collections.*sort*(personList, (Person person1, Person person2) -> {  **int** result = person1.getId().compareTo(person2.getId());  **if** (result == 0) {  result = person1.getName().compareTo(person2.getName());  }  **return** result;  });  personList.forEach(person -> {  System.***out***.println(person);  });  } |

### Set集合元素排序

* HashSet/LinkedHashSet本身就是无序 无法对元素顺序排列
* 只能针对TreeSet元素排列(要求Treeset集合元素类型都必须实现Comparable)

#### 案例

|  |
| --- |
| **private** **static** **void** testHashSet() {  Set<Integer> set = **new** HashSet<>();  set.add(1);  set.add(10);  set.add(101);  set.add(21);  //获得元素的最值  System.***out***.println("max:"+Collections.*max*(set));  System.***out***.println("min:"+Collections.*min*(set));  Set<Person> set1 = **new** HashSet<>();  set1.add(**new** Person(0, "tom"));  set1.add(**new** Person(10, "jim1"));  set1.add(**new** Person(10, "jam"));  set1.add(**new** Person(21, "abc"));  // System.out.println("max:"+(Collections.max(set1, (p1,p2)->{  // return p1.getId().compareTo(p2.getId());  // })));  System.***out***.println(set1);  } |
| **private** **static** **void** testTreeSet() {  TreeSet<Integer> set = **new** TreeSet<>();  set.add(1);  set.add(10);  set.add(101);  set.add(21);  System.***out***.println(set);  System.***out***.println(set.last());  System.***out***.println(set.first());  System.***out***.println("---------------------");  TreeSet<Person> set1 = **new** TreeSet<>((p1,p2)->{  **return** p2.getId().compareTo(p1.getId());  }) ;  set1.add(**new** Person(0, "tom"));  set1.add(**new** Person(10, "jim1"));  set1.add(**new** Person(10, "jam"));  set1.add(**new** Person(21, "tom"));  System.***out***.println(set1);  //[Person [id=21, name=abc], Person [id=10, name=jim1], Person [id=0, name=tom]]  // [Person [id=21, name=tom], Person [id=10, name=jim1], Person [id=0, name=tom]]  } |

### Map集合元素排序

* HashMap/LinkedHashMap的key本身就是无序 无法对元素顺序排列
* 只能针对TreeMap元素排列(要求map的key值类型都必须实现Comparable)

#### 案例

|  |
| --- |
| **private** **static** **void** testTreeMap() {  //map的key排序  TreeMap<Integer, Integer> map = **new** TreeMap<Integer, Integer>();  map.put(1, 10);  map.put(100, 10);  map.put(11, 10);  map.put(21, 10);  System.***out***.println(map);  TreeMap<Person, Integer> map1 = **new** TreeMap<>((p1,p2)->{  **return** p2.getId().compareTo(p1.getId());  }) ;  map1.put(**new** Person(0, "tom"),1);  map1.put(**new** Person(10, "jim1"),1);  map1.put(**new** Person(10, "jam"),1);  map1.put(**new** Person(21, "abc"),1);  System.***out***.println(map1);  } |

## 集合并行化Stream

* 在遍历集合元素执行删除，修改或者新增功能，会出现并发修改的异常，jdk1.8之后，弥补了这个缺陷，提供了对集合元素的并行化操作。

### 案例

|  |
| --- |
| **public** **static** **void** main(String[] args) {  List<Person> personList = **new** ArrayList<>();  personList.add(**new** Person(0, "tom"));  personList.add(**new** Person(10, "jim1"));  personList.add(**new** Person(10, "jam"));  personList.add(**new** Person(21, "abc"));    System.***out***.println(personList);  //删除用户名里面包含m的对象  // for(Person person:personList) {  // if(person.getName().contains("m")) {  // personList.remove(person);  // }  // }  //获得集合对象的Stream实例  // Stream<Person> stream = personList.stream();  // //过滤掉一些不符合条件的元素数据  // stream = stream.filter(person->!person.getName().contains("m"));  // //转换成List/set集合对象 将过滤之后的数据进行收集  // personList = stream.collect(Collectors.toList());//Collectors  personList = personList.parallelStream().filter(person-> !person.getName().contains("m")).collect(Collectors.*toList*());  //count()  // System.out.println(personList.stream().count());  System.***out***.println(personList);  // Map<String, Integer> map = new HashMap<String, Integer>();  } |

## 作业

1. 创建好员工基类和程序员、项目经理子类的基础上，创建一个测试类。 创建一个ArrayList集合对象，要求保存两个程序员和两个项目经理的对象，并循环调用show的方法显示详细信息
2. 创建一个HashMap对象，要求保存两个程序员对象和两个项目经理对象，使用员工的编号做为主键，并循环调用show的方法显示详细信息
3. 基于集合，传入字符串时间，返回该时间是周几，中文显示（星期一）
4. 创建Student类，属性包括id[1-40], score[0-100],所有属性随机生成。创建Set集合，保存20个对象，找到分数最高与最低的学生
5. 已知有十六支男子足球队参加2008 北京奥运会。写一个程序，把这16 支球队随机分为4 个组。采用List集合和随机数 2008 北京奥运会男足参赛国家： 科特迪瓦,阿根廷,澳大利亚,塞尔维亚,荷兰,尼日利亚,日本,美国,中国,新西 兰,巴西,比利时,韩国,喀麦隆,洪都拉斯,意大利
6. 设计Account 类如下：

private long id;

private double balance;

private String password; 要求完善设计，使得该Account 对象能够自动分配自增id。给定一个List 如下：

List<Account> list = new ArrayList<>();

list.add(new Account(10.00, “1234”));

list.add(new Account(15.00, “5678”));

list.add(new Account(0, “1010”)); 要求把List 中的内容放到一个Map 中，该Map 的键为id，值为相应的Account 对象。 最后遍历这个Map，打印所有Account 对象的id 和余额