day11【List、Collections、set】

今日内容

- List集合
- Collections工具类
- Set集合

教学目标

- [ ] 能够说出List集合特点
- [ ] 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

第一章 List接口

我们掌握了Collection接口的使用后，再来看看Collection接口中的子类，他们都具备那些特性呢?

接下来，我们一起学习Collection中的常用几个子类（`java.util.List`集合、`java.util.Set`集合）。

1.1 List接口介绍

`java.util.List`接口继承自`Collection`接口，是单列集合的一个重要分支，习惯性地会将实现了`List`接口的对象称为List集合。在List集合中允许出现重复的元素，所有的元素是以一种线性方式进行存储的，在程序中可以通过索引来访问集合中的指定元素。另外，List集合还有一个特点就是元素有序，即元素的存入顺序和取出顺序一致。

看完API，我们总结一下：

List接口特点：

1. 它是一个元素存取有序的集合。例如，存元素的顺序是11、22、33。那么集合中，元素的存储就是按照11、22、33的顺序完成的）。
2. 它是一个带有索引的集合，通过索引就可以精确的操作集合中的元素（与数组的索引是一个道理）。
3. 集合中可以有重复的元素，通过元素的equals方法，来比较是否为重复的元素。

> tips:我们在基础班的时候已经学习过List接口的子类java.util.ArrayList类，该类中的方法都是来自List中定义。

1.2 List接口中常用方法

List作为Collection集合的子接口，不但继承了Collection接口中的全部方法，而且还增加了一些根据元素索引来操作集合的特有方法，如下：

- `public void add(int index, E element)`: 将指定的元素，添加到该集合中的指定位置上。
- `public E get(int index)`:返回集合中指定位置的元素。
- `public E remove(int index)`: 移除列表中指定位置的元素, 返回的是被移除的元素。
- `public E set(int index, E element)`:用指定元素替换集合中指定位置的元素,返回值的更新前的元素。

List集合特有的方法都是跟索引相关，我们在基础班都学习过。

> tips:我们之前学习Colletion体系的时候，发现List集合下有很多集合，它们的存储结构不同，这样就导致了这些集合它们有各自的特点，供我们在不同的环境下使用，那么常见的数据结构有哪些呢？在下一章我们来介绍：

1.3 ArrayList集合

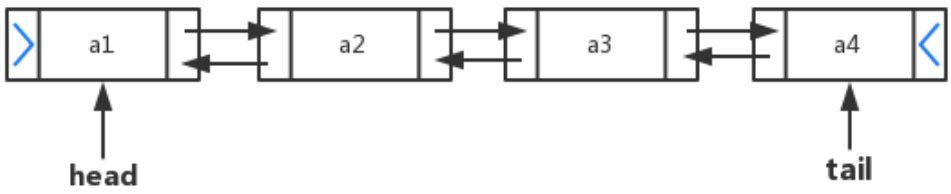
`java.util.ArrayList`集合数据存储的结构是数组结构。元素增删慢，查找快，由于日常开发中使用最多的功能为查询数据、遍历数据，所以`ArrayList`是最常用的集合。

许多程序员开发时非常随意地使用ArrayList完成任何需求，并不严谨，这种用法是不提倡的。

1.4 LinkedList集合

`java.util.LinkedList`集合数据存储的结构是链表结构。方便元素添加、删除的集合。

> LinkedList是一个双向链表，那么双向链表是什么样子的呢，我们用个图了解下



实际开发中对一个集合元素的添加与删除经常涉及到首尾操作，而LinkedList提供了大量首尾操作的方法。这些方法我们作为**了解即可**：

- `public void addFirst(E e)`:将指定元素插入此列表的开头。
- `public void addLast(E e)`:将指定元素添加到此列表的结尾。
- `public E getFirst()`:返回此列表的第一个元素。
- `public E getLast()`:返回此列表的最后一个元素。
- `public E removeFirst()`:移除并返回此列表的第一个元素。
- `public E removeLast()`:移除并返回此列表的最后一个元素。
- `public E pop()`:从此列表所表示的堆栈处弹出一个元素。
- `public void push(E e)`:将元素推入此列表所表示的堆栈。
- `public boolean isEmpty()`：如果列表不包含元素，则返回true。

LinkedList是List的子类，List中的方法LinkedList都是可以使用，这里就不做详细介绍，我们只需要了解LinkedList的特有方法即可。在开发时，LinkedList集合也可以作为堆栈，队列的结构使用。

```
public class Demo04LinkedList {
    public static void main(String[] args) {
        method4();
    }
    /*
     *  void push(E e)：  压入。把元素添加到集合的第一个位置。
     *  E pop()：  弹出。把第一个元素删除，然后返回这个元素。
     */
    public static void method4() {
        //创建LinkedList对象
        LinkedList<String> list = new LinkedList<>();
        //添加元素
        list.add("达尔文");
        list.add("达芬奇");
        list.add("达尔优");
        System.out.println("list:" + list);
        //调用push在集合的第一个位置添加元素
        //list.push("爱迪生");
        //System.out.println("list:" + list);//[爱迪生, 达尔文, 达芬奇, 达尔优]

        //E pop()：  弹出。把第一个元素删除，然后返回这个元素。
        String value = list.pop();
        System.out.println("value:" + value);//达尔文
        System.out.println("list:" + list);//[达芬奇，达尔优]
    }

    /*
     * E removeFirst()：删除第一个元素
     * E removeLast()：删除最后一个元素。
     */
    public static void method3() {
        //创建LinkedList对象
        LinkedList<String> list = new LinkedList<>();
        //添加元素
        list.add("达尔文");
        list.add("达芬奇");
        list.add("达尔优");
        //删除集合的第一个元素
//      String value = list.removeFirst();
//      System.out.println("value:" + value);//达尔文
//      System.out.println("list:" + list);//[达芬奇，达尔优]

        //删除最后一个元素
        String value = list.removeLast();
        System.out.println("value:" + value);//达尔优
        System.out.println("list:" + list);//[达尔文，  达芬奇]
    }

    /*
     * E getFirst()：  获取集合中的第一个元素
     * E getLast()：  获取集合中的最后一个元素
     */
    public static void method2() {
        //创建LinkedList对象
        LinkedList<String> list = new LinkedList<>();
        //添加元素
        list.add("达尔文");
```

```
        list.add("达芬奇");
        list.add("达尔优");

        System.out.println("list:" + list);
        //获取集合中的第一个元素
        System.out.println("第一个元素是：" + list.getFirst());
        //获取集合中的最后一个元素怒
        System.out.println("最后一个元素是：" + list.getLast());
    }


    /*
     * void addFirst(E e)：  在集合的开头位置添加元素。
     * void addLast(E e)：  在集合的尾部添加元素。
     */
    public static void method1() {
        //创建LinkedList对象
        LinkedList<String> list = new LinkedList<>();
        //添加元素
        list.add("达尔文");
        list.add("达芬奇");
        list.add("达尔优");
        //打印这个集合
        System.out.println("list:" + list);//[达尔文, 达芬奇, 达尔优]
        //调用addFirst添加元素
        list.addFirst("曹操");
        System.out.println("list:" + list);//[曹操, 达尔文, 达芬奇, 达尔优]
        //调用addLast方法添加元素
        list.addLast("大乔");
        System.out.println("list:" + list);//[曹操, 达尔文, 达芬奇, 达尔优, 大乔]

    }
}
```

1.5 LinkedList源码分析

- LinkedList的成员变量源码分析：

```
public class LinkedList<E> extends AbstractSequentialList<E>
                    implements List<E>, Deque<E>, Cloneable,
java.io.Serializable{
    transient int size = 0;
    /**
     *存储第一个节点的引用
     */
    transient Node<E> first;

    /**
     * 存储最后一个节点的引用
     */
    transient Node<E> last;

    //......
    //......
```

```
}
```

- LinkedList的内部类Node类源码分析：

```
public class LinkedList<E> extends AbstractSequentialList<E>
                    implements List<E>, Deque<E>, Cloneable,
java.io.Serializable{
    //......
    private static class Node<E> {
        E item;//被存储的对象
        Node<E> next;//下一个节点
        Node<E> prev;//前一个节点

        //构造方法
        Node(Node<E> prev, E element, Node<E> next) {
            this.item = element;
            this.next = next;
            this.prev = prev;
        }
    }
    //......
}
```

- LinkedList的add()方法源码分析：

```
public boolean add(E e) {
    linkLast(e);//调用linkLast()方法
    return true;//永远返回true
}
void linkLast(E e) {
  final Node<E> l = last;//一个临时变量，存储最后一个节点
  final Node<E> newNode = new Node<>(l, e, null);//创建一个Node对象
  last = newNode;//将新Node对象存储到last
  if (l == null)//如果没有最后一个元素，说明当前是第一个节点
    first = newNode;//将新节点存为第一个节点
  else
    l.next = newNode;//否则不是第一个节点，就赋值到当前的last的next成员
  size++;//总数量 + 1
  modCount++;//
}
```

- LinkedList的get()方法：

```
public E get(int index) {
    checkElementIndex(index);//检查索引的合法性(必须在0-size之间)，如果不合法，此方
法抛出异常
    return node(index).item;
}
Node<E> node(int index) {//此方法接收一个索引，返回一个Node
        // assert isElementIndex(index);
        if (index < (size >> 1)) {//判断要查找的index是否小于size / 2，二分法查找
            Node<E> x = first;// x = 第一个节点--从前往后找
            for (int i = 0; i < index; i++)//从0开始，条件：i < index，此循环只
控制次数
                x = x.next;//每次 x = 当前节点.next;
            return x;//循环完毕，x就是index索引的节点。
```

```
        } else {
            Node<E> x = last;// x = 最后一个节点--从后往前找
            for (int i = size - 1; i > index; i--)//从最后位置开始，条件：i > index
                x = x.prev;//每次 x = 当前节点.prev;
            return x;//循环完毕，x就是index索引的节点
        }
    }
```

第二章 Collections类

2.1 Collections常用功能

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

  常用方法如下：

- `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。
- `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。

代码演示：

```
public class CollectionsDemo {
    public static void main(String[] args) {
        ArrayList<Integer> list = new ArrayList<Integer>();

        list.add(100);
        list.add(300);
        list.add(200);
        list.add(50);
        //排序方法
        Collections.sort(list);
        System.out.println(list);
    }
}
结果：
[50,100, 200, 300]
```

我们的集合按照默认的自然顺序进行了排列，如果想要指定顺序那该怎么办呢？

2.2 Comparator比较器

创建一个学生类，存储到ArrayList集合中完成指定排序操作。

Student 类

```
public class Student{
    private String name;
    private int age;
    //构造方法
    //get/set
    //toString
}
```

测试类:

```
public class Demo {
    public static void main(String[] args) {
        // 创建四个学生对象 存储到集合中
        ArrayList<Student> list = new ArrayList<Student>();

        list.add(new Student("rose",18));
        list.add(new Student("jack",16));
        list.add(new Student("abc",20));
        Collections.sort(list, new Comparator<Student>() {
          @Override
            public int compare(Student o1, Student o2) {
            return o1.getAge()-o2.getAge();//以学生的年龄升序
         }
        });


        for (Student student : list) {
            System.out.println(student);
        }
    }
}
Student{name='jack', age=16}
Student{name='rose', age=18}
Student{name='abc', age=20}
```

2.3 可变参数

在**JDK1.5**之后，如果我们定义一个方法需要接受多个参数，并且多个参数类型一致，我们可以对其简化.

格式:

```
修饰符 返回值类型 方法名(参数类型... 形参名){  }
```

代码演示:

```
  public class ChangeArgs {
    public static void main(String[] args) {
        int sum = getSum(6, 7, 2, 12, 2121);
        System.out.println(sum);
    }
    public static int getSum(int... arr) {
        int sum = 0;
         for (int a : arr) {
         sum += a;
        }
         return sum;
    }
}
```

注意：

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。

应用场景: Collections

在Collections中也提供了添加一些元素方法：

`public static <T> boolean addAll(Collection<T> c, T... elements)`:往集合中添加一些元素。

代码演示:

```
public class CollectionsDemo {
    public static void main(String[] args) {
      ArrayList<Integer> list = new ArrayList<Integer>();
      //原来写法
      //list.add(12);
      //list.add(14);
      //list.add(15);
      //list.add(1000);
      //采用工具类 完成 往集合中添加元素
      Collections.addAll(list, 5, 222, 1, 2);
      System.out.println(list);
}
```

第三章 Set接口

`java.util.Set`接口和`java.util.List`接口一样，同样继承自`Collection`接口，它与`Collection`接口中的方法基本一致，并没有对`Collection`接口进行功能上的扩充，只是比`Collection`接口更加严格了。与`List`接口不同的是，`Set`接口都会以某种规则保证存入的元素不出现重复。

`Set`集合有多个子类，这里我们介绍其中的`java.util.HashSet`、`java.util.LinkedHashSet`、`java.util.TreeSet`这两个集合。

> tips:Set集合取出元素的方式可以采用：迭代器、增强for。

3.1 HashSet集合介绍

`java.util.HashSet`是`Set`接口的一个实现类，它所存储的元素是不可重复的，并且元素都是无序的(即存取顺序不能保证不一致)。`java.util.HashSet`底层的实现其实是一个`java.util.HashMap`支持，由于我们暂时还未学习，先做了解。

`HashSet`是根据对象的哈希值来确定元素在集合中的存储位置，因此具有良好的存储和查找性能。保证元素唯一性的方式依赖于：`hashCode`与`equals`方法。

我们先来使用一下Set集合存储，看下现象，再进行原理的讲解:

```
public class HashSetDemo {
    public static void main(String[] args) {
        //创建 Set集合
        HashSet<String>  set = new HashSet<String>();

        //添加元素
        set.add(new String("cba"));
        set.add("abc");
        set.add("bac");
        set.add("cba");
        //遍历
        for (String name : set) {
            System.out.println(name);
        }
    }
}
```

输出结果如下，说明集合中不能存储重复元素:

```
cba
abc
bac
```

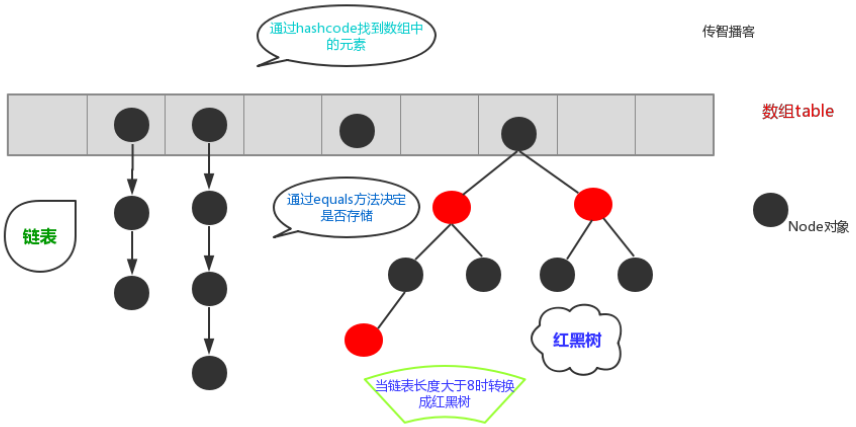
> tips:根据结果我们发现字符串"cba"只存储了一个，也就是说重复的元素set集合不存储。

3.2 HashSet集合存储数据的结构（哈希表）

什么是哈希表呢?

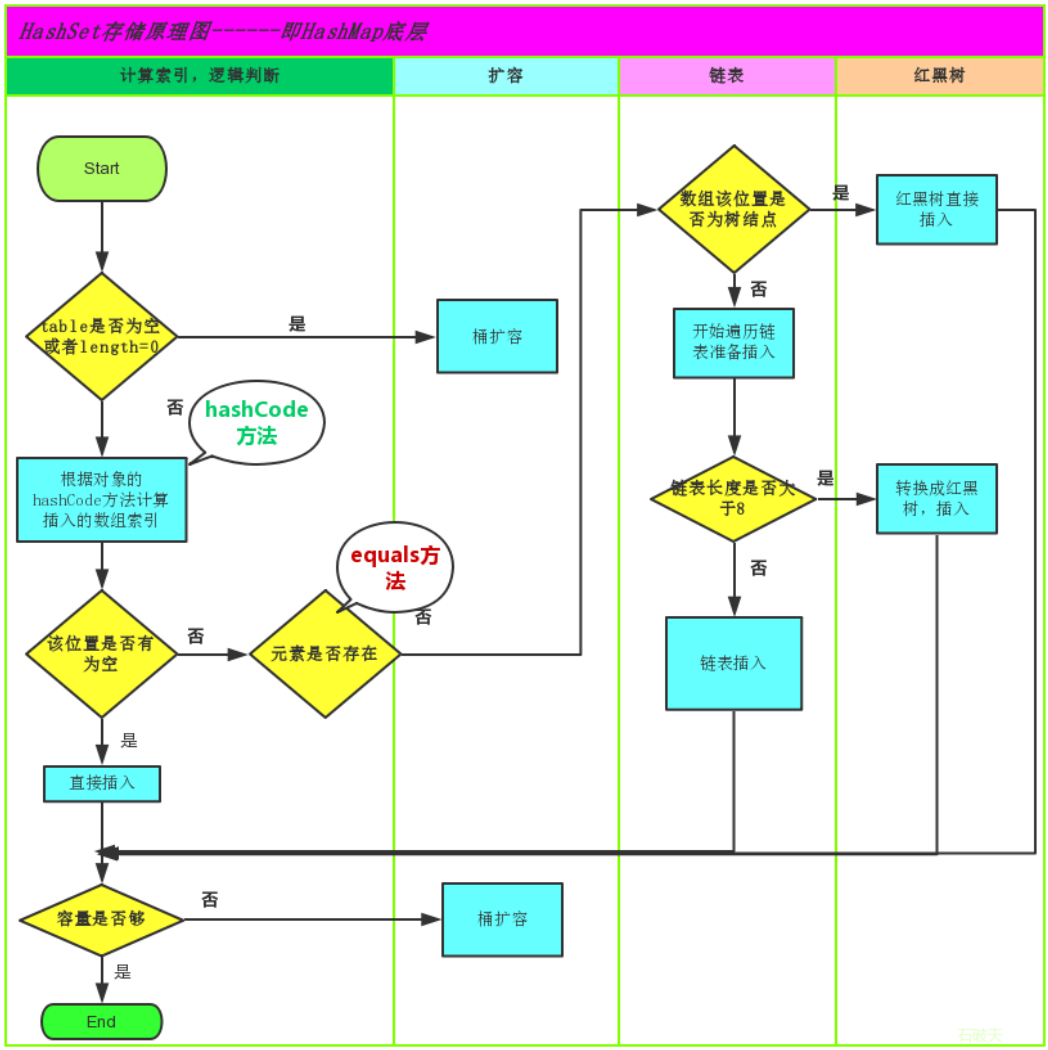
在**JDK1.8**之前，哈希表底层采用数组+链表实现，即使用数组处理冲突，同一hash值的链表都存储在一个数组里。但是当位于一个桶中的元素较多，即hash值相等的元素较多时，通过key值依次查找的效率较低。而JDK1.8中，哈希表存储采用数组+链表+红黑树实现，当链表长度超过阈值（8）时，将链表转换为红黑树，这样大大减少了查找时间。

简单的来说，哈希表是由数组+链表+红黑树（JDK1.8增加了红黑树部分）实现的，如下图所示。



看到这张图就有人要问了，这个是怎么存储的呢？

为了方便大家的理解我们结合一个存储流程图来说明一下：



总而言之，**JDK1.8**引入红黑树大程度优化了HashMap的性能，那么对于我们来讲保证HashSet集合元素的唯一，其实就是根据对象的hashCode和equals方法来决定的。如果我们往集合中存放自定义的对象，那么保证其唯一，就必须复写hashCode和equals方法建立属于当前对象的比较方式。

3.3 HashSet存储自定义类型元素

给HashSet中存放自定义类型元素时，需要重写对象中的hashCode和equals方法，建立自己的比较方式，才能保证HashSet集合中的对象唯一.

创建自定义Student类:

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

    //get/set
    @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);
    }
}
```

创建测试类:

```
public class HashSetDemo2 {
    public static void main(String[] args) {
        //创建集合对象    该集合中存储 Student类型对象
        HashSet<Student> stuSet = new HashSet<Student>();
        //存储
        Student stu = new Student("于谦", 43);
        stuSet.add(stu);
        stuSet.add(new Student("郭德纲", 44));
        stuSet.add(new Student("于谦", 43));
        stuSet.add(new Student("郭麒麟", 23));
        stuSet.add(stu);

        for (Student stu2 : stuSet) {
            System.out.println(stu2);
        }
    }
}
执行结果:
Student [name=郭德纲, age=44]
Student [name=于谦, age=43]
Student [name=郭麒麟, age=23]
```

3.4 HashSet的源码分析

3.4.1 HashSet的成员属性及构造方法

```
public class HashSet<E> extends AbstractSet<E>
                        implements Set<E>, Cloneable, java.io.Serializable{

    //内部一个HashMap--HashSet内部实际上是用HashMap实现的
    private transient HashMap<E,Object> map;
    // 用于做map的值
    private static final Object PRESENT = new Object();
    /**
     * 构造一个新的HashSet,
     * 内部实际上是构造了一个HashMap
     */
    public HashSet() {
        map = new HashMap<>();
    }

}
```

- 通过构造方法可以看出，HashSet构造时，实际上是构造一个HashMap

3.4.2 HashSet的add方法源码解析

```
public class HashSet{
    //......
    public boolean add(E e) {
       return map.put(e, PRESENT)==null;//内部实际上添加到map中，键：要添加的对象，值：
Object对象
    }
    //......
}
```

3.4.3 HashMap的put方法源码解析

```
public class HashMap{
    //......
    public V put(K key, V value) {
        return putVal(hash(key), key, value, false, true);
    }
    //......
    static final int hash(Object key) {//根据参数，产生一个哈希值
        int h;
        return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
    }
    //......
    final V putVal(int hash, K key, V value, boolean onlyIfAbsent,
                   boolean evict) {
        Node<K,V>[] tab; //临时变量，存储"哈希表"--由此可见，哈希表是一个Node[]数组
        Node<K,V> p;//临时变量，用于存储从"哈希表"中获取的Node
        int n, i;//n存储哈希表长度；i存储哈希表索引
```

```
        if ((tab = table) == null || (n = tab.length) == 0)//判断当前是否还没有生成
哈希表
            n = (tab = resize()).length;//resize()方法用于生成一个哈希表，默认长度：
16，赋给n
        if ((p = tab[i = (n - 1) & hash]) == null)//(n-1)&hash等效于hash % n，转换
为数组索引
            tab[i] = newNode(hash, key, value, null);//此位置没有元素，直接存储
        else {//否则此位置已经有元素了
            Node<K,V> e; K k;
            if (p.hash == hash &&
                ((k = p.key) == key || (key != null && key.equals(k))))//判断哈希
值和equals
                e = p;//将哈希表中的元素存储为e
            else if (p instanceof TreeNode)//判断是否为"树"结构
                e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);
            else {//排除以上两种情况，将其存为新的Node节点
                for (int binCount = 0; ; ++binCount) {//遍历链表
                    if ((e = p.next) == null) {//找到最后一个节点
                        p.next = newNode(hash, key, value, null);//产生一个新节点，
赋值到链表
                        if (binCount >= TREEIFY_THRESHOLD - 1) //判断链表长度是否大
于了8
                            treeifyBin(tab, hash);//树形化
                        break;
                    }
                    if (e.hash == hash &&
                        ((k = e.key) == key || (key != null &&
key.equals(k))))//跟当前变量的元素比较，如果hashCode相同，equals也相同
                        break;//结束循环
                    p = e;//将p设为当前遍历的Node节点
                }
            }
            if (e != null) { // 如果存在此键
                V oldValue = e.value;//取出value
                if (!onlyIfAbsent || oldValue == null)
                    e.value = value;//设置为新value
                afterNodeAccess(e);//空方法，什么都不做
                return oldValue;//返回旧值
            }
        }
        ++modCount;
        if (++size > threshold)
            resize();
        afterNodeInsertion(evict);
        return null;
    }
}
```

3.5 LinkedHashSet

我们知道HashSet保证元素唯一，可是元素存放进去是没有顺序的，那么我们要保证有序，怎么办呢？

在HashSet下面有一个子类`java.util.LinkedHashSet`，它是链表和哈希表组合的一个数据存储结构。

演示代码如下:

```
public class LinkedHashSetDemo {
    public static void main(String[] args) {
        Set<String> set = new LinkedHashSet<String>();
        set.add("bbb");
        set.add("aaa");
        set.add("abc");
        set.add("bbc");
        Iterator<String> it = set.iterator();
        while (it.hasNext()) {
            System.out.println(it.next());
        }
    }
}
结果:
  bbb
  aaa
  abc
  bbc
```

3.6 TreeSet集合

3.6.1 特点

TreeSet集合是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,其特点为:

1. 元素唯一
2. 元素没有索引
3. 使用元素的自然顺序对元素进行排序，或者根据创建 TreeSet 时提供的 `Comparator` 比较器 进行排序，具体取决于使用的构造方法:

```
public TreeSet():                                根据其元素的自然排序进行排序
public TreeSet(Comparator<E> comparator):    根据指定的比较器进行排序
```

3.6.2 演示

案例演示**自然排序**(20,18,23,22,17,24,19):

```
public static void main(String[] args) {
    //无参构造,默认使用元素的自然顺序进行排序
    TreeSet<Integer> set = new TreeSet<Integer>();
    set.add(20);
    set.add(18);
    set.add(23);
    set.add(22);
    set.add(17);
    set.add(24);
    set.add(19);
    System.out.println(set);
}

控制台的输出结果为:
```

```
[17, 18, 19, 20, 22, 23, 24]
```

案例演示**比较器排序**(20,18,23,22,17,24,19):

```
public static void main(String[] args) {
    //有参构造,传入比较器,使用比较器对元素进行排序
    TreeSet<Integer> set = new TreeSet<Integer>(new Comparator<Integer>() {
        @Override
        public int compare(Integer o1, Integer o2) {
            //元素前 - 元素后 : 升序
            //元素后 - 元素前 : 降序
            return o2 - o1;
        }
    });
    set.add(20);
    set.add(18);
    set.add(23);
    set.add(22);
    set.add(17);
    set.add(24);
    set.add(19);
    System.out.println(set);
}

控制台的输出结果为:
[24, 23, 22, 20, 19, 18, 17]
```