

Collection

Collection	1
1. Set	3
1.1. SortedSet.....	3
1.1.1. TreeSet	3
1.2. HashSet	9
1.2.1. LinkedHashSet.....	9
2. Queue.....	10
2.1. Queue 接口中定义的方法.....	10
2.1.1. void add(Object e).....	10
2.1.2. boolean offer(Object e)	10
2.1.3. Object element()	10
2.1.4. Object peek().....	10
2.1.5. Object poll()	10
2.1.6. Object remove()	11
2.2. PriorityQueue.....	11
2.3. Deque.....	11
2.3.1. ArrayDeque	11
2.3.2. Deque 接口里定义的双端队列的方法.....	12
3. List	14
3.1. LinkedList	15
3.2. ArrayList	15
3.3. Vector.....	15
3.3.1. Stack.....	15
3.4. List 集合增加根据索引来操作集合元素的方法	15
3.4.1. void add(int index,Object element).....	15
3.4.2. boolean addAll(int index, Colleciton c).....	15
3.4.3. Object get(int index)	15
3.4.4. int indexOf(Object o)	16
3.4.5. int lastIndexOf(Object o).....	16
3.4.6. Object remove(int index)	16
3.4.7. Object set(int index, Object element)	16
3.4.8. List subList(int fromIndex,int toIndex).....	16
3.4.9. void sort(Comparator c).....	16
3.4.10. void replaceAll(UnaryOperator operator).....	16
3.5. List 提供了一个额外 listIterator()方法	16
3.5.1. boolean hasPrevious()	17
3.5.2. Object previous()	17
3.5.3. void add(Object o)	17
4. 集合遍历.....	17

4.1. Lambda 表达式遍历集合.....	17
4.2. Java8 增强的 Iterator 遍历集合元素	17
4.3. 使用 Lambda 表达式遍历 Iterator	18
4.4. 使用 foreach 遍历集合元素	18
5. 接口中的方法.....	18
5.1. boolean add(Object o)	18
5.2. boolean addAll(Collection c)	18
5.3. void clear()	18
5.4. boolean contains(Object o)	18
5.5. boolean containsAll(Collection c)	18
5.6. boolean empty().....	18
5.7. Iterator iterator()	19
5.8. boolean remove(Object c)	19
5.9. boolean removeAll(Colleciton o)	19
5.10. boolean retainAll(Collection c).....	19
5.11. int size()	19
5.12. Object[] toArray()	19
6.	19

Java 集合大致可以分为 Set、List、Queue 和 Map。

- Set 代表无序、不可重复的集合；
- List 代表有序、可重复的集合；
- Map 代表具有映射关系的集合；
- Queue 代表一种队列集合的实现。

Java 的集合类主要由两个接口派生出来：Collection 和 Map，这两个接口又包含了一些子接口或实现类。

Set 和 List 接口是 Collection 接口派生的两个子接口，他们分别代表了无序集合和有序集合。

Collection 接口是 List、Set 和 Queue 接口的父接口，该接口里定义的方法既可用于操作 Set 集合，也可用于操作 List 和 Queue 集合。

1. Set

比较：

- HashSet 的性能总是比 TreeSet 好，特别是最常用的添加、查询元素等操作。当需要一个保持排序的 Set 时，才是用 TreeSet，否则都使用 HashSet。
- LinkedHashMap 比 HashSet 在普通插入、删除操作上要稍微慢一点，但是遍历 LinkedHashMap 会更快。
- EnumSet 是所有 Set 实现类中性能最好的，但它只保存同一个枚举类的枚举值作为集合元素。

Set 的三个实现类 HashSet、TreeSet 和 EnumSet 都是线程不安全的。如果有多个线程同时访问一个 Set 集合，并且有一个线程修改了该 Set，则必须手动保证该 Set 集合的同步性。通常可以通过 Collections 工具类的 synchronizedSortedSet 方法来“包装”该 Set 集合。此操作最好在创建时进行，以防止对 Set 集合的以外非同步访问。

```
SortedSet s = Collections.synchronizedSortedSet(new TreeSet(....));
```

1.1. SortedSet

1.1.1. TreeSet

- 1) TreeSet 是 SortedSet 接口是实现类，它可以确保集合元素处于排序状态。与 HashSet 集合相比，TreeSet 还提供了如下几个额外的方法。
- 2) 因为 TreeSet 是有序的，所以增加了访问第一个、前一个、后一个、最后一个元素的方法，并提供了三个从 TreeSet 中截取子 TreeSet 的方法。
- 3) TreeSet 并不是根据元素的插入顺序进行排序的，而是根据实际值的大小进行排序的。
- 4) 与 HashSet 采用 hash 算法来决定元素的存储位置不同，TreeSet 采用红黑树的数据结构来存储集合元素。TreeSet 支持两种排序方法：自然排序和定制排序
- 5) TreeSet 可以删除没有被修改实例变量、且与其他被修改实例变量的对象重复的对象。同时在 TreeSet 中新增元素同样不会改变原有数据的先后顺序。
- 6) 说明

```

package collection;

import java.util.TreeSet;

class Err implements Comparable{
    private int age;

    public Err(){ }
    public Err(int age){
        this.age = age;
    }
    public int getAge(){
        return this.age;
    }

    @Override
    public int compareTo(Object o) {
        return this.age - ((Err)o).getAge();
    }

    public String toString(){
        return this.age + "";
    }

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

public class TreeSetTest {

    public static void main(String[] args) {
        TreeSet ts = new TreeSet();
        Err err = new Err(5);
        ts.add(err);
        ts.add(new Err(-3));
        ts.add(new Err(9));
        ts.add(new Err(-2));
        System.out.println(ts);//1
        ((Err)ts.first()).setAge(20);
        ((Err)ts.last()).setAge(-2);
    }
}

```

```

        System.out.println(ts);//2
        ts.remove(new Err(-2));
        System.out.println(ts);//3
        ts.remove(new Err(5));
        System.out.println(ts);//4
        ts.remove(new Err(-2));
        System.out.println(ts);//5
    }
}

```

当执行了 4 代码后，TreeSet 会对集合中的元素重新索引（不是重新排序），接下来就可以删除 TreeSet 中的所有元素了，包括哪些被修改过实例变量的元素。

与 HashSet 相比，TreeSet 还提供了额外的方法

Comparator comparator()

如果 TreeSet 采用了定制排序，则该方法返回定制排序所使用的 Comparator；如果 TreeSet 采用了自然排序，则返回 null。

Object first()

返回集合中第一个元素

Object last()

返回集合中最后一个元素

Object lower(Object e)

返回集合中位于指定元素之前的元素（即小于指定元素的最大元素，参考元素不需要是 TreeSet 集合里的元素）

Object higher(Object e)

返回集合中位于指定元素之后的元素（即大于指定元素的最小元素，参考元素不需要是 `TreeSet` 集合里的元素）

SortedSet subSet(Object fromElement, Object toElement)

返回此 `Set` 的子集合，范围从 `fromElement`（包含）到 `toElement`（不包含）

SortedSet headSet(Object toElement)

返回此 `Set` 的子集，由小于 `toElement` 的元素组成

SortedSet tailSet(Object fromElement)

返回此 `Set` 的子集，由大于或等于 `fromElement` 的元素组成。

排序方法

Java 提供了一个 `Comparable` 接口，该接口里定义了一个 `compareTo(Object obj)` 方法，该方法返回一个整数值，实现该接口的类必须实现该方法，实现了该接口的类的对象就可以比较大小。

`obj1.compareTo(obj2)`，如果返回 0 表示两个对象相等；如果返回一个正整数，则表明 `obj1 > obj2`，如果返回一个负整数，则表明 `obj1 < obj2`。

java 中一些常用类已经实现了 `Comparable` 接口，并提供了比较大小的标准。

- 1) `BigDecimal`、`BigInteger` 以及所有的数值型对应的包装类：按它们对应的数值大小进行比较。
- 2) `Character`：按字符的 `UNICODE` 值进行比较
- 3) `Boolean`：true 对应的包装类实例大于 false 对应的包装类实例。
- 4) `String`：按照字符串中字符的 `UNICODE` 值进行比较。
- 5) `Date`、`Time`：后面的时间、日期比前面的时间、日期大

如果试图把一个对象添加到 `TreeSet` 时，则该对象的类必须实现 `Comparable` 接口，否则程序将会抛出异常。

错误示例为：

```
class Err{ }  
public class TreeSetTest {  
    public static void main(String[] args) {
```

```

        TreeSet ts = new TreeSet();
        ts.add(new Err());
    }
}

```

在重写 `equals()` 方法时，应保证方法与 `compareTo(Object obj)` 方法有一致的结果，其规则是如果两个对象通过 `equals()` 方法返回比较 `true` 时，这两个对象通过 `compareTo(Object obj)` 方法比较应返回 `0`；

`TreeSet` 中判断两个对象是否相等的唯一标准是：两个对象通过 `compareTo(Object obj)` 方法比较是否返回 `0`，如果返回 `0`，`TreeSet` 就认为他们相等，否则不等。

当把一个对象加入 `TreeSet` 集合中时，`TreeSet` 调用该对象的 `compareTo(Object obj)` 方法与容器中的其他对象比较大小，然后根据红黑树结构找到它的存储位置。如果两个对象通过 `compareTo` 方法比较相等时，新对象将无法添加到 `TreeSet` 集合中。

1.自然排序

`TreeSet` 会调用元素的 `compareTo(Object obj)` 方法来比较元素之间的大小关系，然后将集合元素按升序排列，这种方式是自然排序。

```
package collection;
```

```
import java.util.TreeSet;
```

```

class Err implements Comparable{
    private int age;

    public Err(){
    public Err(int age){
        this.age = age;
    }
    public int getAge(){
        return this.age;
    }

    @Override
    public int compareTo(Object o) {
        return this.age - ((Err)o).getAge();
    }
}

```

```

    public String toString(){
        return this.age + "";
    }
}

public class TreeSetTest {

    public static void main(String[] args) {
        TreeSet ts = new TreeSet();
        ts.add(new Err(5));
        ts.add(new Err(5));
        ts.add(new Err(6));
        ts.add(new Err(7));
        System.out.println(ts);
    }

}

```

2.定制排序

如果需要实现定制排序，则可以通过 `Comparator` 接口的帮助。该接口里包含了一个 `int compare(T o1,T o2)`方法，该方法用于比较 `o1` 和 `o2` 的大小：如果返回正整数，则表明 `o1>o2`，返回 `0` 表明 `o1=o2`，返回负整数表明 `o1<o2`。

如果需要实现定制排序，则需要在创建 `TreeSet` 集合对象时，提供一个 `Comparator` 对象与该 `TreeSet` 集合关联，由该 `Comparator` 对象负责集合元素的排序逻辑。由于 `Comparator` 是一个函数式接口，因此可使用 `Lambda` 表达式来代替 `Comparator` 对象。

```

Class M{
    int age;
    public M(int age){
        this.age = age;
    }
    public String toString(){
        return this.age+"";
    }
}

public static void main(String[] args){
    TreeSet ts2 = new TreeSet<>(new Comparator() {
        @Override

```



```

    public int compare(Object o1, Object o2) {
        // TODO Auto-generated method stub
        return 0;
    };
});

```


或者:

```

TreeSet ts = new TreeSet((o1,o2)->
{
    M m1 = (M) o1;
    M m2 = (M) o2;
    return m1.age-m2.age;
});
}

```

1.2. HashSet

-  1) HashSet 是 Set 接口的典型实现，大多数时候使用 Set 集合时就是使用这个实现类。HashSet 按 Hash 算法存储集合中的元素，因此具有很好的存取和查找性能。
- 2) HashSet 不能保证元素的排列顺序，顺序可能与添加顺序不同，顺序也有可能发生变化
- 3) HashSet 不是同步的，如果多个线程同时访问一个 HashSet，假设有两个或者两个以上线程同时修改 HashSet 集合时，则必须通过代码来保证其同步。
- 4) 集合元素值可以是 null
- 5) 当向 HashSet 集合中存入一个元素时，HashSet 会调用该对象的 hashCode()方法来得到该对象的 hashCode 值，然后根据该 hashCode 值决定该对象在 HashSet 中的存放位置。如果有两个元素通过 equals()方法比较返回 true，但他们的 hashCode()方法返回值不相等，HashSet 将会把他们存储在不同位置，依然可以添加成功。也就是说，HashSet 集合判断两个元素相等的标准是两个对象通过 equals()方法比较相等并且两个对象的 hashCode()方法返回值也相等。
- 6) 如果 HashSet 中两个以上的元素具有相同的 hashCode 值，性能将会下降（存在 hashCode 值相同的原因是因为在比较是需要 equals 和 hashCode 方法都相同时才认为是同一个值，否则如果 equals 方法不同但是 hashCode 值相同时，认为不是同一个值，此时在这个 hashCode 值的位置上用链式结构保存多个对象，导致性能下降。）
- 7) 当程序把可变对象添加到 HashSet 中之后，尽量不要去修改该集合元素中参与计算 hashCode()和 equals()的实例变量，否则将会导致 HashSet 无法正确操作这些集合。

1.2.1. LinkedHashSet

1) HashSet 还有一个子类 LinkedHashSet，它也是根据元素的 **hashCode** 值来决定元素的存储位置，但它同时使用链表维护元素的次序，这样使得元素看起来是以插入的顺序保存的。也就是说，当遍历 LinkedHashSet 集合里的元素时，LinkedHashSet 将会按元素的添加顺序来访问集合里的元素。

2) LinkedHashSet 需要维护元素的插入顺序，因此性能略低于 HashSet 的性能，但在迭代访问 Set 里的全部元素时将有好的性能，因为它以链表来维护内部顺序。

例如：

```
LinkedHashSet books = new LinkedHashSet();
books.add("a");
books.add("b");
System.out.println(books);
books.remove("a");
books.add("a");
System.out.println(books);
```

输出的结果为：

[a,b]

[b,a]

输出 LinkedHashSet 集合的元素时，元素的顺序总是与添加顺序一致。

2. Queue

Queue 用于模拟队列这种数据结构，队列通常是指“先进先出”FIFO 的容器。

2.1. Queue 接口中定义的方法

2.1.1. void add(Object e)

将元素 e 添加到此队列尾部

2.1.2. boolean offer(Object e)

将指定元素加入此队列的尾部，当使用有容量限制的队列时，此方法通常比 add(Object e)方法更好。

2.1.3. Object element()

获得队列头部元素，但是不删除该元素

2.1.4. Object peek()

获取队列头部元素，但不删除该元素。如果队列为空，则返回 null。

2.1.5. Object poll()

获取队列头部元素，并删除该元素。如果队列为空，则返回 null。

2.1.6. Object remove()

获取队列头部的元素，并删除该元素。

2.2. PriorityQueue

PriorityQueue 是一个比较标准的队列实现类。它保存元素的顺序是根据元素的大小进行重新排序，而不是加入队列的顺序。

因此在调用 poll()和 peek()取出队列中的元素时，并不是取出最先进入队列的元素，而是取出队列中最小的元素。

例子：

```
PriorityQueue pq = new PriorityQueue();
pq.offer(6);
pq.offer(-3);
pq.offer(20);
pq.offer(18);
System.out.println(pq);
```

输出的结果为：

```
[-3, 6, 20, 18]
```

输出结果没有很好地按照大小进行排序，但这只是收到 PriorityQueue 的 toString()方法的返回值的影响。实际上程序多次调用 poll()方法时，就可以看到元素从小到大地顺序移出队列。

priorityqueue 不允许插入 null 元素，它还需要对队列元素进行排序：

- 1) 自然排序：集合中的元素必须实现了 Comparable 接口，而且应该是同一个类的多个实例。
- 2) 定制排序：创建 PriorityQueue 时传入一个 Comparator 对象，该对象负责对队列中的所有元素进行排序，采用定制排序时，不要求队列元素实现 Comparable 接口。

2.3. Deque

参见: [LinkedList](#)

Deque 代表一个“双端队列”，可以同时从两端来添加、删除元素，因此 Deque 的实现类既可以当成队列使用，也可以当成栈使用。

2.3.1. ArrayDeque



ArrayDeque 也是 List 的实现类，既实现了 List 接口，也实现了 Deque 接口，由于实现了 Deque 接口，因此可以当做栈来使用；而 ArrayDeque 底层也是基于数组的实现，因此性能也很好。

创建 Deque 时同样可指定一个 numElements 参数，该参数用于指定 Object[] 数组的长度；如果不指定 numElements 参数，Deque 底层数组的长度为 16。

当做栈

```
ArrayDeque stack = new ArrayDeque();
```

```

stack.push("a");
stack.push("b");
stack.push("c");
stack.push("d");
System.out.println(stack);
System.out.println(stack.peek());
System.out.println(stack.pop());
System.out.println(stack);

```

输出结果为：

```

[d, c, b, a]
d
d
[c, b, a]

```

！！！！注意！！！！

作为栈的方法有 push()、pop()和 peek()

作为队列的方法有 poll()、add()/offer()、peek()

当做队列

```

ArrayDeque stack = new ArrayDeque();
stack.offer("a");
stack.offer("b");
stack.offer("c");
stack.offer("d");
System.out.println(stack);
System.out.println(stack.peek());
System.out.println(stack.poll());
System.out.println(stack);

```

输出结果：

```

[a, b, c, d]
a
a
[b, c, d]

```

2.3.2. Deque 接口里定义的双端队列的方法

void addFirst(Object e)

将指定元素插入双端队列的开头

boolean offerFirst(Object e)

将指定元素插入该双端队列的开头

void addLast(Object e)

将指定元素插入该双端队列的末尾

boolean offerLast(Object e)

将指定元素插入该双端队列的末尾

Iterator decendingIterator()

返回该双端队列对应的迭代器，该迭代器将以逆向顺序来迭代队列中的元素。

Object getFirst()

获取但不删除双端队列的第一个元素

Object peekFirst()

获取但不删除双端队列的第一个元素，如果队列为空，则返回 `null`

Object pollFist()

获取并删除双端队列的第一个元素，如果队列为空，则返回 `null`

Object getLast()

获取但不删除双端队列的最后一个元素

Object peekLast()

获取但不删除双端队列的最后一个元素，如果队列为空，则返回 `null`

Object pollLast()

获取并删除双端队列的最后一个元素，如果队列为空，则返回 `null`

Object pop()

（栈方法）`pop` 出该双端队列所表示的栈的栈顶元素。相当于 `removeFirst()`

Object push()

（栈方法）将一个元素 `push` 进该双端队列所表示的栈的栈顶。相当于 `addFirst(e)`

Object removeFist()

获取并删除该双端队列的第一个元素

boolean removeFirstOccurrence(Object o)

删除该队列的第一次出现的元素 o

Object removeLast()

获取并删除该双端队列的最后一个元素

boolean removeLastOccurrence(Object e)

删除该队列的最后一次出现的元素 o

3. List

List 集合代表一个元素有序、可重复的集合，集合中每个元素都有其对应的顺序索引。List 集合允许使用重复元素，可以通过索引来访问指定位置的集合元素。

索引从 0 开始

对于例子：

```
class A{
    public boolean equals(Object obj){
        return true;
    }
}
```

```
List books = new ArrayList();
```

```
books.add("a");
```

```
books.add("b");
```

```
books.add("c");
```

```
books.remove(new A());
```

对于执行的 remove 操作，会调用 A 的 equals 方法一次与每个元素进行比较，如果该 equals()方法以某个集合元素作为参数是返回 true，list 将会删除该元素——A 类重写了 equals()方法，总是返回 true，因此每次从 list 中删除 a 对象时，总是删除 list 的第一个元素。

ArrayList 和 Vector 都是基于数组实现的 List 类，所以 ArrayList 和 Vector 类封装了一个动态的、允许再分配的 Object[]数组。ArrayList 或 Vector 对象使用 initialCapacity 参数来设置该数组的长度，当 ArrayList 或 Vector 中添加元素超出了该数组的长度时，他们的 initialCapacity 会自动增加。

固定长度的 List：

Arrays 工具类里提供了 asList(Object... a)方法，该方法可以把一个数组或指定个数的对象转换成一个 List 集合，这个 List 集合既不是 ArrayList 实现类的实例，也不是 Vector 实现类的实例，而是 Arrays 的内部类 ArrayList 的实例。Arrays.ArrayList 是一个固定长度的 List 集合，程序只能遍历访问该集合里的元素，不可增加、删除该集合里的元素。

```
List books = Arrays.asList("a","b","c","d");
```

关于使用 List 集合的建议：

- 1) 如果需要遍历 List 集合元素，ArrayList、Vector 使用随机访问方法（get）来遍历集合元素。对于 LinkedList 则应该使用 Iterator 迭代器来遍历
- 2) 如果需要经常执行插入、删除操作来改变包括大量数据的 List 集合的大小，可以考虑使用 LinkedList 集合。ArrayList、Vector 集合可能需要经常重新分配内部数组的大小，效果可能较差。
- 3) 如果有多个线程需要同时访问 List 集合中的元素，开发者可考虑使用 Collections 将集合包装成线程安全的集合。


3.1. LinkedList

 参见: [Deque](#)

LinkedList 是 List 接口的实现类，这意味着它是一个 List 集合，可以根据索引来随机访问集合中的元素。除此之外，LinkedList 还实现了 Deque 接口，可以被当成双端队列来使用，因此既可以当做栈，也可以当做队列使用。

LinkedList 与 ArrayList、ArrayDeque 的实现机制完全不同。前者内部以链表的形式来保存集合中的元素，因此随机访问元素时性能较差，但在插入、删除元素时性能较出色。后者内部以数组的形式来保存集合中的元素，使用随机访问的性能比使用 Iterator 迭代访问的性能要好，因为随机访问会被映射成对数组元素的访问。

3.2. ArrayList

 ArrayList 是线程不安全的，可以使用 Collections.synchronizedList(new ArrayList()) 来保证线程安全。

Vector 是线程安全的，无顺序保证该集合的同步性。

3.3. Vector

Vector 有一个子类 Stack，可以实现进栈(push(Object item))、出栈(pop())、获得第一个元素(peek())。但是它线程安全，性能较差，如果要用到“栈”这种数据结构，可以考虑使用 ArrayDeque。

3.3.1. Stack

3.4. List 集合增加根据索引来操作集合元素的方法

3.4.1. void add(int index, Object element)

将元素 element 插入到 List 集合的 index 处，index 之后的往后移动

3.4.2. boolean addAll(int index, Collection c)

将集合 c 所包含的所有元素都插入大 List 集合的 index 处。

3.4.3. Object get(int index)

返回集合 index 索引处的元素

3.4.4. int indexOf(Object o)

返回对象 o 在 List 集合中第一次出现的位置索引

3.4.5. int lastIndexOf(Object o)

返回对象 o 在 List 集合中最后一次出现的位置索引。

3.4.6. Object remove(int index)

删除并返回 index 索引处的元素。这个 remove 方法的参数是 Int，另外一个 remove 方法的参数是 Object，表示根据 obj 的 equals 方法找到指定的元素删除第一个。

3.4.7. Object set(int index, Object element)

将 index 索引处的元素替换成 element 对象，返回被替换的旧元素。

索引必须是 List 集合的有效索引。

3.4.8. List subList(int fromIndex,int toIndex)

返回从索引 fromIndex（包含）到索引 toIndex（不包含）处所有集合元素组成的子集。

3.4.9. void sort(Comparator c)

根据 Comparator 参数对 list 集合的元素排序。

例子：

....

```
books.sort((o1,o2)->((String)s1).length()-((String)s2).length());
```

3.4.10. void replaceAll(UnaryOperator operator)

根据 operator 指定的计算规则重新设置 List 集合的所有元素

例子：

```
books.replace(ele->((String)ele).length());
```

使用每个字符串的长度作为新的集合元素。

3.5. List 提供了一个额外 listIterator()方法

与 set 只提供一个 iterator()方法不同，List 还额外提供了一个 listIterator()方法，该方法返回一个 ListIterator 对象，ListIterator 接口继承了 Iterator 接口，还提供了专门操作 List 的方法。ListIterator 接口在 Iterator 接口基础上增加了一些方法。

ListIterator 增加了向前迭代的功能，还可通过 add()方法添加元素。

```
String[] strs = {"a","b","c","d"};
```

```
List books = new ArrayList();
```



```

for(String str:str){
    books.add(str);
}
ListIterator li = books.listIterator();
// 正向遍历数据
while(li.hasNext()){
    System.out.print(li.next() + " ");
}
System.out.println("=====");
// 逆向遍历数据
while(li.hasPrevious()){
    System.out.print(li.previous() + " ");
}
System.out.println("=====");

```

3.5.1. boolean hasPrevious()

返回该迭代器关联的集合是否还有上一个元素

3.5.2. Object previous()

返回迭代器的上一个元素

3.5.3. void add(Object o)

在指定位置插入一个元素

4. 集合遍历

4.1. Lambda 表达式遍历集合

```

Collection books = new HashSet();
books.add("a");
books.add("b");
books.add("c");
books.forEach(obj->System.out.println(obj));

```

4.2. Java8 增强的 Iterator 遍历集合元素

```

// 创建集合
....
//遍历
Iterator it = books.iterator();
while(it.hasNext()){
    String book = (String)it.next();
}

```

```
if(book.equals("a")){  
    it.remove();// 从集合中删除元素
```

在使用 Iterator 迭代过程中，不可修改集合元素：如 books.remove(book)就会报错。

```
}  
}
```

4.3. 使用 Lambda 表达式遍历 Iterator

Java8 为 Iterator 新增了一个 forEachRemaining(Consumer action)方法。

```
Iterator it = books.iterator();  
it.forEachRemaining(obj->System.out.println(obj));
```

4.4. 使用 foreach 遍历集合元素

```
for(Object obj:books){  
    //在迭代过程中不能修改集合元素，会引发 ConcurrentModificationException  
    String book = (String)obj;  
}
```

5. 接口中的方法

这些方法来自于 Java API 文档。在现实生活中容器的功能无非是，添加对象、删除对象、清空容器、判断容器是否为空等操作。

5.1. boolean add(Object o)

如果集合对象被改变了，则返回 true

5.2. boolean addAll(Collection c)

将集合 c 中的所有元素添加到指定集合里。如果集合对象被添加操作改变了，则返回 true。

5.3. void clear()

清除集合中的所有元素，将集合长度变为 0

5.4. boolean contains(Object o)

返回集合是否包含指定元素

5.5. boolean containsAll(Collection c)

返回集合里是否包含集合 c 里的所有元素。

5.6. boolean empty()

返回集合是否为空。当长度为 0 时返回 true，否则返回 false。

5.7.Iterator iterator()

返回一个迭代器对象 Iterator，用于遍历集合里的元素

5.8.boolean remove(Object c)

删除集合中的指定元素 o，当集合中包含一个或多个元素 o 时，该方法只删除第一个符合条件的元素，该方法返回 true。

5.9.boolean removeAll(Collection o)

从集合中删除 c 中包含的所有元素（相当于调用该方法的集合减集合 c）。如果删除了一个或一个以上的元素，则方法返回 true。

5.10. boolean retainAll(Collection c)

从集合中删除集合 c 里不包含的元素，相当于调用该方法的集合变成该集合和集合 c 的交集。如果该操作改变了调用该方法的集合，则该方法返回 true。

5.11. int size()

返回元素个数

5.12. Object[] toArray()

该方法把集合转换成一个数组，所有的集合元素变成对应的数组元素。

6.

// 第二次添加同一个对象，输出 true，表明添加成功
System.out.println(set.add(z1)); //①
// 下面输出 set 集合，将看到有两个元素
System.out.println(set);
// 修改 set 集合的第一个元素的 age 变量
((Z)(set.first())).age = 9;
// 输出 set 集合的最后一个元素的 age 变量，将看到也变成了 9
System.out.println(((Z)(set.last())).age);

程序中①代码行把同一个对象再次添加到 TreeSet 集合中，因为 z1 对象的 compareTo(Object obj) 方法总是返回 1，虽然它的 equals() 方法总是返回 true，但 TreeSet 会认为 z1 对象和它自己也不相等，因此 TreeSet 可以添加两个 z1 对象。图 8.5 显示了 TreeSet 及 Z 对象在内存中的存储示意图。

从图 8.5 可以看到 TreeSet 对象保存的两个元素（集合里的元素总是引用，但习惯上把被引用的对象称为集合元素），实际上是同一个元素。所以当修改 TreeSet 集合里第一个元素的 age 变量后，该 TreeSet 集合里最后一个元素的 age 变量也随之改变了。

由此应该注意一个问题：当需要把一个对象放入 TreeSet 中，重写该对象对应对应的 equals() 方法时，应保证该方法与 compareTo(Object obj) 方法有一致的结果，其规则是：如果两个对象通过 equals() 方法比较返回 true 时，这两个对象通过 compareTo(Object obj) 方法比较应返回 0。

如果两个对象通过 compareTo(Object obj) 方法比较返回 0 时，但它们通过 equals() 方法比较返回 false，将很麻烦，因为两个对象通过 compareTo(Object obj) 方法比较相等，TreeSet 不会让第二个元素添加进去，这就会与 Set 集合的规则产生冲突。

如果白书 6.5 中添加一个可变对象后，并且后面程序修改了该可变对象的实例变量，这将导致它

图 8.5 TreeSet 及 Z 对象在内存中的存储示意图