Set接口和Map接口

回顾

```

```

今天任务

```
1.Set接口
        1.1 Set接口的常用方法
        1.2 存储特点
        1.3 Set常用实现类
2.Map接口
        2.1 概述
        2.2 Map接口的常用方法
        2.3 Map常用实现类
        2.4 Map集合的遍历
3.Collections工具类
        3.1 Collections中的常用方法
```

教学目标

```
1.了解Set集合的特点
2.掌握Set集合下常用实现类的使用
3.了解Map集合的特点
4.掌握Map集合下常用实现类的使用
5.掌握Map集合的遍历方式
```

第一节 Set接口

1.1 Set接口常用方法

方法名	描述
add(E e)	确保此 collection 包含指定的元素（可选操作）。
addAll(Collection<? extends E> c)	将指定 collection 中的所有元素都添加到此 collection 中（可选操作）。
clear()	移除此 collection 中的所有元素（可选操作）。
contains(Object o)	如果此 collection 包含指定的元素，则返回true。
containsAll(Collection<?> c)	如果此 collection 包含指定 collection 中的所有元素，则返回 true。
equals(Object o)	比较此 collection 与指定对象是否相等。
isEmpty()	如果此 collection 不包含元素，则返回true。
iterator()	返回在此 collection 的元素上进行迭代的迭代器。
remove(Object o)	从此 collection 中移除指定元素的单个实例，如果存在的话（可选操作）。
removeAll(Collection<?> c)	移除此 collection 中那些也包含在指定 collection 中的所有元素（可选操作）。
retainAll(Collection<?> c)	仅保留此 collection 中那些也包含在指定 collection 的元素（可选操作）。
size()	返回此 collection 中的元素数。
toArray()	返回包含此 collection 中所有元素的数组。

1.2 存储特点

相对无序存储，不可以存储相同的元素（排重），不能通过下标访问

1.3 Set常用实现类

1.3.1 HashSet

此类实现`Set`接口，由哈希表（实际上是一个`HashMap`实例）支持。它不保证`set`的迭代顺序；特别是它不保证该顺序恒久不变。此类允许使用`null`元素。

存储特点：

相对无序存储，不可以存储相同元素（排重），通过哈希表实现的集合

1.3.2 重写hashCode()

`hashCode()`是`Object`中的方法，每个对象的`hashCode`值是唯一的，所以可以理解成`hashCode`值表示这个对象在内存中的位置

HashSet集合排重时，需要判断两个对象是否相同，对象相同的判断可以通过hashCode值判断，所以需要重写hashCode()方法

案例：设计一个Animal类，重写hashCode方法，向一个HashSet集合中添加Animal对象，

检验是否排重（若所有属性都相同，视为相同元素）

代码实现：

```
public class Animal {
    private String name;
    private int age;
    @Override
    public String toString() {
        return "Animal [name=" + name + ", age=" + age + "]";
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
    public Animal() {
        super();
    }
    public Animal(String name, int age) {
        super();
        this.name = name;
        this.age = age;
    }
    @Override
    /*重写hashCode()方法，单纯为了检查此方法是否可以实现排重效果，所以返回一个固定的值，使所有本类对象的
hashCode值都是相同的*/
    public int hashCode() {
        return 1;
    }
}
```

向HashSet集合中添加多个Animal对象时，所有属性都相同时，并没有完成想要的排重效果；

所以只重写hashCode方法并不能实现我们想要的排重效果

1.3.3 重写equals()

equals()方法是Object类中的方法，表示比较两个对象是否相等，若不重写相当于比较对象的地址，

所以我们可以尝试重写equals方法，检查是否排重

案例：设计一个Animal类，重写equals方法，向一个HashSet集合中添加Animal对象，
检验是否排重（若所有属性都相同，视为相同元素）

代码实现：

```
public class Animal {
    private String name;
    private int age;
    @Override
    public String toString() {
        return "Animal [name=" + name + ", age=" + age + "]";
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
    public Animal() {
        super();
    }
    public Animal(String name, int age) {
        super();
        this.name = name;
        this.age = age;
    }
    @Override
    /*重写hashCode()方法，单纯为了检查此方法是否可以实现排重效果，所以返回true，使得所有本类对象使用
equals方法比较时，都是相等的*/
    public boolean equals(Object obj) {
        return true;
    }
}
```

向HashSet集合中添加多个Animal对象时，所有属性都相同时，并没有完成想要的排重效果；

所以只重写equals方法，也不能完全实现我们想要的排重效果。

1.3.4 Set集合实现排重

同时重写hashCode和equals方法，实现排重

案例：设计一个Student类，同时重写hashCode和equals方法，检查是否实现排重

代码实现：

```java
public class Student {
    private String name;
    public Student(String name) {
        super();
        this.name = name;
    }
    public Student() {
        super();
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    @Override
    public String toString() {
        return "Student [name=" + name + "]";
    }
    @Override
    //重写equals
    public boolean equals(Object obj) {
        //先判断传入的参数对象是否是Student对象，若不是直接返回false
        if(obj instanceof Student) {
            //若是，强转成Student对象，并比较属性的值
            Student s = (Student) obj;
            if(this.name.equals(s.name)) {
                //若属性的值相同，则返回true
                return true;
            }
        }
        return false;
    }
    @Override
    public int hashCode(){
        /*hashCode方法返回值是int类型，所以重写时需要找到int类型的数据返回，还要保证此方法的返回值与对象的所有属性都相关,所以返回姓名属性的字符串的长度*/
        return this.name.length();
    }
}
```

> 同时重写hashCode和equals两个方法，可以实现元素的排重效果

1.3.5 LinkedHashSet

LinkedHashSet类是具有可预知迭代顺序(相对有序)的Set接口的哈希表和链接列表实现。是HashSet的子类。

存储特点：
　　相对有序存储，不可以存储相同元素（排重），通过链表实现的集合（注定相对有序）

LinkedHashSet集合的元素排重与HashSet集合排重方法一致。

1.3.6 TreeSet集合

TreeSet集合是可以给元素进行重新排序的一个Set接口的实现。使用元素的自然顺序对元素进行排序，或者根据创建 set 时提供的Comparator进行排序，具体取决于使用的构造方法。

存储特点：
　　相对无序存储，排重，通过二叉树实现的集合，可以给元素进行重新排序

1.3.7 SortedSet接口

TreeSet除了实现了Set接口外，还实现了SortedSet接口

> SortedSet接口中常用的方法：

方法名	描述
first()	返回此 set 中当前第一个（最低）元素。
last()	返回此 set 中当前最后一个（最高）元素。
headSet(E toElement)	返回此 set 的部分视图，其元素严格小于toElement。
tailSet(E fromElement)	返回此 set 的部分视图，其元素大于等于fromElement。
subSet(E fromElement, E toElement)	返回此 set 的部分视图，其元素从 fromElement（包括）到 toElement（不包括）。

1.3.8 TreeSet集合的元素排序

自然排序

元素所属的类需要实现java.lang.Comparable接口，并重写compareTo方法。

compareTo方法除了可以进行排序外，还有排重的功能，但是必须在compareTo方法中对类中所有的属性值都进行判断，否则不比较那个属性，排重就会忽略哪个属性

> 案例：设计一个Person类，实现将Person对象添加到TreeSet集合中时，对所有的元素进行排序

代码实现：

```java
public class Person implements Comparable<Person> {
    private String name;
    private int age;
    public Person() {
        super();
    }
    public Person(String name, int age) {
        super();
        this.name = name;
        this.age = age;
    }
    @Override
    public String toString() {
        return "Person [name=" + name + ", age=" + age + "]";
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
    @Override
    //重写compareTo方法，按照年龄升序，若年龄相同按照姓名降序排序
    public int compareTo(Person o) {
        //Person中有两个属性，此方法中必须对name和age两个属性都进行比较，否则将无法正确排重
        if(this.age != o.age) {
            return this.age - o.age;
        }else {
            return o.name.compareTo(this.name);
        }
    }
}
```

> 定制排序

元素需要通过java.util.Comparator接口（比较器）中的compare方法进行比较大小，并排序。

compare方法除了可以进行排序外，还有排重的功能，但是必须在compare方法中对类中所有的属性值都进行判断，否则不比较那个属性，排重就会忽略哪个属性

TreeSet集合中的无参数构造方法默认使用自然排序的方式对元素进行排序，使用TreeSet集合的定制排序时，创建集合对象不可以直接使用无参数构造方法，需要使用传入一个Comparator比较器的构造方法创建集合对象。对于此操作，我们有两种操作方式：

> 第一种方式：元素所属的类实现Comparator接口，创建集合对象时，传入一个元素所属类的对象作为比较器

代码实现：

```java
import java.util.Comparator;
/*
元素所属的类实现Compararor接口，重写compare方法
*/
public class Animal implements Comparator<Animal> {
    private String name;
    private int age;
    @Override
    public String toString() {
        return "Animal [name=" + name + ", age=" + age + "]";
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
    public Animal(String name, int age) {
        super();
        this.name = name;
        this.age = age;
    }
    public Animal() {
        super();
    }
    @Override
    public int compare(Animal o1, Animal o2) {
        if(o1.age!=o2.age) {
            return o2.age - o1.age;
        }else {
            return o1.name.compareTo(o2.name);
        }
    }
}
public class AnimalDemo{
    public static void main(String[]args){
        //创建一个TreeSet集合，传入一个Comparator比较器，Animal是Comparator实现类
        TreeSet<Animal> treeSet = new TreeSet<>(new Animal());//参数的Animal对象充当比较器
        //添加元素
        treeSet.add(new Animal("大黄", 1));
        treeSet.add(new Animal("旺财", 2));
        //遍历集合
        Iterator<Animal> it = treeSet.iterator();
        while(it.hasNext()){
            System.out.println(it.next());
```

```
        }
    }
}
```

> 第二种方式：元素所属的类不实现Comparator接口，创建集合对象是使用Comparator接口的匿名内部类

代码实现：

```
public class Animal {
    private String name;
    private int age;
    @Override
    public String toString() {
        return "Animal [name=" + name + ", age=" + age + "]";
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
    public Animal(String name, int age) {
        super();
        this.name = name;
        this.age = age;
    }
    public Animal() {
        super();
    }
}
public class AnimalDemo{
    public static void main(String[]args){
        //创建一个TreeSet集合，使用Comparator接口的匿名内部类的匿名对象作为比较器
        TreeSet<Animal> treeSet = new TreeSet<>(new Comparator() {
            @Override
            public int compare(Animal o1, Animal o2) {
                if(o1.age!=o2.age) {
                    return o2.age - o1.age;
                }else {
                    return o1.name.compareTo(o2.name);
                }
            }
        });
        //添加元素
        treeSet.add(new Animal("大黄", 1));
        treeSet.add(new Animal("旺财", 2));
        //遍历集合
        Iterator<Animal> it = treeSet.iterator();
        while(it.hasNext()){
            System.out.println(it.next());
        }
    }

}
```

第二节 **Map**接口

2.1 概述

Map接口是将键映射到值的对象。一个映射不能包含重复的键；每个键最多只能映射到一个值。

2.2 Map接口的常用方法

方法名	描述
clear()	从此映射中移除所有映射关系（可选操作）。
containsKey(Object key)	如果此映射包含指定键的映射关系，则返回 true。
containsValue(Object value)	如果此映射将一个或多个键映射到指定值，则返回 true。
entrySet()	返回此映射中包含的映射关系的 Set 视图。
equals(Object o)	比较指定的对象与此映射是否相等。
get(Object key)	返回指定键所映射的值；如果此映射不包含该键的映射关系，则返回 null。
hashCode()	返回此映射的哈希码值。
isEmpty()	如果此映射未包含键-值映射关系，则返回 true。
keySet()	返回此映射中包含的键的 Set 视图。
put(K key, V value)	将指定的值与此映射中的指定键关联（可选操作）。
putAll(Map<? extends K,? extends V> m)	从指定映射中将所有映射关系复制到此映射中（可选操作）。
remove(Object key)	如果存在一个键的映射关系，则将其从此映射中移除（可选操作）。
size()	返回此映射中的键-值映射关系数。

2.3 Map常用实现类

2.3.1 HashMap

> 基于哈希表的`Map`接口的实现。此实现提供所有可选的映射操作，并允许使用`null`值和`null`键。此类不保证映射的顺序。
>
> 存储特点：
> 　　相对无序存储，元素以键值对形式存在，键不可以重复，值可以重复，元素整体排重，可以快速的通过键查找到所对应的值，通过哈希表实现的集合。
>
> `Map`集合的排重，只需要重写键所属的类的`hashCode`和`equals`方法即可。

代码实现：

```java
//Person作为键
public class Person {
    private String name;
    private int age;
    public Person() {
        super();
    }
    public Person(String name, int age) {
        super();
        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 + "]";
    }
    @Override
    public int hashCode() {
        return age + name.length();
    }
    @Override
    public boolean equals(Object obj) {
        if(obj instanceof Person) {
            Person p = (Person) obj;
            if(p.name.equals(name)&&p.age==age) {
                return true;
            }
        }
        return false;
    }
}
//Dog类作为值
public class Dog {
    private String name;
    public Dog(String name) {
        super();
        this.name = name;

    }
```

```
    public Dog() {
        super();
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    @Override
    public String toString() {
        return "Dog [name=" + name + "]";
    }
}
//测试类
public class Demo {
    public static void main(String[] args) {
        HashMap<Person, Dog> map = new HashMap<>();
        map.put(new Person("zhangsan", 12), new Dog("大黄"));
        map.put(new Person("lisi", 12), new Dog("旺财"));
        map.put(new Person("zhangsan", 12), new Dog("二哈"));
        System.out.println(map.get("zhangsan"));//通过键找到对应的值
        Set<Person> set = map.keySet();//获取map集合中所有的键
        Collection<Dog> c = map.values();//获取map集合中所有的值
    }
}
```

map集合中若向集合中添加相同键的键值对时，新的值会将旧的值覆盖。
上述代码中map集合中有两个键值对，分别为：张三-12---二哈，lisi-12---旺财

2.3.2 LinkedHashSet

LinkedHashSet集合是具有可预知迭代顺序的Set接口的哈希表和链接列表实现。此实现与HashSet的不同之外在于，后者维护着一个运行于所有条目的双重链接列表。用法与HashSet类似。

存储特点：
相对有序存储，元素排重，通过链表实现的集合。

2.3.3 Hashtable

此类实现一个哈希表，该哈希表将键映射到相应的值。任何非null对象都可以用作键或值。

存储特点：
相对无序存储，元素排重，通过哈希表实现的集合。

2.3.4 HashMap与Hashtable的区别

1）方法一样，使用一样
2）Hashtable中的方法都是线程安全的，而HashMap中的方法是非线程安全的
3）Hashtable中不允许存在null的键和值，但是HashMap中允许null的键和值

2.4 Map集合的遍历

2.4.1 使用keySet方法与get方法结合

```
public class Demo {
    public static void main(String[] args){
        HashMap<String, Integer> map = new HashMap<>();
        map.put("aaa", 12);
        map.put("bbb", 13);
        map.put("ccc", 14);
        //通过keySet获取map中所有键
        Set<String> set = map.keySet();
        //获取set的迭代器
        Iterator<String> it = set.iterator();
        while(it.hasNext()){
            String s = it.next();
            //通过迭代的键，找到对应的值，一起输出
            System.out.println(s+"---"+map.get(s));
        }
    }
}
```

2.4.2 使用Map.Entry方法：

调用Map集合的entrySet方法，相当于将Map集合转成一个Set集合，再通过Set集合的遍历方式遍历即可。

代码实现：

```
public class Demo {
    public static void main(String[] args) {
        HashMap<String, Integer> map = new HashMap<>();
        map.put("aaa", 111);
        map.put("bbb", 222);
        map.put("ccc", 333);
        //将map转成一个Set集合
        Set<Map.Entry<String, Integer>> set = map.entrySet();
        //遍历set
        Iterator<Map.Entry<String, Integer>> it = set.iterator();
        while(it.hasNext()) {
            System.out.println(it.next());
        }
    }
}
```

第三节 Collections工具类

此类完全由在 collection 上进行操作或返回 collection 的静态方法组成。对集合进行操作时，可以使用这个类中的静态方法。

3.1 Collections中常用方法

> 1.同步线程

```
/*
        static <T> Collection<T>
    synchronizedCollection(Collection<T> c)
          返回指定 collection 支持的同步（线程安全的）collection。
static <T> List<T>
 synchronizedList(List<T> list)
          返回指定列表支持的同步（线程安全的）列表。
static <K,V> Map<K,V>
 synchronizedMap(Map<K,V> m)
          返回由指定映射支持的同步（线程安全的）映射。
static <T> Set<T>
 synchronizedSet(Set<T> s)
          返回指定 set 支持的同步（线程安全的）set。
        */
        Collection c = Collections.synchronizedCollection(new ArrayList());
        List s = Collections.synchronizedList(new ArrayList());
        Map m = Collections.synchronizedMap(new HashMap());
```

2.排序

```
/*
        static <T extends Comparable<? super T>>
void sort(List<T> list)
          根据元素的自然顺序 对指定列表按升序进行排序。
        */
        ArrayList<Integer> list = new ArrayList<>();
        list.add(-10);
        list.add(5);
        list.add(3);
        list.add(7);

        System.out.println(list);
        Collections.sort(list);//默认升序
        System.out.println(list);
```

3.将集合中的元素进行反转

```
        /*
        static void reverse(List<?> list)
         反转指定列表中元素的顺序。
        */
        Collections.reverse();
```

4.将集合元素随机输出

```
        /*
        static void shuffle(List<?> list)
         使用默认随机源对指定列表进行置换。
        */
```

5.获取集合中的最值

```
        /*
        static T max(Collection coll)
         根据元素的自然顺序，返回给定 collection 的最大元素。

          static <T extends Object & Comparable<? super T>>
        T min(Collection<? extends T> coll)
         根据元素的自然顺序 返回给定 collection 的最小元素。
        */
        int n1 = Collections.max(list);
```

6.替换

```
        /*
        static <T> boolean
    replaceAll(List<T> list, T oldVal, T newVal)
         使用另一个值替换列表中出现的所有某一指定值
        */
        //原因：List集合是不排重的，使用新的元素将集合中出现的所有的旧的元素替换掉
        Collections.replaceAll(list,5,100);
```

7.统计指定元素在集合中出现的次数

```
        /*
        static int frequency(Collection<?> c, Object o)
         返回指定 collection 中等于指定对象的元素数。
        */
        int num = Collections.frequency(list,5);
```

> 8.二分法查找

```
        /*
        static <T> int
binarySearch(List<? extends Comparable<? super T>> list, T key)
         使用二分搜索法搜索指定列表，以获得指定对象。
        */
        //前提：必须是排好序的集合
        int index = Collections.binarySearch(list,-10);

        //注意：Collections工具类中的方法只操作Collection接口，主要操作的是List接口
```

总结

课前默写

```
1.实例化一个ArrayList对象  ArrayList<String> list = new ArrayList<>()
2.简述ArrayList和LinkedList之间的区别
```

作业

1.按照要求完成下面的题目
1>创建一个List，在List中增加三个工人，基本信息如下：
姓名 年龄 工资
zhang3 18 3000
li4 25 3500
wang5 22 3200
2> 在li4之前插入一个工人，信息为：姓名：zhao6，年龄：24，工资3300
3> 删除wang5 的信息
4> 利用for 循环遍历，打印List 中所有工人的信息
5> 利用迭代遍历，对List中所有的工人调用work 方法。
6> 为工人类Worker重写equals方法，当姓名、年龄、工资全部相等时候才返回true

2.创建一个Student类，有成员变量name和cardId。如果两个学生对象的姓名和学号一样视为同一个学生，在HashSet中添加学生对象 并遍历打印学生信息。

3.向TreeSet集合中加入5个员工的对象，根据员工的年龄（升序）进行排序，若年龄相同，再根据工龄（降序）来排序，若工龄相同，根据薪水（降序）排序

面试题

1.简述HashSet和LinkedHashSet之间的区别