day06【Collections、Set、Map、斗地主排序】

今日内容

- Collections工具类
  - 常用方法
  - 比较器
  - 可变参数
- Set集合
  - 概述
  - HashSet集合
  - HashSet存储数据的结构
  - HashSet存储自定义类型元素
  - LinkedHashSet存储自定义类型元素
  - TreeSett存储自定义类型元素
- Map集合
  - 概述
  - 常用子类
  - 常用方法
  - Map集合遍历
  - HashMap存储自定义类型
  - LInkedHashMap存储自定义类型
  - TreeMap集合
  - 练习
- 集合的综合案例

教学目标

- [ ] 能够使用集合工具类
- [ ] 能够使用Comparator比较器进行排序
- [ ] 能够使用可变参数
- [ ] 能够说出Set集合的特点
- [ ] 能够说出哈希表的特点
- [ ] 使用HashSet集合存储自定义元素
- [ ] 能够说出Map集合特点、常用的方法
- [ ] 使用Map集合添加方法保存数据
- [ ] 使用"键找值"的方式遍历Map集合
- [ ] 使用"键值对"的方式遍历Map集合
- [ ] 能够使用HashMap存储自定义键值对的数据
- [ ] 能够完成斗地主洗牌发牌案例

第一章 Collections类

知识点--Collections常用功能

目标

- 掌握工具类Collections的使用

路径

- 概述
- 常用方法
- 演示Collections工具的使用

讲解

1.1.1概述

`java.utils.Collections`是集合工具类，用来对集合进行操作。

1.1.2常用方法

- `public static void shuffle(List<?> list)`:打乱集合顺序。
- `public static <T> void sort(List<T> list)`:将集合中元素按照默认规则排序。

1.1.3演示Collections的工具的使用

需求：演示集合工具类中的功能

//测试类代码

```
public class Test {
    public static void main(String[] args) {
        //创建List集合
        List<String> list = new ArrayList<>();
        //添加数据
        list.add("a");
        list.add("b");
        list.add("c");
        System.out.println(list);
        //public static void shuffle(List<?> list)
        Collections.shuffle(list);
        System.out.println(list);
        // public static <T> void sort(List<T> list)
        Collections.sort(list);
        System.out.println(list);
    }
}
//结果
[a, b, c]
[c, b, a]
[a, b, c]
```

小结

知识点--Comparator比较器

目标

- 掌握Comparator比较器使用

路径

- 概述
- 常用方法
- 演示Comparator比较器使用

讲解

1.2.1概述

`public interface Comparator<T>` 定义比较对象规则的接口

1.2.2常用方法

```
public int compare(T  o1,T  o2) 比较用来排序的两个参数
    o1一般代表正着添加的元素，o2表示已经添加过的元素。根据第o1与o2的比较结果，返回负整数、零或正整数实现排序。
    默认使用方式
        升序    根据o1的值与o2的值做比较(o1-o2)
        降序    根据o2的值与o1的值做比较(o2-o1)
```

1.2.3 应用场景

Collections工具类中: `public static <T> void sort(List<T> list, Comparator<? super T> )`:将集合中元素按照指定规则排序。

1.2.4演示Comparator比较器的应用

需求：演示Comparator在集合工具类排序功能中的使用

//Comparator实现类

```
public class MyComparator implements Comparator<Integer> {

    @Override
    public int compare(Integer o1, Integer o2) {
        System.out.println("o1: " + o1+"---"+"o2: " + o2);
        //o2表示前面的那个数据
        //o1表示后面的那个数据

        //返回值
            // 返回的是0   不操作
            // 如果是负数   o1放到前面
            // 如果是正数   o2放到前面
```

```
        int num = o1 - o2;//升序
        // int num = o2 - o1;//降序
        return num;
    }
}
```

//测试类代码

```
public class Test {
    public static void main(String[] args) {
        List<Integer> list = new ArrayList<>();
        list.add(4);
        list.add(1);
        list.add(2);
        list.add(5);
        list.add(3);
        System.out.println(list);
        //进行排序
        // <T> void sort(List<T> list, Comparator<? super T> )
        MyComparator mc = new MyComparator();
        Collections.sort(list, mc);
        System.out.println(list);
    }
}
//结果
[4, 1, 2, 5, 3]
[3, 5, 2, 1, 4]
```

小结

```
```

知识点--可变参数

目标

- 掌握可变参数的使用

路径

- 概述
- 格式
- 注意事项
- 应用场景
- 演示可变参数应用场景

讲解

1.3.1

在JDK1.5之后，定义了可变参数，用来表示一个方法需要接受的多个同类型参数。

1.3.2 格式

```
参数类型... 形参名
```

1.3.3 注意事项

- 1.一个方法只能有一个可变参数
- 2.如果方法中有多个参数，可变参数要放到最后。

1.3.4应用场景

Collections工具类中的添加元素等方法：`public static <T> boolean addAll(Collection<T> c,T...elements)`:往集合中添加一些元素。

1.3.5演示可变参的使用

需求：演示可变参数在集合工具列的添加方法中的应用

//测试类代码

```
public class Test {
    public static void main(String[] args) {
        show(1, 2, 3, 4, 5);
        System.out.println("--------");
        show2(100, 1, 2, 3, 4, 5);
        System.out.println("--------");
        // public static <T> boolean addAll(Collection<T> c,T...elements)
        List<String> list = new ArrayList();
        list.add("a");
        list.add("b");
        list.add("c");
        System.out.println(list);
        System.out.println("--------");
        Collections.addAll(list, "d", "e", "f");
        System.out.println(list);
    }

    public static void show(int... arr) {
        System.out.println(Arrays.toString(arr));
    }

    public static void show2(int num, int... arr) {
        System.out.println(num);
        System.out.println(Arrays.toString(arr));
    }
    //如果方法中有多个参数，可变参数要放到最后。
    // public static void show3(int... arr,int num) {
    //     System.out.println(num);
    //     System.out.println(Arrays.toString(arr));
    // }
}
//结果
[1, 2, 3, 4, 5]
```

```
--------
100
[1, 2, 3, 4, 5]
--------
[a, b, c]
--------
[a, b, c, d, e, f]
```

小结

```
```

第二章 Set接口

知识点--Set接口概述

目标

- 了解Set集合的特点

路径

- Set集合概述
- Set接口特点
- Set集合常用子类

讲解

2.1.1Set集合概述

`java.util.Set`接口继承自Collection接口，是单列集合的一个重要分支。实现了Set`接口的对象称为Set集合。

Set集合没有对于Collection功能的额外扩充,但是有更加完善的存储机制。

Set集合无索引，只能是用增强for和迭代器遍历

2.1.2Set接口特点

- 元素无索引,元素存取无序,元素不可重复(唯一)

2.1.3Set集合常用子类

`java.util.HashSet`：哈希表结构集合

`java.util.LinkedHashSet`：链表结构集合

`java.util.TreeSet`：树结构集合

小结

知识点--HashSet基本使用

目标

- 理解HashSet集合的特点

路径

- 概述
- 特点
- 演示HashSet的使用

讲解

2.2.1概述

`java.util.HashSet`是`Set`接口的一个实现类

底层的实现其实是一个`java.util.HashMap`支持

根据对象的哈希值来确定元素在集合中的存储位置，具有良好的存储和查找性能

元素唯一，底层依赖`hashCode`与`equals`方法。

2.2.2特点

- 元素无索引,元素存取无序,元素不可重复(唯一)

2.2.3演示HashSet的使用

需求：在测试类中演示HashSet的基本使用

```
public class Test {
    public static void main(String[] args) {
        HashSet<String> hs = new HashSet<>();
        hs.add("a");
        hs.add("a");
        hs.add("a");
        hs.add("b");
        hs.add("c");
        System.out.println(hs);

        System.out.println("Aa".hashCode());
        System.out.println("BB".hashCode());
    }
}
//结果
[a, b, c]
2112
2112
```

小结

知识点--HashSet集合数据结构

目标

- 能够简单说明HashSet的底层结构

路径

- 数据结构
- 元素唯一代码原理
- 哈希表原理图
- 哈希表存储流程图

讲解

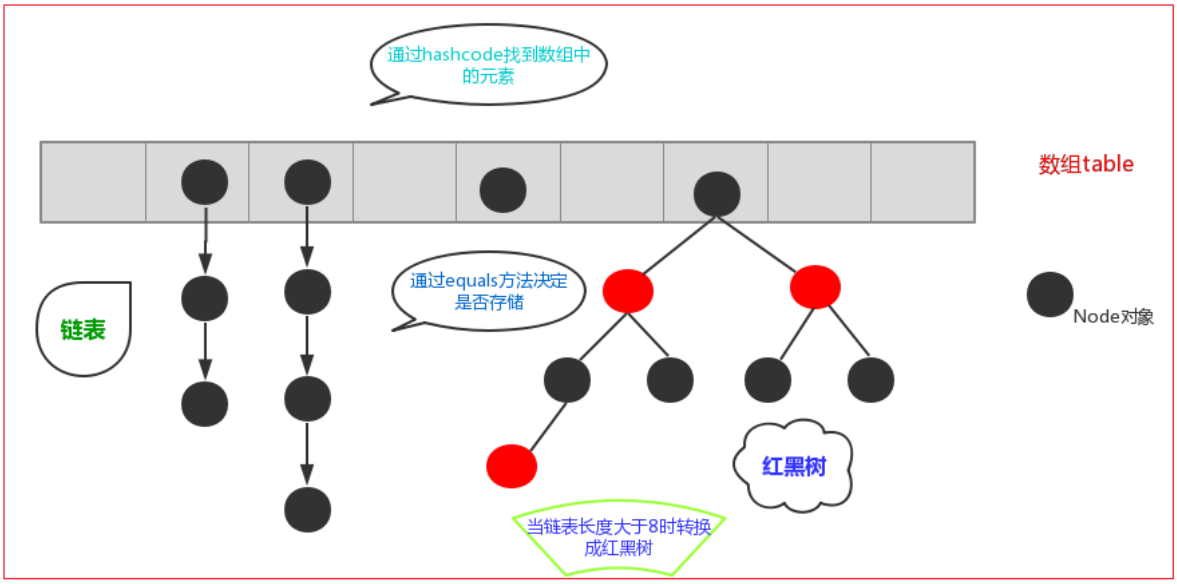
2.3.1数据结构

JDK1.8之前，哈希表底层采用数组+链表，
JDK1.8开始，哈希表存储采用数组+链表+红黑树。

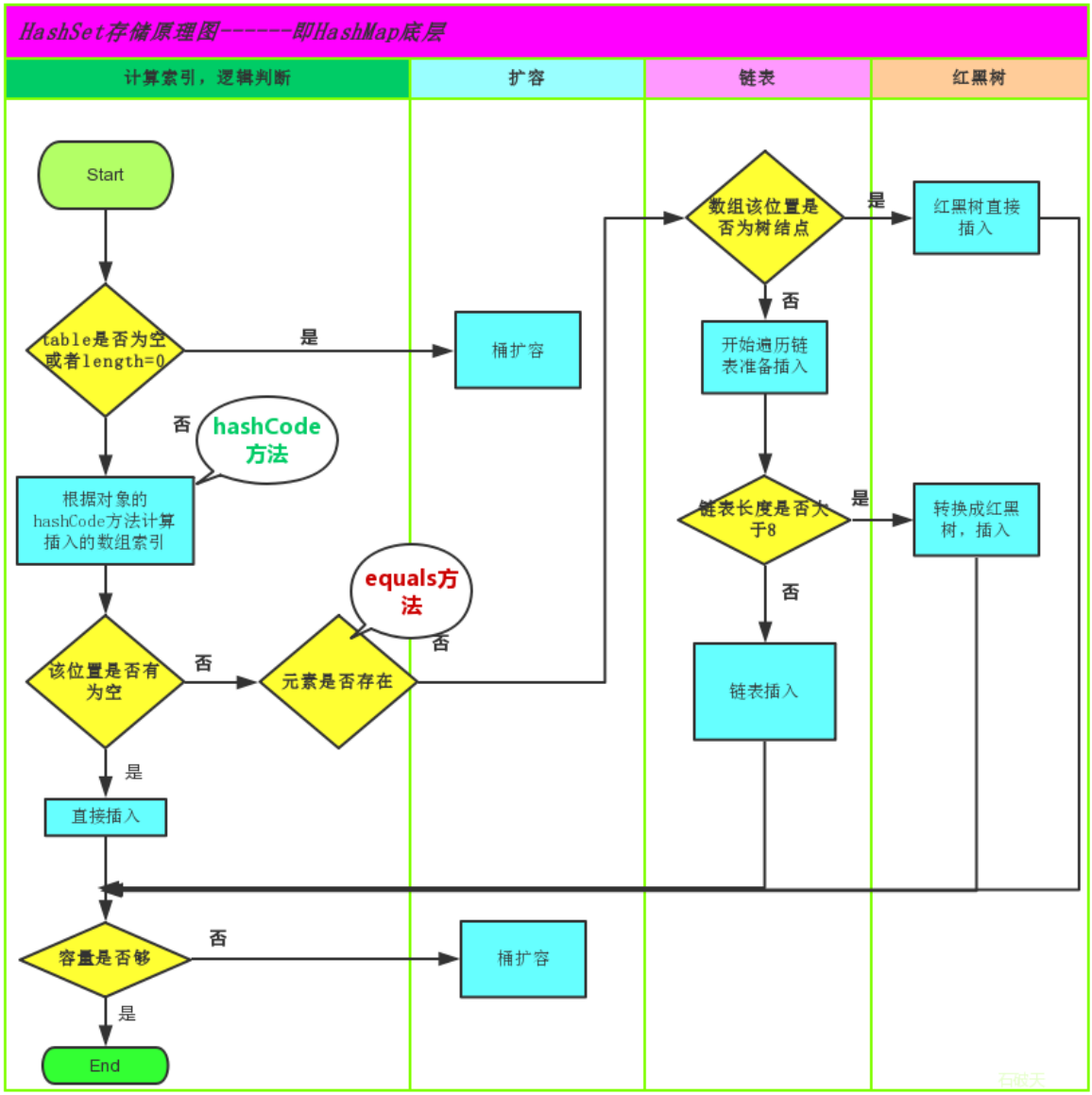
2.3.2元素唯一代码原理

- hashCode决定存储的列数(相同则一列)，equals是否存在相同元素
- jdk8为提高查询效率，当一列数据达到8个且总数据达到64个，则增加数组长度，重新排列数据

2.3.3哈希表原理图



2.3.3哈希表存储流程图



小结

知识点--HashSet存储自定义类型元素

目标

- 掌握HashSet存储自定义类型元素

路径

- 概述
- 演示HashSet存储自定义类型元素

讲解

2.4.1概述

- HashSet存储对象,是根据继承自Object类中的hashCode方法和equals方法的值进行判定存储
- 默认的hashCode和equals方法是通过地址值计算，在实际开发中我们一般需要重写对象这两个方法

2.4.2演示HashSet存储自定义类型元素

需求：通过HashSet存储自定义学生类型对象

//自定义学生类代码

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

    public Student() {
    }

    public Student(String name, int age) {
        this.name = name;
        this.age = age;
    }
    public Student(String name, int age,String flag) {
        this.name = name;
        this.age = age;
        this.flag=flag;
    }
    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 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 &&
                Objects.equals(name, student.name);
    }

    @Override
```

```
    public int hashCode() {
        return Objects.hash(name, age);//得到一个哈希值
    }

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

//测试类代码

```
public class Test {
    public static void main(String[] args) {
        //创建集合对象
        HashSet<Student> hs = new HashSet<>();
        //创建学生对象--学生类(重写hashCode和equals)
        Student s1 = new Student("张三", 18, "1");
        Student s2 = new Student("李四", 20);
        Student s3 = new Student("张三", 20);
        Student s4 = new Student("张三", 18, "2");
        //添加学生对象
        hs.add(s1);
        hs.add(s2);
        hs.add(s3);
        hs.add(s4);
        //查看数据
        for (Student s : hs) {
            System.out.println(s.getName() + "..." + s.getAge() + "..." +
s.flag);
        }
    }
}
//结果
张三...20...null
张三...18...1
李四...20...null
```

小结

```

```

知识点--LinkedHashSet存储自定义类型元素

目标

- 掌握LinkedHashSet使用及特点

路径

- 概述
- 特点
- 演示LinkedHashSet使用

讲解

2.5.1概述

`java.util.LinkedHashSet` 是HashSet的一个子类，底层采用链表+哈希表

LinkedHashSet类在保留HashSet元素唯一的基础上，增加了有序性

2.5.2特点

元素无索引,元素**存取有序**,元素不可重复(唯一)

- 保证元素唯一:由哈希表保证元素唯一,哈希表保证元素唯一依赖hashCode()和equals()方法
- 保证元素存取有序: 由链表保证元素存取有序

2.5.3演示LinkedHashSet使用

需求：通过LinkedHashSet存储自定义学生类型对象

//学生类代码

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

    public Student() {
    }

    public Student(String name, int age) {
        this.name = name;
        this.age = age;
    }
    public Student(String name, int age, String flag) {
        this.name = name;
        this.age = age;
        this.flag=flag;
    }
    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 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 &&
                Objects.equals(name, student.name);
    }

    @Override
    public int hashCode() {
        return Objects.hash(name, age);//得到一个哈希值
    }

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

//测试类代码

```
public class Test {
    public static void main(String[] args) {
        LinkedHashSet<Student> lhs = new LinkedHashSet<>();
        Student s1 = new Student("张三", 18);
        Student s2 = new Student("李四", 20);
        Student s3 = new Student("张三", 20);
        Student s4 = new Student("张三", 18);
        lhs.add(s1);
        lhs.add(s2);
        lhs.add(s3);
        lhs.add(s4);
        //查看数据
        for (Student s : lhs) {
            System.out.println(s.getName() + "..." + s.getAge() + "..." +
s.flag);
        }
    }
}
//结果
张三...18...null
李四...20...null
张三...20...null
```

小结

知识点--TreeSet存储自定义类型元素

目标

- 掌握TreeSet的使用及特点

路径

- 概述
- 特点
- 演示TreeSet的使用

讲解

2.6.1概述

`java.util.TreeSet` 是是Set接口的一个实现类,底层依赖于TreeMap,是一种基于**红黑树**的实现。

TreeSet集合存储的对象必须拥有排序规则(比较器)，否则会报出异常`cannot be cast to java.lang.Comparable`。

自然排序：

- 对象类实现 java.lang.Comparable接口，重写compare方法，使用TreeSet无参构造创建集合对象

比较器排序：

- 创建重写compare方法的Comparator接口实现类对象,使用TreeSet有参构造【TreeSet(Comparator comparator)】创建集合对象

> tips：以谁为优先排序，就把谁放前面

2.6.2特点

元素无索引,元素存取无序,元素不可重复(唯一)，元素可排序

2.6.3演示TreeSet存储的使用

需求：通过TreeSet存储自定义学生类和老师类型对象演示两种排序方式

//学生类代码

```
public class Student implements Comparable<Student> {
    String num;
    private String name;
    private int age;

    public Student() {
    }

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

```
    public Student(String num, String name, int age) {
        this.num = num;
        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 "Student{" +
                "num='" + num + '\'' +
                ", name='" + name + '\'' +
                ", age=" + age +
                '}';
    }

    @Override
    public int compareTo(Student s) {
        //this代表的是正则添加的元素
        //s 代表的是集合中已经存储的元素

        // 返回值:
            //正数   添加的放到 被比较元素的后面
            //0     代表两个内容相同，不添加
            //负数   添加的放到 被比较元素的前面
        //(重点记忆)升序   this  vs  s
        //(重点记忆)降序   s  vs  this
        // System.out.println("this:" + this);
        // System.out.println("s:" + s);
        //考虑到 要用过姓名和年龄综合排序，所以需要使用这两个内容得到最终的num值
        //考虑到 排序，有可能优先按照姓名排序，也有可能优先按照年龄排序
            //想用谁优先做比较，就把谁放到前面，优先比较差值
        //优先按照年龄排序
        /*
        int num = s.age-this.age;
        // num =(num==0)?s.name.compareTo(this.name):s.age-this.age;
        num =(num==0)?s.name.compareTo(this.name):num;
        return num;
        */
        // return  (s.age-this.age==0)?s.name.compareTo(this.name):s.age-
this.age;
```

```
        return  (s.name.compareTo(this.name)==0)?s.age-
this.age:s.name.compareTo(this.name);
    }
}
```

//老师类代码

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

    public Teacher() {
    }

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

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

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

//Comparable实现类

```
public class MyComprator implements Comparator<Teacher> {
    @Override
    public int compare(Teacher o1, Teacher o2) {
        // System.out.println("o1:" + o1);
        // System.out.println("o2:" + o2);
        //o1 正则添加的数据
        //o2 集合中已经添加的数据
        //(重点记忆)升序  o1  vs  o2
        //(重点记忆)降序  o2  vs  o1
```

```
        //升序
        //优先按照姓名排序
        int num = o1.getName().compareTo(o2.getName());
        num = (num == 0) ? o1.getAge() - o2.getAge() : num;
        return num;
    }
}
```

//测试类代码

```
public class Test {
    public static void main(String[] args) {
       /* // 自然排序
        TreeSet<Student> ts = new TreeSet<>();
        // Student s = new Student("张三", 18);
        // ts.add(s);
        //Student cannot be cast to class java.lang.Comparable
        ts.add(new Student("1", "zhangsan", 18));
        System.out.println("--------");
        ts.add(new Student("2", "lisi", 20));
        System.out.println("--------");
        ts.add(new Student("3", "zhangsan", 20));
        System.out.println("--------");
        ts.add(new Student("4", "zhangsan", 18));

        for (Student s : ts) {
            System.out.println(s);
        }*/

        // 比较器排序
        MyComprator mc = new MyComprator();
        TreeSet<Teacher> ts2 = new TreeSet<>( mc);
        ts2.add(new Teacher("zhangsan", 18));
        System.out.println("--------");
        ts2.add(new Teacher("lisi", 20));
        System.out.println("--------");
        ts2.add(new Teacher("zhangsan", 20));
        System.out.println("--------");
        ts2.add(new Teacher("zhangsan", 18));
        for (Teacher s : ts2) {
            System.out.println(s);
        }
    }
}
//结果
--------
--------
--------
Student{num='3', name='zhangsan', age=20}
Student{num='1', name='zhangsan', age=18}
Student{num='2', name='lisi', age=20}
--------
--------
--------
Teacher{name='lisi', age=20}
Teacher{name='zhangsan', age=18}
```

```
Teacher{name='zhangsan', age=20}
```

小结

```

```

第三章 Map集合

知识点--Map集合概述

目标

- 能够说出Map集合特点

路径

- Map集合概述
- Map集合与Collection集合区别
- 单列/双列集合原理图
- Map的常用子类介绍

讲解

3.1.1概述

`java.util.Map`双列集合的顶层接口，用来存储具备映射关系对象的集合接口定义。

映射关系：生活中类似`IP地址与主机名`、`身份证号与个人`等一一对应的对应关系。

Map集合中存储的内容根据映射关系分为两部分，称为键值对

- 键(Key)不能包含重复的键，
- 值(Value)可以重复；
- 每个键只能对应一个值。

3.1.2Map集合与Collection集合区别

- `Collection`集合中的元素，是以单个的形式存储。称为单列集合
- `Map`集合中的元素，是以键值对的形式存储。称为双列集合

3.1.3单列/双列集合原理图

3.1.4Map的常用子类介绍

HashMap<K,V>

- 存储数据采用的哈希表结构，元素的存取顺序不能保证一致。
- 由于要保证键的唯一、不重复，需要重写键的hashCode()方法、equals()方法。

LinkedHashMap<K,V>

- HashMap的子类，存储数据采用的哈希表结构+链表结构。
- 通过链表结构可以保证元素的存取顺序一致；
- 通过哈希表结构可以保证键唯一、不重复，需要重写键的hashCode()方法、equals()方法。

TreeMap<K,V>

- TreeMap集合和Map相比没有特有的功能，底层的数据结构是红黑树；
- 可以对元素的*键*进行排序，排序方式有两种:自然排序和比较器排序

> tips：Map接口中的集合都有两个泛型变量<K,V>,在使用时，要为两个泛型变量赋予数据类型。两个泛型变量<K,V>的数据类型可以相同，也可以不同。

小结

知识点--Map的常用方法

目标

- 掌握Map的常用方法

路径

- 常用方法
- 演示Map的常用方法

讲解

3.2.1常用方法

- `public V put(K key, V value)`：把指定的键与值添加到Map集合中。
  - 若键(key)在集合中不存在，添加指定的键和值到集合中，返回null；
  - 若键(key)在集合中存在，对应键的原值被新值替代，返回该键对应的原值。
- `public V remove(Object key)`：把指定的键对应的键值对元素在Map集合删除，返回被删除元素的值。
- `public V get(Object key)` 根据指定的键，在Map集合中获取对应的值。
- `public Set<K> keySet()`：获取Map集合中所有的键，存储到Set集合中。
- `public Set<Map.Entry<K,V>> entrySet()`：获取到Map集合中所有的键值对对象的集合(Set集合)。
- `public boolean containKey(Object key)`:判断该集合中是否有此键。

3.2.2演示Map的常用方法

需求：通过HashMap演示Map集合常用方法

//测试类代码

```
public class Test {
    public static void main(String[] args) {
        //创建一个Map集合对象
        Map<Integer, String> m = new HashMap<>();
        //public V put(K key, V value)
        // System.out.println(m.put(1, "a"));
        // System.out.println(m.put(1, "b"));
        m.put(1, "a");
        m.put(2, "b");
        m.put(3, "c");
        System.out.println(m);
        System.out.println("--------");
        // public V remove(Object key)
        System.out.println(m.remove(1));
        System.out.println(m.remove(4));
        System.out.println(m);
        System.out.println("--------");
        // public V get(Object key)
        System.out.println(m.get(1));
        System.out.println(m.get(2));
        System.out.println(m.get(3));
        System.out.println("--------");
        // public Set<K> keySet()
        Set<Integer> set = m.keySet();
        System.out.println(set);
        System.out.println("--------");
        // public Set<Map.Entry<K,V>> entrySet()
        Set<Map.Entry<Integer, String>> entries = m.entrySet();
```

```
        System.out.println(entries);
        System.out.println("--------");
        // public boolean containKey(Object key)
        System.out.println(m.containsKey(1));
        System.out.println(m.containsKey(2));
    }
}
//结果
{1=a, 2=b, 3=c}
--------
a
null
{2=b, 3=c}
--------
null
b
c
--------
[2, 3]
--------
[2=b, 3=c]
--------
false
true
```

小结

```

```

知识点-Map的遍历

目标

- 掌握Map集合的遍历

路径

- 方式1:键找值
- 方式:2键值对
- 演示两种遍历方式

讲解

3.3.1方式1:键找值

通过元素中的键，获取键所对应的值

步骤:

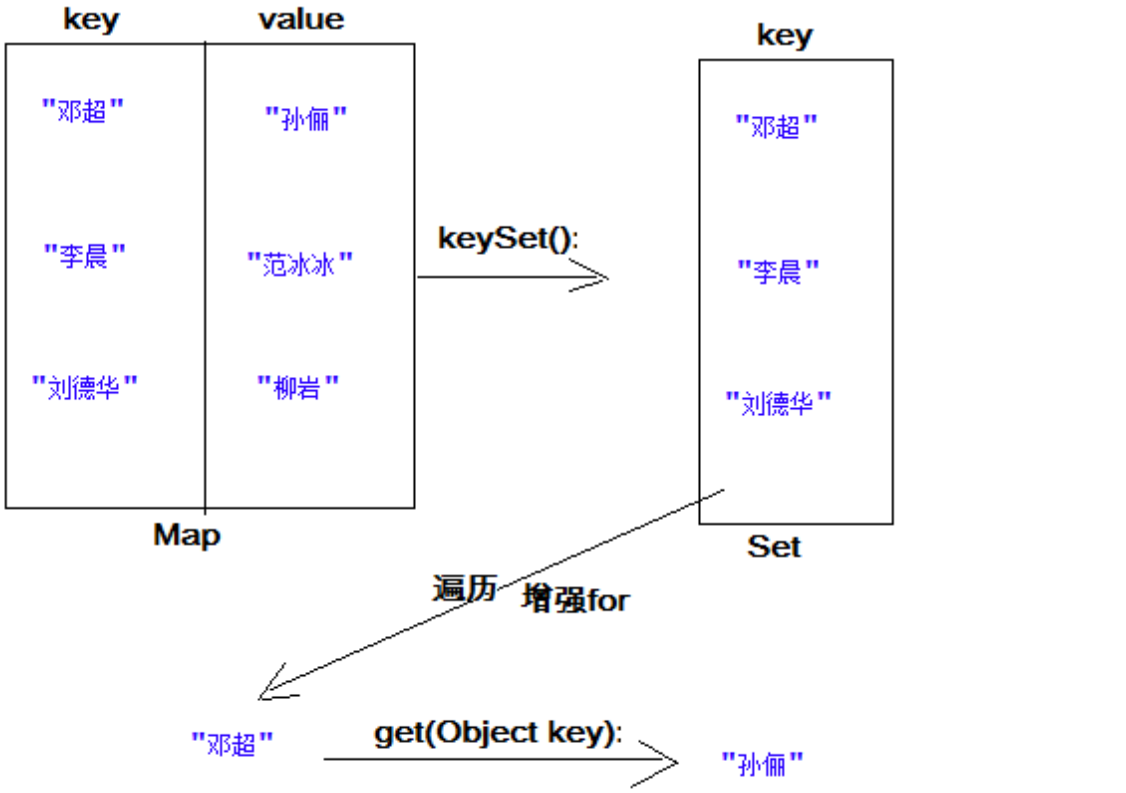
- 获取Map中所有的键，由于键是唯一的，所以返回一个Set集合存储所有的键。方法提示:`keyset()`
- 遍历键的Set集合，得到每一个键。
- 根据键，获取键所对应的值。方法提示:`get(K key)`

Map集合遍历方式1 ： 键找值

Map集合方法：

keySet(): 得到Map集合中所有的键

get(Object key): 通过指定的键，从map集合中找对应的值



3.3.2演示键找值方式遍历

需求：演示通过键找值的方式实现Map集合遍历

//测试类代码

```
public class Test {
    public static void main(String[] args) {
        Map<Integer, String> m = new HashMap();
        m.put(1, "a");
        m.put(2, "b");
        m.put(3, "c");
        //通过键，找值的方式遍历
        // 获取Map中所有的键
        Set<Integer> set = m.keySet();
        // 遍历键的Set集合，得到每一个键。
        for (Integer key : set) {
            String value = m.get(key);
            System.out.println("key=" + key + "---" + "value=" + value);
        }
    }
}
//结果
key=1---value=a
key=2---value=b
key=3---value=c
```

3.3.3方式2:键值对

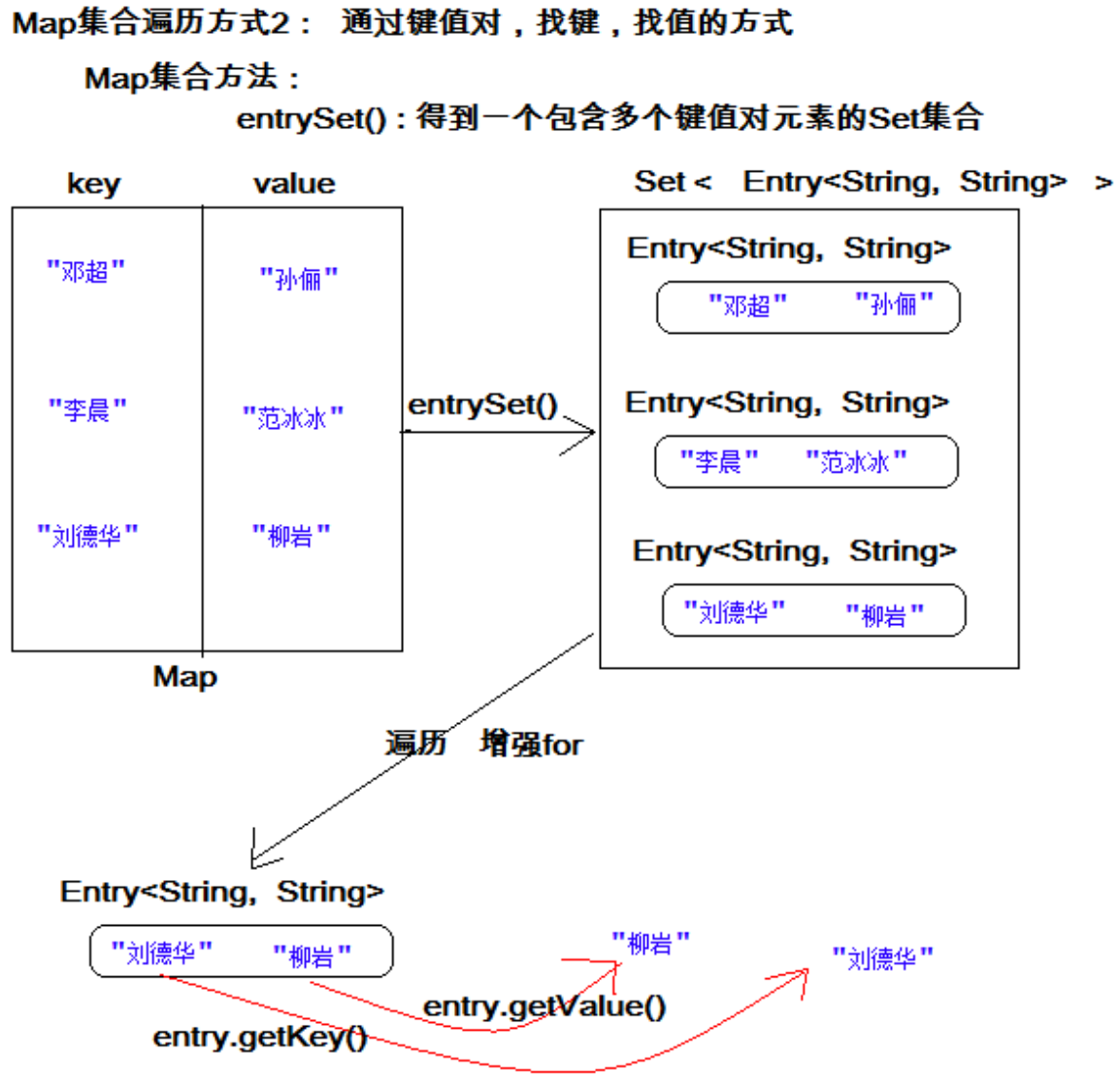
`java.util.Map.Entry` 将键值对的对应关系封装成了对象。Map的内部接口定义，具体功能由Map子类负责具体实现。

Entry中的常用方法

- `public K getKey()`：获取Entry对象中的键。
- `public V getValue()`：获取Entry对象中的值。

步骤：

- 获取Map集合中，所有的键值对(Entry)对象，以Set集合形式返回。方法提示:`entrySet()`。
- 遍历包含键值对(Entry)对象的Set集合，得到每一个键值对(Entry)对象。
- 通过键值对(Entry)对象，获取Entry对象中的键与值。 方法提示:`getkey() getValue()`



> tips：Map集合不能直接使用迭代器或者foreach进行遍历。但是转成Set之后就可以使用了。

3.3.4演示键值对方式遍历

需求：演示通过键找值的方式实现Map集合遍历

//测试类代码

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

```
        Map<Integer, String> m = new HashMap();
        m.put(1, "a");
        m.put(2, "b");
        m.put(3, "c");
        //通过键值对的方式遍历
        // 获取Map集合中，所有的键值对(Entry)对象，以Set集合形式返回。方法提
示:entrySet()。
        Set<Map.Entry<Integer, String>> entries = m.entrySet();
        // 遍历包含键值对(Entry)对象的Set集合，得到每一个键值对(Entry)对象。
        for (Map.Entry<Integer, String> entry : entries) {
            // 通过键值对(Entry)对象，获取Entry对象中的键与值。  方法提示:getkey()
getValue()
            Integer key = entry.getKey();
            String value = entry.getValue();
            System.out.println("key=" + key + "---" + "value=" + value);
        }
    }
}
key=1---value=a
key=2---value=b
key=3---value=c
```

小结

知识点--HashMap存储自定义类型

目标

- 掌握HashMap的使用

路径

- 概述
- 演示HashMap存储自定义类型

讲解

3.4.1概述

当给HashMap中存放自定义对象时，如果自定义对象作为key存在，这时要保证对象唯一，必须复写对象的hashCode和equals方法。

3.4.2演示HashMap存储自定义类型

需求：将包含姓名和年龄的学生对象作为键, 家庭住址作为值，存储到map集合中(学生姓名相同并且年龄相同视为同一名学生)。

学生类代码

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

    public Student() {
```

```
    }

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

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    @Override
    public 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 &&
                Objects.equals(name, student.name);
    }

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

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

//学生类代码

```
public class Test {
    public static void main(String[] args) {
        //创建集合对象
        HashMap<Student, String> hm = new HashMap<>();
        //创建学生对象
        Student s1 = new Student("张三", 18);
        Student s2 = new Student("李四", 20);
        Student s3 = new Student("张三", 20);
```

```
        Student s4 = new Student("张三", 18);
        //添加学生对象和对应的地址
        hm.put(s1, "北京");
        hm.put(s2, "上海");
        hm.put(s3, "广州");
        hm.put(s4, "深圳");
        //遍历数据
        Set<Student> stus = hm.keySet();
        for (Student key : stus) {
            String value = hm.get(key);
            System.out.println(key + "---" + value);
        }
    }
}
//结果
Student{name='张三', age=20}---广州
Student{name='张三', age=18}---深圳
Student{name='李四', age=20}---上海
```

小结

知识点--LinkedHashMap存储自定义类型

目标

- 理解LinkedHashMap的使用

路径

- 概述
- 演示LinkedHashMap存储自定义类型

讲解

3.5.1概述

LinkedHashMap是HashMap子类，底层由链表和哈希表组合。要保证map中存放的key和取出的顺序一致，可以使用java.util.LinkedHashMap集合来存放。

3.5.2演示LinkedHashMap存储自定义类型

需求：使用LinkedHashMap类存储自定义学生类(学生姓名相同并且年龄相同视为同一名学生)

//学生类代码

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

    public Student() {
    }

    public Student(String name, int age) {
```

```
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    @Override
    public 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 &&
                Objects.equals(name, student.name);
    }

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

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

//测试类代码

```
public class Test {
    public static void main(String[] args) {
        //创建集合对象
        LinkedHashMap<Student, String> lhm = new LinkedHashMap<>();
        //创建学生对象
        Student s1 = new Student("张三", 18);
        Student s2 = new Student("李四", 20);
        Student s3 = new Student("张三", 20);
        Student s4 = new Student("张三", 18);
        //添加学生对象和对应的地址
```

```
        lhm.put(s1, "北京");
        lhm.put(s2, "上海");
        lhm.put(s3, "广州");
        lhm.put(s4, "深圳");
        //遍历数据
        Set<Student> stus = lhm.keySet();
        for (Student key : stus) {
            String value = lhm.get(key);
            System.out.println(key + "---" + value);
        }
    }
}
//结果
Student{name='张三', age=18}---深圳
Student{name='李四', age=20}---上海
Student{name='张三', age=20}---广州
```

小结

知识点--TreeMap存储自定义类型

目标

- 理解TreeMap集合

路径

- 概述
- 演示TreeMap集合
- 演示自然排序
- 演示比较器排序

讲解

3.6.1概述

`java.util.TreeMap` 是是Map接口的一个实现类,是一种基于**红黑树**的实现，可以对元素的***键***进行排序。

TreeMap集合存储的对象如果作为键，必须拥有排序规则(比较器)，否则会报出异常。`cannot be cast to java.lang.Comparable`

自然排序：

- 对象类型自身拥有的排序规则，由对象类实现java.lang.Comparable接口，重写compare方法
- 使用TreeSet无参构造创建对象

比较器排序：

- 存储容器拥有的排序规则，创建重写compare方法的Comparator接口子类对象,作为参数传递给TreeMap构造方法
- 使用TreeSet有参构造创建对象。public TreeMap(Comparator comparator)： 根据指定的比较器进行排序

3.6.2演示TreeMap集合

需求：通过TreeMap存储自定义学生类和老师类型对象演示两种排序方式

学生类代码

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

    public Student() {
    }

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

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    @Override
    public int compareTo(Student s) {
        //this 正在添加的      s代表的是已存在
        //升序  this  vs  s
        //降序  s  vs  this

        int num = this.name.compareTo(s.name);
        num = (num == 0) ? this.age - s.age : num;
        return num;
    }

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

老师类代码

```
public class Teacher {
```

```
    private String name;
    private int age;

    public Teacher() {
    }

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

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

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

//Comparator实现类

```
public class MyComparator implements Comparator<Teacher> {
    @Override
    public int compare(Teacher t1, Teacher t2) {
        //t1 正在添加的      t2代表的是已存在
        //升序 t1  vs  t2
        //降序  t2  vs  t1

        int num = t2.getName().compareTo(t1.getName());
        num = (num == 0) ? t2.getAge() - t1.getAge() : num;
        return num;
    }
}
```

//测试类代码

```
public class Test {
```

```
    public static void main(String[] args) {
        //自然排序
        /*TreeMap<Student, String> ts = new TreeMap<>();
        ts.put(new Student("张三", 18), "北京");
        ts.put(new Student("李四", 20), "上海");
        ts.put(new Student("张三", 20), "广州");
        ts.put(new Student("张三", 18), "深圳");
        //遍历
        Set<Map.Entry<Student, String>> entries = ts.entrySet();
        for (Map.Entry<Student, String> entry : entries) {
            Student key = entry.getKey();
            String value = entry.getValue();
            System.out.println(key + "---" + value);
        }*/
        System.out.println("--------");
        // 比较器排序
        MyComparator mc = new MyComparator();
        TreeMap<Teacher, String> ts2 = new TreeMap<>(mc);
        ts2.put(new Teacher("张三", 18), "北京");
        ts2.put(new Teacher("李四", 20), "上海");
        ts2.put(new Teacher("张三", 20), "广州");
        ts2.put(new Teacher("张三", 18), "深圳");
        //遍历
        Set<Map.Entry<Teacher, String>> entries2 = ts2.entrySet();
        for (Map.Entry<Teacher, String> entry : entries2) {
            Teacher key = entry.getKey();
            String value = entry.getValue();
            System.out.println(key + "---" + value);
        }
    }
}
```

小结

案例-Map集合练习

需求

输出一个字符串中每个字符出现次数。

分析

```
获取一个字符串对象
创建一个Map集合，键代表字符，值代表次数。
遍历字符串得到每个字符。
判断Map中是否有该键。boolean containKey(Object key)
    如果没有，第一次出现，存储次数为1；
    如果有，则说明已经出现过，获取到对应的值进行++，再次存储。
打印最终结果
```

实现

//测试类代码

```
public class Test {
    public static void main(String[] args) {
        // 获取一个字符串对象
        Scanner sc = new Scanner(System.in);
        System.out.println("请输入一段字符串内容");
        String line = sc.nextLine();
        // 创建一个Map集合，键代表字符，值代表次数。
        HashMap<Character, Integer> hm = new HashMap<>();
        // 遍历字符串得到每个字符。
        for (int i = 0; i < line.length(); i++) {
            char key = line.charAt(i);
            // 判断Map中是否有该键(public boolean containKey(Object key):判断该集合中
是否有此键)
            if (!hm.containsKey(key)) {
                // 如果没有，第一次出现，存储次数为1；
                hm.put(key, 1);
            } else {
                // 如果有，则说明已经出现过，获取到对应的值进行++，再次存储。
                Integer count = hm.get(key);
                count++;
                hm.put(key, count);
            }
        }
        // 打印最终结果
        System.out.println(hm);
    }
}
```

小结

第四章 模拟斗地主

需求:

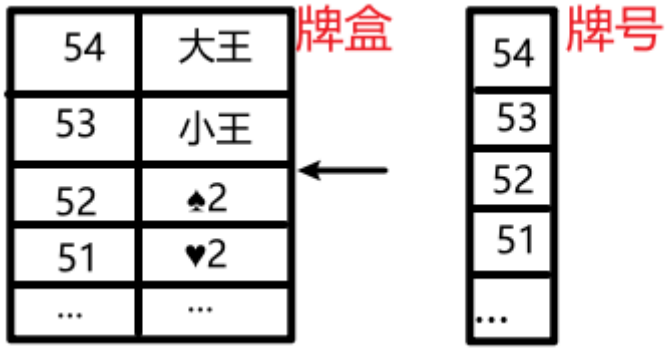
- 按照斗地主的规则，完成洗牌发牌的动作。

```
令狐冲：[♠2, ♦A, ♥A, ♠A, ♣K, ♥Q, ♦J, ♣J, ♥J, ♦9, ♣7, ♦5, ♥4, ♠4, ♣3, ♥3, ♠3]
石破天：[小王, ♦2, ♣2, ♥2, ♣A, ♦K, ♣Q, ♦10, ♥10, ♠10, ♠8, ♣6, ♥6, ♣5, ♠5, ♦4, ♣4]
鸠摩智：[大王, ♥K, ♦Q, ♠Q, ♣10, ♥9, ♠9, ♦8, ♣8, ♥8, ♦7, ♥7, ♠7, ♦6, ♠6, ♥5, ♦3]
底牌：[♠K, ♠J, ♣9]
```

- 具体规则：
    - 使用54张牌打乱顺序,三个玩家参与游戏，三人交替摸牌，每人17张牌，最后三张留作底牌
    - 牌面展示规则：大王,小王,2,A,K,Q,J,10,9,8,7,6,5,4,3...
    - 花色按照黑红梅方的顺序，优先花色排列。

分析

- 准备牌



- 洗牌:
  对牌号操作即可
- 发牌
  使用牌号作为发牌对象，使用单列集合作为玩家记录手中牌号
- 看牌
  对单列集合排序
  遍历手中牌号,从牌盒中获取对应牌面并打印。

步骤

- 准备牌
  将牌盒设计为一个HashMap<Integer，String>集合
  将牌号作为键，牌面(牌面由花色和数字组成)作为值，按照映射关系，放进牌盒集合
  使用一个ArrayList集合记录所有牌号。
- 洗牌:
  使用Collections类的shuffle方法对牌号进行打乱
- 发牌
  为每个玩家和剩余底牌各分配一个ArrayList记录牌号。
  将牌号通过对3取模判断，依次发牌，存入玩家集合中
  将最后3张牌号，直接存放于底牌集合中
- 看牌
  使用Collections类中的sort方法对玩家的牌号进行排序
  遍历手中牌号,从牌盒中获取对应牌面并打印。。

实现

演示代码

```
public class Test {
    public static void main(String[] args) {
        // 准备牌
        // 将牌盒设计为一个HashMap<Integer，String>集合
        HashMap<Integer, String> cardBox = new HashMap<>();
        // 使用一个ArrayList集合记录所有牌号
        ArrayList<Integer> list = new ArrayList<>();
        // 将牌号作为键，牌面(牌面由花色和数字组成)作为值，按照映射关系，放进牌盒集合
        int index = 1;
        String[] nums = {"3", "4", "5", "6", "7", "8", "9", "10", "J", "Q", "K",
"A", "2"};
        String[] colors = "♦,♣,♥,♠".split(",");
```

```
        for (String num : nums) {
            for (String color : colors) {
                String card = color + num;
                cardBox.put(index, card);
                list.add(index);
                index++;
            }
        }
        cardBox.put(index, "小王");
        list.add(index);
        index++;
        cardBox.put(index, "大王");
        list.add(index);
        System.out.println(cardBox);

        // 洗牌：
        // 使用Collections类的shuffle方法对牌号进行打乱
        Collections.shuffle(list);
        //发牌
        // 为每个玩家和剩余底牌各分配一个ArrayList<Integer>记录牌号。
        ArrayList<Integer> player1 = new ArrayList<>();
        ArrayList<Integer> player2 = new ArrayList<>();
        ArrayList<Integer> player3 = new ArrayList<>();
        ArrayList<Integer> diPai = new ArrayList<>();
        // 将牌号通过对3取模判断，依次发牌，存入玩家集合中
        for (int i = 0; i < list.size(); i++) {
            int cardNum = list.get(i);
            if (i >= 51) {
                // 将最后3张牌号，直接存放于底牌集合中
                diPai.add(cardNum);
            } else {
                if (i % 3 == 0) {
                    player1.add(cardNum);
                } else if (i % 3 == 1) {
                    player2.add(cardNum);
                } else {
                    player3.add(cardNum);
                }
            }
        }

        //看牌
        // 使用Collections类中的sort方法对玩家的牌号进行排序
        Collections.sort(player1);
        Collections.sort(player2);
        Collections.sort(player3);
        Collections.sort(diPai);
        // 遍历手中牌号,从牌盒中获取对应牌面并打印。
        showCard(player1, cardBox);
        showCard(player2, cardBox);
        showCard(player3, cardBox);
        showCard(diPai, cardBox);

    }

    public static void showCard(ArrayList<Integer> player, HashMap<Integer,
String> cardBox) {
        System.out.print("[");
```

```
        for (int i = 0; i < player.size(); i++) {
            Integer cardNum = player.get(i);
            String card = cardBox.get(cardNum);
            if (i == player.size() - 1) {
                System.out.print(card);
            } else {
                System.out.print(card + ",");
            }
        }
        System.out.println("]");
    }
}
//结果
{1=♦3, 2=♣3, 3=♥3, 4=♠3, 5=♦4, 6=♣4, 7=♥4, 8=♠4, 9=♦5, 10=♣5, 11=♥5, 12=♠5,
13=♦6, 14=♣6, 15=♥6, 16=♠6, 17=♦7, 18=♣7, 19=♥7, 20=♠7, 21=♦8, 22=♣8, 23=♥8,
24=♠8, 25=♦9, 26=♣9, 27=♥9, 28=♠9, 29=♦10, 30=♣10, 31=♥10, 32=♠10, 33=♦J, 34=♣J,
35=♥J, 36=♠J, 37=♦Q, 38=♣Q, 39=♥Q, 40=♠Q, 41=♦K, 42=♣K, 43=♥K, 44=♠K, 45=♦A,
46=♣A, 47=♥A, 48=♠A, 49=♦2, 50=♣2, 51=♥2, 52=♠2, 53=小王, 54=大王}
[♦3,♦4,♣4,♦6,♠6,♦7,♠7,♦8,♠8,♠J,♦Q,♠Q,♥A,♠A,♣2,♠2,小王]
[♣3,♥3,♠3,♠4,♣5,♠5,♣6,♣7,♣8,♦9,♠10,♦J,♥Q,♣K,♠K,♣A,大王]
[♥4,♦5,♥5,♥6,♥7,♥8,♥9,♠9,♣10,♥10,♣J,♥J,♣Q,♥K,♦A,♦2,♥2]
[♣9,♦10,♦K]
```

小结