# 是什么

<https://www.bilibili.com/video/BV1YA411T76k?p=1>

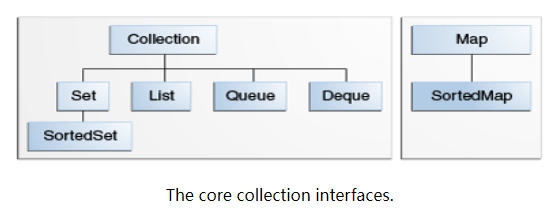
电子游戏的截图

描述已自动生成少了Queue

图形用户界面, 应用程序

描述已自动生成

<https://docs.oracle.com/javase/tutorial/collections/interfaces/index.html>



# 具体类/接口

## （1） Map接口

### 【1】 是什么

### 【2】 实现类

#### HashMap类

##### 1） 本身

见表格, 日程表

中度可信度描述已自动生成

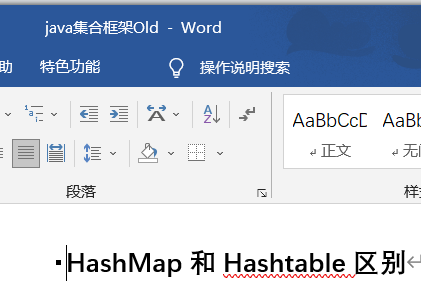
##### 2） 子类

###### 1》 LinkedHashMap类

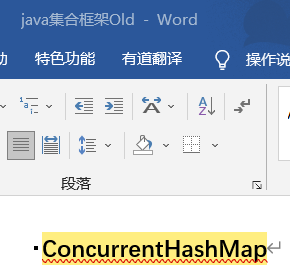
见图形用户界面

中度可信度描述已自动生成

#### Hashtable类

见

#### ConcurrentHashMap类

见

#### TreeMap类

## （2） Queue接口

### 【1】 是什么

### 【2】 子接口

#### 《1》 Deque

<https://www.liaoxuefeng.com/wiki/1252599548343744/1265122668445536>

#### 《2》 BlockingQueue

见

### 【3】 实现类

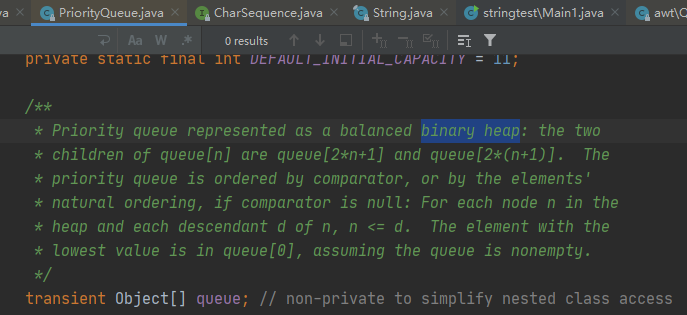
#### 《1》 PriorityQueue类

参1：<https://juejin.cn/post/6859724629616164871>





参2：jdk源码





## （3） Set接口

### 【1】 是什么

### 【2】 实现类

#### 《1》 HashSet类

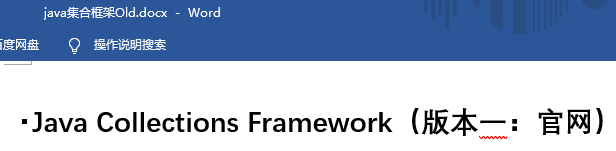
##### 是什么

##### 子类

###### 1》 LinkedHashSet

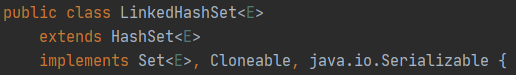
A） 是什么

参考：

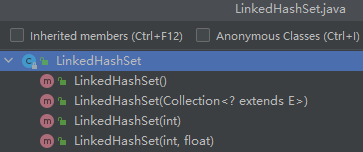
1】 搜“LinkedHashSet”

2】

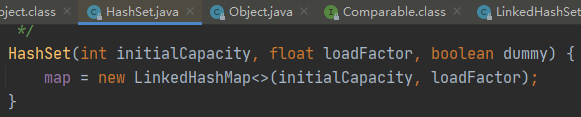
<https://zhuanlan.zhihu.com/p/348610464#:~:text=LinkedHa,%E5%8A%A0%E9%A1%BA%E5%BA%8F%E9%81%8D%E5%8E%86%E8%BE%93%E5%87%BA%E3%80%82>

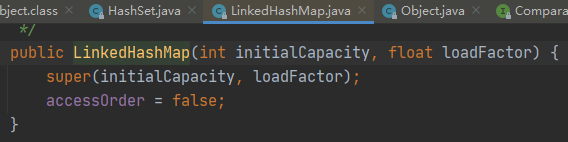


它底层基于LinkedHashMap，通过构造器实现：



4个构造器均调了





即使用LinkedHashMap的插入/添加顺序：insertion-order来实现自己的有序。

B） 使用

<https://www.cainiaojc.com/java/java-linkedhashset.html>

插入+遍历：

|  |
| --- |
| LinkedHashSet<Integer> evenNumber = new LinkedHashSet<>();  // 使用add()方法 evenNumber.add(2); evenNumber.add(4); evenNumber.add(6); System.*out*.println("LinkedHashSet的toString()输出: " + evenNumber);  //调用iterator()方法访问元素 System.*out*.print("LinkedHashSet的Iterator输出: "); Iterator<Integer> iterate = evenNumber.iterator(); while(iterate.hasNext()) {  System.*out*.print(iterate.next());  System.*out*.print(", "); } |

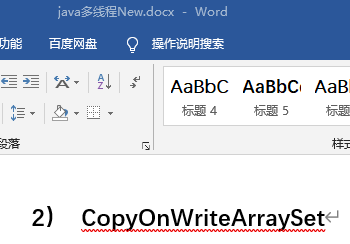


#### 《2》 TreeSet类

见图形用户界面, 应用程序

描述已自动生成

#### 《3》 CopyOnWriteArraySet类

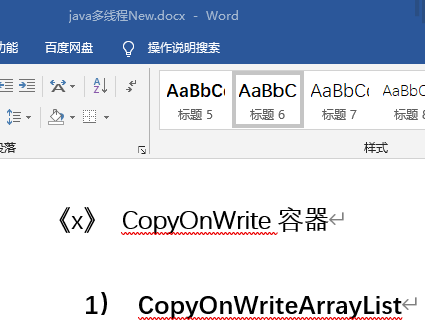
见

## （4） List接口

### 【1】 是什么

### 【2】 实现类

#### 《1》 CopyOnWriteArrayList类

见

#### 《2》 Vector类

#### 《3》 ArrayList类

##### 相关操作

###### 缩容

<https://juejin.cn/post/6844904195812753415>

<https://www.zhihu.com/question/396893900?utm_id=0>

ArrayList没有在删除元素【如remove(),clear()】时进行自动缩容，但能通过trimToSize()进行手动缩容。

# 常用操作

## （1） 遍历

<https://www.cnblogs.com/leskang/p/6031282.html>

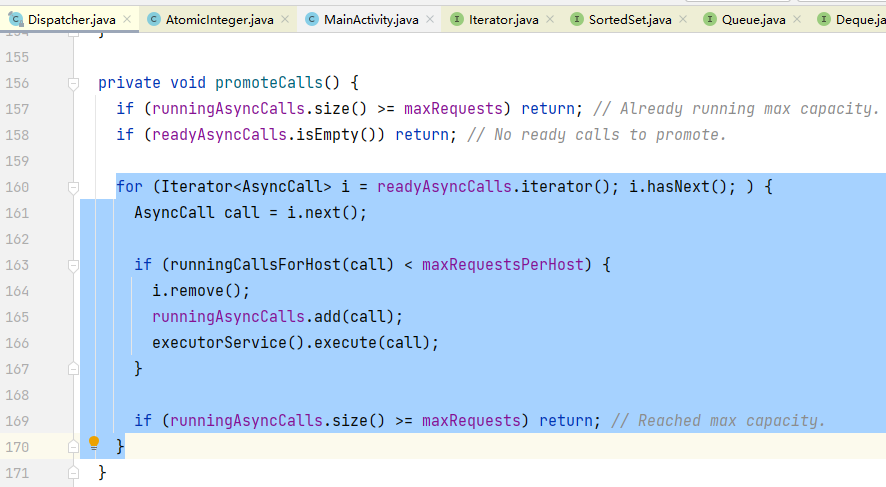
<https://blog.csdn.net/bluuusea/article/details/79949592>

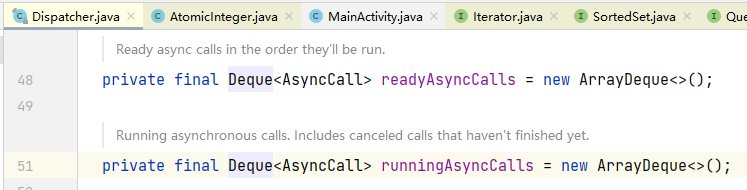
<https://www.runoob.com/java/method-for.html>

<https://cloud.tencent.com/developer/news/700876>

### 【x】 实例

#### 例1：okhttp的分发器





# Comparator和Comparable

<https://blog.csdn.net/tolcf/article/details/52229068>

<https://www.cnblogs.com/xujian2014/p/5215082.html>

<https://www.geeksforgeeks.org/comparable-vs-comparator-in-java/>

<https://www.huaweicloud.com/articles/e892b3fc321f6f17f12376b43e71a2d1.html>

<https://www.cnblogs.com/skywang12345/p/3324788.html>

<http://www.itwanger.com/java/2020/01/04/java-comparable-comparator.html>

## （1） 是什么

参考：

1】 https://docs.oracle.com/javase/7/docs/api/java/lang/Comparable.html

2】 <https://docs.oracle.com/javase/7/docs/api/java/util/Comparator.html>

### 总体

参1：<https://www.geeksforgeeks.org/comparable-vs-comparator-in-java/>

Java provides two interfaces to sort objects using data members of the class: 

1. Comparable
2. Comparator

参2：<https://blog.csdn.net/tolcf/article/details/52229068>

java中，对集合对象或者数组对象排序，有两种实现方式。

（1）对象实现Comparable 接口

     （2）定义比较器，实现Comparator接口。

参3：<https://www.cnblogs.com/skywang12345/p/3324788.html>

Comparable是排序接口；若一个类实现了Comparable接口，就意味着“该类支持排序”。  
而Comparator是比较器；我们若需要控制某个类的次序，可以建立一个“该类的比较器”来进行排序。

Comparable相当于“内部比较器”，而Comparator相当于“外部比较器”。

### 【1】 Comparable

图形用户界面, 文本

描述已自动生成

参1：<https://docs.oracle.com/javase/7/docs/api/java/lang/Comparable.html>

This interface imposes a total ordering on the objects of each class that implements it. This ordering is referred to as the class's ***natural ordering***, and the class's compareTo method is referred to as its ***natural comparison method***.

Lists (and arrays) of objects that implement this interface can be sorted automatically by [Collections.sort](https://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#sort(java.util.List)) (and [Arrays.sort](https://docs.oracle.com/javase/7/docs/api/java/util/Arrays.html#sort(java.lang.Object[]))). Objects that implement this interface can be used as keys in a [sorted map](https://docs.oracle.com/javase/7/docs/api/java/util/SortedMap.html) or as elements in a [sorted set](https://docs.oracle.com/javase/7/docs/api/java/util/SortedSet.html), without the need to specify a [comparator](https://docs.oracle.com/javase/7/docs/api/java/util/Comparator.html).

参2：

<https://www.cnblogs.com/skywang12345/p/3324788.html>

https://blog.csdn.net/tolcf/article/details/52229068

若一个类实现了Comparable接口，就意味着“**该类支持排序**”。

* 实现了Comparable接口的类的对象的 列表或数组 可以通过Collections.sort或Arrays.sort进行排序，通过

**Collections.sort(list,Collections.reverseOrder());** 对list进行倒序排列。

在用Collections类的sort方法排序时若不指定Comparator，那就以自然顺序：***natural ordering***排序。所谓自然顺序就是实现Comparable接口设定的排序方式。

* “实现Comparable接口的类的对象”可以用作“有序映射(如TreeMap)”中的键或“有序集合(TreeSet)”中的元素，而不需要指定Comparator。

参3：

https://www.huaweicloud.com/articles/e892b3fc321f6f17f12376b43e71a2d1.html

在java中Integer，Double等基本数据类型封装类和String都实现了Comparable接口，**可完成比较大小操作，**不用我们在自己去实现。

文本

中度可信度描述已自动生成

文本

描述已自动生成

参4：<https://blog.csdn.net/tolcf/article/details/52229068>

自定义类要在加入集合或数组中后能够排序，可以实现Comparable接口。

### 【2】 Comparator

图形用户界面, 文本, 应用程序

描述已自动生成

参1：<https://docs.oracle.com/javase/7/docs/api/java/util/Comparator.html>

A comparison function:功能, which imposes a ***total ordering*** on some collection of objects. Comparators can be passed to a sort method (such as [Collections.sort](https://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#sort(java.util.List,%20java.util.Comparator)) or [Arrays.sort](https://docs.oracle.com/javase/7/docs/api/java/util/Arrays.html#sort(T[],%20java.util.Comparator))) to allow precise control over the sort order. Comparators can also be used to control the order of certain data structures (such as [sorted sets](https://docs.oracle.com/javase/7/docs/api/java/util/SortedSet.html) or [sorted maps](https://docs.oracle.com/javase/7/docs/api/java/util/SortedMap.html)), or to provide an ordering for collections of objects that don't have a [natural ordering](https://docs.oracle.com/javase/7/docs/api/java/lang/Comparable.html).

参2：https://www.cnblogs.com/skywang12345/p/3324788.html

Comparator 是比较器接口。

我们若需要控制某个类的次序，而该类本身不支持排序(即没有实现Comparable接口)；那么，我们可以建立一个“该类的比较器”来进行排序。这个“比较器”只需要实现Comparator接口即可。

也就是说，我们可以通过“**实现Comparator来新建一个比较器**”，然后通过该比较器对类进行排序。

参3：https://blog.csdn.net/tolcf/article/details/52229068

Comparator是一个专用的比较器，当这个对象不支持自比较或者自比较函数不能满足要求时，可写一个比较器来完成两个对象之间大小的比较。

Comparator体现了一种策略模式(strategy design pattern)，就是不改变对象自身，而用一个策略对象(strategy object)来改变它的行为。

## （2） 使用

### 【1】 Comparable

#### 如何用

<https://www.cnblogs.com/skywang12345/p/3324788.html>

Comparable 接口定义如下：

|  |
| --- |
| public interface Comparable<T> {  */\*\*  \* Compares this object with the specified object for order. Returns a  \* negative integer, zero, or a positive integer as this object is less  \* than, equal to, or greater than the specified object.  \*  \* <p>The implementor must ensure  \* {****@code*** *sgn(x.compareTo(y)) == -sgn(y.compareTo(x))}  \* for all {****@code*** *x} and {****@code*** *y}. (This  \* implies that {****@code*** *x.compareTo(y)} must throw an exception iff  \* {****@code*** *y.compareTo(x)} throws an exception.)  \*  \* <p>The implementor must also ensure that the relation is transitive:  \* {****@code*** *(x.compareTo(y) > 0 && y.compareTo(z) > 0)} implies  \* {****@code*** *x.compareTo(z) > 0}.  \*  \* <p>Finally, the implementor must ensure that {****@code*** *x.compareTo(y)==0}  \* implies that {****@code*** *sgn(x.compareTo(z)) == sgn(y.compareTo(z))}, for  \* all {****@code*** *z}.  \*  \* <p>It is strongly recommended, but <i>not</i> strictly required that  \* {****@code*** *(x.compareTo(y)==0) == (x.equals(y))}. Generally speaking, any  \* class that implements the {****@code*** *Comparable} interface and violates  \* this condition should clearly indicate this fact. The recommended  \* language is "Note: this class has a natural ordering that is  \* inconsistent with equals."  \*  \* <p>In the foregoing description, the notation  \* {****@code*** *sgn(}<i>expression</i>{****@code*** *)} designates the mathematical  \* <i>signum</i> function, which is defined to return one of {****@code*** *-1},  \* {****@code*** *0}, or {****@code*** *1} according to whether the value of  \* <i>expression</i> is negative, zero, or positive, respectively.  \*  \** ***@param*** *o the object to be compared.  \** ***@return*** *a negative integer, zero, or a positive integer as this object  \* is less than, equal to, or greater than the specified object.  \*  \** ***@throws*** *NullPointerException if the specified object is null  \** ***@throws*** *ClassCastException if the specified object's type prevents it  \* from being compared to this object.  \*/* public int compareTo(T o); } |

#### 实例

https://blog.csdn.net/tolcf/article/details/52229068

|  |
| --- |
| public class Main1 {   public static void main(String[] args) {  List<UserInfo> list = new ArrayList<>();  list.add(new UserInfo(1,21,"name1"));  list.add(new UserInfo(2,27,"name2"));  list.add(new UserInfo(3,15,"name3"));  list.add(new UserInfo(5,24,"name5"));  list.add(new UserInfo(4,24,"name4"));   //排序  Collections.*sort*(list);  for (int i = 0; i < list.size(); i++) {  System.*out*.println(list.get(i)); //  }  } }  class UserInfo implements Comparable<UserInfo> {   private int userid;  private int age;  private String name;   public UserInfo(int userid, int age, String name) {  this.userid = userid;  this.age = age;  this.name = name;  }   public int getUserid() {  return userid;  }   public void setUserid(int userid) {  this.userid = userid;  }   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 String toString() {  return "UserInfo{" +  "userid=" + userid +  ", age=" + age +  ", name='" + name + '\'' +  '}';  }   @Override  public int compareTo(UserInfo o) {  //如果年龄相同，则比较userId，否则比较年龄  if (this.age - o.age == 0) {  return this.userid - o.userid;  } else {  return this.age - o.age;  }  } } |

文本

描述已自动生成

### 【2】 Comparator

#### 《1》 如何用

Comparator 接口包含2个抽象方法，定义如下：

|  |
| --- |
| @FunctionalInterface public interface Comparator<T> {  */\*\*  \* Compares its two arguments for order. Returns a negative integer,  \* zero, or a positive integer as the first argument is less than, equal  \* to, or greater than the second.<p>  \*  \* The implementor must ensure that {****@code*** *sgn(compare(x, y)) ==  \* -sgn(compare(y, x))} for all {****@code*** *x} and {****@code*** *y}. (This  \* implies that {****@code*** *compare(x, y)} must throw an exception if and only  \* if {****@code*** *compare(y, x)} throws an exception.)<p>  \*  \* The implementor must also ensure that the relation is transitive:  \* {****@code*** *((compare(x, y)>0) && (compare(y, z)>0))} implies  \* {****@code*** *compare(x, z)>0}.<p>  \*  \* Finally, the implementor must ensure that {****@code*** *compare(x, y)==0}  \* implies that {****@code*** *sgn(compare(x, z))==sgn(compare(y, z))} for all  \* {****@code*** *z}.<p>  \*  \* It is generally the case, but <i>not</i> strictly required that  \* {****@code*** *(compare(x, y)==0) == (x.equals(y))}. Generally speaking,  \* any comparator that violates this condition should clearly indicate  \* this fact. The recommended language is "Note: this comparator  \* imposes orderings that are inconsistent with equals."<p>  \*  \* In the foregoing description, the notation  \* {****@code*** *sgn(}<i>expression</i>{****@code*** *)} designates the mathematical  \* <i>signum</i> function, which is defined to return one of {****@code*** *-1},  \* {****@code*** *0}, or {****@code*** *1} according to whether the value of  \* <i>expression</i> is negative, zero, or positive, respectively.  \*  \** ***@param*** *o1 the first object to be compared.  \** ***@param*** *o2 the second object to be compared.  \** ***@return*** *a negative integer, zero, or a positive integer as the  \* first argument is less than, equal to, or greater than the  \* second.  \** ***@throws*** *NullPointerException if an argument is null and this  \* comparator does not permit null arguments  \** ***@throws*** *ClassCastException if the arguments' types prevent them from  \* being compared by this comparator.  \*/* int compare(T o1, T o2);   */\*\*  \* Indicates whether some other object is &quot;equal to&quot; this  \* comparator. This method must obey the general contract of  \* {****@link*** *Object#equals(Object)}. Additionally, this method can return  \* {****@code*** *true} <i>only</i> if the specified object is also a comparator  \* and it imposes the same ordering as this comparator. Thus,  \* {****@code*** *comp1.equals(comp2)} implies that {****@code*** *sgn(comp1.compare(o1,  \* o2))==sgn(comp2.compare(o1, o2))} for every object reference  \* {****@code*** *o1} and {****@code*** *o2}.<p>  \*  \* Note that it is <i>always</i> safe <i>not</i> to override  \* {****@code*** *Object.equals(Object)}. However, overriding this method may,  \* in some cases, improve performance by allowing programs to determine  \* that two distinct comparators impose the same order.  \*  \** ***@param*** *obj the reference object with which to compare.  \** ***@return*** *{****@code*** *true} only if the specified object is also  \* a comparator and it imposes the same ordering as this  \* comparator.  \** ***@see*** *Object#equals(Object)  \** ***@see*** *Object#hashCode()  \*/* boolean equals(Object obj);  。。。 //剩下的default和static方法省略  } |

1. 若一个类要实现Comparator接口：它一定要实现compare(T o1, T o2) 方法，但可以不实现 equals(Object obj) 方法。

这涉及Comparator是函数式接口以及其继承Object的关系

#### 《2》 实例

##### 1） 例1：类未实现Comparable+未重写equals

https://blog.csdn.net/tolcf/article/details/52229068

|  |
| --- |
| import java.util.ArrayList; import java.util.Collections; import java.util.Comparator; import java.util.List;  public class Main2 {   public static void main(String[] args) {  List<UserInfo> list = new ArrayList<UserInfo>();  list.add(new UserInfo(1,21,"name1"));  list.add(new UserInfo(2,27,"name2"));  list.add(new UserInfo(3,15,"name3"));  list.add(new UserInfo(5,24,"name5"));  list.add(new UserInfo(4,24,"name4"));   //new一个比较器  MyComparator myComparator = new MyComparator();  //对list排序  Collections.*sort*(list, myComparator);   for (UserInfo userInfo : list) {  System.*out*.println(userInfo);  }  } }  class MyComparator implements Comparator<UserInfo> {    @Override  public int compare(UserInfo o1, UserInfo o2) {  //如果年龄相同，则比较userId，否则比较年龄  if (o1.getAge() - o2.getAge() == 0) {  return o1.getUserid() - o2.getUserid();  } else {  return o1.getAge() - o2.getAge();  }  } }  class UserInfo {   private int userid;  private int age;  private String name;   public UserInfo(int userid, int age, String name) {  this.userid = userid;  this.age = age;  this.name = name;  }   public int getUserid() {  return userid;  }   public void setUserid(int userid) {  this.userid = userid;  }   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 String toString() {  return "UserInfo{" +  "userid=" + userid +  ", age=" + age +  ", name='" + name + '\'' +  '}';  } } |

文本

描述已自动生成

##### 2） 例2：类实现Comparable+未重写equals

https://blog.csdn.net/tolcf/article/details/52229068

|  |
| --- |
| import java.util.ArrayList; import java.util.Collections; import java.util.Comparator; import java.util.List;  public class Main2 {   public static void main(String[] args) {  List<UserInfo> list = new ArrayList<UserInfo>();  list.add(new UserInfo(1,21,"name1"));  list.add(new UserInfo(2,27,"name2"));  list.add(new UserInfo(3,15,"name3"));  list.add(new UserInfo(5,24,"name5"));  list.add(new UserInfo(4,24,"name4"));   System.*out*.println("Origin sort: " + list);  Collections.*sort*(list);  System.*out*.println("Comparable sort: " + list);  //new一个比较器  MyComparator myComparator = new MyComparator();  //对list排序  Collections.*sort*(list, myComparator); //comparator会替换comparable算法  System.*out*.println("Comparator sort: " + list);  } }  class MyComparator implements Comparator<UserInfo> {   @Override  public int compare(UserInfo o1, UserInfo o2) {  //如果年龄相同，则比较userId，否则比较年龄  if (o1.getAge() - o2.getAge() == 0) {  return o1.getUserid() - o2.getUserid();  } else {  return o1.getAge() - o2.getAge();  }  } }  class UserInfo implements Comparable<UserInfo> {   private int userid;  private int age;  private String name;   public UserInfo(int userid, int age, String name) {  this.userid = userid;  this.age = age;  this.name = name;  }   public int getUserid() {  return userid;  }   public void setUserid(int userid) {  this.userid = userid;  }   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 String toString() {  return "{userid=" + userid +  ", age=" + age +  ", name='" + name + '\'' +  '}';  }   @Override  public int compareTo(UserInfo o) {  //若年龄相同，则倒序排id；否则，倒序排age  if (o.getAge() - this.age == 0) {  return o.getUserid() - this.userid;  } else {  return o.getAge() - this.age;  }  } } |

结果：

|  |
| --- |
| Origin sort: [{userid=1, age=21, name='name1'}, {userid=2, age=27, name='name2'}, {userid=3, age=15, name='name3'}, {userid=5, age=24, name='name5'}, {userid=4, age=24, name='name4'}]  Comparable sort: [{userid=2, age=27, name='name2'}, {userid=5, age=24, name='name5'}, {userid=4, age=24, name='name4'}, {userid=1, age=21, name='name1'}, {userid=3, age=15, name='name3'}]  Comparator sort: [{userid=3, age=15, name='name3'}, {userid=1, age=21, name='name1'}, {userid=4, age=24, name='name4'}, {userid=5, age=24, name='name5'}, {userid=2, age=27, name='name2'}] |

##### 2） 例3：类实现Comparable+多个比较器+重写equals

<https://www.cnblogs.com/skywang12345/p/3324788.html>

Person类：

|  |
| --- |
|  |

|  |
| --- |
|  |

##### 3） 例3：重写equals

https://www.cnblogs.com/skywang12345/p/3324788.html

#### 自归纳：

参考：

1. 升降序记法：

<https://blog.csdn.net/u011277123/article/details/70780135>

<https://blog.csdn.net/u013066244/article/details/78895747>

<https://zhuanlan.zhihu.com/p/54004622>

<https://www.cnblogs.com/daixianjun/p/java-compare.html>

1. 源码分析：

<https://blog.csdn.net/u013066244/article/details/78997869>

1） compare（T，T）记法+使用

1》 compare（Integer num1，Integer num2）

升序：return num1-num2；

降序：return num2-num1；

2） Comparator对基本数据类型包装类的写法

模板写法：

|  |
| --- |
| Integer[] integers = {1, 2, 3}; Arrays.sort(integers, new Comparator<Integer>() {  public int compare(Integer a, Integer b) {  return xx;  } }); |

### 使用对比

参1：

<https://www.huaweicloud.com/articles/e892b3fc321f6f17f12376b43e71a2d1.html>

两者的区别是实现Comparator接口代码更加灵活，可以定义某个类的多个比较器，从而在排序时根据实际场景自由调用，而Comparable接口实现后便不能改动。

参2：<http://www.itwanger.com/java/2020/01/04/java-comparable-comparator.html>

通过上面的两个例子可以比较出 Comparable 和 Comparator 两者之间的区别：

* 一个类实现了 Comparable 接口，意味着该类的对象可以直接进行比较（排序），但比较（排序）的方式只有一种，很单一。
* 一个类如果想要保持原样，又需要进行不同方式的比较（排序），就可以定制比较器（实现 Comparator 接口）。
* Comparable 接口在 java.lang 包下，而 Comparator 接口在 java.util 包下，算不上是亲兄弟，但可以称得上是表（堂）兄弟。

举个不恰当的例子。我想从洛阳出发去北京看长城，体验一下好汉的感觉，要么坐飞机，要么坐高铁；但如果是孙悟空的话，翻个筋斗就到了。我和孙悟空之间有什么区别呢？孙悟空自己实现了 Comparable 接口（他那年代也没有飞机和高铁，没得选），而我可以借助 Comparator 接口（现代化的交通工具）。

参3：<https://blog.csdn.net/tolcf/article/details/52229068>

我们若需要控制某个类的次序,而该类本身不支持排序(即没有实现Comparable接口);那么，我们可以新建一个该类的比较器来进行排序。这个比较器只需要实现comparator即可。

如果引用的为第三方jar包，这时候，没办法改变类本身，可是使用这种方式。

参4：https://www.cnblogs.com/xujian2014/p/5215082.html

两种方法各有优劣， 用Comparable 简单， 只要实现Comparable 接口的对象直接就成为一个可以比较的对象，但是需要修改源代码。 用Comparator 的好处是不需要修改源代码， 而是另外实现一个比较器， 当某个自定义的对象需要作比较的时候，把比较器和对象一起传递过去就可以比大小了， 并且在Comparator 里面用户可以自己实现复杂的可以通用的逻辑。

# Aggregate Operations

<https://docs.oracle.com/javase/tutorial/collections/streams/index.html>

# 企业规范

## （1） oppo

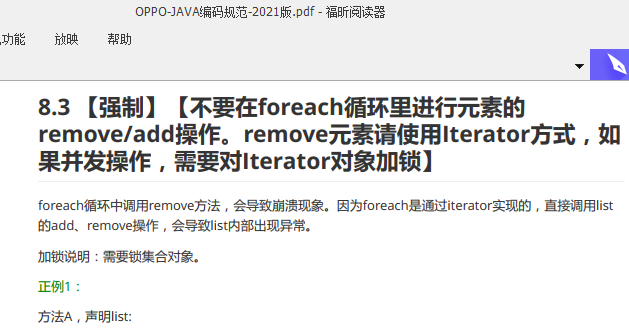
### 【1】



### 【2】



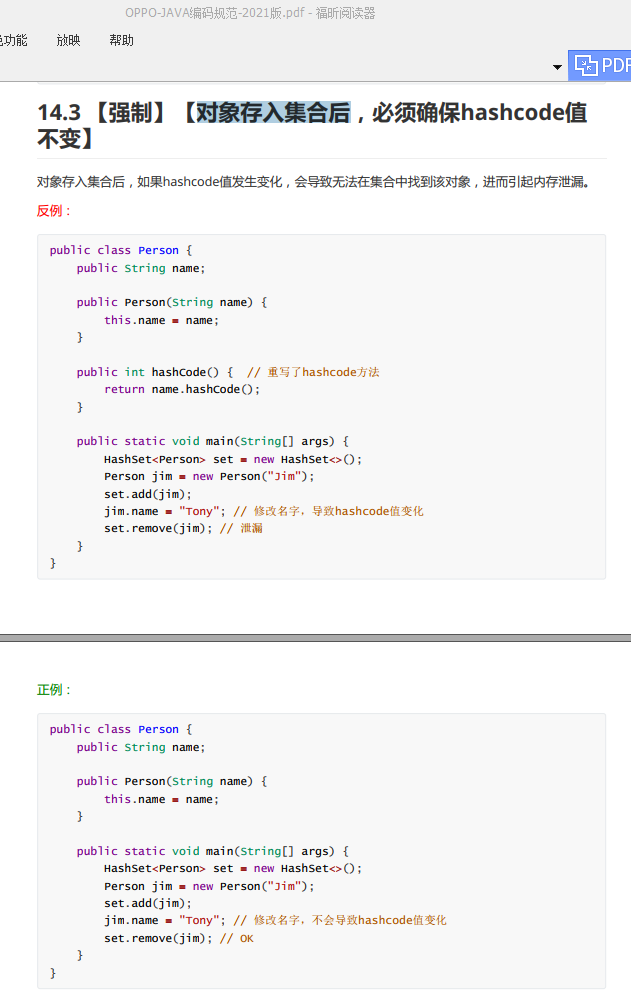
### 【3】







### 【4】



# 常见面试题

## （1） HashMap相关

### 问题集1：

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

#### 问1：

1. Hash底层原理：

在jdk7中，是通过 数组+链表 实现的。

在jdk8中，是通过 数组+（链表/红黑树） 实现的。

1. HashMap 是非线程安全类，在多线程环境下可能会存在问题。

* 可通过Collections.synchronizedMap(HashMap对象) 将已有HashMap转为线程安全的。
* jdk中提供的线程安全的Map实现有Hashtable，ConcurrentHashMap

#### 问2：

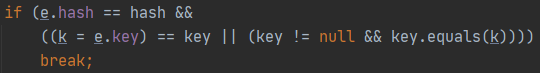
在jdk8中，流程如下：

1. 如果桶数组table为null或长度为0，则通过扩容的方式初始化table
2. 根据key定位要插入的键值对在数组中的位置/属于哪个桶

定位相关的代码为和

1. 定位到桶后，再判断桶是否为空：

* 如果为空，则将键值对（作为链表头结点）存入即可。
* 如果不为空，则（在链表或红黑树中）查找要put的键值对是否已存在：
* 若存在，则根据条件判断是否用新值替换旧值

判断存在的代码为；

判断是否用新值替换旧值的代码为，因为put方法的onlyIfAbsent参数为false，所以始终会替换。

* 若不存在，则将键值对插入链表尾/红黑树。对于链表，若此时其长度大于TREEIFY\_THRESHOLD(8)则将链表转为红黑树

【3】 最后，如果此时键值对数量超过阈值threshold，则进行扩容。

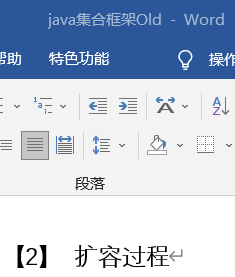
#### 问3：

基于jdk8

1. 扩容时机

在put操作中，有2处：

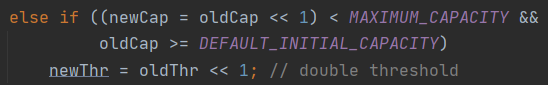
1. 桶数组table为null或长度为0时，通过扩容的方式初始化table
2. 插入新元素后，若此时键值对总量大于阈值threshold，则进行扩容
3. 扩容resize（）如何实现？

具体见。

主要分3步：

1. 计算新桶数组的容量newCap和新扩容阈值newThr

存在多个条件分支根据不同情况分别设值。一般情况下newCap和newThr变为旧容量oldCap和旧阈值oldThr的两倍，对应代码

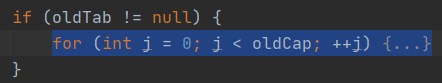
。

1. 根据计算出的newCap创建新的桶数组，并赋给全局变量table

对应代码：

。

1. 按老数组中桶的顺序遍历每个桶的键值对节点，重新放置/映射到新数组中

对应代码：。

对于老数组中的每个桶的处理分三种情况：

* 单节点（Node类型）：通过e.hash & (newCap - 1)定位到在新数组中的位置，并将其直接放置到根节点。
* 链表（Node类型）

将原链表按原相对顺序分成可能的两组，并放进新数组中。

分两步：

* 1. 遍历原链表

对原链表中的每个节点e：当e.hash & oldCap的结果为0时插到loHead为链表头、loTail为链表尾的链表中；否则插到hiHead为链表头、hiTail为链表尾的链表中。

* 1. 将链表loHead存到newTab[j]中，链表hiHead存到newTab[j+oldCap]中
* 红黑树（TreeNode类型）：

和链表节点的处理类似：将原红黑树拆分成可能的两组，并放进新数组中。只是将Node类型换成了TreeNode类型 且 多了树化相关操作。

同样分两步：

* 1. 遍历原红黑树

因红黑树节点仍保留了next引用，故仍可以按链表方式遍历红黑树。

对原红黑树中的每个节点e：当e.hash & oldCap的结果为0时插到loHead为链表头、loTail为链表尾的链表中；否则插到hiHead为链表头、hiTail为链表尾的链表中。

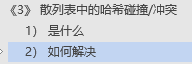
* 1. 将loHead存到newTab[j]中，hiHead存到newTab[j+oldCap]中。这里相比链表的对应步骤多的操作是：如果链表loHead/hiHead的长度小于等于UNTREEIFY\_THRESHOLD(6)，则将TreeNode类型的链表转为Node类型的普通链表；否则，根据条件重新将TreeNode类型的链表树化。

#### 问4：

1. 什么是哈希碰撞？

见“密码学.docx”

1. 如何解决哈希碰撞？

见“密码学.docx”

HashMap采用拉链法。

#### 问5：HashMap和Hashtable对比

见

#### 问6：

同问3

#### 问7：HashMap和ConcurrentHashMap对比

见

#### 问8：

##### 《1》 ArrayList和HashMap对比

<https://blog.csdn.net/yaodieteng1510/article/details/79582925>

<https://segmentfault.com/a/1190000012926722>

###### 1） 相同

1》 都是线程不安全，不同步

2》 都可以储存null值

3》 获取元素个数方法一样，都用size()方法获取

###### 2） 不同

1》 实现的接口

ArrayList实现了List接口，底层使用的是（动态）数组。

HashMap实现了Map接口，底层使用的是哈希表。

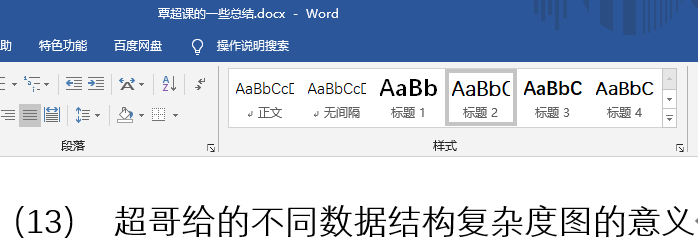
2》 默认的大小

ArrayList的默认大小为10，HashMap的默认大小为16。

3》 扩容

ArrayList扩容为原容量的1.5倍，HashMap扩容为原容量的2倍。

4》 时间复杂度

参考：

平均时间复杂度上：

* 随机访问：ArrayList为O（1），HashMap无随机访问
* 查找（给某个元素，查该元素）：ArrayList为O（n），HashMap为O（1）
* 插入：ArrayList为O（n），HashMap为O（1）
* 删除：ArrayList为O（n），HashMap为O（1）

5》 使用场景

若存储不需要键值对形式的数据，应该用ArrayList；否则，应该用

HashMap。

##### 《2》 ArrayList和HashMap谁取数快？

<https://blog.csdn.net/Ambitiouss/article/details/100081190>

<https://blog.csdn.net/yaodieteng1510/article/details/79582925> 评论区

因为HashMap只能查询操作（给元素，查元素），无法随机访问，所以按查询操作的时间复杂度来比较两者，平均情况下ArrayList为O（n），HashMap为O（1），所以HashMap更快。

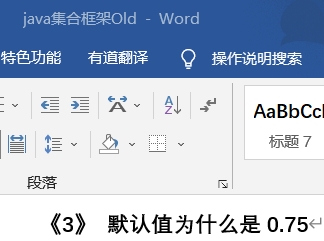
### 问题2：Hashtable和ConcurrentHashMap对比

问题源：享学公开课

见

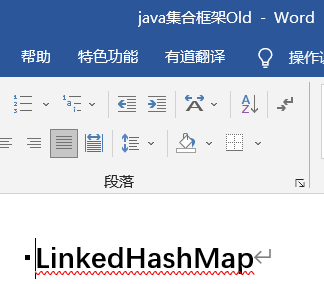
### 问题3：HashMap的默认负载因子为什么是0.75？

问题源：<https://leetcode.cn/circle/discuss/hZSCJk/>

回答：见

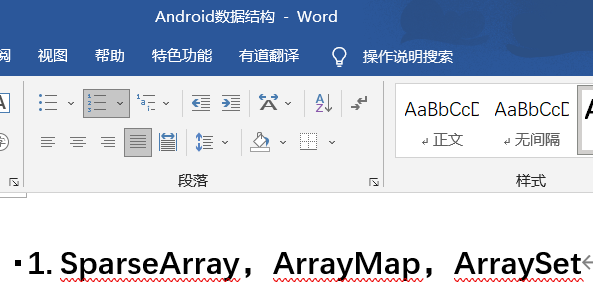
### 问题4：HashMap和LinkedHashMap对比

问题源：<https://blog.csdn.net/qq_27053103/article/details/79564062>

回答：见

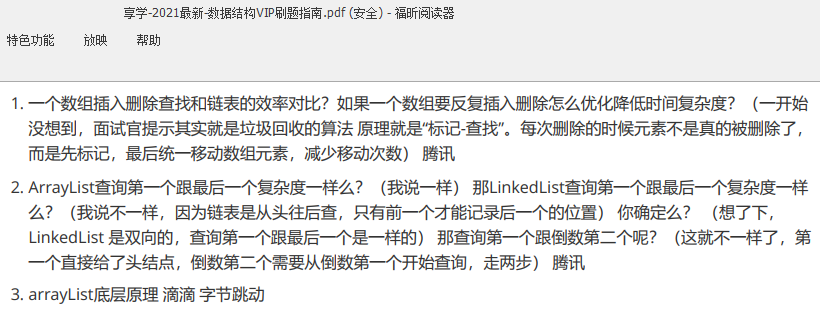
### 问题4：HashMap和SparseArray对比

问题源：<https://blog.csdn.net/qq_27053103/article/details/79564062>

回答：见

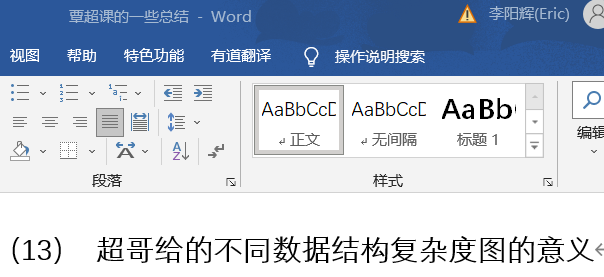
## （2） List相关

### 问题集1：



#### 问1：

1. 数组和链表的效率对比

见

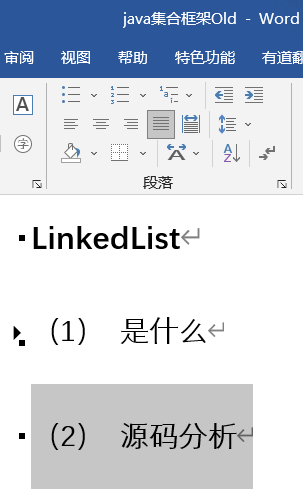
表格

描述已自动生成

1. 按给的题解，应该是参考了GC“标记-清除”算法，同见SparseArray的delete方法原理

#### 问2：

1. ArrayList：查询第一个和最后一个复杂度一样，都是O（1），通过get方法对数组随机访问
2. LinkedList

具体见

* + 1. 查询第一个和最后一个复杂度一样，都是O（1），因为有first和last指针
    2. 查询第一个和倒数第二个不同，第一个直接用first指针，倒数第二个要从last指针往前走一步

#### 问3：

<https://www.cnblogs.com/neverth/p/11786048.html>

<https://juejin.cn/post/6872669538341617671>

<https://juejin.cn/post/6986833022427349005>

1. 底层是（动态）数组

1》 实现RandomAccess接口：支持随机访问

2》 数组容量不够时会扩容为原来的1.5倍

1. 非线程安全
2. 时间复杂度：随机访问、获取表长O（1），增删平均O（n）

### 问题2：ArrayList和LinkedList对比

<https://zhuanlan.zhihu.com/p/68478784>

<http://c.biancheng.net/view/6843.html>

<https://zhuanlan.zhihu.com/p/58515780>

<https://blog.csdn.net/eson_15/article/details/51145788>

<https://zhuanlan.zhihu.com/p/33141246>

<https://cloud.tencent.com/developer/article/1741097>

<https://juejin.cn/post/6950578141068787720>

#### 《1》 相同

1） 都实现了List接口

2） 都是非线程安全，多线程环境下容易造成脏读问题。若要使ArrayList是线程安全的，可用Collections.synchronizedList(已有ArrayList对象)，也可使用CopyOnWriteArrayList或Vector；若要使LinkedList是线程安全的，可用Collections.synchronizedList(new LinkedList())。

#### 《2》 不同

1） ArrayList基于动态数组【即数组大小可变】，连续一块内存空间；LinkedList基于双（向）链表，不一定是连续的内存空间

2）随机访问【即根据索引/下标来访问】的get和set方法，ArrayList【时复为O（1）】优于LinkedList【时复为O（n）】

注：LinkedList的随机访问是根据index大小从前/后开始搜索

3） 添加（add方法）、删除（remove方法）元素，LinkedList【时复为O（1）】优于ArrayList【时复为O（n）】

注：上面的说法是通用的答案，但若仔细分析则需要结合数量和增删位置（头、尾、中间）来判断

4） 存储数据相同时，LinkedList比ArrayList更占内存，因为LinkedList的节点除了存储数据，还存储了两个引用，一个指向前一个元素，一个指向后一个元素。

5） 扩容：

1》 LinkedList不存在扩容的说法，因为是链表结构；

2》 ArrayList底层是动态数组存在扩容说法，默认的数组大小是10，在检测是否容量不够需要扩容后，如果要扩容，会扩容为原来数组容量的1.5倍大小

|  |
| --- |
| **int** oldCapacity **=** elementData**.**length**;**  **int** newCapacity **=** oldCapacity **+** **(**oldCapacity **>>** 1**);** |

然后把老数组的元素复制到新数组里面

|  |
| --- |
| elementData **=** Arrays**.**copyOf**(**elementData**,** newCapacity**)** |

### 问题3：ArrayList和Vector对比

<https://zhuanlan.zhihu.com/p/28241176>

<https://cloud.tencent.com/developer/article/1835276>

<https://www.nowcoder.com/questionTerminal/0953369f92054cbfbf1024a1e723e04f>

<https://www.runoob.com/java/java-vector-class.html>

[https://www.cnblogs.com/wanlipeng/archive/2010/10/21/1857791.html 非jdk8](https://www.cnblogs.com/wanlipeng/archive/2010/10/21/1857791.html%20非jdk8)

<https://cloud.tencent.com/developer/article/1339659>

#### 《1》 相同

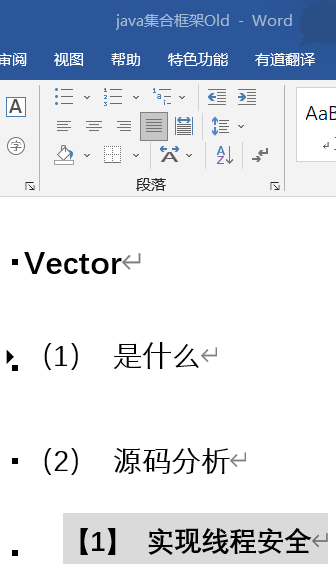
1） 都实现了List接口

2） 底层都是动态数组，默认大小都是10

#### 《2》 不同

1. Vector是线程安全的，ArrayList不是线程安全。所以ArrayList比Vector快/效率高，因为加锁和释放锁在系统中是有开销的。

注：Vector是通过synchronized方法实现线程安全的，具体见

。

1. 扩容：Vector默认扩展1倍，ArrayList默认扩展50%
2. Vector目前很少用了，主要是性能差；ArrayList常用

### 问题4：Vector和CopyOnWriteArrayList对比

问题源：<https://leetcode.cn/circle/discuss/hZSCJk/>

回答：见