

# Set集合

## Set集合的基本概述和特点

概述: public interface Set<E> extends Collection{}

1.不可重复的单列集合容器

2.主要有两个实现类:HashSet, TreeSet

特点:

1.可以去除重复

2.存储顺序不一致

3.没有带索引的方法,所以不能使用for循环遍历,也不能通过索引来获取,删除Set集合里面的元素

## TreeSet集合

TreeSet集合特点

1.不包含重复元素的集合

2.没有带索引的方法

3.可以将元素按照规则进行排序

<在存的时候给元素按照一定规则排序,也就是说,打印的时候时按照顺序打印出来的>

**特别的:**

**自定义类使用TreeSet集合存储元素/对象,需要制定排序规则**

## TreeSet自然排序

1.自定义类实现Comparable接口,重写CompareTo方法,指定规则.

2.在使用TreeSet集合存储自定义对象时,就会按照指定的规则进行排序,并且在排序的时候会不断调用CompareTo方法,直到排序完成.

@Override

public int compareTo(Student o) {

/\*这里只比较了年龄

return this.age - o.age;

\*/

//主要判断

int result = this.age - o.age;

//次要判断

//this.name.compareTo(o.name)是String类中的方法

result = result == 0 ? this.name.compareTo(o.name) : result;

return result;

}

## TreeSet比较器排序

比较器排序Comparator的使用

1.TreeSet的带参构造方法使用的是比较器排序对元素进行排序的

<TreeSet​(Comparator<? super E> comparator) 构造一个新的，空的树集，根据指定的比较器进行排序>

2.比较器,就是让集合构造方法接收Comparator的实现类对象,重写Compar(T o1, T o2)方法

3.重写方法时,一定要注意排序规则按照要求指定主要条件和次要条件来重写Compar(T o1, T o2)方法

举例:

TreeSet<Teacher> ts = new TreeSet<>(new Comparator<Teacher>() {

@Override

public int compare(Teacher o1, Teacher o2) {

//o1代表将要录入的数据,o2代表已经存在的数据

//主要条件

int result = o1.getAge() - o2.getAge();

//次要条件

result = result == 0 ? o1.getName().compareTo(o2.getName()) : result;

return result;

}

});

Teacher t1 = new Teacher("zhangsan", 23);

Teacher t2 = new Teacher("lisi", 22);

Teacher t3 = new Teacher("wangwu", 24);

Teacher t4 = new Teacher("zhuliu", 24);

Teacher t5 = new Teacher("zhangsan", 23);

ts.add(t1);

ts.add(t2);

ts.add(t3);

ts.add(t4);

ts.add(t5);

System.out.println(ts);

## 两种排序方式小结

1.自然排序:自定义类实现Comparable接口,重写comparTo方法,根据返回值进行排序;

2.比较器排序:创建TreeSet对象的时候传递Comparator的实现类对象,重写compare方法, 根据返回值进行排序;

3.在使用的时候,默认使用自然排序,当自然排序不满足现在的需求的时候<比如JDK的类>,使用比较器排序

## 数据结构-树

TreeSet底层数据结构-树

树的分类:

1.二叉树

2.二叉查找树

3.平衡二叉树

4.红黑树 TreeSet的底层是红黑树

特别的<理解>:

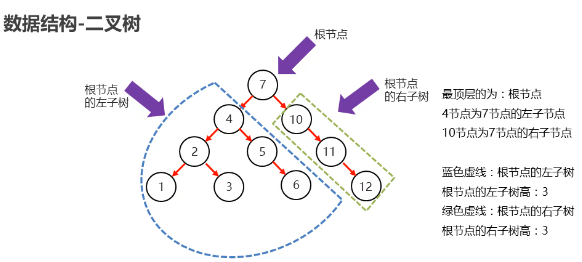
数据结构底层有三种,数组(ArrayList),链表(LinkeList),树

## 二叉树

**节点:就是每个数据,或者说是对象**

**度:每个节点的子节点数量**

**二叉树:节点的度<=2的树的数据结构.**



二叉树的专业名词

整棵树高度 = 层数;

层数:几层,几个子节点 + 1,就是层数, + 1是加的自己;

根节点:祖宗节点;

左子节点: 父节点的左边子节点;

右子节点: 父节点的右边子节点;

左子树:根节点的左子节点下面的小树;

右子树:根节点的右子节点下面的小树;

最后:如图.

## 二叉查找树

或者叫二叉排序树,二叉搜索树

任意一个节点的左子节点都比本节点小,右子节点都比本节点大.

二叉查找树的存入(添加)规则:

小的存左边

大的存右边

相同的不存

## 平衡二叉树

平衡二叉树的概述

提高二叉树的查找效率

二叉树左右两个子树的高度差不超过1

任意节点的左右两个子树都是一颗平衡二叉树

平衡二叉树-旋转

触发机制:

当添加一个节点以后,该树不再是一颗平衡二叉树时

**左旋:**就是将根节点的右侧往左拉,原先的右子节点变成新的父节点,并把原先的右子节点的左子节点出让,给已经降级的根节点,,原先的根节点作为新父节点的左子节点,并且如果新父节点原来有左子节点,则新的左子节点(降级的根节点)接收它为右子节点

**右旋:**就是和左旋相反.

平衡二叉树的左左和左右

左左:当根节点左子树的左子树有节点插入,导致二叉树的不平衡

进行一次右旋就可以了

左右:当根节点左子树的右子树有节点插入,导致二叉树的不平衡

先进行左子节点左旋,再整体进行右旋才可以

平衡二叉树的右右和右左

右右:当根节点右子树的右子树有节点插入,导致二叉树的不平衡

进行一次左旋就可以了

右左:当根节点右子树的左子树有节点插入,导致二叉树的不平衡

先进行右子节点右旋,再整体进行左旋才可以

## 红黑树

红黑树概述:

1.红黑树是一种自平衡的二叉查找树,是计算机科学中用到的一种数据结构.

2.每一个节点可以是红或者黑

3.红黑树不是高度平衡的,他的平衡时通过”自己的红黑规则”进行实现的

红黑规则

1.每一个节点要么时红色的,要么是黑色的;

2.根节点必须是黑色的;

3.如果一个节点没有子节点或者父节点,则该节点相应的指针属性值为Nil,这些Nil视为叶节点,每个叶节点(Nil)是黑色的;

4.如果某一个节点是红色,那么它的子节点必须是黑色(不能出现两个红色节点相连的情况);

5.对每个节点,从该节点到其所有后代叶节点的简单路径上,均包含相同数目的黑色节点;

红黑树添加节点

添加的节点的颜色,可以是红色也可以是黑色

添加三个元素时如果默认颜色是黑色,则在使用红黑树添加元素时需要调整两次颜色;

添加三个元素时如果默认颜色是红色,则在使用红黑树添加元素时需要调整一次颜色;

所以,默认红色,效率高

红黑树添加多个节点

添加的节点默认是红色的.

## HashSet

HashSet集合的概述和特点

概述: 该类实现Set接口，由哈希表（实际为HashMap实例）支持。

public class HashSet<E> extends AbstractSet<E> implements Set<E>, Cloneable, Serializable

特点:

1.底层数据结构是哈希表;

2.不能保证存取顺序一致;

3.没有带索引的方法,所以不能使用普通for循环遍历;

4.实现Set集合,所以元素唯一

## 哈希值

哈希值:是JDK根据对象的地址或者字符串或者数字算出来的int类型的数值

Object类中有一个方法可以获取对象的哈希值

public int hashCode(): 返回对象的哈希码值

对象的哈希值特点:

如果没有重写hashCode()方法,那么是根据对象的地址值计算出来的哈希值.

同一个对象多次调用hashCode()方法返回的哈希值是相同的

不同对象的哈希值是不一样的.

如果重写了hashCode()方法,一般都是通过对象的属性计算出哈希值.

如果不同的对象属性值是一样的,那么计算出来的那细致也是一样的.

(同一个对象多次调用hashCode()方法返回的哈希码值是相同的;

默认情况下,不同对象的哈希码值是不同的.而重写hashCode()方法,可以实现让不同对象的哈希值相同.)

## 常见数据结构之哈希表

哈希表

JDK8之前,底层采用数组+链表实现.

JDK8之后,底层进行了优化,由数组+链表+红黑树实现

HashSet 1.7版本原理

1.底层结构:哈希表,根据元素的哈希值跟数组长度计算出应该存入的位置.(数组+链表);

2.数组长度默认为16(索引0-15),加载因子为0.75(存入元素个数/数组长度>0.75时,数组长度\*2);

3.首先会先获取元素的哈希值,计算出在数组中应存入的索引:

1> 判断该索引处是否为null;

2> 如果是null,直接添加;

3> 如果不是null,则与链表中所有元素通过equals()方法比较属性值;

4> 只要有一个相同就不存,如果都不一样,才会存入集合;

HashSet 1.8版本原理

1.底层结构:哈希表(同上),(数组,链表,红黑树的结合体);

2.当华仔下面的元素过多,不利于添加和查询;

3.所以当链表长度超过8的时候,自动转换为红黑树;

4.存储流程不变,只是说,在数组每个索引位置可能是链表也可能是红黑树

特别的:

如果使用HashSet集合存储自定义对象,则需要重写hashCode()和equals()方法,不然两个方法比较的都是地址值和地址值属性,而应该比较的是成员属性.

## Set集合小结

# Map集合

## Map集合的概述

public interface Map<K,V>

1.双列集合

2.K:键的数据类型 V:值的数据类型

3.键不可以重复,值可以重复

4.键和值是一一对应的,每一个键只能找到自己对应的值

5.(键 + 值) 这个整体称之为”键值对”或者”键值对对象”,在Java中叫做”Entry对象”.

## Map集合的常用方法

Map<String, String> map = new HashMap<>();

map.put(K key, V value); 添加元素

map.remove(Object key); 根据键删除键值对元素

map.clear(); 移除所有键值对元素

map.containsKey(Object key); 判断集合是否包含指定的键

map.containsValue(Object value); 判断集合是否包含指定的值

map.isEmpty(); 判断集合是否为空

map.size(); 返回集合长度,也就是键值对个数

几个注意点:

put方法添加键值对时如果键不存在那就直接添加键值对,如果添加的键值对的键已存,在那么会替换键上的值,并把替换下来的返回.

remove方法也会返回被删除的值,不是键值对.

## Map集合的遍历

第一种

Set<K>keyset(): 获取Map集合中所有的键的一个集合

V get(Object key): 根据键获取值

Map<String, String> map = new HashMap<>();

map.put("1号老公", "1号老婆");

map.put("2号老公", "2号老婆");

map.put("3号老公", "3号老婆");

map.put("4号老公", "4号老婆");

map.put("5号老公", "5号老婆");

Set<String> keys = map.keySet();

for (String key : keys) {

String value = map.get(key);

System.out.println(key + "-" + value);}

第二种

Set<Map.Entry<K, V>>entrySet(): 获取所有键值对对象的Set集合

K getKey(): 获取键

V getValue(): 获取值

Set<Map.Entry<String, String>> entries = map.entrySet();

for (Map.Entry<String, String> entry : entries) {

String key = entry.getKey();

String value = entry.getValue();

System.out.println(key + "-" + value);

}

## HashMap

概述: public class HashMap<K,V> extends AbstractMap<K,V> implements Map<K,V>, Cloneable, Serializable

基于哈希表的实现的Map接口。 该实现提供了所有可选的映射操作，并允许null值和null密钥。

特点:

1.HashMap是Map里面的一个实现类

2.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

3.HashMap跟HashSet一样底层是哈希表结构的

小结:

1.HashMap底层是哈希表结构的

2.依赖hashCode()和equals()方法来保证键的唯一

3.如果键要存储的是自定义类的对象,则需要在自定义类中重写hashCode()和equals()方法

## HashMap的三种遍历方式

第一种和第二种同Map,获取键集合,遍历得到每个值,打印;获取键值对的Set集合,遍历得到每个键和值,在打印.

第三种遍历方式:

//第三种遍历方式 **源码论证**

map.forEach((Student key, String value)->{

System.out.println(key + "-" + value);

});

测试类中: 集合对象map调用forEach方法,传入一个匿名内部类对象,遍历打印key和value;

源码中:

1.首先forEach方法是HashMap的对象调用的方法

@Override

public void forEach(BiConsumer<? super K, ? super V> action) {

Node<K,V>[] tab;

if (action == null)

throw new NullPointerException();

if (size > 0 && (tab = table) != null) {

int mc = modCount;

for (Node<K,V> e : tab) {

for (; e != null; e = e.next)

action.accept(e.key, e.value);

}

if (modCount != mc)

throw new ConcurrentModificationException();

}

}

此处:重写了Map的forEach

2.再看Map中的forEach方法

default void forEach(BiConsumer<? super K, ? super V> action) {

Objects.requireNonNull(action);

for (Map.Entry<K, V> entry : entrySet()) {

K k;

V v;

try {

k = entry.getKey();

v = entry.getValue();

} catch (IllegalStateException ise) {

// this usually means the entry is no longer in the map.

throw new ConcurrentModificationException(ise);

}

action.accept(k, v);

}

}

此处:方法的形参传递了一个BiConsumer接口的实现类对象,所以调用这个方法需要用到多态<接口不能直接创建对象>.因此在测试类中使用map的forEach方法时,使用Lambda表达式,简化了匿名内部类对象编码.然后此对象需要重写BiConsumer中的action方法,此方法需要传入键和值对象.在方法体中第一行略过,第二行是一个增强for循环遍历得到每一个键值对的元素,并赋值给对象k和v.中间在get方法得到k和v的时候抓取了一个异常,for循环外调用了action方法.而在测试类中,重写了action方法<本来就是要重写的,因为BiConsumer中action是抽象方法>,重写内容就是打印k和v.由此就是第三种遍历方式

## TreeMap

概述: public class TreeMap<K, V> extends AbstractMap<K, V> implements NavigableMap<K, V>, Cloneable, Serializable

一个基于红黑树的NavigableMap实现。 该地图根据其键值的natural ordering或地图创建时提供的Comparator进行排序，具体取决于使用的构造函数。

意思就是说,是Map的实现类,在创建时使用自然排序或者比较器排序.

特点:

1.TreeMap是Map的一个实现类.

2.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(抄视频,不是特点)

3.TreeMap跟TreeSet一样底层是红黑树结构的.

小结:

1.TreeMap底层是红黑树结构的

2.依赖自然排序或者比较器排序,对键进行排序

3.如果键存储的是自定义类的对象,则需要在自定义类中重写hashCode()和equals()方法

## TreeMap集合的遍历

和HashMap一样有三种遍历方式

不同的是,使用TreeMap集合遍历自定义对象的时候,要么自定义类实现Comparable接口,重写ComparTo方法;要么创建TreeMap对象时使用有参构造传入一个比较器对象Comparator匿名内部类重写Comparator

# 可变参数

可变参数:就是形参个数是可以变换的

格式:修饰符 返回值类型 方法名(数据类型...变量名){}

举例: public static int getSum(int...a){}

可变参数注意事项

1.这里的变量其实是一个数组

2.如果一个方法有多个参数,并包含可变参数,可变参数要放在最后

## 创建不可变集合

Static<E> List<E> of(E...elements) 创建一个具有指定元素的List集合对象

Static<E> Set<E> of(E...elements) 创建一个具有指定元素的Set集合对象

Static<K, V> Map<K, V> of(E...elements) 创建一个具有指定元素的Map集合对象

**例如:**

List<String> ofList = List.of("zhangsan", "lisi", "wangwu", "zhuliu");

System.out.println(ofList);

Set<Integer> ofSet = Set.of(1, 2, 3, 4, 5, 6);

System.out.println(ofSet);

Map<Integer, Integer> ofMap = Map.of(0, 1, 2, 3, 4, 5, 6, 7, 8, 9);

System.out.println(ofMap);

**特别的:** 以后再创建ArrayList等集合对象在添加集合元素时,可以不再需要使用put或者add方法了,简化为:

ArrayList<String> array = new ArrayList<>(List.of("zhangsan", "lisi", "wangwu", "zhuliu"));

在不影响长度可变的前提下,将已经确定的元素用一个不可变集合对象一次性添加到集合中,不用挨个添加了.