集合

一、泛型

（一）为什么要使用泛型

1.需求：我们需要一个箱子，可以放入一个对象，也可以取出对象。

解决办法：

```
public class Box {
    private Object object;
//  从箱子中取对象
    public Object getObject() {
        return object;
    }
// 往箱子中存对象
    public void setObject(Object object) {
        this.object = object;
    }
}

public class BoxTest {
    public static void main(String[] args) {
//        创建一个Box对象
        Box box1 = new Box();
        box1.setObject("hello");            //往箱子对象中存放数据或对象
        String str = (String) box1.getObject();     //从箱子中取出存储的对象，必须要进行强制类型转化
        box1.setObject(13);
        int i = (Integer) box1.getObject();

        String str1 = (String)box1.getObject();
        System.out.println(str1);   //因为强制类型转化过程中，类型不匹配，导致异常的发生
        //Exception in thread "main" java.lang.ClassCastException:
java.lang.Integer //cannot be cast to java.lang.String
//  at com.baizhi.java.BoxTest.main(BoxTest.java:12)
    }
}
```

总结：往Box对象中取对象时每次都要进行强制类型转化，而且强制类型转化可能出现问题或异常。

如何解决?

解决思路：给箱子贴上一个标签，要求往箱子中存放的对象必须是标签声明的类型，取对象就不需要进行强制类型转化了。

具体实现使用泛型。

java5.0中引入了泛型，在实例化箱子的时候可以“贴上标签”，规定箱子必须放入对象的类型、取出的对象类型，这样就无需强制类型转换了

案例：

```
public class Box1<T> {
     T t;       //T 必须和<T>声明的类型一致

    public T getT() {
        return t;
    }

    public void setT(T t) {
        this.t = t;
    }
}

//        创建一个箱子，该箱子上贴上是一个String类型的标签，表示该箱子只能存放字符串类型的对象
        Box1<String> box1 = new Box1<String>();
        box1.setT("hello");
//        box1.setT(12);         //编译出错，类型没有和泛型定义的类型保持一致。
//        取出对象时，不在需要进行强制类型转化了，使用起来更加方便了
        String str1 = box1.getT();
        System.out.println(str1);
```

（二）泛型的类型擦除

概念：泛型我们可以看作是一种标签，泛型并不会影响原对象的时机类型。在程序允许期间泛型类型会被擦除。

Java泛型只存在于编译期，运行期类型参数被擦除，如果类型参数无上届，就擦除为Object类型，如果有上届，就擦除为上届，比如 会被擦除为Animal类型

例：

```
public class Animal {

}
class Dog extends Animal{

}
class Cat extends Animal{

}
//泛型类型中<T> 相当于<T extends Object>
public class Box1<T> {
     T t;       //T 必须和<T>声明的类型一致

    public T getT() {
        return t;
    }

    public void setT(T t) {
        this.t = t;
    }
}
public class BoxTest1 {
    public static void main(String[] args) {

        Box1<Dog> dogBox1 = new Box1<>();
        dogBox1.setT(new Dog());
```

```
        Dog dog = dogBox1.getT();

        Box1<Animal> animalBox1 = new Box1<>();
//          在运行期，狗箱子和动物箱子都是箱子，没有"类型参数 "的信息

        System.out.println(dogBox1.getClass() == animalBox1.getClass());
 //true
//          无论存放Dog的狗箱子，还是存放Animal的动物箱子，在程序允许期间，本质都是一样，都是箱子对象。
        System.out.println(dogBox1 instanceof Box1);
        System.out.println(animalBox1 instanceof Box1);
        //Java泛型只存在于编译期，运行期类型参数被擦除，由于运行期的类型参数被擦除，类型参数被擦除为Object类型
        // 所以可以通过其他的引用将任何类型的对象扔到dogBox中，通过了编译，运行期也没抛异常
        //          将原来贴上dog标签的箱子，还原成箱子对象，（撕掉标签）
        Box1 box1 = dogBox1;
        //撕掉标签后的箱子，本质上任然时箱子，此时可以往箱子中存放任何类型的对象。
        box1.setT(new Cat());
        box1.setT("hello");

    }
}
```

extend 限定泛型范围

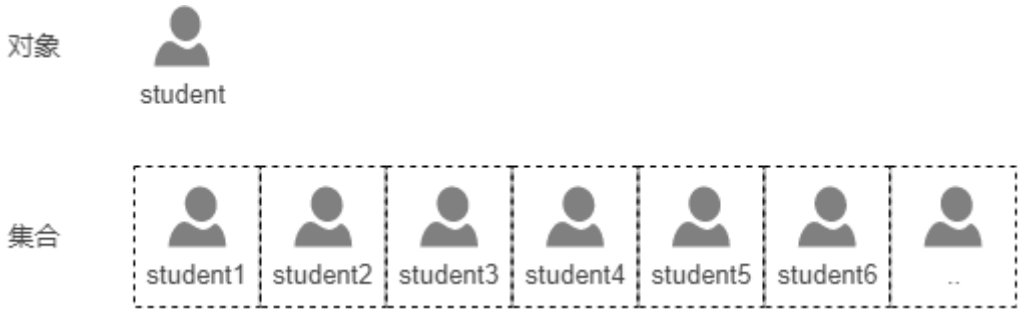
```
/**
 * 通过 T extends Animal 限定泛型的范围，表示该箱子只能存放Animal以及Animal的子类对象
 * @param <T>
 */
public class Box2<T extends Animal> {
    T t;

    public T getT() {
        return t;
    }

    public void setT(T t) {
        this.t = t;
    }
}
public class BoxTest2 {
    public static void main(String[] args) {
//          创建一个狗箱子
        Box2<Dog> dogBox2 = new Box2<>();
//          往狗箱子中存放Dog对象
        dogBox2.setT(new Dog());
//          类型擦除
        Box2 box2 = dogBox2;
        //编译出错，Box2箱子由类型限定 Box2<T extends Animal>,box2在进行类型擦除后，参数类型为Animal
//          box2.setT("hello");
//          如果泛型类型有上界，则类型擦除后泛型类型为上界。<T extends Animal> 会被擦除为Animal类型
        box2.setT(new Cat());       //可以编译通过，box2中此时只能存放Animal以及子类对象
    }
```

```
}
```

二、集合

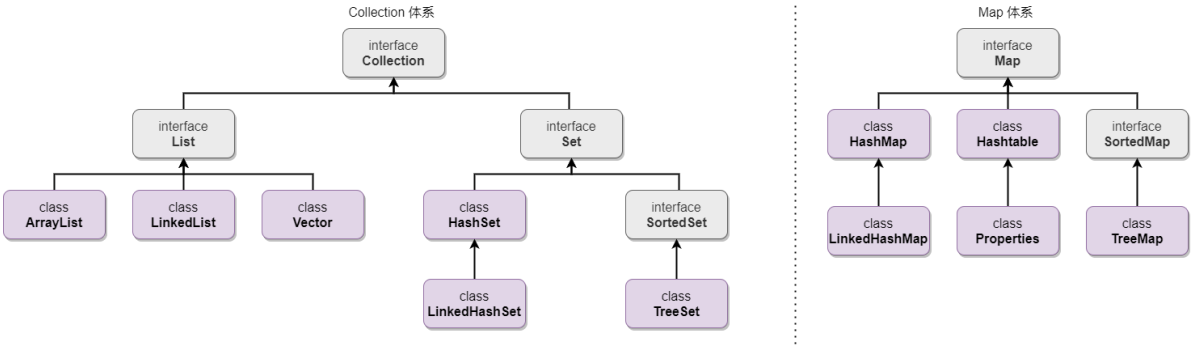
(一) 概述

作用：存储对象的容器，代替数组，使用起来比数组更加方便。



所在位置：java.util包

体系结构



(二) Collection接口

特点：内部的每个元素都是一个对象

常用的方法：

方法名	作用	返回值
add(Object o)	将对象o添加到集合中	`boolean`
contains(Object o)	判断集合中是否包含指定元素	boolean
remove(Object o)	将对象o从集合中移除	boolean
isEmpty()	判断集合中是否存在有效元素，返回为true,表示集合为空	boolean
size()	返回集合中元素的个数	int

三、List接口

特点：Collection接口的子接口，有序、有下标、元素可重复

List接口常用方法

方法名	作用	返回值
add(int index,Object o)	将元素o保持到集合的指定位置（不会覆盖原有元素）	boolean
remove(int index)	移除指定index下标的元素，（其他元素会向前移动一位）	Object(被移除的对象)
get(int index)	根据index返回值指定下标的元素。	E 指定下标的元素
subList(int beginIndex,int endIndex)	截取集合中元素，下标从beginIndex开始，到endIndex-1结束	List

四、List接口的实现类

（一）、ArrayList[重点]

特点:

```
1.内部由一个数组负责存储数据，通过下标访问元素
2.查询快，增删慢
3.线程不安全（线程章节补充）
```

总结:

1. ArrayList底层以数组方式实现，内存连续，根据下标查找元素效率比较高
2. ArrayList如何按照下标添加元素，和删除元素，因为涉及到元素的移动，效率比较低
3. 正常的添加操作，add(元素) 会默认将元素追加的集合末尾，效率比较高。

（二）、LinkedList

特点：

```
1.jdk1.2提供的实现类，操作速度快，线程不安全
2.内部由链表负责存储数据
3.查询慢，增删快
```

总结：

1. LinkedList底层以链表方式实现，内存不连续，查询效率比较低
2. LinkedList在进行按照下标添加元素和删除元素时，只需要将执行重新指向新的下一个位置即可，不需要移动集合中的元素，添加，删除的效率比较高

（三）、Vector（了解）

特点：

```
1.jdk1.0提供的实现类，操作速度慢，线程安全
2.内部也是由一个数组负责存储数据
3.查询快，增删慢
```

五、List集合中常用的方法

以ArrayList为例

```
public class Test1 {
    public static void main(String[] args) {
//        创建List集合对象
        List list = new ArrayList();
//        添加元素
        list.add("hello");
//        添加自定义对象
        list.add(new Animal());
//        根据下标移除指定下标的元素
```

```
        list.remove(1);
//          根据下标修改元素
        list.set(0,"world");
//        根据下标获取指定下标的元素
        String str = (String)list.get(0);
        System.out.println(str);
//        创建一个只能存储String类型的List集合，
        List<String> strList = new ArrayList<String>();
//        往使用泛型的集合中添加数据
        strList.add("hello");
//        strList.add(new Animal());   此时strList使用了泛型进行类型的约束，该集合只能存储String类型的数据
        String str1 = strList.get(0);   //使用泛型的集合，在获取元素时，就不用在进行强制类型转化了
        System.out.println(str1);
    }
}
```

六、遍历集合

（一）for循环遍历

特点：使用for循环的循环遍历充当下标，获取集合中每个元素，只能遍历List对应实现类的集合

规范：

```
for(int i=0;i<集合名.size();i++){
    //根据下标获取集合中的元素
    System.out.println(集合名.get(i));
}
```

案例：

```
    public static void main(String[] args) {
//        创建集合
        List<Student> studentList = new ArrayList<>();
//        创建学生对象
        Student stu1 = new Student("css",21,98.9);
        Student stu2 = new Student("lt",22,88.9);
        Student stu3 = new Student("pjf",20,95.9);
        Student stu4 = new Student("lfy",21,93);
        Student stu5 = new Student("hbh",23,97);
        Student stu6 = new Student("fs",22,96);
        Student stu7 = new Student("ssf",21,93);
        Student stu8 = new Student("hk",20,98);
        Student stu9 = new Student("lyb",21,92);
//        将学生添加到list集合中
        studentList.add(stu1);
        studentList.add(stu2);
        studentList.add(stu3);
        studentList.add(stu4);
        studentList.add(stu5);
        studentList.add(stu6);
        studentList.add(stu7);
        studentList.add(stu8);
        studentList.add(stu9);
```

```
//        使用普通for循环进行遍历
        for(int i=0;i< studentList.size();i++){
            Student stu = studentList.get(i);
            System.out.println(stu);
        }
    }
```

（二）迭代器遍历（Iterator）

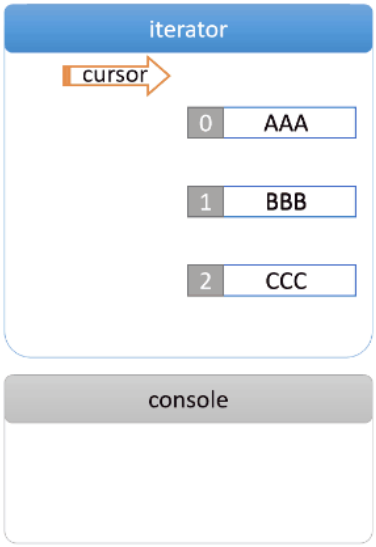
特点:内部使用游标遍历，无需使用下标，可以遍历Collection所有集合实现类，并保证遍历的完整性

规范:

```
Iterator<集合中存储的数据类型>  it = 集合名.iterator();
while(it.hasNext()){          //游标往下移动一位
 Object o = it.next();        //取出右边指向的一个对象
}
```

例:

```
List list = new ArrayList();
list.add("AAA");
list.add("BBB");
list.add("CCC");

Iterator iterator = list.iterator();

while( iterator.hasNext() ){

     Object obj = iterator.next();

     System.out.println( obj );

}
```



迭代器常用方法

方法名	作用	返回值
hasNext()	判断是否有下一个元素	boolean
next()	返回当前指向的元素	E 具体的元素
remove()	移除当前元素	void

案例代码

```
    public static void main(String[] args) {
        //        创建集合
        List<Student> studentList = new ArrayList<>();
//        创建学生对象
        Student stu1 = new Student("css",21,98.9);
        Student stu2 = new Student("lt",22,88.9);
```

```
        Student stu3 = new Student("pjf",20,95.9);
        Student stu4 = new Student("lfy",21,93);
        Student stu5 = new Student("hbh",23,97);
        Student stu6 = new Student("fs",22,96);
        Student stu7 = new Student("ssf",21,93);
        Student stu8 = new Student("hk",20,98);
        Student stu9 = new Student("lyb",21,92);
//        将学生添加到list集合中
        studentList.add(stu1);
        studentList.add(stu2);
        studentList.add(stu3);
        studentList.add(stu4);
        studentList.add(stu5);
        studentList.add(stu6);
        studentList.add(stu7);
        studentList.add(stu8);
        studentList.add(stu9);
//        使用迭代器进行遍历list集合
//        集合对象.iterator(); 返回一个迭代器对象
       Iterator<Student> it =  studentList.iterator();
//       使用while循环进行遍历
        while (it.hasNext()){       //判断是否有下一个元素，如果有返回值true,
            Student stu =  it.next();          //获取指向的对象
            System.out.println(stu);
        }
    }
```

（三）加强for遍历

jdk1.5提供的一种特殊的遍历方式，统一所有集合的遍历形式。

实现原理：底层使用的迭代器进行集合的遍历

语法：

```
for(元素类型 变量名: 集合(数组)){
    //每次循环都会从数组或集合中取出一个元素，赋值给变量
}
```

例：遍历数组

```
public class Test4 {
    public static void main(String[] args) {
//        定义数组
        int[] array = {12,32,43,23,12,32,14,76,47,9,47};
        for(int i : array ){    //每次循环从a数组中取出一个元素为变量i赋值
            System.out.println(i);
        }
    }
}
```

例：遍历list集合

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

```
        //        创建集合
        List<Student> studentList = new ArrayList<>();
//        创建学生对象
        Student stu1 = new Student("css",21,98.9);
        Student stu2 = new Student("lt",22,88.9);
        Student stu3 = new Student("pjf",20,95.9);
        Student stu4 = new Student("lfy",21,93);
        Student stu5 = new Student("hbh",23,97);
        Student stu6 = new Student("fs",22,96);
        Student stu7 = new Student("ssf",21,93);
        Student stu8 = new Student("hk",20,98);
        Student stu9 = new Student("lyb",21,92);
//        将学生添加到list集合中
        studentList.add(stu1);
        studentList.add(stu2);
        studentList.add(stu3);
        studentList.add(stu4);
        studentList.add(stu5);
        studentList.add(stu6);
        studentList.add(stu7);
        studentList.add(stu8);
        studentList.add(stu9);
//        使用forEach遍历
        for(Student stu:studentList){       //每次从studentList集合中取出一个对象赋值给stu遍历
            System.out.println(stu);
        }
    }
```

（四）forEach方法遍历

JDK8提供的一种遍历方法，可以配合Lambda表达式一起使用。

实现原理：内部使用迭代器进行遍历

方法声明：该方法需要传入一个Consumer接口的实现类对象

```
void froEach(Consumer action);
Consumer 接口
interface Consumer{
    void accept(T t);
}
```

实现Consumer接口，重写accept方法，参数为集合中元素，forEach方法进行遍历时，会自动向该方法传递参数。

例：匿名内部类实现

```
    public static void main(String[] args) {
        //        创建集合
        List<Student> studentList = new ArrayList<>();
//        创建学生对象
        Student stu1 = new Student("css",21,98.9);
        Student stu2 = new Student("lt",22,88.9);
        Student stu3 = new Student("pjf",20,95.9);
```

```
        Student stu4 = new Student("lfy",21,93);
        Student stu5 = new Student("hbh",23,97);
        Student stu6 = new Student("fs",22,96);
        Student stu7 = new Student("ssf",21,93);
        Student stu8 = new Student("hk",20,98);
        Student stu9 = new Student("lyb",21,92);
//        将学生添加到list集合中
        studentList.add(stu1);
        studentList.add(stu2);
        studentList.add(stu3);
        studentList.add(stu4);
        studentList.add(stu5);
        studentList.add(stu6);
        studentList.add(stu7);
        studentList.add(stu8);
        studentList.add(stu9);
//        使用匿名内部类遍历集合
        Consumer<Student> con = new Consumer<Student>() {
            @Override
            public void accept(Student student) {
                System.out.println(student);
            }
        };
//        调用forEach方法，传入Consumer实现类对象
        studentList.forEach(con);

//        使用Lambda表达式遍历集合
        studentList.forEach(stu -> System.out.println(stu));
    }
```

七、List排序

概念：List集合默认有序（下标），需要根据list集合中元素的某个属性值进行排序

（一）Comparable排序

1.将list集合中的元素对用的类，要实现Coomparable接口，重写comparTo()方法

案例：

```
public class Student implements Comparable<Student>{
    private String name;
    private int age;
    private String gender;
    private Double score;

    public Student(String name, int age, String gender, double score) {
        this.name = name;
        this.age = age;
        this.gender = gender;
        this.score = score;
    }

    public Student() {
    }

    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 String getGender() {
        return gender;
    }

    public void setGender(String gender) {
        this.gender = gender;
    }

    public double getScore() {
        return score;
    }

    public void setScore(Double score) {
        this.score = score;
    }

    @Override
    public String toString() {
        return "Student{" +
                "name='" + name + '\'' +
                ", age=" + age +
                ", gender='" + gender + '\'' +
                ", score=" + score +
                '}';
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Student student = (Student) o;
        return age == student.age &&
                Double.compare(student.score, score) == 0 &&
                Objects.equals(name, student.name) &&
                Objects.equals(gender, student.gender);
    }

    @Override
    public int hashCode() {
        return Objects.hash(name, age, gender, score);
    }
//  在compareTo方法中，需要进行编写排序的规则
    @Override
```

```
    public int compareTo(Student student) {
//根据学生的成绩进行排序，
        return (int)((this.score - student.getScore())*10);
    }
}
```

2.使用Collections.sort(集合)进行排序

案例：

```
public class StudentTest1 {
    public static void main(String[] args) {
        List<Student> students = new ArrayList<>();
        students.add(new Student("师赛飞",21,"男",89.5));
        students.add(new Student("张宁波",20,"男",91.1));
        students.add(new Student("化柯",22,"男",82));
        students.add(new Student("李博",21,"男",75));
        students.add(new Student("高正",20,"男",86));
        students.add(new Student("侯迎坤",20,"男",78));
        for(Student stu:students){
            System.out.println(stu);
        }
        System.out.println("--------排序之后的元素-------");
//        Collections.sort(students);
        // Collections 集合的工具类，
        // sort()根据集合中元素进行按照条件进行排序，排序的条件依据集合元素重写compartTo方法的返回值。
        Collections.sort(students);

        for(Student stu:students){
            System.out.println(stu);
        }
    }
}
```

Comparable接口排序的缺点：因为使用该接口进行排序，集合中的元素类需要实现Comparable接口，重写comparTo()方法，导致原来的实体类造成污染。

（二）Comparator接口

概念：Comparator接口可以进行排序，但是不需要实体类进重写，

如何使用?

在排序时，可以将Comparator的一个匿名内部类对象，传递给Collection.sort(list,Comparator匿名内部类);

案例：

```
public class Student{
    private String name;
    private int age;
    private String gender;
    private Double score;
```

```
    public Student(String name, int age, String gender, double score) {
        this.name = name;
        this.age = age;
        this.gender = gender;
        this.score = score;
    }

    public Student() {
    }

    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 String getGender() {
        return gender;
    }

    public void setGender(String gender) {
        this.gender = gender;
    }

    public double getScore() {
        return score;
    }

    public void setScore(Double score) {
        this.score = score;
    }

    @Override
    public String toString() {
        return "Student{" +
                "name='" + name + '\'' +
                ", age=" + age +
                ", gender='" + gender + '\'' +
                ", score=" + score +
                '}';
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Student student = (Student) o;
        return age == student.age &&
```

```
                Double.compare(student.score, score) == 0 &&
                Objects.equals(name, student.name) &&
                Objects.equals(gender, student.gender);
    }

    @Override
    public int hashCode() {
        return Objects.hash(name, age, gender, score);
    }

}

public class StudentTest1 {
    public static void main(String[] args) {
        List<Student> students = new ArrayList<>();
        students.add(new Student("师赛飞",21,"男",89.5));
        students.add(new Student("张宁波",20,"男",91.1));
        students.add(new Student("化柯",22,"男",82));
        students.add(new Student("李博",21,"男",75));
        students.add(new Student("高正",20,"男",86));
        students.add(new Student("侯迎坤",20,"男",78));
        for(Student stu:students){
            System.out.println(stu);
        }
        System.out.println("--------排序之后的元素-------");
//通过Comparator匿名内部类进行排序，排序依据根据compare方法的返回值进行的
        Collections.sort(students, new Comparator<Student>() {
//              Comparator比较器，根据重写的compare方法进行排序
            @Override
            public int compare(Student student, Student t1) {
//                  根据学生的年龄进行排序
                return student.getAge() - t1.getAge();
            }
        });

        for(Student stu:students){
            System.out.println(stu);
        }
    }
}
```

注意：使用Comparator进行排序时，实体类不需要做任何操作，减少了对实体类的污染，只需要在排序时，调用方法Collections.sort(list集合，Comparator匿名内部类对象)，

排序规则，在Comparator匿名内部类中重写的compar方法中进行定义的。

八、set接口

特点：Collection子接口，无序，无下标，元素不可重复（无序，不可重复）

常用方法：Set接口中的方法均继承与Collection接口

方法名	作用	返回值
`add(E e)`	将元素e添加到set集合中	boolean
`contains(Object o)`	判断set集合中是否包含元素O	boolean
`isEmpty()`	判断set集合中是否包含元素，如果不包含返回true	boolean
`iterator()`	返回一个迭代器，通常进行遍历set	Iterator
`remove(Object o)`	将元素O从set集合中移除	boolean
size()	返回set集合中元素的数量	int

案例：Set集合常用方法

```
 public static void main(String[] args) {
//        1.创建set集合
        Set<String> set = new HashSet<>();
//        2.往set集合中存放元素
        set.add("zhangsan");
        set.add("lisi");
        set.add("wangwu");
        set.add("zhaoliu");
//        3.判断集合是否包含元素
        System.out.println(set.isEmpty());
        System.out.println(set.size());
//        4.移除元素：set集合没有下标，只能通过对象进行移除
        set.remove("zhangsan");
//      5.输出set集合中的元素数量
        System.out.println(set.size());
//        6.判断set集合中是否包含lisi字符串
        System.out.println(set.contains("lisi"));
    }
```

案例：set接口特点（无序，不可重复）

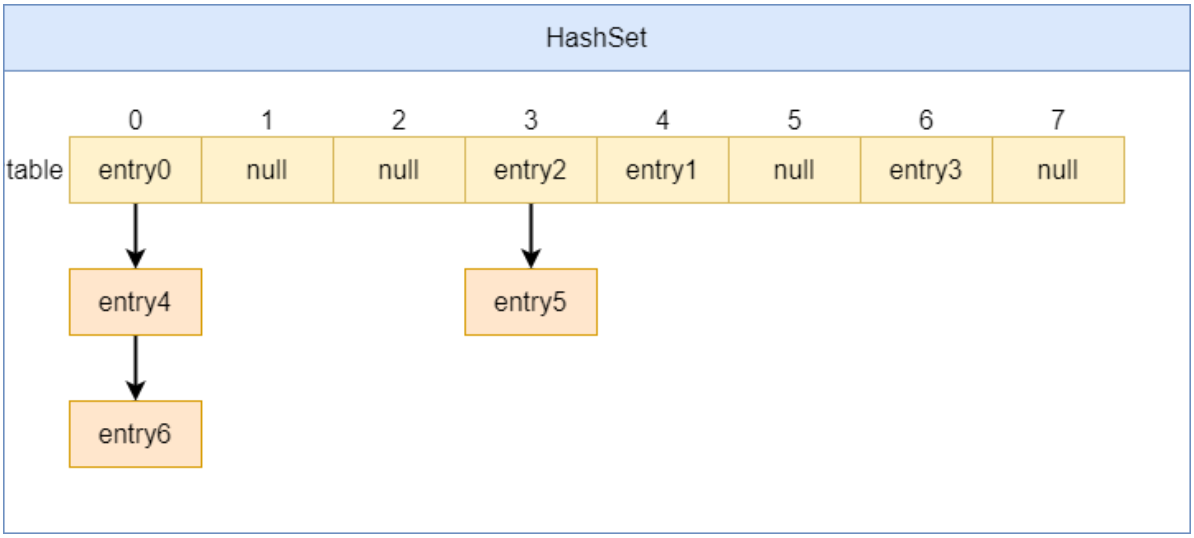
```
    public static void main(String[] args) {
//        1.创建set集合
        Set<String> set = new HashSet<>();
//        2.往set集合中存放元素
        set.add("zhangsan");
        set.add("lisi");
        set.add("wangwu");
        set.add("zhaoliu");
//        输出set集合的大小
        System.out.println(set.size());
        set.add("zhangsan");            //添加重复元素，重复元素不会真正的添加到set集合中，（不可重复型）
        System.out.println(set.size());
//      在进行遍历set集合时，遍历的顺序和元素添加的顺序不一致。（无序性）
        for(String str:set){
            System.out.println(str);
        }
    }
```

九、Set接口的实现类

（一）、HashSet

特点：有数组+链表进行存储，高效存取，内部的元素也被称为entry.

HashSet的存储结构：



hashCode(哈希码)，是一个尽量唯一的整数标识，可以用来区分对象，例如生活中的身份证号，hashCode由程序员进行设计，并没有具体的算法，但是要遵循以下原则，每个相同对象拥有相同的hashCode值,而不同对象尽量 拥有不同的hashCode值。（两个对象的不同，hashCode值一定不同的，两个对象hashCode值相同，则两个对象不一定相同）

案例：测试HahsCode方法

```
public class Student {
 private String name;
 private int age;
 private double score;

 public Student() {
 }

 public Student(String name, int age, double score) {
     this.name = name;
     this.age = age;
     this.score = score;
 }

 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 double getScore() {
        return score;
    }

    public void setScore(double score) {
        this.score = score;
    }

    @Override
    public String toString() {
        return "Student{" +
                "name='" + name + '\'' +
                ", age=" + age +
                ", score=" + score +
                '}';
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Student student = (Student) o;
        return age == student.age &&
                Double.compare(student.score, score) == 0 &&
                Objects.equals(name, student.name);
    }
// hashCode（）：重写Ojbet类中的方法，根据对象的属性值，重新生成新的hashCode值。
    @Override
    public int hashCode() {
        return Objects.hash(name, age, score);
    }
}
public static void main(String[] args) {
        Student stu1 = new Student("wangyx",16,100D);
        Student stu2 = new Student("yangdd",35,59.9);
        Student stu3 = new Student("chechangtong",21,89);
        Student stu4 = new Student("chechangtong",21,89);
        System.out.println(stu1.hashCode());
        System.out.println(stu2.hashCode());
//        不同对象，他们的属性值全部相同，对应的hashCode值也是相同的
        System.out.println(stu3.hashCode());        //1061597779
        System.out.println(stu4.hashCode());        //1061597779

    }
```

总结：内容相同的对象使用重写后的HashCode方法，会返回相同的hashcode值。

如果要向HashSet中添加自定义类型的对象时，该类需要重写HashCode方法和equal方法。

案例：

```
import java.util.Objects;

public class Student {
```

```
    private String name;
    private int age;
    private double score;

    public Student() {
    }

    public Student(String name, int age, double score) {
        this.name = name;
        this.age = age;
        this.score = score;
    }

    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 double getScore() {
        return score;
    }

    public void setScore(double score) {
        this.score = score;
    }

    @Override
    public String toString() {
        return "Student{" +
                "name='" + name + '\'' +
                ", age=" + age +
                ", score=" + score +
                '}';
    }

    @Override
    public int hashCode() {
        return Objects.hash(name, age, score);
    }
}

-- 测试代码
    public static void main(String[] args) {
        Student stu1 = new Student("wangyx",16,100D);
        Student stu2 = new Student("yangdd",35,59.9);
        Student stu3 = new Student("chechangtong",21,89);
        Student stu4 = new Student("chechangtong",21,89);
```

```
//        创建HashSet对象
        Set<Student> set = new HashSet<>();
        set.add(stu1);
        set.add(stu2);
        set.add(stu3);
        set.add(stu4);
        System.out.println(set.size());
//        当set集合中的元素没有重写Hahcode方法时，默认使用Object类中的hashcode
//        Object类中的hashcode方法，默认返回的是对象的内存地址，stu3,stu4内存地址不一样
//        如果Set集合中的元素重写了hashCode方法，则往set集合中添加元素时，会默认调用重写后的
hashcode方法，
//        重写后的hashCode方法会根据对象的属性值生成新的hashCode值，所以，当两个对象的所有属
性值全部相同时，
//        会返回相同的hashcode,HashSet集合就会根据对象的hashCode进行去重
        for(Student stu:set){
            System.out.println(stu);
        }
    }
```

元素属性值不同，但是hashCode相同的情况

案例：

```
    Student stu1 = new Student("柳柴",16,100D);
    Student stu2 = new Student("柴柳",16,100d);
    System.out.println(stu1.hashCode());          //1897947248
    System.out.println(stu1.hashCode());          //1897947248
```

因为hashcode算法，并不是完美的算法，所以会出现两个不同对象，拥有相同的hashCode值。此时调用HashSet的add方法，只会添加一个元素，为了保证不同的元素都可以添加到Set集合中，需要再次调用equals方法进行验证，而Object类中的equals方法默认比较的时对象的内存地址，并不能正对对象的属性进行验证，所以还需要重写equals方法。

```
import java.util.Objects;
public class Student {
    private String name;
    private int age;
    private double score;

    public Student() {
    }

    public Student(String name, int age, double score) {
        this.name = name;
        this.age = age;
        this.score = score;
    }

    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 double getScore() {
        return score;
    }

    public void setScore(double score) {
        this.score = score;
    }

    @Override
    public String toString() {
        return "Student{" +
                "name='" + name + '\'' +
                ", age=" + age +
                ", score=" + score +
                '}';
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Student student = (Student) o;
        return age == student.age &&
                Double.compare(student.score, score) == 0 &&
                Objects.equals(name, student.name);
    }

    @Override
    public int hashCode() {
        return Objects.hash(name, age, score);
    }
}
    public static void main(String[] args) {

        Student stu1 = new Student("柳柴",16,100D);
        Student stu2 = new Student("柴柺",16,100d);
        System.out.println(stu1.hashCode());
        System.out.println(stu1.hashCode());
        HashSet<Student> set = new HashSet<>();
        set.add(stu1);
        set.add(stu2);
        for(Student stu:set){
            System.out.println(stu);
        }
        stu1.equals(stu2);
    }
```

总结：

HashSet添加元素的依据
1.比较两个对象的hashCode值，如果hashCode值相同，调用对象的equals方法
2.如果equals方法返回true表示两个对象为相同对象，不会进行添加，如果equals方法返回false表示不同对象，都会添加，

向HashSet集合添加元素，HashSet会先调用元素中的hashCode方法，如果hashCode于其中的元素HashCode相同，则会再次调用equals方法验证两个元素是否真正的相同。
为了确保HashSet能够真正的去重，请为元素重写hashCode和equals方法。

为什么不直接使用equals方法进行验证两个对象是否重复？
因为equlas方法验证比较严谨的，步骤较多，所以效率较低，hashcode只是一个整数，比较整数效率高很多。

（二）、LinkedHashSet

特点：去重机制和HashSet相同的，但是可以维护元素的添加顺序

案例：

```
    public static void main(String[] args) {
        Student stu1 = new Student("wangyx",16,100D);
        Student stu2 = new Student("yangdd",35,59.9);
        Student stu3 = new Student("chechangtong",21,89);
        Student stu4 = new Student("gaozheng",21,89);
//        创建LinkedHashSet对象
        Set<Student> set = new LinkedHashSet<>();
        set.add(stu1);
        set.add(stu2);
        set.add(stu3);
        set.add(stu4);
//        遍历set集合
        for(Student stu:set){
            System.out.println(stu);
        }
      //  输出结果：默认和添加元素的顺序一致的
//Student{name='wangyx', age=16, score=100.0}
//Student{name='yangdd', age=35, score=59.9}
//Student{name='chechangtong', age=21, score=89.0}
//Student{name='gaozheng', age=21, score=89.0}
    }
```

（三）、TreeSet

特点：可以自动对集合中的元素进行排序，元素必须实现Comparable接口

去重机制：根据Comparable接口中重写的compareTo方法的返回值进行去重，如果返回值为0 表示相同对象。

compareTo方法的实现规范：遵循同异原则，当equals方法返回为true时，compareTo方法返回0，当equals方法返回值false时，compareTo方法返回非0，

例：Student实现Comparable接口

```
public class Student implements Comparable<Student>{
```

```
    private String name;
    private int age;
    private double score;

    public Student() {
    }

    public Student(String name, int age, double score) {
        this.name = name;
        this.age = age;
        this.score = score;
    }

    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 double getScore() {
        return score;
    }

    public void setScore(double score) {
        this.score = score;
    }

    @Override
    public String toString() {
        return "Student{" +
                "name='" + name + '\'' +
                ", age=" + age +
                ", score=" + score +
                '}';
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Student student = (Student) o;
        return age == student.age &&
                Double.compare(student.score, score) == 0 &&
                Objects.equals(name, student.name);
    }

    @Override
    public int hashCode() {
```

```
        return Objects.hash(name, age, score);
    }

    //实现compareTo方法，如果name相同，验证age,如果age相同验证score,排序规则亦如此
    @Override
    public int compareTo(Student student) {
        int result = name.compareTo(student.getName());
        if(result ==0){
            result  = age-student.age;
        }
        if(result==0){
            result = (int)((score-student.getScore())*10);
        }

        return result;
//        根据age属性值进行排序
//        return this.age - student.getAge();
    }
}


 public static void main(String[] args) {
        Student stu1 = new Student("wangyx",16,100D);
        Student stu2 = new Student("yangdd",35,59.9);
        Student stu3 = new Student("gaozheng",21,89);
        Student stu4 = new Student("chechangtong",21,84);
        Student stu5 = new Student("anglababy",22,69);
        Student stu6 = new Student("anglababy",22,61);
//      创建treeSet集合对象
        Set<Student>  set = new TreeSet<Student>();
        set.add(stu1);
        set.add(stu2);
        set.add(stu3);
        set.add(stu4);
        set.add(stu5);
        set.add(stu6);
//        遍历set集合
        for(Student stu:set){
            System.out.println(stu);
        }
    }
```

总结：TressSet的去重机制

根据元素compareTo方法的返回值进行去重和排序的，标准的实体，需要在compareTo方法中进行所有属性的比较。
如果只是在compareTo方法中根据某一个属性值进行排序，则也会根据该属性的值进行去重。不符合实际要求

十、Map体系结构

（一）Map接口

特点：一个元素由两个对象组成，分别称为key和Value, 无序，无下标，key不可以重复，值可以重复的，通过键访问值。

Map接口的特点与字典相同

例：

字典	
英文（key）	中文（value）
Like	欢喜
Happy	欢喜
Delete	删除
Remove	删除
…	…

除此之外，例如身份证号和姓名，经纬度和地点都具备这样的关系

（二）Map常用方法

方法名	作用	返回值
put(Object key,Object value)	向map集合中添加一组键值对，如果键以及存在，则覆盖	Value(不用)
get(key)	根据key返回value,如果key值不存在，返回null	Value
remove(key)	根据key移除一组键值对，	Value
containsKey(key)	判断map的key值中是否包含指定的值	boolean
containsValue(value)	判断value是否在map中存在	boolean
size()	返回map中键值对数量	int
keySet()	返回所有key值对应的set集合	Set<>
values()	返回所有的value()	Collection
entrySet()	返回所有的entry(键值对)对应的set集合	`Set<Map.Entry<K,V>>`

（三）使用方式

例：创建一个Map集合，存储对应的值

```
    public static void main(String[] args) {
//        创建一个Map集合,key为Integer类型，Value为String类型
        Map<Integer,String> map = new HashMap<>();
```

```
//        向map集合中添加元素
        map.put(99,"A");
        map.put(80,"B");
        map.put(70,"C");
        map.put(60,"D");
//        根据key获取Value
        String v1 = map.get(60);
        String v2 = map.get(65);    //如果key不存在，返回null
        System.out.println(v1);
        System.out.println(v2);    //null
//        根据key删除一组键值对
        String v3 = map.remove(60);
        System.out.println(v3);
//      获取map集合键值对数量
        System.out.println(map.size());
//        判断map中是否包含指定的key
        System.out.println(map.containsKey(60));        //false ,不包含
//        判断是否包含指定的值
        System.out.println(map.containsValue("A"));     //true ,包含
    }
```

十一、Map的遍历方式[重点]

（一）、键遍历

特点：通过Map中keySet方法，返回一个所有key的set集合，通过遍历set集合，获取所有的key

案例：

```
    public static void main(String[] args) {
        //        创建一个Map集合,key为Integer类型，Value为String类型
        Map<Integer,String> map = new HashMap<>();
//        向map集合中添加元素
        map.put(99,"A");
        map.put(80,"B");
        map.put(70,"C");
        map.put(60,"D");
//        键遍历：map.keySet()
        Set<Integer> keyset = map.keySet();
//        遍历keySet,
        for(Integer key:keyset){
            System.out.println(key+"\t"+map.get(key));
        }
    }
```

（二）、值遍历

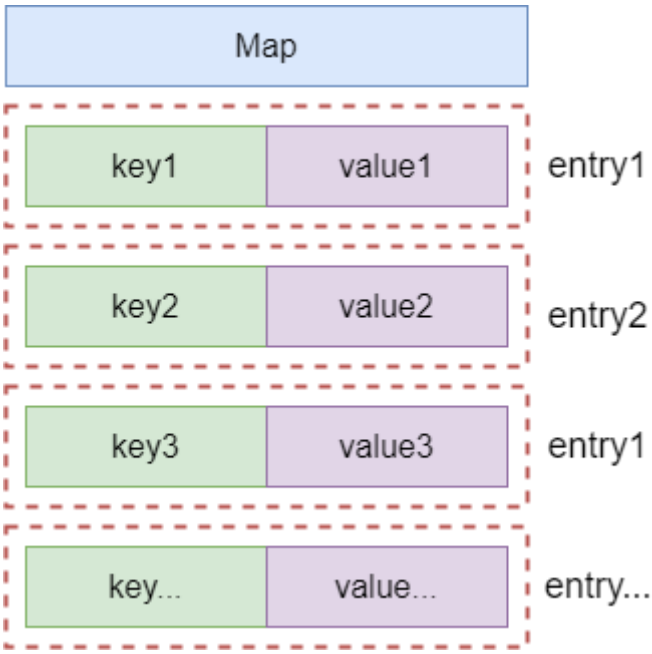
特点： 通过Map中的values()返回一个存储所有值的集合，遍历该集合取得所有的value,不能通过值遍历，获取对应的key

案例：

```
    public static void main(String[] args) {
        //        创建一个Map集合,key为Integer类型，Value为String类型
        Map<Integer,String> map = new HashMap<>();
//        向map集合中添加元素
        map.put(99,"A");
        map.put(80,"B");
        map.put(70,"C");
        map.put(60,"D");
//        值遍历：通过map.values()获取存储所有值的集合，通过遍历该集合获取对应的值
        Collection<String> cls =  map.values();
//            遍历集合中的所有值
        for(String str:cls){
            System.out.println(str);
        }
    }
```

（三）、键值对遍历

特点：通过Map中entrySet(),该方法会返回一个存储所有的entry(键值对)的set集合，遍历该集合获取所有的键值对。



案例：

```
    public static void main(String[] args) {
        //        创建一个Map集合,key为Integer类型，Value为String类型
        Map<Integer,String> map = new HashMap<>();
//        向map集合中添加元素
        map.put(99,"A");
        map.put(80,"B");
        map.put(70,"C");
        map.put(60,"D");
//        键值对遍历：通过map集合对象的entrySet(),返回一个包含所有马匹中键值对的set集合。
//        Map.Entry<Integer,String> 一个Map.Entry对象代表一个键值对
        Set<Map.Entry<Integer,String>> entrySet = map.entrySet();
//        遍历set集合
```

```
        for(Map.Entry<Integer,String> en:entrySet){
//            en.getkey():通过键值对对象.getkey()获取对应的key
//            en.getValue():通过键值对对象.getValue()获取对应的值
            System.out.println(en.getKey()+"\t"+en.getValue());
        }
    }
```

（四）、forEach方法遍历(了解)

特点：该方法需要一个BiComsumer接口的实现类，需要实现该接口accept()方法，accept方法中的两个参数分别代表Map集合中key和value

例：使用匿名内部类的方式实现，打印所有键值对

```
    public static void main(String[] args) {
        //        创建一个Map集合,key为Integer类型，Value为String类型
        Map<Integer,String> map = new HashMap<>();
//        向map集合中添加元素
        map.put(99,"A");
        map.put(80,"B");
        map.put(70,"C");
        map.put(60,"D");
//        使用匿名内部类，该匿名内部类要实现BiConsumer接口
        map.forEach(new BiConsumer<Integer, String>() {
            @Override
            public void accept(Integer integer, String s) {
                System.out.println(integer+"\t"+s);
            }
        });
//        Lambda表达式实现，打印所有的键值对
        map.forEach((a,b)-> System.out.println(a+"\t"+b));
    }
```

十二、Map常用的实现类

（一）、HashMap【重点】

特点：

```
1.JDK1.2 开始提供，操作速度快，线程不安全
2.允许null作为key或者value的值，但是作为key的值只能出现一次
3.使用hashcode和equals方法进行去重，如果key为自定义类型，要求该类型重写hashcode和equals方法，保证去重成功。
```

案例：

```
public class Student{
    private String name;
    private int age;
```

```
    private String gender;
    private Double score;

    public Student(String name, int age, String gender, double score) {
        this.name = name;
        this.age = age;
        this.gender = gender;
        this.score = score;
    }

    public Student() {
    }

    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 String getGender() {
        return gender;
    }

    public void setGender(String gender) {
        this.gender = gender;
    }

    public double getScore() {
        return score;
    }

    public void setScore(Double score) {
        this.score = score;
    }

    @Override
    public String toString() {
        return "Student{" +
                "name='" + name + '\'' +
                ", age=" + age +
                ", gender='" + gender + '\'' +
                ", score=" + score +
                '}';
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
```

```
        if (o == null || getClass() != o.getClass()) return false;
        Student student = (Student) o;
        return age == student.age &&
                Double.compare(student.score, score) == 0 &&
                Objects.equals(name, student.name) &&
                Objects.equals(gender, student.gender);
    }

    @Override
    public int hashCode() {
        return Objects.hash(name, age, gender, score);
    }
}



public static void main(String[] args) {
//        定义一个Map集合，key为学生对象，value为学生的成绩
       Student stu1 =  new Student("师赛飞",21,"男",89.5);
       Student stu2 =  new Student("张宁波",20,"男",91.1);
       Student stu3 =  new Student("化柯",22,"男",82);
       Student stu4 =  new Student("李博",21,"男",75);
       Student stu5 =  new Student("高正",20,"男",86);
       Student stu6 =  new Student("侯迎坤",20,"男",78);
       Student stu7 =  new Student("侯迎坤",20,"男",78);
//HashMap去重依据key元素对应的hashCode值，
        Map<Student,Double> map = new HashMap<>();
        map.put(stu1,stu1.getScore());
        map.put(stu2,stu2.getScore());
        map.put(stu3,stu3.getScore());
        map.put(stu4,stu4.getScore());
        map.put(stu5,stu4.getScore());
        map.put(stu6,stu6.getScore());
        map.put(stu7,stu7.getScore());
        System.out.println(map.size());     //结果6，Map中key元素重写hashCode和
equals方法
    }
```

(二)、HashTable

特点:

```
1.jdk1.0 线程安全 操作速度慢
2.不允许null作为key和value的值，会触发空指针异常
3.去重依据也是根据HashCode值进行去重，key为自定义类型的元素，则该元素需要重写hashcode和
equals方法
```

例:

```
    public static void main(String[] args) {
        //        定义一个Map集合，key为学生对象，value为学生的成绩
        Student stu1 =  new Student("师赛飞",21,"男",89.5);
        Student stu2 =  new Student("张宁波",20,"男",91.1);
        Student stu3 =  new Student("化柯",22,"男",82);
```

```
        Student stu4 =  new Student("李博",21,"男",75);
        Student stu5 =  new Student("高正",20,"男",86);
        Student stu6 =  new Student("侯迎坤",20,"男",78);
        Student stu7 =  new Student("侯迎坤",20,"男",78);
//HashMap去重依据key元素对应的hashCode值，
        Map<Student,Double> map = new Hashtable<>();
        map.put(stu1,stu1.getScore());
        map.put(stu2,stu2.getScore());
        map.put(stu3,stu3.getScore());
        map.put(stu4,stu4.getScore());
        map.put(stu5,stu4.getScore());
        map.put(stu6,stu6.getScore());
        map.put(stu7,stu7.getScore());
        System.out.println(map.size());     //6  key会根据元素的hashcode值进行去重

//        不允许将null作为key值
//        map.put(null,23d);      //hashtable作为线程安全对象，不允许null作为key
        //hashtable作为线程安全对象，不允许null作为value
        map.put(new Student("李峰阳",20,"男",89),null);
        System.out.println(map.size());
    }
Exception in thread "main" java.lang.NullPointerException
```

面试题：简述HashMap和Hashtable的区别

```
HashMap是非线程安全对象，效率较高，可以将null作为key或value的值
Hashtable是线程安全对象，效率低，不允许将null作为key或value的值
```

（三）、Properties

特点：

```
1.Hashtable的子类，主要用于配置文件的读取，可以将.properties文件加载为key-value结构的数据
2.所有元素都为String类型
```

案例：

```
 public static void main(String[] args) {
//        创建Properties属性对象
        Properties properties = new Properties();
//        往属性对象中添加数据，
        properties.put("userName","zhangsan");
        properties.put("password","123456");
//        获取数据
//        根据key获取对应的value
        String userName = properties.getProperty("userName");
        String password = properties.getProperty("password");
        System.out.println(userName);
        System.out.println(password);

    }
```

（四）、TreeMap

特点：

```
1.SortedMap接口的实现类，可以对key进行自动排序
2.使用Comparable接口中的compareTo方法的返回值进行排序的，如果key为自定义类型，则要求可以元素对应的类必须实现Comparable接口，从小compareTo方法
3.TreeMap去重依据根据compareTo方法的返回值进行去重，
```

案例：按照学生的年龄进行排序，要求key为学生对象，value为学生对应的成绩

```
public class Student implements Comparable<Student>{
    private String name;
    private int age;
    private String gender;
    private Double score;

    public Student(String name, int age, String gender, double score) {
        this.name = name;
        this.age = age;
        this.gender = gender;
        this.score = score;
    }

    public Student() {
    }

    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 String getGender() {
        return gender;
    }

    public void setGender(String gender) {
        this.gender = gender;
    }

    public double getScore() {
        return score;
    }

    public void setScore(Double score) {
```

```
        this.score = score;
    }

    @Override
    public String toString() {
        return "Student{" +
                "name='" + name + '\'' +
                ", age=" + age +
                ", gender='" + gender + '\'' +
                ", score=" + score +
                '}';
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Student student = (Student) o;
        return age == student.age &&
                Double.compare(student.score, score) == 0 &&
                Objects.equals(name, student.name) &&
                Objects.equals(gender, student.gender);
    }

    @Override
    public int hashCode() {
        return Objects.hash(name, age, gender, score);
    }

//    重写Comparable接口中的compareTo方法，以age属性值进行排序
    @Override
    public int compareTo(Student student) {

        return this.age - student.getAge();
    }
}


public static void main(String[] args) {
        Student stu1 =  new Student("师赛飞",21,"男",89.5);
        Student stu2 =  new Student("张宁波",25,"男",91.1);
        Student stu3 =  new Student("化柯",22,"男",82);
        Student stu4 =  new Student("李博",29,"男",75);
        Student stu5 =  new Student("高正",23,"男",86);
//        因为在Student类中，compareTo方法中根据age属性值进行排序的，所有默认也会根据age属性值进行去重，
//        所以年龄相同的学生对象不会重复进行添加
        Student stu6 =  new Student("侯迎坤",22,"男",78);
        Student stu7 =  new Student("侯迎坤",22,"男",78);
//HashMap去重依据key元素对应的hashCode值，
        Map<Student,Double> map = new TreeMap<>();
        map.put(stu1,stu1.getScore());
        map.put(stu2,stu2.getScore());
        map.put(stu3,stu3.getScore());
        map.put(stu4,stu4.getScore());
        map.put(stu5,stu5.getScore());
        map.put(stu6,stu6.getScore());
        map.put(stu7,stu7.getScore());
```

```
//        遍历TreMap结合
        map.forEach((k,v)-> System.out.println(k+"\t"+v));
    }
```

案例：TreeMap的多重排序，先根据age,age相同根据name进行排序，name相同根据分数进行排序

```
import java.util.Objects;

public class Student implements Comparable<Student>{
    private String name;
    private int age;
    private String gender;
    private Double score;

    public Student(String name, int age, String gender, double score) {
        this.name = name;
        this.age = age;
        this.gender = gender;
        this.score = score;
    }

    public Student() {
    }

    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 String getGender() {
        return gender;
    }

    public void setGender(String gender) {
        this.gender = gender;
    }

    public double getScore() {
        return score;
    }

    public void setScore(Double score) {
        this.score = score;
    }

    @Override
```

```
    public String toString() {
        return "Student{" +
                "name='" + name + '\'' +
                ", age=" + age +
                ", gender='" + gender + '\'' +
                ", score=" + score +
                '}';
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Student student = (Student) o;
        return age == student.age &&
                Double.compare(student.score, score) == 0 &&
                Objects.equals(name, student.name) &&
                Objects.equals(gender, student.gender);
    }

    @Override
    public int hashCode() {
        return Objects.hash(name, age, gender, score);
    }

// 根据对象的所有属性值进行排序和去重
//    排序规则：先以年龄进行排序，age相同比较姓名，姓名相同比较分数
    @Override
    public int compareTo(Student student) {
        int result = this.age - student.getAge();
        if(result==0){
            result = this.name.compareTo(student.getName());
        }
        if(result==0){
            result = (int)((this.score - student.getScore())*10);
        }
        return result;
    }
}


import java.util.Map;
import java.util.TreeMap;

public class StudentTest {
    public static void main(String[] args) {
        Student stu1 =  new Student("师赛飞",21,"男",89.5);
        Student stu2 =  new Student("张宁波",25,"男",91.1);
        Student stu3 =  new Student("化柯",22,"男",82);
        Student stu4 =  new Student("李博",29,"男",75);
        Student stu5 =  new Student("高正",23,"男",86);
//        因为在Student类中，compareTo方法中根据age属性值进行排序的，所有默认也会根据age属性值进行去重，
//        所以年龄相同的学生对象不会重复进行添加
        Student stu6 =  new Student("侯迎坤",22,"男",78);
        Student stu7 =  new Student("侯迎坤",22,"男",90);
//HashMap去重依据key元素对应的hashCode值，
        Map<Student,Double> map = new TreeMap<>();
```

```
        map.put(stu1,stu1.getScore());
        map.put(stu2,stu2.getScore());
        map.put(stu3,stu3.getScore());
        map.put(stu4,stu4.getScore());
        map.put(stu5,stu5.getScore());
        map.put(stu6,stu6.getScore());
        map.put(stu7,stu7.getScore());
//          遍历TreMap结合
        map.forEach((k,v)-> System.out.println(k+"\t"+v));
    }
}
```

（五）、LinkedHashMap

特点：

```
1.和HashMap特点相同
2.可以保持元素的添加顺序
```

案例：

```
    public static void main(String[] args) {
        Student stu1 =  new Student("师赛飞",21,"男",89.5);
        Student stu2 =  new Student("张宁波",25,"男",91.1);
        Student stu3 =  new Student("化柯",22,"男",82);
        Student stu4 =  new Student("李博",29,"男",75);
        Student stu5 =  new Student("高正",23,"男",86);
        Student stu6 =  new Student("侯迎坤",22,"男",78);
        Student stu7 =  new Student("侯迎坤",22,"男",90);
//      创建LinkedHashMap对象，
        Map<Student,Double> map = new LinkedHashMap<>();
        map.put(stu1,stu1.getScore());
        map.put(stu2,stu2.getScore());
        map.put(stu3,stu3.getScore());
        map.put(stu4,stu4.getScore());
        map.put(stu5,stu5.getScore());
        map.put(stu6,stu6.getScore());
        map.put(stu7,stu7.getScore());
//          遍历LinkedHashMap，
//          默认的遍历属性和添加元素的顺序保持一致。
        map.forEach((k,v)-> System.out.println(k+"\t"+v));
    }
```

十三、综合案例【了解】

需求：使用Map保存用户信息以及用对应的订单信息和订单详情

用户对象(User)：name,age,gender,

订单对象(order)：oid,orderName,date,totalPrice,proNum

商品对象(product)：pid,pname,price,address

对象关系分析：

一个用户可以有多个订单　　user　1: n　order

一个订单可以包含多个商品　　order 1:n product

1.如何体现用户和订单之间的关系?

　需要在用户的属性中添加一个List 表示一个用户对应多个订单

2.如何体现订单和商品之间的关系?

　需要在订单类中添加一个属性 Set 表示一个订单对应多个商品

实体类

```
import java.util.List;

/**
 * 用户对象(User)：name,age,gender,
 */
public class User {
    private String name;
    private int age;
    private String gender;
//    在用户类中添加订单的List,体现一个用户对应多个订单。
    private List<Order> orders;

    public User() {
    }

    public User(String name, int age, String gender) {
        this.name = name;
        this.age = age;
        this.gender = gender;
    }

    public User(String name, int age, String gender, List<Order> orders) {
        this.name = name;
        this.age = age;
        this.gender = gender;
        this.orders = orders;
    }

    public List<Order> getOrders() {
        return orders;
    }

    public void setOrders(List<Order> orders) {
        this.orders = orders;
    }

    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 String getGender() {
        return gender;
    }

    public void setGender(String gender) {
        this.gender = gender;
    }

    @Override
    public String toString() {
        return "User{" +
                "name='" + name + '\'' +
                ", age=" + age +
                ", gender='" + gender + '\'' +
                ", orders=" + orders +
                '}';
    }
}

/**
 * 订单对象(order)：oid,orderName,date,totalPrice,proNum
 */
public class Order {
    private Integer oid;
    private String orderName;
    private String date;
    private double totalPrice;
    private int proNum;
//    在Order类上添加对应的Product的set集合，表示一个订单对应多个商品
    private Set<Product> productSet;
    public Order() {
    }

    public Order(Integer oid, String orderName, String date, double totalPrice,
int proNum) {
        this.oid = oid;
        this.orderName = orderName;
        this.date = date;
        this.totalPrice = totalPrice;
        this.proNum = proNum;
    }

    public Order(Integer oid, String orderName, String date, double totalPrice,
int proNum, Set<Product> productSet) {
        this.oid = oid;
        this.orderName = orderName;
        this.date = date;
        this.totalPrice = totalPrice;
```

```
        this.proNum = proNum;
        this.productSet = productSet;
    }

    public Set<Product> getProductSet() {
        return productSet;
    }

    public void setProductSet(Set<Product> productSet) {
        this.productSet = productSet;
    }

    public Integer getOid() {
        return oid;
    }

    public void setOid(Integer oid) {
        this.oid = oid;
    }

    public String getOrderName() {
        return orderName;
    }

    public void setOrderName(String orderName) {
        this.orderName = orderName;
    }

    public String getDate() {
        return date;
    }

    public void setDate(String date) {
        this.date = date;
    }

    public double getTotalPrice() {
        return totalPrice;
    }

    public void setTotalPrice(double totalPrice) {
        this.totalPrice = totalPrice;
    }

    public int getProNum() {
        return proNum;
    }

    public void setProNum(int proNum) {
        this.proNum = proNum;
    }

    @Override
    public String toString() {
        return "Order{" +
                "oid=" + oid +
                ", orderName='" + orderName + '\'' +
                ", date='" + date + '\'' +
```

```
                ", totalPrice=" + totalPrice +
                ", proNum=" + proNum +
                ", productSet=" + productSet +
                '}';
    }
}

/**
 * 商品对象(product)：pid,pname,price,address
 */
public class Product {
    private Integer pid;
    private String pname;
    private double price;
    private String address;

    public Product() {
    }

    public Product(Integer pid, String pname, double price, String address) {
        this.pid = pid;
        this.pname = pname;
        this.price = price;
        this.address = address;
    }

    public Integer getPid() {
        return pid;
    }

    public void setPid(Integer pid) {
        this.pid = pid;
    }

    public String getPname() {
        return pname;
    }

    public void setPname(String pname) {
        this.pname = pname;
    }

    public double getPrice() {
        return price;
    }

    public void setPrice(double price) {
        this.price = price;
    }

    public String getAddress() {
        return address;
    }

    public void setAddress(String address) {
        this.address = address;
    }
```

```
    @Override
    public String toString() {
        return "Product{" +
                "pid=" + pid +
                ", pname='" + pname + '\'' +
                ", price=" + price +
                ", address='" + address + '\'' +
                '}';
    }
}


public class UserTest {
    public static void main(String[] args) {
//        为订单创建对应的商品
            Product p1 = new Product(1001,"恰恰香瓜子",8.9,"新疆");
            Product p2 = new Product(1002,"三只松鼠夏威夷果",35,"云南");
            Product p3 = new Product(1003,"迪迦奥特曼玩具",99,"郑州");
            Product p4 = new Product(1004,"良品铺子",30.9,"浙江");
            Product p5 = new Product(1005,"ipad",3999,"深圳");
            Product p6 = new Product(1006,"HUAWEI",4999,"深圳");
            Product p7 = new Product(1007,"小米",2999,"深圳");
            Product p8 = new Product(1008,"Vivo",1999,"深圳");

//        在为两个对象创建对应的订单
            Order o1 = new Order(10001,"我的小零食","2023-1-4",200,5);
//            给订单绑定对应的商品
                Set<Product> set1 = new HashSet<>();
                set1.add(p1);
                set1.add(p2);
                set1.add(p4);
                o1.setProductSet(set1);
            Order o2 = new Order(10002,"我的电子产品","2023-1-2",20000,3);
                Set<Product> set2 = new HashSet<>();
                set2.add(p3);
                set2.add(p5);
                set2.add(p6);
                o2.setProductSet(set2);
            Order o3 = new Order(10003,"生日礼物","2023-1-1",10000,3);
                Set<Product> set3 = new HashSet<>();
                set3.add(p7);
                set3.add(p8);
        //        先创建两个对象
            User user1 = new User("安建龙",21,"男");
//            给用户绑定对应的订单信息
                List<Order> orders1 = new ArrayList<>();
                orders1.add(o1);
                orders1.add(o2);
                user1.setOrders(orders1);

            User user2 = new User("车畅通",22,"男");
            List<Order> orders2 = new ArrayList<>();
            orders2.add(o3);
            user2.setOrders(orders2);


        System.out.println(user1);
    }
```

```
}


public class UserTest1 {
    public static void main(String[] args) {
        //        为订单创建对应的商品
        Product p1 = new Product(1001,"恰恰香瓜子",8.9,"新疆");
        Product p2 = new Product(1002,"三只松鼠夏威夷果",35,"云南");
        Product p3 = new Product(1003,"迪迦奥特曼玩具",99,"郑州");
        Product p4 = new Product(1004,"良品铺子",30.9,"浙江");
        Product p5 = new Product(1005,"ipad",3999,"深圳");
        Product p6 = new Product(1006,"HUAWEI",4999,"深圳");
        Product p7 = new Product(1007,"小米",2999,"深圳");
        Product p8 = new Product(1008,"Vivo",1999,"深圳");

        Order o1 = new Order(10001,"我的小零食","2023-1-4",200,5);
        Order o2 = new Order(10002,"我的电子产品","2023-1-2",20000,3);
        Order o3 = new Order(10003,"生日礼物","2023-1-1",10000,3);

        User user1 = new User("安建龙",21,"男");
        User user2 = new User("车畅通",22,"男");
//        定义用来保存用户的信息的map对象，key为用户对象，value为用户对应的订单列表
        Map<User, List<Map<Order,Set<Product>>>> map = new HashMap<>();
//            订单1
        Map<Order,Set<Product>> orderMap1 = new HashMap<>();
        Set<Product> set1 = new HashSet<>();
        set1.add(p1);
        set1.add(p2);
        orderMap1.put(o1,set1);
//        订单2
        Map<Order,Set<Product>> orderMap2 = new HashMap<>();
        Set<Product> set2 = new HashSet<>();
        set2.add(p3);
        set2.add(p4);
        orderMap2.put(o2,set2);
//        将多个订单封装为list
        List<Map<Order,Set<Product>>> list1 = new ArrayList<>();
        list1.add(orderMap1);
        list1.add(orderMap2);

        map.put(user1,list1);

    }
}
```