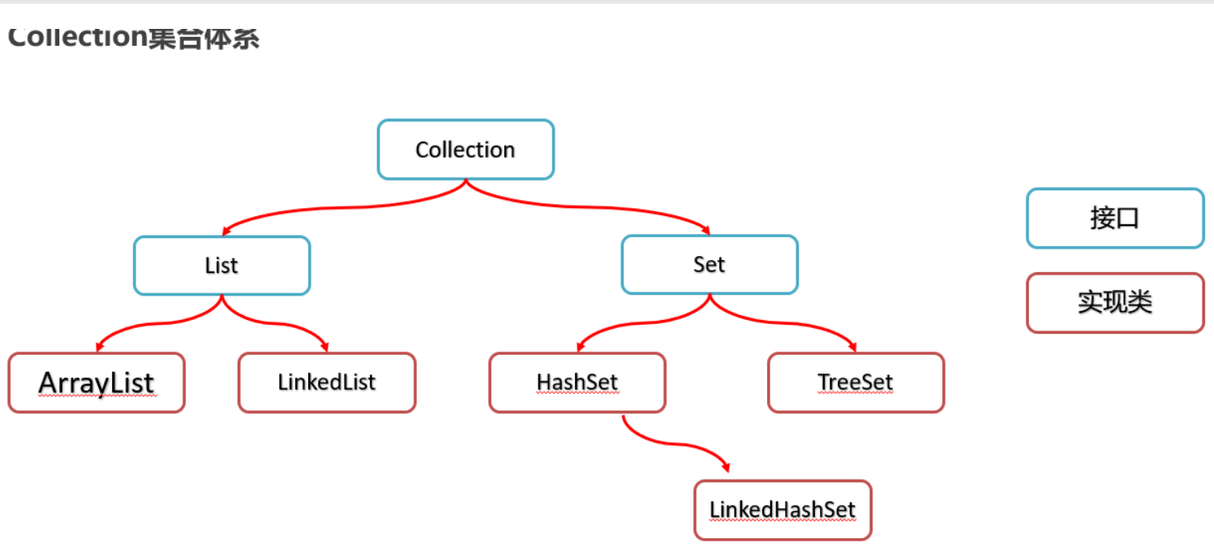
Set系列集合的特点



- **Set**系列集合特点

无序：存取顺序不一致
不重复：可以去除重复
无索引：没有带索引的方法，所以不能使用普通for循环遍历，也不能通过索引来获取元素。

- **Set**集合实现类特点

HashSet : 无序、不重复、无索引。

LinkedHashSet：有序(按插入顺序)、不重复、无索引。

TreeSet：排序(非基本数据类型需要定义比较规则)、不重复、无索引。

Set集合的功能上基本上与Collection的API一致

```
    @Test
    public void test1() {
        final Set<String> set = new HashSet<>();//多态写法,hashset无序 不重复 无索引
        set.add("java");
        System.out.println(set.add("java"));
        set.add("spring");
        set.add("springboot");
        set.add("springmvc");
        System.out.println(set);
        System.out.println("------------------");

         Set<String> set1 = new LinkedHashSet<>();//多态写法，有序 不重复 无索引
        set1.add("java");
        System.out.println(set1.add("java"));
        set1.add("spring");
        set1.add("springboot");
        set1.add("springmvc");
        System.out.println(set1);
        System.out.println("------------------");

        Set<String> set3 = new TreeSet<>();//多态写法，排序 不重复 无索引
        set3.add("java");
        System.out.println(set3.add("java"));
        set3.add("spring");
        set3.add("mybatis");
        set3.add("apache");
        System.out.println(set3);
        System.out.println(set3.remove("java"));
        System.out.println(set3.isEmpty());
        final Object[] array = set3.toArray();
        System.out.println(Arrays.toString(array));
        set3.clear();
        System.out.println(set3);
    }
```

```
false
[spring, java, springboot, springmvc]
------------------
false
[java, spring, springboot, springmvc]
------------------
false
[apache, java, mybatis, spring]
true
false
[apache, mybatis, spring]
[]
```

HashSet元素无序的底层原理：

- **HashSet**底层原理

HashSet集合底层采取哈希表存储的数据。
哈希表是一种对于增删改查数据性能都较好的结构

- 哈希表的组成

JDK8之前的，底层使用数组+链表组成
JDK8开始后，底层采用数组+链表|红黑树组成。

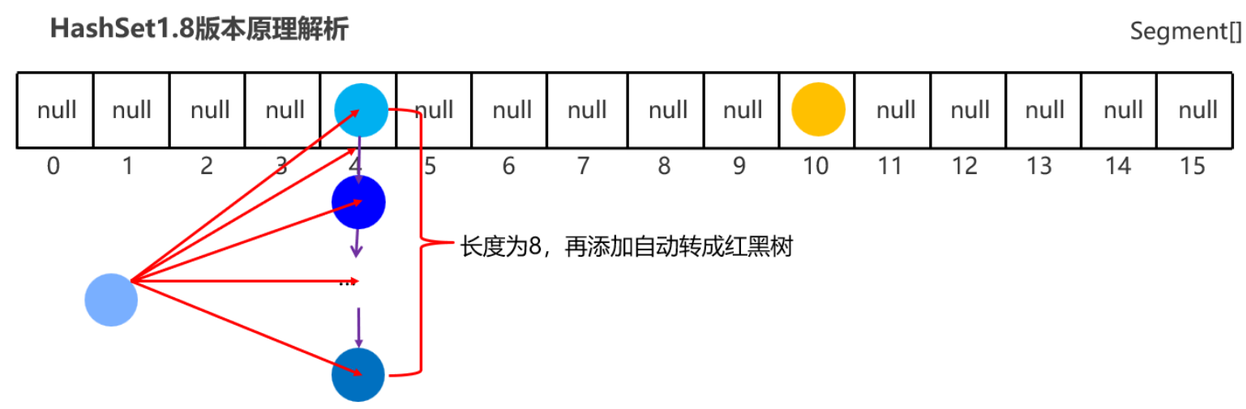
- 哈希值

是JDK根据对象的地址，按照某种规则算出来的int类型的数值。

Object类的API：
　public int hashCode()：返回对象的哈希值

同一个对象多次调用hashCode()方法返回的哈希值是相同的
默认情况下，不同对象的哈希值是不同的

创建一个默认长度16的数组，数组名table
根据元素的哈希值跟数组的长度求余计算出应存入的位置（哈希算法）
判断当前位置是否为null，如果是null直接存入
如果位置不为null，表示有元素，则调用equals方法比较
如果一样，则不存，如果不一样，则存入数组，
JDK 7头插法
JDK 8尾插法
结论：
　哈希表是一种对于增删改查数据性能都较好的结构



底层结构：哈希表（数组、链表、红黑树的结合体）
当挂在元素下面的数据过多时，查询性能降低，从JDK8开始后，当链表长度超过8的时候，自动转换为红黑树;进一步提高了操作数据的性能

- 哈希表的详细流程

创建一个默认长度16，默认加载因为0.75的数组，数组名table

根据元素的哈希值跟数组的长度计算出应存入的位置

判断当前位置是否为null，如果是null直接存入，如果位置不为null，表示有元素，则调用equals方法比较属性值，如果一样，则不存，如果不一样，则存入数组。

当数组存满到16*0.75=12时，就自动扩容，每次扩容原先的两倍

HashSet元素去重复的底层原理

1. 创建一个默认长度16的数组，数组名table
2. 根据元素的哈希值跟数组的长度求余计算出应存入的位置（哈希算法）
3. 判断当前位置是否为null，如果是null直接存入
4. 如果位置不为null，表示有元素，则调用equals方法比较
5. 如果一样，则不存，如果不一样，则存入数组，

结论：如果希望Set集合认为2个内容一样的对象是重复的，
必须重写对象的hashCode()【原本的hashcode是根据对象地址通过一定的规则生成的，因为new出来的对象其地址一定不同，所以其hashcode大概率不同，所以要重写】和equals()方法

```
class Student {
```

```
    private String name;
    private int age;
    private char sex;


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

    @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 && sex == student.sex && Objects.equals(name, student.name);
    }

    /**
     * 入参相同，得到的结果值也相同
     * 先比较hashcode是否相同，如果相同，再调用重写的equals方法比较内容是否相同，相同就认为是同一个对象，不同就认为是不同对象
     * 重写hashcode后，hashcode不同，一定是不同对象；但hashcode相同，不一定是同一个对象，需要判断内容
     *
     * @return
     */
    @Override
    public int hashCode() {
        return Objects.hash(name, age, sex);
    }

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

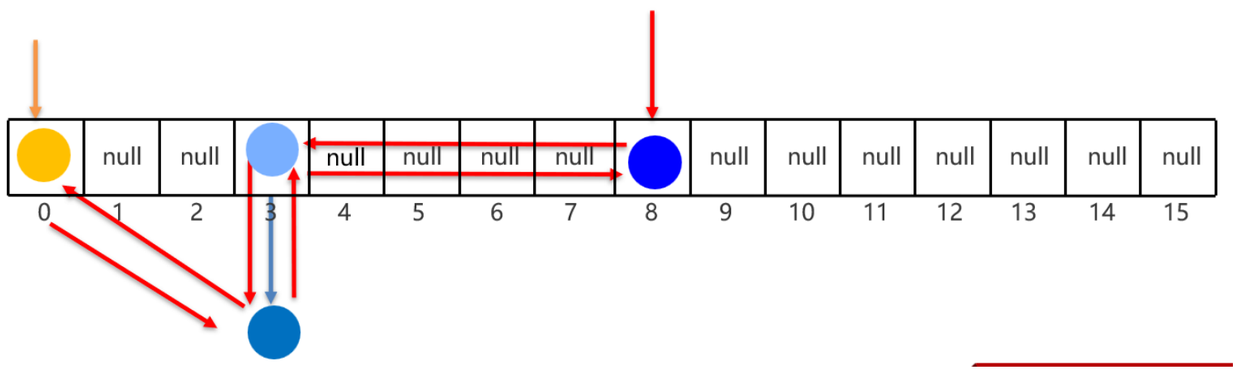
```
public class SetDemo2 {
    public static void main(String[] args) {
        Set<Student> set = new HashSet<>();
        final Student student1 = new Student("吴晓阳", 50, '男');
        final Student student2 = new Student("吴晓阳", 50, '男');
        final Student student3 = new Student("刘畅", 20, '男');
        System.out.println(student1.hashCode());
        System.out.println(student2.hashCode());
        System.out.println(student3.hashCode());
        Collections.addAll(set, student1, student2, student3);
        System.out.println(set);
    }
}
```

```
-750029800
-750029800
654998255
[Student{name='吴晓阳', age=50, sex=男}, Student{name='刘畅', age=20, sex=男}]
```

实现类：LinkedHashSet

- **LinkedHashSet**集合概述和特点

有序、不重复、无索引。
这里的有序指的是保证存储和取出的元素顺序一致
原理：底层数据结构是依然哈希表，只是每个元素又额外的多了一个双链表的机制记录存储的顺序



实现类：TreeSet

- TreeSet集合概述和特点

不重复、无索引、可排序
可排序：按照元素的大小默认升序（有小到大）排序。
TreeSet集合底层是基于红黑树的数据结构实现排序的，增删改查性能都较好。
注意：TreeSet集合是一定要排序的，可以将元素按照指定的规则进行排序

- TreeSet集合默认规则

对于数值类型：Integer , Double，官方默认按照大小进行升序排序。（集合、泛型 引用数据类型）
对于字符串类型：默认按照首字符的编号升序排序。
对于自定义类型如Student对象，TreeSet无法直接排序

想要使用TreeSet存储自定义类型，需要制定排序规则

- **自定义排序规则**

方式一
让自定义的类（如学生类）实现Comparable接口重写里面的compareTo方法来定制比较规则。

方式二
TreeSet集合有参数构造器，可以设置Comparator接口对应的比较器对象，来定制比较规则

返回值的规则：
　如果认为第一个元素大于第二个元素返回正整数即可。
　如果认为第一个元素小于第二个元素返回负整数即可。
　o1-o2是升序排序；
　如果认为第一个元素等于第二个元素返回0即可，此时Treeset集合只会保留一个元素，认为两者重复。
如果TreeSet集合存储的对象有实现比较规则，集合也自带比较器，默认使用集合自带的比较器排序；（比较器优先级更高）

- 小结

TreeSet集合的特点是怎么样的？
　可排序、不重复、无索引
　底层基于红黑树实现排序，增删改查性能较好

TreeSet集合自定义排序规则有几种方式
　2种。
　类实现Comparable接口，重写比较规则。
　集合自定义Comparator比较器对象，重写比较规则。

Collection体系的特点、使用场景

1.如果希望元素可以重复，又有索引，索引查询要快？
　用ArrayList集合，基于数组的。（用的最多）

2. 如果希望元素可以重复，又有索引，增删首尾操作快？
　用LinkedList集合，基于链表的。
3. 如果希望增删改查都快，但是元素不重复、无序、无索引。
　用HashSet集合，基于哈希表的。jdk1.8(数组，链表|红黑树)
4. 如果希望增删改查都快，但是元素不重复、有序、无索引。
　用LinkedHashSet集合，基于哈希表和双链表。
5. 如果要对对象进行排序。
　用TreeSet集合，基于红黑树。后续也可以用List集合实现排序。

可变参数

- 概述

可变参数用在形参中可以接收多个数据。
可变参数的格式：数据类型...参数名称
　public void test(int...a){}

- 作用

传输参数非常灵活，方便。可以不传输参数，可以传输1个或者多个，也可以传输一个数组
可变参数在方法内部本质上就是一个数组；数组的下标，长度等功能可变长参数都拥有

- **可变参数的注意事项：**

1.一个形参列表中可变参数只能有一个
2.可变参数必须放在形参列表的最后面

集合工具类Collections

java.utils.Collections:是集合工具类
作用：Collections并不属于集合，是用来操作集合的工具类。

Collections常用的API

方法名称	说明
`public static <T> boolean addAll(Collection<? super T> c, T... elements)`	给集合对象批量添加元素
`public static void shuffle(List<?> list)`	打乱List集合元素的顺序

Method	第一个参数中泛型存放的类型是第二个参数类型的父类或本类
`addAll(Collection<? super T> c, T... elements)`	

Collections排序相关API

- 使用范围：只能对于List集合的排序。

排序方式1：

方法名称	说明
`public static <T> void sort(List<T> list)`	将集合中元素按照默认规则排序

注意：本方式不可以直接对自定义类型的List集合排序，除非自定义类型实现了比较规则Comparable接口。

排序方式2：

方法名称	说明
`public static <T> void sort(List<T> list, Comparator<? super T> c)`	将集合中元素按照指定规则排序

```
class Apple implements Comparable<Apple> {
    private String name;
    private double price;
    private int weight;

    @Override
    public String toString() {
        return "Apple{" +
                "name='" + name + '\'' +
                ", price=" + price +
                ", weight=" + weight +
                '}';
    }

    public Apple(String name, double price, int weight) {
        this.name = name;
        this.price = price;
        this.weight = weight;
    }

    @Override
    public int compareTo(Apple o) {
        return this.weight - o.weight;//去除中重量重复的元素
//        return this.weight - o.weight > 0 ? 1 : -1;//不会去除
    }

    public String getName() {
        return name;
    }

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

    public double getPrice() {
        return price;
    }

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

    public int getWeight() {
        return weight;
    }

    public void setWeight(int weight) {
        this.weight = weight;
    }
}
```

```
    public static void main(String[] args) {
        List<Apple> list = new ArrayList<>();
        Collections.addAll(list,
                           new Apple("富士康", 12.31, 20),
                           new Apple("红苹果", 41.2, 11),
                           new Apple("青苹果", 12.11, 14));
        System.out.println(list);
        Collections.shuffle(list);
        System.out.println(list);
//        Collections.sort(list);//Apple类实现了comparable接口，可以排序
        Collections.sort(list, ((o1, o2) -> Double.compare(o1.getPrice(), o2.getPrice())));//通过价格比较，可以重复，比较器有优先级
        System.out.println(list);
    }
```

```
[Apple{name='富士康', price=12.31, weight=20}, Apple{name='红苹果', price=41.2, weight=11}, Apple{name='青苹果', price=12.11, weight=14}]
[Apple{name='红苹果', price=41.2, weight=11}, Apple{name='富士康', price=12.31, weight=20}, Apple{name='青苹果', price=12.11, weight=14}]
[Apple{name='青苹果', price=12.11, weight=14}, Apple{name='富士康', price=12.31, weight=20}, Apple{name='红苹果', price=41.2, weight=11}]
```

1. 为何Collections的API只能针对于List集合排序。

方法名称
`public static <T> void sort(List<T> list)`
`public static <T> void sort(List<T> list, Comparator<? super T> c)`

List是基于数组的，元素值之间可以换位置；
Set是基于哈希表的，元素值的位置相对固定，不易更换位置；

综合案例

- 斗地主

```
/*
 * 点数："3","4","5","6","7","8","9","10","J","Q","K","A","2"
 * 花色："♠", "♥", "♣", "♦"
 * 大小王："🃏" , "🂠"
 */
public class GameDemo {
    public static List<Card> cards = new ArrayList<>();

    static {
        final String[] sizes = {"3", "4", "5", "6", "7", "8", "9", "10", "J", "Q", "K", "A", "2"};
        final String[] colors = {"♠", "♥", "♣", "♦"};
        int counter = 0;
        for (String size : sizes) {
            for (String color : colors) {
                cards.add(new Card(size, color, counter++));
            }
        }
        cards.add(new Card("🂠", "", counter++));
        cards.add(new Card("🃏", "", counter));
    }

    public static void main(String[] args) {
        System.out.println(cards);
        Collections.shuffle(cards);
        List<Card> chong = new ArrayList<>();
        List<Card> jiu = new ArrayList<>();
        List<Card> ying = new ArrayList<>();


        for (int i = 0; i < cards.size() - 3; i++) {
            if (i % 3 == 0) {
                chong.add(cards.get(i));
            } else if (i % 3 == 1) {
                jiu.add(cards.get(i));
            } else if (i % 3 == 2) {
                ying.add(cards.get(i));
            }
        }
        //截取集合中的某段区间，作为一个新的集合
        final List<Card> lastThree = GameDemo.cards.subList(GameDemo.cards.size() - 3,
GameDemo.cards.size());

        sortCard(chong);
        sortCard(jiu);
        sortCard(ying);
        System.out.println(chong);
        System.out.println(jiu);
        System.out.println(ying);
        System.out.println(lastThree);
    }

    private static void sortCard(List<Card> cards) {
        Collections.sort(cards, ((o1, o2) -> o1.getIndex() - o2.getIndex()));
    }
}
```

```
/**
 * ClassName: Card <br/>
 * Description: <br/>
 * date: 2022/3/12 19:21<br/>
 *
 * @author fgcy<br />
 * @since JDK 1.8
 */
public class Card {
    private String size;
    private String color;
    private int index;

    public String getSize() {
        return size;
    }

    public void setSize(String size) {
        this.size = size;
    }

    public String getColor() {
        return color;
    }

    public void setColor(String color) {
        this.color = color;
    }
```

```
    public int getIndex() {
        return index;
    }

    public void setIndex(int index) {
        this.index = index;
    }

    @Override
    public String toString() {
        return size + color;
    }

    public Card() {
    }

    public Card(String size, String color, int index) {
        this.size = size;
        this.color = color;
        this.index = index;
    }
}
```

```
[3♥, 4♦, 6♥, 8♦, 9♣, 9♦, 10♥, 10♣, J♥, J♣, Q♠, Q♥, Q♣, K♠, A♠, A♥, A♣]
[3♠, 3♦, 4♠, 5♠, 5♦, 6♠, 6♦, 7♥, 7♣, 8♠, 9♠, 10♦, K♥, K♣, A♦, 2♥, 2♦]
[4♥, 4♣, 5♥, 5♣, 6♣, 7♠, 8♥, 8♣, 9♥, 10♠, J♠, Q♦, K♦, 2♠, 2♣, 🃏, 🦹]
[3♣, J♦, 7♦]
```

Map集合体系

Map集合的概述

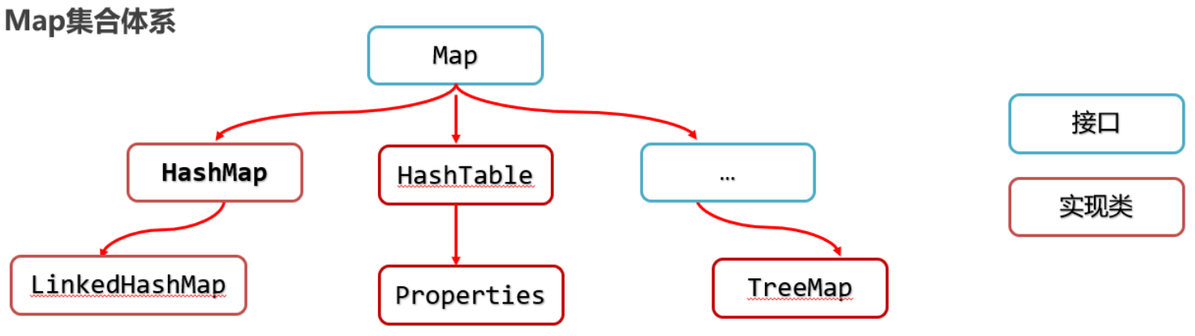
- 概述

Map集合是一种双列集合，每个元素包含两个数据。
Map集合的每个元素的格式：key=value(键值对元素)。
Map集合也被称为“键值对集合”。

Map集合的完整格式：{key1=value1 , key2=value2 , key3=value3}双列集合，键值对集合
Collection集合的格式: [元素1,元素2,元素3..] 单列集合

Map集合是什么？使用场景是什么样的？
Map集合是键值对集合
Map集合非常适合做类购物车这样的业务场景

Map集合体系特点



- 说明

使用最多的Map集合是HashMap。
重点掌握HashMap , LinkedHashMap , TreeMap。

- **Map集合体系特点**

Map集合的特点都是由键决定的。
Map集合的键是无序,不重复的，无索引的，值可以重复。
Map集合后面重复的键对应的值会覆盖前面重复键的值。
Map集合的键值对都可以为null {name=xxx,null=null,name=ccc}

- **Map集合实现类特点**

HashMap:元素按照键是无序，不重复，无索引，值不做要求。（与Map体系一致）
LinkedHashMap:元素按照键是有序，不重复，无索引，值可重复
TreeMap：元素按照建是排序，不重复，无索引的，值可重复

```
    @Test
    public void test1() {
        Map<String, String> maps = new HashMap<>();//键无序，不重复，无索引
        maps.put("name", "fgcy");
        maps.put("java", "12");
        maps.put("age", "18");
        maps.put("like", "no");
        maps.put(null, null);
        maps.put("like", "girl");//当当前元素的键与之前元素相同时，值会更新成当前元素的值
        System.out.println(maps);
        System.out.println("------------------------------");


        Map<String, String> maps2 = new LinkedHashMap<>();//键有序，不重复，无索引
        maps2.put("name", "fgcy");
        maps2.put("java", "12");
        maps2.put("age", "18");
        maps2.put("like", "no");
        maps2.put(null, null);
        maps2.put("like", "girl");//当当前元素的键与之前元素相同时，值会更新成当前元素的值
        System.out.println(maps2);
```

```
    }
```

```
{null=null, java=12, like=girl, name=fgcy, age=18}
----------------------------
{name=fgcy, java=12, age=18, like=girl, null=null}
```

Map集合常用API

Map集合
Map是双列集合的祖宗接口，它的功能是全部双列集合都可以继承使用的

Map API如下:

方法名称	说明
V put(K key,V value)	添加元素
V remove(Object key)	根据键删除键值对元素
void clear()	移除所有的键值对元素
boolean containsKey(Object key)	判断集合是否包含指定的键
boolean containsValue(Object value)	判断集合是否包含指定的值
boolean isEmpty()	判断集合是否为空
int size()	集合的长度，也就是集合中键值对的个数

```
@Test
    public void test2() {
        Map<String, String> maps = new HashMap<>();
        Map<String, String> maps2 = new HashMap<>();

        System.out.println(maps.put("name", "苏未晓"));//null
        maps.put("age", "11");
        maps.put("sex", "男");
        maps.put("hobby", "女");
        maps.put("age", "22");//key为age的值被覆盖为22
        System.out.println(maps.size());
        System.out.println(maps);
        System.out.println("是否包含键\'name\':" + maps.containsKey("name"));
        System.out.println("是否包含值\'22\'" + maps.containsValue("22"));
        System.out.println("集合是否为空:" + maps.isEmpty());
        System.out.println("删除键为name的值为：" + maps.remove("name"));
        System.out.println(maps);

        final Set<String> keySet = maps.keySet();//用set来接，应为Map的键时无序 不重复 无索引
        for (String s : keySet) {
            System.out.print(s + "=");
            System.out.print(maps.get(s) + " ");
        }
        System.out.println();
        System.out.println("当没有这个键时，值为：" + maps.get("name1"));//null
        final Collection<String> values = maps.values();//用Collection来接 可以重复
        System.out.println(values);
        maps.clear();
        System.out.println(maps);
        maps.put("age", "11");
        maps.put("sex", "男");
        maps.put("hobby", "女");

        maps2.put("where", "你爹");
        maps2.put("love", "男");
        maps2.put("honey", "女");
        maps.putAll(maps2);//把maps2中的元素全部拷贝一份到maps中
        System.out.println(maps);
        System.out.println(maps2);
    }
```

```
null
4
{sex=男, name=苏未晓, age=22, hobby=女}
是否包含键'name':true
是否包含值'22'true
集合是否为空:false
删除键为name的值为：苏未晓
{sex=男, age=22, hobby=女}
sex=男 age=22 hobby=女
当没有这个键时，值为：null
[男, 22, 女]
{}
{love=男, honey=女, sex=男, where=你爹, age=11, hobby=女}
{love=男, honey=女, where=你爹}
```

Map集合的遍历方式一：键找值

- Map集合的三种遍历方式：

方式一：键找值的方式遍历：先获取Map集合全部的键，再根据遍历键找值。
方式二：键值对的方式遍历，把“键值对”看成一个整体，难度较大。
方式三：JDK 1.8开始之后的新技术：Lambda表达式

- 步骤：

先获取Map集合的全部键的Set集合。
遍历键的Set集合，然后通过键提取对应值

键找值涉及到的API:

方法名称	说明
Set<K> keySet()	获取所有键的集合
V get(Object key)	根据键获取值

```
    @Test
    public void test3() {
        Map<String, String> maps = new HashMap<>();
        final Set<String> keySet = maps.keySet();
        maps.put("name", "14");
        maps.put("name", "55");//值覆盖
        maps.put("age", "14");
        maps.put("asex", "14");
        for (String key : keySet) {//得到一个键的set集合
            System.out.println(key + ":" + maps.get(key));
        }
    }
```

```
name:55
asex:14
age:14
```

Map集合的遍历方式二：键值对

先把Map集合中的每个元素转换，一个个的Map.Entry<>实例后，再把Map.Entry<>实例添加进Set集合，Set集合中每个元素都是键值对实体类型了。
遍历Set集合，然后提取键以及提取值

键值对涉及到的API:

方法名称	说明
Set<Map.Entry<K,V>> entrySet()	获取所有键值对对象的集合
K getKey()	获得键
V getValue()	获取值

```
    @Test
    public void test4() {
        Map<String, Integer> maps = new HashMap<>();
        maps.put("java", 11);
        maps.put("Mysql", 20);
        maps.put("Oracle", 24);
        maps.put("Mongodb", 54);
        /*
         * Map.Entry<String, Integer>是键值对类型，可以存储键是String值是Integer的键值对
         *因为Entry是接口，所以Map.Entry<>中存储的键值对实例是有maps.entrySet()方法
         通过遍历maps中的键值对，并为其创作出实例赋值给Map.Entry<>,
         然后再把Map.Entry<>的实例添加到set集合中
         * */
        final Set<Map.Entry<String, Integer>> entrySets = maps.entrySet();
        for (Map.Entry<String, Integer> entrySet : entrySets) {
            System.out.println(entrySet.getKey() + ":" + entrySet.getValue());
        }
    }
```

```
java:11
Mysql:20
Mongodb:54
Oracle:24
```

Map集合的遍历方式三：lambda表达式

Map结合Lambda遍历的API

方法名称	说明
default void forEach(BiConsumer<? super K, ? super V> action)	结合lambda遍历Map集合

```
    @Test
    public void test5() {
        Map<String, Integer> maps = new HashMap<>();
        maps.put("java", 11);
        maps.put("Mysql", 20);
        maps.put("Oracle", 24);
        maps.put("Mongodb", 54);
//        maps.forEach(System.out::printf);//方法引用 适用于接收一个参数，并直接将其输出；此处接收有两个参数 【不适用】
        //foreach底层使用的是Map.Entry<>遍历maps的方式
        maps.forEach((k, v) -> System.out.println(k + ":" + v));//一行解决，简洁
    /*
        maps.forEach(new BiConsumer<String, Integer>() {
            @Override
            public void accept(String k, Integer v) {
                System.out.println(k + ":" + v);
```

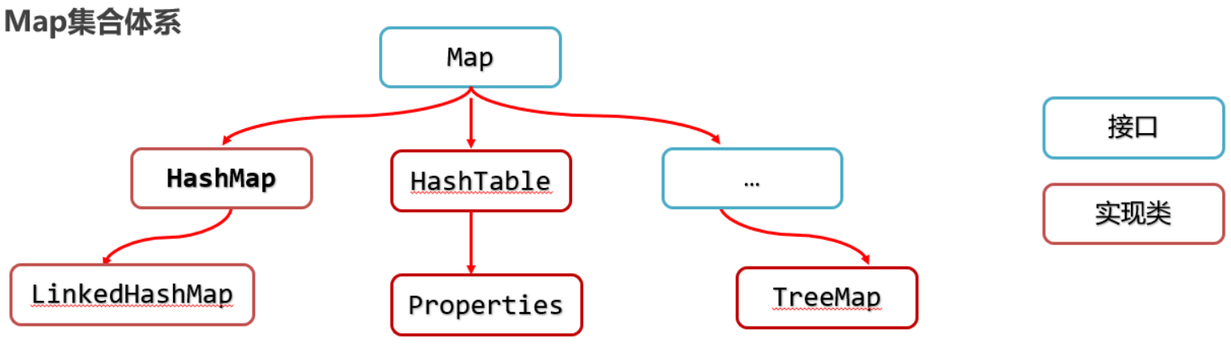
```
            }
        });
     */
    }
```

```
java:11
Mysql:20
Mongodb:54
Oracle:24
```

Map集合的实现类HashMap

使用最多的Map集合是HashMap。
重点掌握HashMap , LinkedHashMap , TreeMap

Map集合体系



- **HashMap特点：**

HashMap是Map里面的一个实现类。特点都是由键决定的：无序、不重复、无索引；值不做要求，只要满足泛型约束就行；

没有额外需要学习的特有方法，直接使用Map里面的方法就可以了。

HashMap跟HashSet底层原理是一模一样的，都是哈希表结构，只是HashMap的每个元素包含两个值而已。

哈希表结构：
1.8之前：数组、链表；元素挂在前
1.8：数组、链表、红黑树，元素挂在后

实际上：Set系列集合的底层就是Map实现的，只是Set集合中的元素只要键数据，不要值数据而已。

```
    public HashSet() {
        map = new HashMap<>();
    }
```

- **HashMap的添加规则**

1. 根据Map键的地址通过一定的规则计算出HashCode，然后使用HashCode对数组长度求余；
2. 通过求余得到的余数找出所在的数组的第几个空间，判断此空间有没有元素，有：通过调用equals方法比较每个元素与该元素是否相等，若相等， 更新值；否则直接放到该数组空间；

- **HashMap**的特点和底层原理

由键决定：无序、不重复、无索引。HashMap底层是哈希表结构的。
依赖hashCode方法和equals方法保证键的唯一。
如果键要存储的是自定义对象，需要重写hashCode和equals方法。
基于哈希表。增删改查的性能都较好

- HashMap存储自定义类型

```
class Student {
    private String name;
    private int age;
    private double weight;

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

    public Student() {
    }

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

    @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.weight, weight) == 0 && Objects.equals(name,
student.name);
    }

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

```
public class HashMapDemo1 {
    public static void main(String[] args) {
        final Student s1 = new Student("张三", 12, 55.1);//重写了hashcode和equals方法 这两个对象相等
        final Student s2 = new Student("张三", 12, 55.1);//重写了hashcode和equals方法 这两个对象相等
        final Student s3 = new Student("李氏", 52, 75.1);


        final HashMap<Student, String> map = new HashMap<>();
        map.put(s1, "北京");
        map.put(s2, "上海");
        map.put(s3, "广州");
        System.out.println(map);
    }
}
```

```
{Student{name='张三', age=12, weight=55.1}=上海, Student{name='李氏', age=52, weight=75.1}=广州}
```

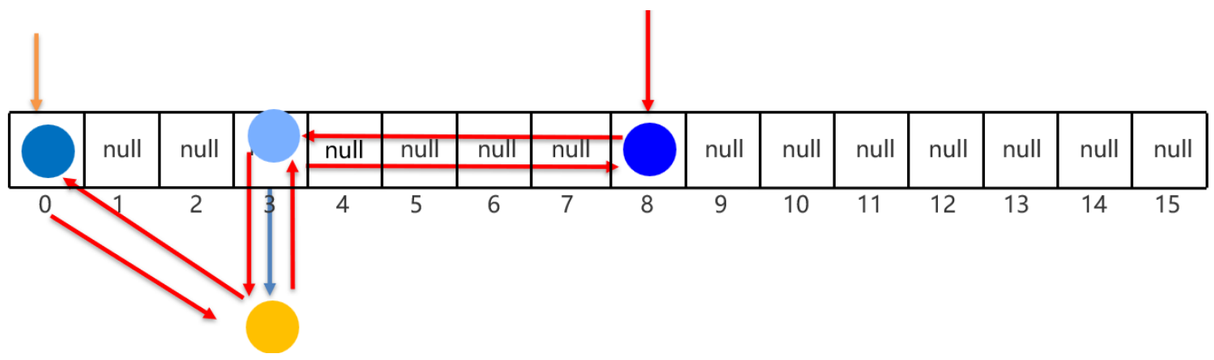
Map集合的实现类LinkedHashMap

- **LinkedHashMap集合概述和特点**

由键决定：有序、不重复、无索引。

这里的有序指的是保证存储和取出的元素顺序一致

原理：底层数据结构是依然哈希表，只是每个键值对元素又额外的多了一个双链表的机制记录存储的顺序。



Map集合的实现类TreeMap

- **TreeMap集合概述和特点**

由键决定特性：不重复、无索引、可排序
可排序：按照键数据的大小默认升序（有小到大）排序。只能对键排序。
注意：TreeMap集合是一定要排序的，可以默认排序，也可以将键按照指定的规则进行排序
TreeMap跟TreeSet一样底层原理是一样的，底层基于红黑树实现排序，增删改查性能较好

- **TreeMap集合自定义排序规则有2种**

类实现Comparable接口，重写比较规则。
集合自定义Comparator比较器对象，重写比较规则

- 案例

```
class Apple implements Comparable<Apple> {
    private String name;
    private double price;
    private int weight;

    @Override
    public String toString() {
        return "Apple{" +
                "name='" + name + '\'' +
                ", price=" + price +
                ", weight=" + weight +
                '}';
    }

    public Apple(String name, double price, int weight) {
        this.name = name;
        this.price = price;
        this.weight = weight;
    }

    @Override
    public int compareTo(Apple o) {
        return this.weight - o.weight;//去除中重量重复的元素
//        return this.weight - o.weight > 0 ? 1 : -1;//不会去除
    }

    public String getName() {
        return name;
    }

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

    public double getPrice() {
        return price;
    }

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

    public int getWeight() {
        return weight;
    }

    public void setWeight(int weight) {
        this.weight = weight;
```

```
    }
}
```

```
public class TreeMapDemo {
    public static void main(String[] args) {
        final Apple a1 = new Apple("红富士", 12.3, 10);//苹果类实现了comparable接口，根据重量比较
        final Apple a2 = new Apple("亲苹果", 22.1, 10);//这个对象与上面的对象，重量是一致的，所以认为这两个对象相同；后值把前值覆盖
        final Apple a3 = new Apple("黄苹果", 21.3, 23);
        //集合的比较器优先级较高，这个是通过价格来比较的
        final TreeMap<Apple, Integer> maps = new TreeMap<>(((o1, o2) -> Double.compare(o1.getPrice(), o2.getPrice())));
        maps.put(a1, 12);
        maps.put(a2, 52);
        maps.put(a3, 2);
        System.out.println(maps);
    }
}
```

```
 比较器比较：
{Apple{name='红富士', price=12.3, weight=10}=12, Apple{name='黄苹果', price=21.3, weight=23}=2, Apple{name='亲苹果', price=22.1, weight=10}=52}

可比较接口比较：
    {Apple{name='红富士', price=12.3, weight=10}=52, Apple{name='黄苹果', price=21.3, weight=23}=2}
```

集合的嵌套

```
public class TestMapDemo {
    public static void main(String[] args) {
        final List<String> wants1 = new ArrayList<>();
        final List<String> wants2 = new ArrayList<>();
        final List<String> wants3 = new ArrayList<>();
        Collections.addAll(wants1, "A", "B", "C");
        Collections.addAll(wants2, "A", "C");
        Collections.addAll(wants3, "B", "C");
        //集合嵌套
        final HashMap<String, List<String>> maps = new HashMap<>();
        final HashMap<String, Integer> col = new HashMap<>();
        maps.put("苏未晓", wants1);
        maps.put("李石", wants2);
        maps.put("张三", wants3);
//        System.out.println(maps);// {张三=[B, C], 李石=[A, C], 苏未晓=[A, B, C]}

        final Collection<List<String>> values = maps.values();
//        System.out.println(values);//   [[B, C], [A, C], [A, B, C]]
        for (List<String> value : values) {
            for (String v : value) {
                col.put(v, col.containsKey(v) ? col.get(v) + 1 : 1);
            }
        }
        System.out.println(col);
    }
}
```

```
{A=2, B=2, C=3}
```