一、 常见集合面试题

1. 集合类型可以归纳为三种 List、Map、Set

Map 接口和 Collection 接口是所有集合框架的父接口

Collection 接口的子接口包括：Set 接口和 List 接口

Map 接口的实现类主要有：HashMap、TreeMap、Hashtable、LinkedHashMap、ConcurrentHashMap 以及 Properties 等

Set 接口的实现类主要有：HashSet、TreeSet、LinkedHashSet 等

List 接口的实现类主要有：ArrayList、LinkedList、Stack 以及 Vector 等

结构特点：

List 和 Set 是存储单列数据的集合，Map 是存储键和值这样的双列数据的集合；

List 中存储的数据是有顺序，并且允许重复；

Map 中存储的数据是没有顺序的，其键是不能重复的，它的值是可以有重复的，

Set 中存储的数据是无序的，且不允许有重复，但元素在集合中的位置由元素的 hashcode 决定，位置是固定的（Set 集合根据 hashcode 来进行数据的存储，所以位置是固定的，但是位置不是用户可以控制的，所以对于用户来说 set 中的元素还是无序的）；

实现类：

List 接口有三个实现类

LinkedList：基于链表实现，链表内存是散乱的，每一个元素存储本身内存地址的同时还存储下一个元素的地址。链表增删快，查找慢；

ArrayList：基于数组实现，非线程安全的，效率高，便于索引，但不便于插入删除；

Vector：基于数组实现，线程安全的，效率低。

Map 接口有三个实现类

HashMap：基于 hash 表的 Map 接口实现，非线程安全，高效，支持 null 值和 null 键；

HashTable：线程安全，低效，不支持 null 值和 null 键；

HashMap 和 HashTable 的 key 值均不能重复，若添加 key 相同的键值对，后面的 value 会自动覆盖前面的 value，但不会报错。

LinkedHashMap：是 HashMap 的一个子类，保存了记录的插入顺序；

SortMap 接口：TreeMap，能够把它保存的记录根据键排序，默认是键值的升序排序。

Set 接口有两个实现类

HashSet：底层是由 HashMap 实现，不允许集合中有重复的值，使用该方式时需

要重写 equals()和 hashCode()方法；

LinkedHashSet：继承于 HashSet，同时又基于 LinkedHashMap 来进行实现，底层使用的是 LinkedHashMap

区别：

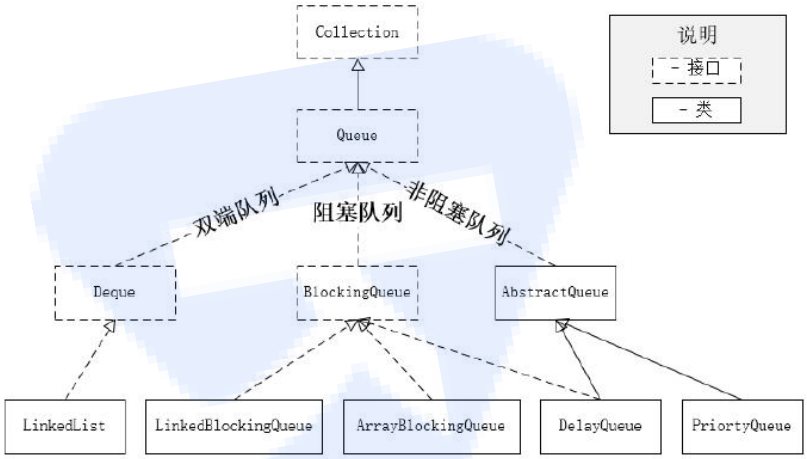
List 集合中对象按照索引位置排序，可以有重复对象，允许按照对象在集合中的索引位置检索对象，例如通过 list.get(i)方法来获取集合中的元素；Map 中的每一个元素包含一个键和一个值，成对出现，键对象不可以重复，值对象可以重复；Set 集合中的对象不按照特定的方式排序，并且没有重复对象，但它的实现类能对集合中的对象按照特定的方式排序，例如 TreeSet 类，可以按照默认顺序，也可以通过实现 Java.util.Comparator<Type>接口来自定义排序方式。

collection/DuplicateAndOrderTest.java

```
package com.open.collection;

import java.util.LinkedList;
import java.util.TreeSet;

public class DuplicateAndOrderTest {
    public static void main(String[] args) {
        LinkedList linkedList = new LinkedList();
        linkedList.add("111");
        linkedList.add("333");
        linkedList.add("222");
        linkedList.add("444");
        linkedList.add("555");
        linkedList.add("111");
        linkedList.get(1);
        System.out.println(linkedList);
        TreeSet treeSet = new TreeSet();
        treeSet.add("111");
        treeSet.add("333");
        treeSet.add("222");
        treeSet.add("444");
        treeSet.add("555");
        treeSet.add("111");

        System.out.println(treeSet);
    }

}
```

2. **队列**

队列（Queue）：与栈相对的一种数据结构， 集合（Collection）的一个子类。队列允许在一端进行插入操作，而在另一端进行删除操作的线性表，栈的特点是后进先出，而队列的特点是先进先出。队列的用处很大，比如实现消息队列。



双端队列：双端队列（Deque）是 Queue 的子类也是 Queue 的补充类，头部和尾部都支持元素插入和获取。

阻塞队列：阻塞队列指的是在元素操作时（添加或删除），如果没有成功，会阻塞等待执行。例如，当添加元素时，如果队列元素已满，队列会阻塞等待直到有空位时再插入。

BlockingQueue 提供了线程安全的队列访问方式，当向队列中插入数据时，如果队列已满，线程则会阻塞等待队列中元素被取出后再插入；当从队列中取数据时，如果队列为空，则线程会阻塞等待队列中有新元素再获取。

非阻塞队列：非阻塞队列和阻塞队列相反，会直接返回操作的结果，而非阻塞等待。双端队列也属于非阻塞队列

ConcurrentLinkedQueue 是一个基于链接节点的无界线程安全队列，它采用先进先出的规则对节点进行排序，当我们添加一个元素的时候，它会添加到队列的尾部；当我们获取一个元素时，它会返回队列头部的元素。

它的入队和出队操作均利用 CAS（Compare And Set）更新，这样允许多个线程并发执行，并且不会因为加锁而阻塞线程，使得并发性能更好。

方法说明：

add(E)：添加元素到队列尾部，成功返回 true，队列超出时抛出异常；

offer(E)：添加元素到队列尾部，成功返回 true，队列超出时返回 false；

remove()：删除元素，成功返回 true，失败返回 false；

poll()：获取并移除此队列的第一个元素，若队列为空，则返回 null；

peek()：获取但不移除此队列的第一个元素，若队列为空，则返回 null；

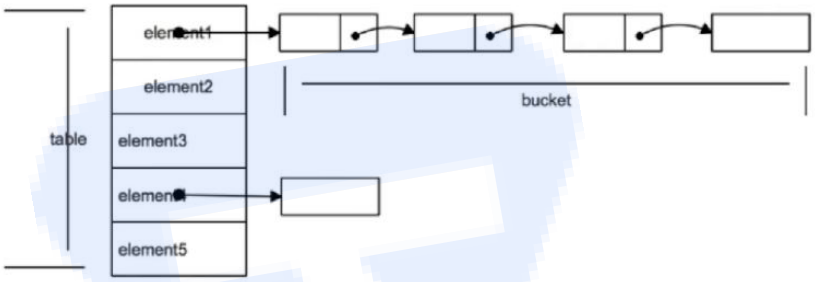
element()：获取但不移除此队列的第一个元素，若队列为空，则抛异常。

3. HashMap

HashMap(Java8 以前)：数组+链表

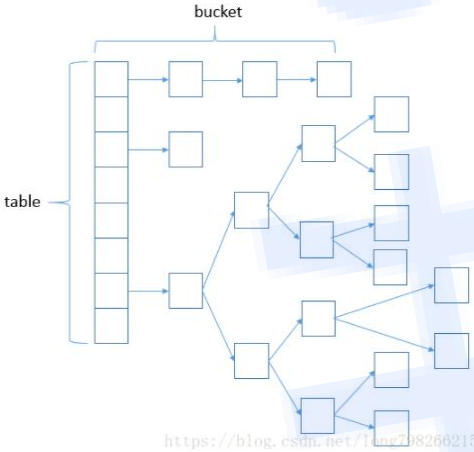
HashMap 在没有为数组赋予长度时，默认是 16。每个元素存储的是链表的头结点，通过 hash(key.hashCode())%len 取模操作获取要添加的元素在数组中的位置。

这里有一个极端情况，添加到 hash 表里的元素的 key 通过取模操作后总是得到同一个值，即所有元素都分配到同一个桶(bucket)，这样在查询链表时要从头部逐个遍历，hashMap 的性能会从 O(1)恶化到 O(n)

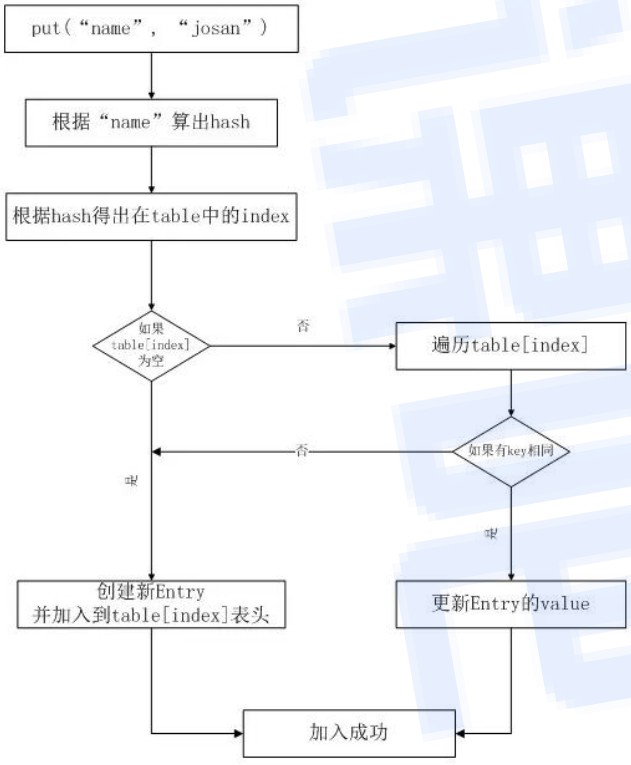


HashMap(Java8 以后)

HashMap 由 数组+链表+红黑树实现，桶中元素可能为链表，也可能为红黑树。为了提高综合（查询、添加、修改）效率，当桶中元素数量超过 TREEIFY_THRESHOLD（默认为 8）时，链表存储改为红黑树存储，当桶中元素数量小于 UNTREEIFY_THRESHOLD（默认为 6）时，红黑树存储改为链表存储



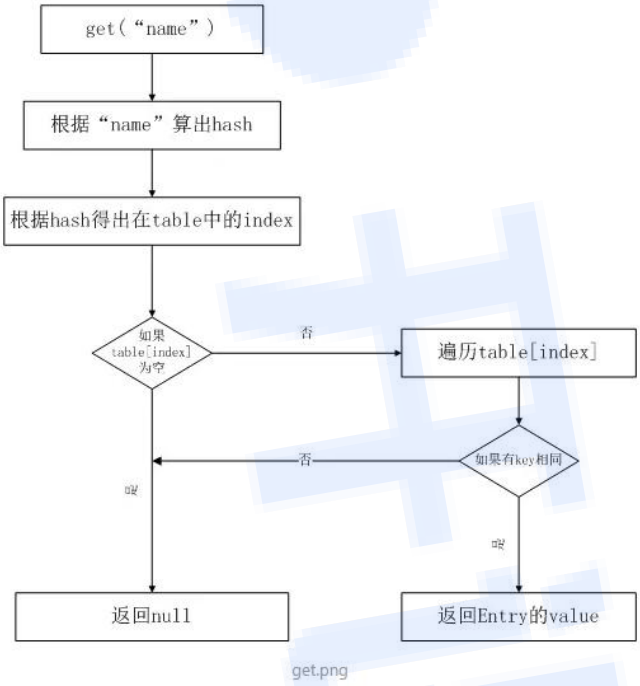
put 方法存值过程如下图所示：



在 JDK1.8 中，HashMap 存储数据的过程可以分为以下几步：

1) 对 key 的 hashCode () 进行 hash 后计算数组获得下标 index;
2) 如果当前数组为 null，进行容量的初始化，初始容量为 16；
3) 如果 hash 计算后没有碰撞，直接放到对应数组下标里；
4) 如果 hash 计算后发生碰撞且节点已存在，则替换掉原来的对象；
5) 如果 hash 计算后发生碰撞且节点已经是树结构，则挂载到树上。
6) 如果 hash 计算后发生碰撞且节点是链表结构，则添加到链表尾，并判断链表是否需要转换成树结构（默认大于 8 的情况会转换成树结构）；
7) 完成 put 后，是否需要 resize () 操作（数据量超过 threshold，threshold 为初始容量和负载因子之积，默认为 12）。

get 方法的取值过程如下图所示：



get.png

总结

JDK 1.8 以前 HashMap 的实现是 数组+链表，即使哈希函数取得再好，也很难达到元素百分百均匀分布。当 HashMap 中有大量的元素都存放到同一个桶中时，这个桶下有一条长长的链表，这个时候 HashMap 就相当于一个单链表，假如单链表有 n 个元素，遍历的时间复杂度就是 O(n)，完全失去了它的优势。针对这种情况，JDK 1.8 中引入了红黑树（查找时间复杂度为 O(logn)）来优化这个问题

为什么线程不安全？

多线程 PUT 操作时可能会覆盖刚 PUT 进去的值；扩容操作会让链表形成环形数据结构，形成死循环。容量的默认大小是 16，负载因子是 0.75，当 HashMap 的 size > 16*0.75 时就会发生扩容(容量和负载因子都可以自由调整)。

为什么容量是 2 的倍数？

在根据 hashcode 查找数组中元素时，取模性能远远低于与性能，且和 2^n-1 进行与操作能保证各种不同的 hashcode 对应的元素也能均匀分布在数组中

HashMap 是线程不安全的，如果有线程安全需求，推荐使用 ConcurrentHashMap。

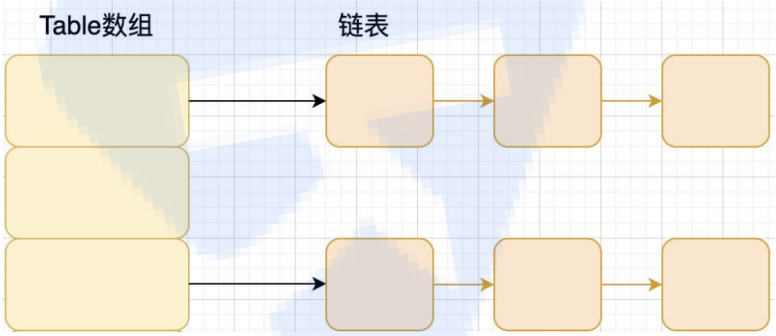
什么是 HashMap 的加载因子？加载因子为什么是 0.75？

判断什么时候进行扩容的，假如加载因子是 0.5，HashMap 的初始化容量是 16，

那么当 HashMap 中有 16*0.5=8 个元素时，HashMap 就会进行扩容。

Hash 冲突及解决？

所谓 “拉链法” 就是：将链表和数组相结合。也就是说创建一个链表数组，数组中每一格就是一个链表。若遇到哈希冲突，则将冲突的值加到链表中即可。



4. **HashMap 遍历**

迭代器（Iterator）方式遍历；

For Each 方式遍历；

Lambda 表达式遍历（JDK 1.8+）；

Streams API 遍历（JDK 1.8+）

5. **迭代器 Iterator 作用**

Iterator 主要是用来遍历集合用的，它的特点是更加安全，因为它可以确保，在当前遍历的集合元素被更改的时候，就会抛出 ConcurrentModificationException 异常。

collection/ IteratorTest.java

```
package com.open.collection;

import java.util.*;

public class IteratorTest {
    public static void main(String[] args) {
        List<Integer> arrayList = new ArrayList<>();
        for (int i = 0; i < 20; i++) {
            arrayList.add(Integer.valueOf(i));
        }

        Iterator<Integer> iterator = arrayList.iterator();
        while (iterator.hasNext()) {
            Integer integer = iterator.next();
            if (integer.intValue() == 5) {
                arrayList.remove(integer);
            }
        }

//        Map<Integer,String> map=new HashMap();
//        map.put(1,"java");
//        map.put(2,"c++");
//        map.put(3,"python");
```

```
//        Iterator<Map.Entry<Integer,String>>
iterator=map.entrySet().iterator();
//        while(((Iterator) iterator).hasNext()){
//            Map.Entry<Integer,String> entry=iterator.next();
//            map.remove(entry);
//            System.out.println(entry.getKey()+":"+entry.getValue());
//        }

    }
}
```

我们不能在遍历中使用集合 map.remove() 来删除数据，这是非安全的操作方式，但我们可以使用迭代器的 iterator.remove() 的方法来删除数据，这是安全的删除集合的方式。

6. **如何让 HashMap 变成线程安全(ConcurrentHashMap)**

可以使用 Collections 类的 synchronizedMap()方法对 HasnMap 进行包装。不建议使用，效率低。可以使用 ConcurrentHashMap

ConcurrentHashMap: 可以看作是线程安全的 HashMap

CopyOnWriteArrayList:可以看作是线程安全的 ArrayList，在读多写少的场合性能非常好，远远好于 Vector.

ConcurrentLinkedQueue:高效的并发队列，使用链表实现。可以看做一个线程安全的 LinkedList，这是一个非阻塞队列。

SafeHashMapDemo.java

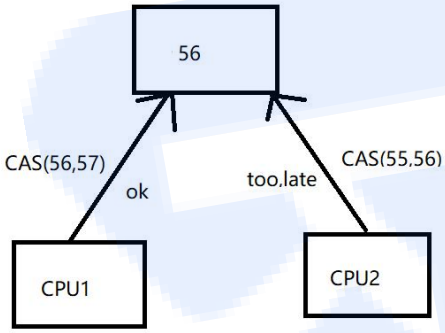
```
package com.open.collection;

import java.util.Collections;
import java.util.HashMap;
import java.util.Map;

public class SafeHashMapDemo {
    public static void main(String[] args) {
        Map hashMap = new HashMap();
        Map safeHashMap = Collections.synchronizedMap(hashMap);
        safeHashMap.put("aa","1");
        safeHashMap.put("bb","2");
        System.out.println(safeHashMap.get("bb"));
    }

}
```

7. **什么是 CAS**

compare and swap,主要用在并发场景中，是一种思想，是一种实现线程安全的算法。在并发编程中实现那些不能被打断的交换操作，从而避免在多线程下执行顺序不确定导致错误。

思路：我认为 V 的值应该是 A,如果是则改成 B,如果不是 A,说明有人修改过，那我就不修改，避免多人同时修改导致出错。

CAS 有三个操作数:内存值 V、预期值 A、要修改的值 B,当且仅当预期值 A 和内存值 V 相同时，才将内存值修改为 B,否则什么都不做。最后返回现在的 V 值。



ABA 问题描述

小明在提款机，提取了 50 元，因为提款机问题，有两个线程，同时把余额从 100 变为 50

线程 1（提款机）：获取当前值 100，期望更新为 50，

线程 2（提款机）：获取当前值 100，期望更新为 50，

线程 1 成功执行，线程 2 某种原因 block 了，这时，某人给小明汇款 50

线程 3（默认）：获取当前值 50，期望更新为 100，

这时候线程 3 成功执行，余额变为 100，

线程 2 从 Block 中恢复，获取到的也是 100，compare 之后，继续更新余额为 50

此时可以看到，实际余额应该为 100（100-50+50），但是实际上变为了 50（100-50+50-50）

解决方法：

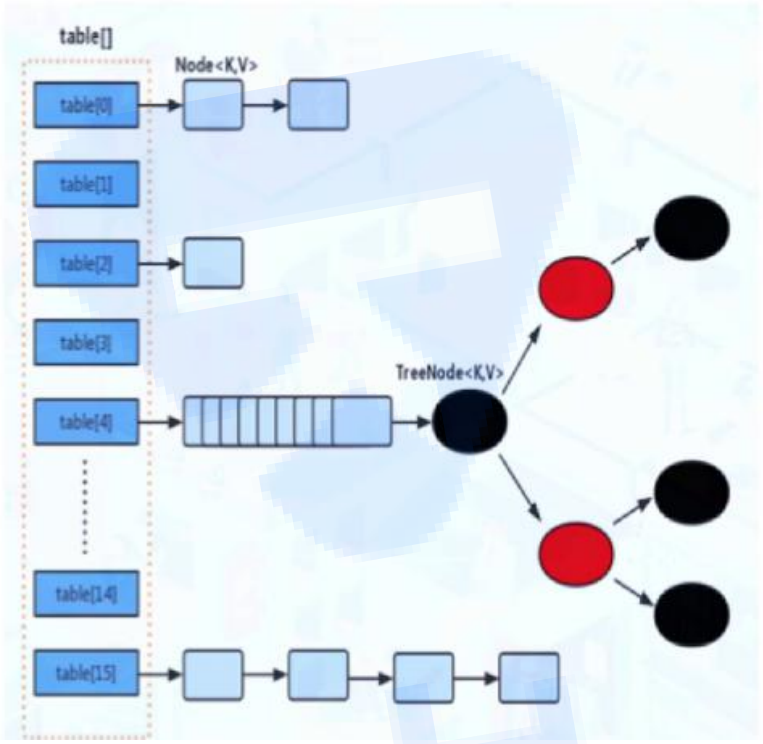
在变量前面加上版本号，每次变量更新的时候变量的版本号都+1

Java 提供了一个 AtomicStampedReference 原子引用变量，通过添加版本号来解决 ABA 的问题

8. **ConcurrentHashMap**

因为 hashtable 在多线程环境下，多个线程要竞争同一把对象锁，效率低。所以在 jdk5 之后，ConcurrentHashMap 出现。

原理：通过锁细粒度化，将整锁拆解成多个锁进行优化。(分段锁+CAS)



synchronized 只锁定当前链表或红黑树的首结点。这样只要 hash 不冲突，就不会产生并发。

ConcurrentHashMap:put 方法逻辑

1.判断 Node[]数组是否初始化，没有则进行初始化操作。

2.通过 hash 定位数组的索引坐标，是否有 Node 节点，如果没有则使用 CAS 进行添加（链表的头节点），添加失败则进入下次循环。

3.检查到内部有其它线程正在扩容，则帮助它一起扩容。

4.如果 f!=null，则使用 synchronized 锁住 f 元素（链表或红黑二叉树的头元素）

如果是 Node(链表结构)则执行链表的添加操作。

如果是 TreeNode(树型结构)则执行树添加操作。

5.判断链表长度是否已经到达默认临界值 8，当超过就将链表转为树结构。

9. ArrayList 和 LinkedList 有哪些区别？

相同点：ArrayList 和 LinkedList 都是 List 接口的实现类，因此都具有 List 的特点，即存取有序，可重复；而且都不是线程安全的。

不同点：ArrayList 基于数组实现，LinkedList 基于双向链表实现。

ArrayList 基于数组存储数据，因此查询元素时可以直接按照数据下标进行索引，而插入元素时，通常涉及到数据元素的复制和移动，所以查询数据快而插入数据慢；

LinkedList 基于双向链表存储数据，因此查询元素时需要前向或后向遍历，而插入数据时只需要修改本元素的前后项即可，所以查询数据慢而插入数据快。

所以，ArrayList 适合查询多（读多）场景，LinkedList 适合插入多（写多）的场景。

10. **比较 HashSet、LinkedHashSet 和 TreeSet 三者的异同**

HashSet 是 Set 接口的主要实现类 ，HashSet 的底层是 HashMap，线程不安全的，可以存储 null 值；

LinkedHashSet 是 HashSet 的子类，能够按照添加的顺序遍历；

TreeSet 底层使用红黑树，能够按照添加元素的顺序进行遍历，排序的方式有自然排序和定制排序。

11. Collections 和 Collection 的区别

Collection 是个 java.util 下的接口，它是各种集合结构的父接口，定义了集合对象的基本操作方法。

Collections 是个 java.util 下的工具类，它包含有各种有关集合操作的静态方法，主要是针对集合类的一个帮助类或者叫包装类，它提供一系列对各种集合的搜索，排序，线程安全化等操作方法。

12. Comparable 和 Comparator 接口是干什么的？列出它们的区别。

comparable 接口实际上是出自 java.lang 包 它有一个 compareTo(Object obj)方法用来排序 comparator 接口实际上是出自 java.util 包它有一个 compare(Object obj1, Object obj2)方法用来排序

一般我们需要对一个集合使用自定义排序时，我们就要重写 compareTo()方法或 compare()方法。

Customer.java

```
package com.open.collection;

import java.util.Set;
import java.util.TreeSet;

public class Customer implements Comparable{
	private String name;
	private int age;

	public Customer(String name, int age) {
	    this.age = age;
	    this.name = name;
	}

	public int getAge() {
	    return age;
	}

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

	public String getName() {
	    return name;
	}

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

	@Override
```

```
        public boolean equals(Object obj) {
            if (this == obj)
                return true;
            if (!(obj instanceof Customer))
                return false;
            final Customer other = (Customer) obj;

            if (this.name.equals(other.getName()) && this.age ==
other.getAge())
                return true;
            else
                return false;
        }

        @Override
        public int compareTo(Object o) {
            Customer other = (Customer) o;
            // 先按照 name 属性排序
            if (this.name.compareTo(other.getName()) > 0)
                return 1;
            if (this.name.compareTo(other.getName()) < 0)
                return -1;

            // 在按照 age 属性排序
            if (this.age > other.getAge())
                return 1;
            if (this.age < other.getAge())
                return -1;
            return 0;

        }
        @Override
        public int hashCode() {
            int result;
            result = (name == null ? 0 : name.hashCode());
            result = 29 * result + age;
            return result;
        }

        public static void main(String[] args) {
            Set<Customer> set = new TreeSet<Customer>();
            Customer customer1 = new Customer("Tom", 16);
            Customer customer2 = new Customer("jerry", 15);
            set.add(customer1);
```

```
            set.add(customer2);
            for (Customer c : set) {
                System.out.println(c.name + " " + c.age);
            }
        }
    }
```

CustomerComparator.java

```
package com.open.collection;

import java.util.Comparator;
import java.util.Iterator;
import java.util.Set;
import java.util.TreeSet;

public class CustomerComparator implements Comparator<Customer> {
    @Override
    public int compare(Customer c1, Customer c2) {
        if (c1.getName().compareTo(c2.getName()) > 0) return -1;
        if (c1.getName().compareTo(c2.getName()) < 0) return 1;
        return 0;
    }

    public static void main(String args[]) {
        Set<Customer> set = new TreeSet<Customer>(new
CustomerComparator());

        Customer customer1 = new Customer("a", 5);
        Customer customer2 = new Customer("b", 9);
        Customer customer3 = new Customer("c", 2);
        set.add(customer1);
        set.add(customer2);
        set.add(customer3);
        Iterator<Customer> it = set.iterator();
        while (it.hasNext()) {
            Customer customer = it.next();
            System.out.println(customer.getName() + " " +
customer.getAge());
        }
    }
}
```

13. **集合中那些类是线程安全类，哪些是不安全的，哪些是支持排序的类**

线程安全类：Vector、Hashtable、Stack。

线程不安全的类：ArrayList、Linkedlist、HashSet、TreeSet、HashMap、TreeMap

支持排序的类有 HashSet、LinkedHashSet、TreeSet 等（Set 接口下的实现都支持排

序）

此题主要考查集合框架的知识。在集合框架中 Collection 接口为集合的根类型，提供集合操作的常用 API 方法，该接口下派生出两个子接口，一个是不支持排序的 List 接口，一个是有自身排序的 Set 接口，所以回答排序与不排序分别从两接口的实现中在作答。线程安全上来说，Vector 类比同属于 List 接口的 ArrayList 要早，是一个线程安全的类，在 JDK1.2 以后才推出一个异步的 ArrayList 类，比 Vector 类效率高。同理 Stack 继承自 Vector 也线程安全的类，另外在在 Map 接口的实现在 Hashtable 也是个线程安全的类。

14. 数组和集合 List 之间的转换

数组和集合 Lis 的转换在我们的日常开发中是很常见的一种操作，主要通过 Arrays.asList 以及 List.toArray 方法来实现。

converTest.java

```
package com.open.collection;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class converTest {
    public static void main(String[] args) {
        //list 集合转换成数组
        ArrayList<String> list=new ArrayList<>();
        list.add("a");
        list.add("b");
        list.add("c");
        Object[] arr=list.toArray();
        for(int i=0;i<arr.length;i++){
            System.out.println(arr[i]);
        }

        //数组转换为 list 集合
        String[] arr2={"a","b","c"};
        List<String> asList= Arrays.asList(arr2);
        for(int i=0;i<asList.size();i++){
            System.out.println(asList.get(i));
        }
    }
}
```

数组转为集合 List：

通过 Arrays.asList 方法搞定，转换之后不可以使用 add/remove 等修改集合的相关方法，因为该方法返回的其实是一个 Arrays 的内部私有的一个类 ArrayList，该类继承于 Abstractlist，并没有实现这些操作方法，调用将会直接抛出 UnsupportOperationException 异常。这种转换体现的是一种适配器模式，只是转换接口，本质上还是一个数组。

集合转换数组：

List.toArray 方法实现了集合转换成数组，这里最好传入一个类型一样的数组，大小就是 list.size()。因为如果入参分配的数组空间不够大时，toArray 方法内部将重新分配内存空间，并返回新数组地址；如果数组元素个数大于实际所需，下标为 list.size()及其之后的数组元素将被置为 null，其它数组元素保持原值。所以，建议该方法入参数组的大小与集合元素个数保持一致。

若是直接使用 toArray 无参方法，此方法返回值只能是 Object[ ]类，若强转其它类型数组将出现 ClassCastException 错误。

15. List 里面如何剔除相同的对象

removeEqObject

```
package com.open.collection;

import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;

public class removeEqObject {
    public static void main(String[] args) {
        List<String> list = new ArrayList<String>();
        list.add("8");
        list.add("8");
        list.add("9");
        list.add("9");
        list.add("0");
        System.out.println(list);
        // 方法:将 List 中数据取出来来存到 Set 中
        HashSet<String> set = new HashSet<String>();
        for (int i = 0; i < list.size(); i++) {
            set.add(list.get(i));
        }
        System.out.println(set);
    }

}
```