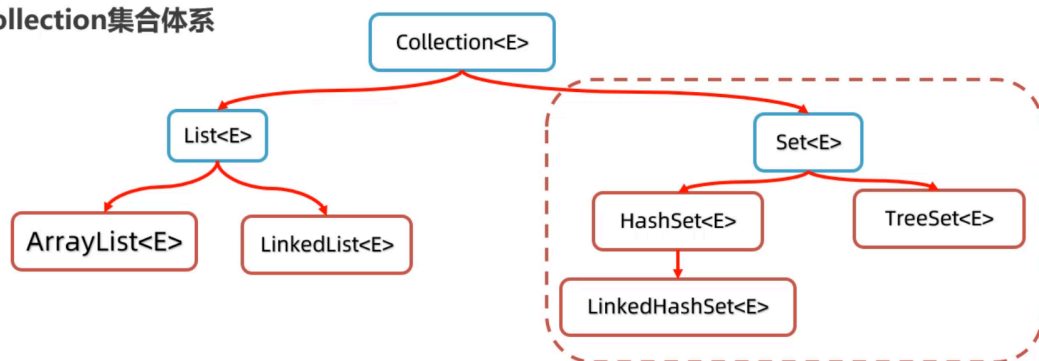


Set 接口

1.Set

1.1 概述

Collection集合体系



接口

实现类

Set系列集合特点：无序：添加数据的顺序和获取出的数据顺序不一致； 不重复； 无索引；

- HashSet：无序、不重复、无索引。
- LinkedHashSet：**有序**、不重复、无索引。
- TreeSet：**排序**、不重复、无索引。

- 1) 概述：是一个接口
- 2) 实现类：HashSet LinkedHashSet TreeSet
- 3) Set 系列集合的特点：无序、不重复、无索引
- 4) 今天介绍的这三个实现类都是线程不安全的

1.2 示例：不同实现类的特点

```
1 package com.itheima.d1_collection_set;
2
3 import com.sun.source.tree.Tree;
4
```

```

5 import java.util.HashSet;
6 import java.util.LinkedHashSet;
7 import java.util.TreeSet;
8
9 /*
10     目标：整体了解一下 Set 系列集合的特点
11
12     Set 集合的体系特点
13         HashSet：无序、不重复、无索引。
14         LinkedHashSet：有序、不重复、无索引。
15         TreeSet：排序、不重复、无索引。
16
17     需求：演示 Set 集合的每一个实现类的特点
18 */
19 public class SetTest1 {
20     public static void main(String[] args) {
21         System.out.println("HashSet: ");
22         HashSet<Object> hs1 = new HashSet<>();
23         // 添加元素
24         hs1.add("bca");
25         hs1.add("bca");
26         hs1.add("cab");
27         hs1.add("abc");
28         // 打印集合对象
29         System.out.println(hs1);
30
31         System.out.println("LinkedHashSet: ");
32         HashSet<Object> hs2 = new LinkedHashSet<>();
33         // 添加元素
34         hs2.add("bca");
35         hs2.add("bca");
36         hs2.add("cab");
37         hs2.add("abc");
38         // 打印集合对象
39         System.out.println(hs2);
40
41         System.out.println("TreeSet: ");
42         TreeSet<Object> hs3 = new TreeSet<>();

```

```

43         // 添加元素
44         hs3.add("bca");
45         hs3.add("bca");
46         hs3.add("cab");
47         hs3.add("abc");
48         // 打印集合对象
49         System.out.println(hs3);
50     }
51 }
52

```

运行结果

```

1 HashSet:
2 [bca, abc, cab]
3 LinkedHashSet:
4 [bca, cab, abc]
5 TreeSet:
6 [abc, bca, cab]

```

1.3 思考

1) 共同特点

- 全部不能存储重复元素，重复添加的 "bca" 只会保留一份。
- 没有索引，不能通过索引取值。

2) HashSet

-  去重； 存取无序，元素顺序和添加顺序无关，依靠哈希值决定存放位置。

3) LinkedHashSet

-  去重； 存取有序，元素顺序等于你调用 `add()` 添加的先后顺序；底层是哈希表 + 双向链表维护顺序。

4) TreeSet

-  去重； 自动排序，字符串按照字典（字母）顺序升序排列；

1.4 关于哈希值

1) 是什么

哈希值：是JDK 根据对象的地址、或者对象的内容，算出来的一个 int 类型整数。

每一个对象都有自己的哈希值

2) 两个关键点

1. 同一个对象，多次调用 `hashCode()`，返回的哈希值一定相同
 2. 不同的对象，哈希值有可能相同 → 哈希冲突
-

2.HashSet

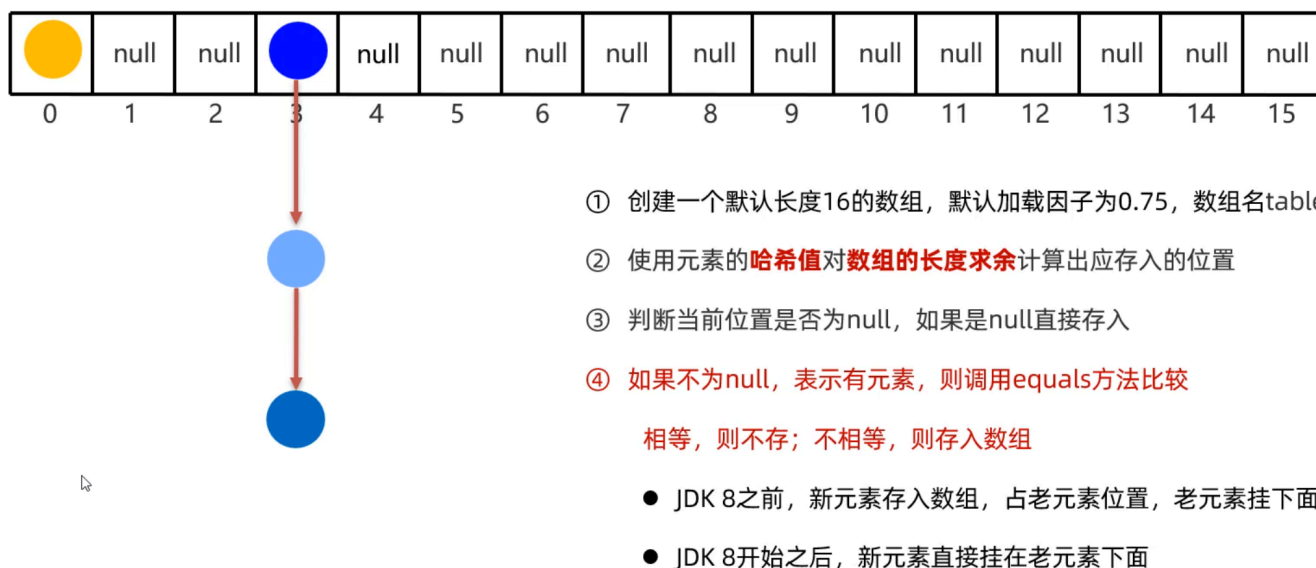
2.1 概述

HashSet 集合底层是基于哈希表实现的，哈希表根据JDK版本的不同，也是有点区别的

- JDK8 以前：哈希表 = 数组+链表
- JDK8 以后：哈希表 = 数组+链表+红黑树

JDK8之前HashSet集合的底层原理，基于哈希表：数组+链表

table[]



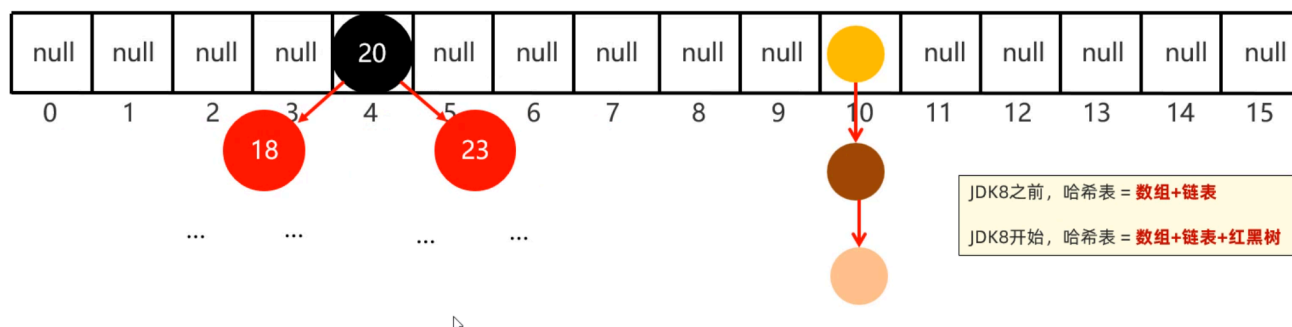
往HashSet集合中存储元素时，底层调用了元素的两个方法：一个是hashCode方法获取元素的hashCode值（哈希值）；另一个是调用了元素的equals方法，用来比较新添加的元素和集合中已有的元素是否相同。

- 只有新添加元素的hashCode值和集合中以后元素的hashCode值相同、新添加的元素调用equals方法和集合中已有元素比较结果为true，才认为元素重复。
- 如果hashCode值相同，equals比较不同，则以链表的形式连接在数组的同一个索引为位置（如上图所示）

在JDK8开始后，为了提高性能，当链表的长度超过8时，就会把链表转换为红黑树，如下图所示：

JDK8开始HashSet集合的底层原理，基于哈希表：数组 + 链表 + 红黑树

Segment[]



JDK8开始，当链表长度超过8，且数组长度 ≥ 64 时，自动将链表转成红黑树

2.2 去重原理

HashSet 存储元素依赖两个方法：一个是 hashCode 方法用来确定在底层数组中存储的位置，另一个是用 equals 方法判断新添加的元素是否和集合中已有的元素相同。

1. 先调用对象 `hashCode()` 算哈希值，定位哈希表数组位置
2. 如果该位置没有元素 → 直接存入
3. 如果位置已有元素：
 - 哈希值不同：判定不同对象，直接存
 - 哈希值相同（哈希冲突）：调用 `equals()` 比较内容
 - `equals()==true`：判定重复，不存储
 - `equals()==false`：哈希冲突，挂在同一个位置链表 / 红黑树上

新建一个学生类和测试类：

```
1 package com.itheima.d1_collection_set;  
2  
3 import java.util.HashSet;
```

```

4
5 public class SetTest3 {
6     public static void main(String[] args) {
7         Student student1 = new Student("张三", "001");
8         Student student2 = new Student("张三", "001");
9         Student student3 = new Student("李四", "003");
10
11         HashSet<Student> hashSet = new HashSet<>();
12         hashSet.add(student1);
13         hashSet.add(student2);
14         hashSet.add(student3);
15
16         System.out.println(hashSet);
17     }
18 }
19

```

```

1 package com.itheima.d1_collection_set;
2
3 public class Student {
4     private String name;
5     private String sid;
6
7     public Student(String name, String sid) {
8         this.name = name;
9         this.sid = sid;
10    }
11
12    public String getName() {
13        return name;
14    }
15
16    public void setName(String name) {
17        this.name = name;
18    }
19

```

```

20     public String getSid() {
21         return sid;
22     }
23
24     public void setSid(String sid) {
25         this.sid = sid;
26     }
27
28     @Override
29     public String toString() {
30         return "Student{" +
31             "name='" + name + '\'' +
32             ", sid='" + sid + '\'' +
33             '}';
34     }
35 }
36

```

如果我不重写 hashCode 方法，HashSet 就会调用 Object 类下的 hashCode 方法，只要 hashCode 不一样，就是不同的对象，其运行结果如下所示：

```

1 [Student{name=' 李 四 ', sid='003'}, Student{name=' 张 三 ', sid='001'},
  Student{name='张三', sid='001'}]

```

现在我重写 hashCode 方法，运行结果就会是这样：

```

1 [Student{name='张三', sid='001'}, Student{name='李四', sid='003'}]

```

就达到去重的效果了！

2.3 总结

1) `HashSet` 无序、不重复、无索引，底层是哈希表

2) `Object` 类已经 JDK 内部帮我们重写好了 `hashCode()` 与 `equals()`，所以存字符串不需要我们手动写；自己写的实体类（`Student`、`User`），要放入 `HashSet` 做内容去重，IDE 一键生成这两个方法即可（IDEA：Alt+Insert → `equals()` and `hashCode()`）。

3) `HashSet` 集合保证元素唯一性，**必须同时重写 `hashCode()` 和 `equals()` 两个方法，缺一不可。**

4) 不能使用普通 for 循环遍历（没有索引），只能使用增强 for、迭代器。

5) 允许存 `null`，最多存一个 `null`（去重）。

6) 判定重复条件：`hashCode` 返回相同 并且 `equals` 返回 `true`

7) JDK1.8 之后哈希表 = 数组+链表+红黑树

8) 链表长度大于 8，链表转成红黑树；当红黑树节点小于 6 时，退化为链表。

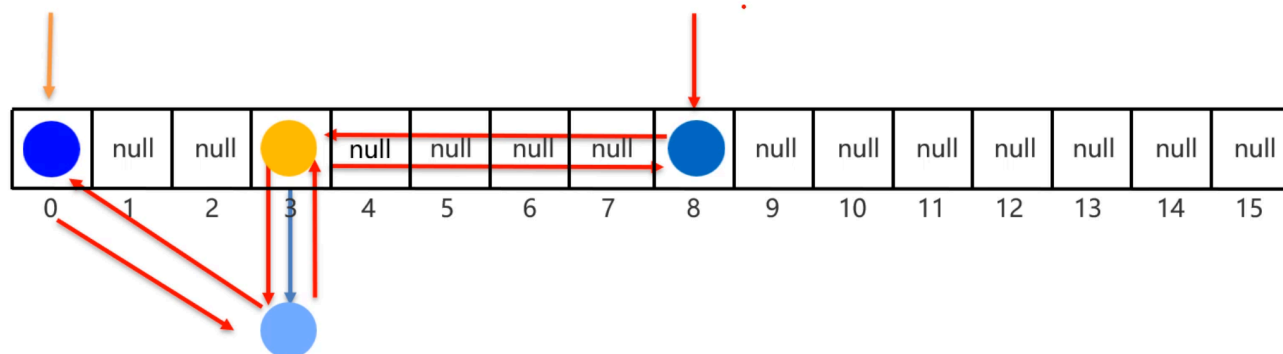
9) `HashSet` 负载因子默认为 0.75：当数组容量达到 75% 就会扩容，数组容量翻倍。

3. LinkedHashSet

3.1 概述

`LinkedHashSet` 底层是**哈希表 + 双向链表**，其特点有：有序，不重复，无索引。和 `HashSet` 相比多了一个双向链表，可以记录存储的顺序。其功能与 `HashSet` 差别不大。

- 依然是基于哈希表(数组、链表、红黑树)实现的。
- 但是，它的每个元素都额外的多了一个双链表的机制记录它前后元素的位置。



每次添加元素，就和上一个元素用双向链表连接一下。第一个添加的元素是双向链表的头节点，最后一个添加的元素是双向链表的尾节点。

4.TreeSet

4.1 概述

TreeSet集合的特点是可以对元素进行排序，但是必须指定元素的排序规则。

如果往集合中存储String类型的元素，或者Integer类型的元素，它们本身就具备排序规则，所以直接就可以排序。

例如：

```
1 Set<Integer> set1= new TreeSet<>();
2 set1.add(8);
3 set1.add(6);
4 set1.add(4);
5 set1.add(3);
6 set1.add(7);
7 set1.add(1);
```

```

8  set1.add(5);
9  set1.add(2);
10 System.out.println(set1); //[1,2,3,4,5,6,7,8]
11
12 Set<Integer> set2= new TreeSet<>();
13 set2.add("a");
14 set2.add("c");
15 set2.add("e");
16 set2.add("b");
17 set2.add("d");
18 set2.add("f");
19 set2.add("g");
20 System.out.println(set1); //[a,b,c,d,e,f,g]

```

如果往TreeSet集合中存储自定义类型的元素，比如说Student类型，则需要我们自己指定排序规则，否则会出现异常。

```

1  package com.itheima.d1_collection_set;
2
3
4  import java.util.Set;
5  import java.util.TreeSet;
6
7
8  public class SetTest3 {
9      public static void main(String[] args) {
10          Student student1 = new Student("张三", "001");
11          Student student2 = new Student("李四", "001");
12          Student student3 = new Student("王五", "003");
13
14          Set<Student> treeSet = new TreeSet<>();
15          treeSet.add(student1);
16          treeSet.add(student2);
17          treeSet.add(student3);
18
19          System.out.println(treeSet);

```

```
20     }
21 }
22
```

```
↑ /Users/liaoshangshaorouzong/Library/Java/JavaVirtualMachines/temurin-21.0.11/Contents/Home/bin/java -javaagent:/Application <
↓ Exception in thread "main" java.lang.ClassCastException: class com.itheima.d1_collection_set.Student cannot
|
| at java.base/java.util.TreeMap.compare(TreeMap.java:1604)
| at java.base/java.util.TreeMap.addEntryToEmptyMap(TreeMap.java:811)
| at java.base/java.util.TreeMap.put(TreeMap.java:820)
| at java.base/java.util.TreeMap.put(TreeMap.java:569)
| at java.base/java.util.TreeSet.add(TreeSet.java:259)
| at com.itheima.d1_collection_set.SetTest3.main(SetTest3.java:15)
```

有两种方法来解决这个问题：

4.2 指定排序规则

4.2.1 自然排序

让元素的类实现Comparable接口，重写compareTo方法

```
1 package com.itheima.d1_collection_set;
2
3 import java.util.Objects;
4
5 public class Student implements Comparable<Student>{
6     private String sid;
7     private String name;
8     private int score;
9
10    public Student(String sid, String name, int score) {
11        this.sid = sid;
12        this.name = name;
13        this.score = score;
14    }
15
16    public String getName() {
17        return name;
18    }
19 }
```

```

18     }
19
20     public void setName(String name) {
21         this.name = name;
22     }
23
24     public String getSid() {
25         return sid;
26     }
27
28     public void setSid(String sid) {
29         this.sid = sid;
30     }
31
32     public int getScore() {
33         return score;
34     }
35
36     public void setScore(int score) {
37         this.score = score;
38     }
39
40     @Override
41     public boolean equals(Object o) {
42         if (o == null || getClass() != o.getClass()) return false;
43         Student student = (Student) o;
44         return score == student.score && Objects.equals(sid,
student.sid) && Objects.equals(name, student.name);
45     }
46
47     @Override
48     public int hashCode() {
49         return Objects.hash(sid, name, score);
50     }
51
52     @Override
53     public String toString() {
54         return "Student{" +

```

```

55         "sid='" + sid + '\'' +
56         ", name='" + name + '\'' +
57         ", score=" + score +
58         '}'';
59     }
60
61     @Override
62     public int compareTo(Student o) {
63         return this.score - o.score;
64     }
65 }
66

```

第一步：实现 Comparable 接口

```

public class Student implements Comparable<Student>{
    private String sid;
    private String name;
    private int score;
}

```

第二步：重写 compareTo() 方法

```

@Override
public int compareTo(Student o) {
    return this.score - o.score;
}

```

排序规则：

根据compareTo方法的返回值进行指定元素位置

如果返回值为负数，表示当前存入的元素是较小值，存左边

如果返回值为0，表示当前存入的元素相等，不存

如果返回值为正数，表示当前存入的元素是较大值，存右边

this：当前添加的元素

o：集中存在的元素

反思：

```

1 @Override
2 public int compareTo(Student o) {
3     return this.score - o.score;
4 }

```

这样子写是不是太笼统了呢？如果两个学生的成绩是一样的话，那他不就只保留最前面那个对象了吗？事实果真如此

```
public static void main(String[] args) {
    Student student1 = new Student(sid: "001", name: "张三", score: 98);
    Student student2 = new Student(sid: "002", name: "李四", score: 98);
    Student student3 = new Student(sid: "003", name: "王五", score: 94);
}
```

```
/Users/liaoshangshaorouzong/Library/Java/JavaVirtualMachines/temurin-21.0.11/Contents/Home/bin/java -Djava.class.path=. Student
[Student{sid='003', name='王五', score=94}, Student{sid='001', name='张三', score=98}]
```

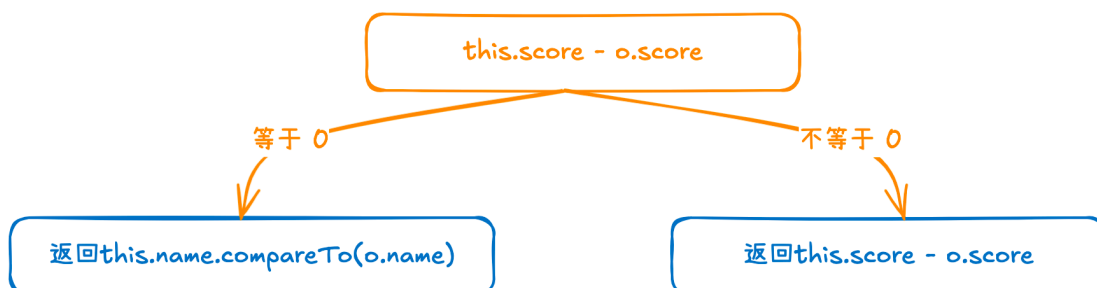
进程已结束，退出代码为 0

可以看到运行结果并没有李四这一项。我们要进行一些改动：

```
1 @Override
2 public int compareTo(Student o) {
3     return this.score - o.score == 0 ? this.name.compareTo(o.name) :
4         this.score - o.score;
5 }
```

这里运用一个三元运算符来进行判断，

```
1 @Override
2 public int compareTo(Student o) {
3     return this.score - o.score == 0 ? this.name.compareTo(o.name) : this.score - o.score;
4 }
```



比较名字 `this.name.compareTo(o.name)` (字符串字典顺序)

直接返回 `this.score - o.score`, 按分数升序排序

改进后

```
1 package com.itheima.d1_collection_set;
2
```

```

3 import java.util.Objects;
4
5 public class Student implements Comparable<Student>{
6     private String sid;
7     private String name;
8     private int score;
9
10    public Student(String sid, String name, int score) {
11        this.sid = sid;
12        this.name = name;
13        this.score = score;
14    }
15
16    public String getName() {
17        return name;
18    }
19
20    public void setName(String name) {
21        this.name = name;
22    }
23
24    public String getSid() {
25        return sid;
26    }
27
28    public void setSid(String sid) {
29        this.sid = sid;
30    }
31
32    public int getScore() {
33        return score;
34    }
35
36    public void setScore(int score) {
37        this.score = score;
38    }
39
40    @Override

```

```

41     public boolean equals(Object o) {
42         if (o == null || getClass() != o.getClass()) return false;
43         Student student = (Student) o;
44         return score == student.score && Objects.equals(sid,
student.sid) && Objects.equals(name, student.name);
45     }
46
47     @Override
48     public int hashCode() {
49         return Objects.hash(sid, name, score);
50     }
51
52     @Override
53     public String toString() {
54         return "Student{" +
55             "sid='" + sid + '\'' +
56             ", name='" + name + '\'' +
57             ", score=" + score +
58             '}';
59     }
60
61     @Override
62     public int compareTo(Student o) {
63         return this.score - o.score == 0 ? this.name.compareTo(o.name)
: this.score - o.score;
64     }
65 }
66

```

```

1  package com.itheima.d1_collection_set;
2
3
4  import java.util.Set;
5  import java.util.TreeSet;
6
7

```

```

8 public class SetTest3 {
9     public static void main(String[] args) {
10         Student student1 = new Student("001", "张三", 98);
11         Student student2 = new Student("002", "李四", 98);
12         Student student3 = new Student("003", "王五", 94);
13
14         Set<Student> treeSet = new TreeSet<>();
15         treeSet.add(student1);
16         treeSet.add(student2);
17         treeSet.add(student3);
18
19         System.out.println(treeSet);
20     }
21 }
22

```

```

1 [Student{sid='003', name='王五', score=94}, Student{sid='001', name='张三', score=98}, Student{sid='002', name='李四', score=98}]

```

总结:

1. **实现接口**: 实体类实现 `Comparable<泛型>`, 重写 `compareTo()` 方法。 `this` 代表当前对象, `o` 代表被比较对象。

2. 返回值规则

- 返回负数: `this < o`, 当前对象排在前面
- 返回0: 判定对象重复, `TreeSet` 拒绝存入该元素
- 返回正数: `this > o`, 传入对象 `o` 排在前面

3. 升降序写法

- 升序: `this.属性 - o.属性`
- 降序: `o.属性 - this.属性`

4. 多条件排序（示例：先分数，后名字）

```
1 return this.score - o.score == 0 ? this.name.compareTo(o.name) :  
   this.score - o.score;
```

逻辑：①优先按分数升序；②分数相等，执行次要条件，调用 String 的 `compareTo()` 按名字字典顺序升序；③只有分数、名字全部相同才返回 0，才判定重复，避免同分不同名对象被错误删除。

5. 核心考点

- TreeSet 不依赖 `hashCode()` 与 `equals()`，完全依靠 `compareTo()` 返回值完成排序 + 去重。
- 如果只写主条件，次要属性不同但主条件相等时，会返回 0，把不同对象当成重复删掉。
- 字符串比较必须调用 `字符串.compareTo()`，不能直接相减。

4.2.2 比较器排序

原理：根据 `compareTo` 方法的返回值进行指定元素位置

如果返回值为负数，表示当前存入的元素是较小值，存左边 -> 升序

如果返回值为 0，表示当前存入的元素相等，不存

如果返回值为正数，表示当前存入的元素是较大值，存右边 -> 降序

this 表示当前要存储的元素，o 表示的是已经存在的元素

```
1 package com.itheima.d1_collection_set;  
2  
3 import com.sun.source.tree.Tree;  
4  
5 import java.util.Comparator;  
6 import java.util.Set;  
7 import java.util.TreeSet;
```

```

8
9 public class SetTest4 {
10     public static void main(String[] args) {
11         Student student1 = new Student("001", "张三", 98);
12         Student student2 = new Student("002", "李四", 98);
13         Student student3 = new Student("003", "王五", 94);
14
15         TreeSet<Student> treeSet = new TreeSet<>(new
16     Comparator<Student>() {
17         @Override
18         public int compare(Student o1, Student o2) {
19             // o1: 当前要存的元素; o2: 已经存的元素
20             return o2.getScore() - o1.getScore() == 0 ?
21                 o2.getName().compareTo(o1.getName()) : o2.getScore() - o1.getScore();
22         }
23     });
24     treeSet.add(student1);
25     treeSet.add(student2);
26     treeSet.add(student3);
27
28     for (Student s : treeSet) {
29         System.out.println(s);
30     }
31 }

```

完

最后修改: 2026 年 8 月 22 日