# 集合概述

为了保存数量不确定的数据，以及保存具有映射关系的数据，java提供了集合类，集合类主要负责保存数据，因此集合类也被称为容器类。所有的集合类都位于java.util包下，后来为了处理多线程环境下的并发安全问题，java 5还在java.util.concurrent包下提供了一些多线程支持的集合类。

# 集合中的接口

Java集合类主要由两个接口派生而出：Collection和Map接口。Collection和Map接口是Java集合的根接口，这两个接口包含了一些子接口或实现类。List和Set接口是Collection接口的子接口。

# 访问集合中的元素

如果要访问List集合中的元素，可以直接根据元素的索引来访问。如果访问Set集合中的元素，则只能根据元素本身来访问。如果访问Map集合中的元素，可以根据每项元素的key来访问value。

# Collection集合的用法

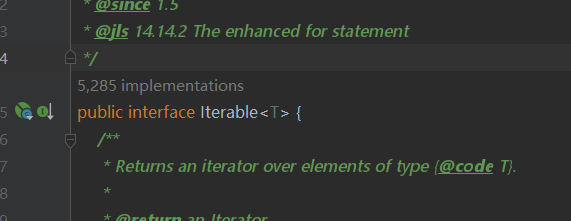
添加元素、删除元素、返回集合中元素的个数和清空整个集合。

# Iterable接口

## 基本概念

在Java中，Iterable接口是所有集合类的父接口，它提供了一种通用的方法来访问集合中的元素。实现了Iterable接口的类可以使用foreach语句进行遍历，即支持for-each循环。

## 源码



## 方法

### default void forEach(Consumer<? super T> action)

### Iterator<T> iterator()

返回一个实现了 Iterator<T> 接口的对象，用于迭代访问集合中的元素。

### default Spliterator<T> spliterator()

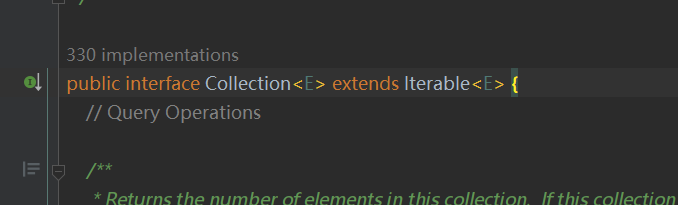
用于支持流式操作（streams）和并行迭代的关键方法

## 总结

Iterable接口是Java集合框架中的一个重要接口，通过实现它可以获得支持foreach循环的能力。我们可以通过调用iterator()方法获取一个迭代器对象，然后在循环中使用该迭代器对象来遍历集合中的元素。同时，为了使用foreach循环，我们还需要保证所遍历的集合实现了Iterable接口。

# Collection接口

## 源码



## 方法

### boolean add(E e)

添加元素

### boolean addAll(Collection c)

添加集合

### void clear()

清空集合

### boolean contains(Object o)

集合中包含指定的元素，返回true

### boolean equals(Object o)

比较集合与指定对象是否相等

### boolean isEmpty()

集合不包含元素，返回true

### Iterator iterator()

返回集合元素上进行的迭代器

### boolean remove(Object o)

删除元素

### boolean removeAll(Collection c)

删除集合

### boolean retainAll(Collection c)

从列表中移除未包含在指定collection中的所有元素，Collection 接口中的 boolean retainAll(Collection c) 方法用于更新集合，使其只保留在指定集合 c 中也存在的元素。换句话说，这个方法会从当前集合中移除所有不在集合 c 中的元素，只保留**当前集合与指定集合的交集部分**。

### int size()

返回集合的大小

### Object[] toArray()

返回包含集合中所有元素的数组（把集合转换为对象数组）

### T[] toArray(T[] a)

返回包含此 collection 中所有元素的数组，返回数组的运行时类型与指定数组的运行时类型相同。

# Collector接口

## 基本概念

Collector接口是一个非常重要的接口，用于将流中的元素收集到一个容器中。Collector接口提供了多种方法来定义如何收集元素，可以用于各种场景，例如将元素收集到List、Set、Map等容器中，或者进行分组、分区等操作。

## 方法

### Supplier<A> supplier()

返回一个用于创建容器的Supplier对象

### BiConsumer<A, T> accumulator()

返回一个用于将元素添加到容器中的BiConsumer对象

# Iterator接口

## 基本概念

Iterator接口也是Java集合框架的成员，但是与Collection、Map接口不同，Collection、Map集合主要用于存储其他对象，而Iterator接口主要用于遍历Collection集合中的元素，Iterator也被称为迭代器。Iterator接口隐藏了各种Collection实现类的底层细节，向应用程序提供了遍历Collection集合元素的统一编程接口，接口中定义了三个方法：

（1）boolean hasNext()：如果仍有元素可以迭代，则返回true。

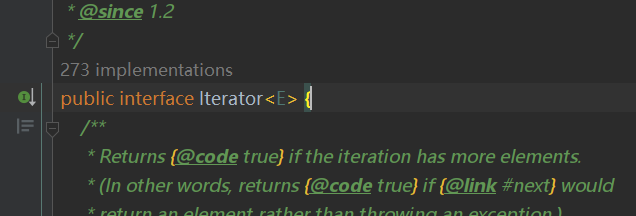
（2）Object next()：返回迭代的下一个元素。

（3）void remove()：从迭代器指向的collection集合中移除迭代器返回的最后一个元素。

Iterator仅用于遍历集合，Iterator本身并不提供存储对象的能力，如果需要创建Iterator对象，则必须有一个被迭代的集合，没有集合的Iterator对象没有任何价值。当使用Iterator对集合进行遍历时，Iterator并不是把集合元素本身传给了迭代变量，而是把集合元素的值传给了迭代变量，所以修改迭代变量的值对集合元素本身没有任何影响。当使用Iterator对集合进行遍历时，集合里的元素不能被改变，只有通过remove()方法删除上一次next()方法返回的集合元素才可以，否则引发程序异常。

Iterator迭代器采用的是快速失败机制，一旦在迭代过程中检测到该集合已经被修改（通常是程序中的其他线程修改），程序就会引发异常，而不是显示修改后的结果，这样可以避免共享资源而引发的潜在问题。

## 源码



## 方法

### boolean hasNext();

如果迭代器中包含元素，则返回true。

### E next();

返回迭代器中的下一个元素

### default void remove()

从迭代器指向的collection集合中移除迭代器返回的最后一个元素。

### default void forEachRemaining(Consumer<? super E> action)

对迭代器中的每个剩余元素执行给定的操作，直到所有元素都已经被处理或者操作引发异常为止。

forEachRemaining方法是Java 8中Iterator接口新增的一个默认方法，它可以极大地简化对迭代器中元素的操作。通过该方法，我们可以避免手动编写while循环来遍历迭代器中的所有元素。

# Iterable接口与Iterator接口的区别

## 1、Iterable 接口

（1）用于支持增强型 for-each 循环。

（2）包含单个方法 iterator()，返回一个 Iterator 对象。

（3）集合类实现 Iterable 接口可以直接用于 foreach 循环。

## 2、Iterator 接口

（1）提供了一种访问集合元素的统一方式。

（2）提供了 hasNext() 和 next() 方法进行逐个访问集合元素。

（3）可以通过 remove() 方法删除集合中的元素。

在实际应用中，Iterable 接口和 Iterator 接口通常一起使用，集合类实现 Iterable 接口以便于使用 foreach 循环遍历，而在需要更多控制的情况下，可以通过获取 Iterator 对象来手动遍历和操作集合元素。

# Set接口

## 基本概念

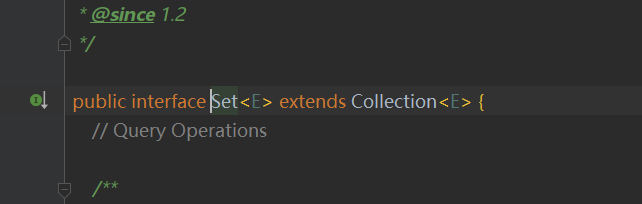
Set集合存储无序、不能重复的数据，Set集合判断两个对象相同不是使用==运算符，而是使用equal()方法进行判断的，也就是说只要两个对象用equal方法进行判断返回true时，Set集合就不会接受这两个对象，反之只要两个对象用equal方法比较返回false，Set就会接受这两个对象。

Set接口的实现类：HashSet、LinkedHashSet、TreeSet。

## 判断两个对象相同的标准

使用equal()方法进行判断，只要两个对象用equal方法进行判断返回true时，Set集合就不会接受这两个对象。

## 源码



## 方法

### boolean add(E e)

添加元素

### void clear()

清空集合中的所有元素

### boolean contains(Object o)

如果此 set 包含指定元素，则返回 true

### boolean equals(Object o)

比较指定对象与Set的相等性

### boolean isEmpty()

集合不包含任何元素，返回true

### Iterator iterator()

返回集合元素的迭代器

### boolean remove(Object o)

删除元素

### int size()

返回集合的大小

### Object[] toArray()

返回Set集合中所有元素的数组

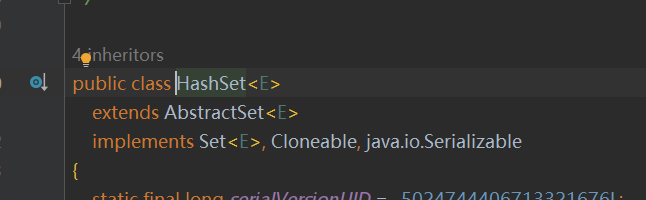
# HashSet实现类

## 基本概念

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算法来存储集合中的元素，因此具有很好的存取和查找性能。

HashSet集合判断两个元素相等的标准是两个对象通过equal()方法比较相等，并且两个对象的hashCode()方法返回值也相等。如果需要把某个类的对象保存到HashSet集合中，重写这个类的equal()方法和hashCode()方法时，应该尽量保证两个对象通过equal()方法比较返回true时，它们的hashCode()方法返回值也相等。

## 源码



## 特点

（1）不能保证元素的排列顺序，顺序有可能发生变化。（无序）

（2）HashSet不是同步的，如果多个线程同时访问一个HashSet，必须通过代码来保证其同步。

（3）集合元素值可以是null。（List集合元素值也可以存储null）

## 底层数据结构

对于HashSet而言，它是基于HashMap来实现的，底层采用HashMap来保存元素。

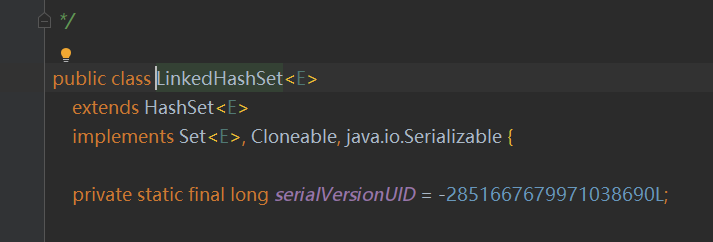
# LinkedHashSet实现类

## 基本概念

LinkedHashSet是HashSet的子类，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。（有序的）

LinkedHashSet需要维护元素的插入顺序，因此性能低于HashSet的性能，但在迭代访问Set里全部元素时将有很好的性能，因为它以链表来维护内部顺序。LinkedHashSet使用了链表来记录集合元素的添加顺序，但LinkedHashSet依然是HashSet，因此它依然不允许集合元素重复。LinkedHashSet是有序不重复的。

## 源码

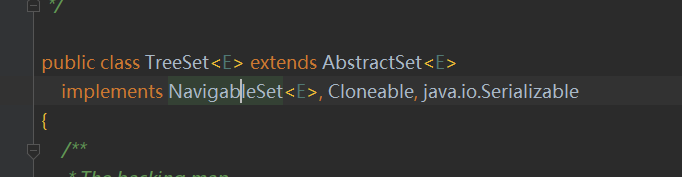


# TreeSet实现类

## 基本概念

TreeSet是SortSet接口的实现类，TreeSet可以确保集合元素处于排序状态。

## 源码



## 方法

### （1）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，默认采用自然排序。

### （2）Object first()

返回集合中的第一个元素。

### （3）Object last()

返回集合中的最后一个元素。

### （4）Object lower(Object e)

返回集合中位于指定元素之前的元素。

### （5）Object higher(Object e)

返回集合中位于指定元素之后的元素。

### （6）SortedSet subSet(fromElement,toElement)

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

### （7）SortedSet headSet(toElement)

返回此Set的子集，由小于toElement的元素组成。

### （8）SortedSet tailSet(fromElement)

返回此Set的子集，由大于或等于fromElement的元素组成。

## 其他说明

TreeSet中的元素是有序的，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。

TreeSet并不是根据元素的插入顺序进行排序的，而是根据元素实际值的大小来进行排序的，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，在默认情况下，TreeSet采用自然排序。 自然排序：TreeSet会调用集合元素的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。

说明：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，还有向TreeSet中添加的应该是同一个类的对象，否则也会引发ClassCastException异常。

## 底层原理

底层是基于TreeMap来实现的，所以底层结构也是红黑树，因为他和HashSet不同的是不需要重写hashCode()和equals()方法，因为它去重是依靠比较器来去重，因为结构是红黑树，所以每次插入都会遍历比较来寻找节点插入位置，如果发现某个节点的值是一样的那就会直接覆盖。

# HashSet与TreeSet的区别

## HashSet

1、实现: 基于哈希表（HashMap），存储元素时不保持任何顺序。

2、时间复杂度: 大部分操作（如 add, remove, contains）的时间复杂度是 O(1)。

3、顺序: 不保证元素的顺序，元素的顺序是不可预测的。

4、允许: 允许存储 null 元素。

5、线程安全: HashSet 不是线程安全的。如果需要线程安全，可以使用 Collections.synchronizedSet(new HashSet<>()) 或 ConcurrentHashMap.newKeySet()。

## TreeSet

1、实现: 基于红黑树（自平衡的二叉搜索树），存储元素时保持自然排序或者提供的比较器。

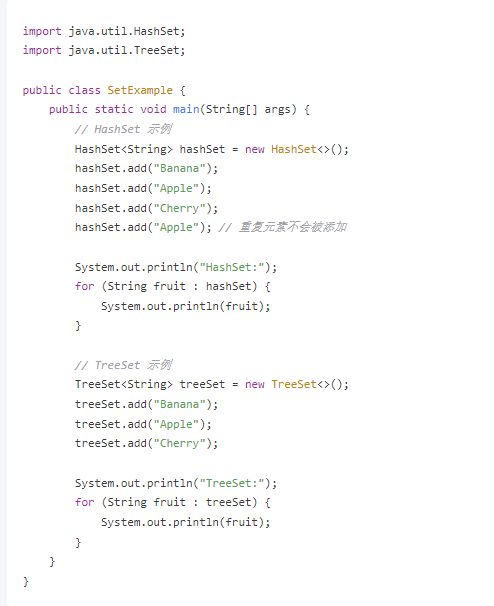
2、时间复杂度: 大部分操作（如 add, remove, contains）的时间复杂度是 O(log n)。

3、顺序: 保证元素的自然排序或使用提供的比较器排序。

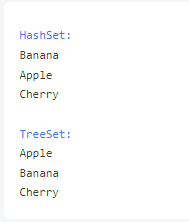
4、允许: 不允许存储 null 元素（如果是自定义比较器的情况也不允许）。

5、线程安全: TreeSet 也不是线程安全的。使用同步包装器或其他线程安全的集合来解决并发问题。

## 示例代码



输出：



**1、HashSet** 输出的顺序是不确定的，因为它不保持任何顺序。

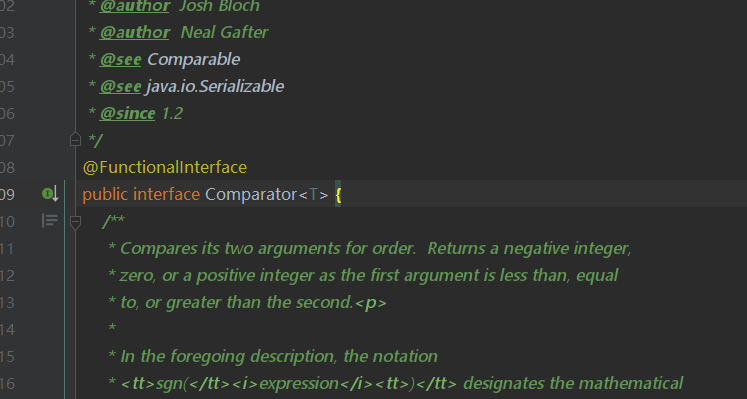
**2、TreeSet** 输出的顺序是自然排序的（按字母顺序），因为它会自动对元素进行排序。

# Comparator接口

## 基本概念

Comparator 接口在 Java 中用于自定义对象的排序逻辑。它允许你定义如何比较两个对象，从而控制排序的顺序。这与 Comparable 接口不同，后者只能为一个类提供自然排序顺序，而 Comparator 可以用于不同的排序标准或多个排序标准。

## 源码



## 方法

### int compare(T o1, T o2);

比较两个对象 o1 和 o2 的顺序。

## 示例

以下是一个使用 Comparator 接口自定义排序规则的示例代码：





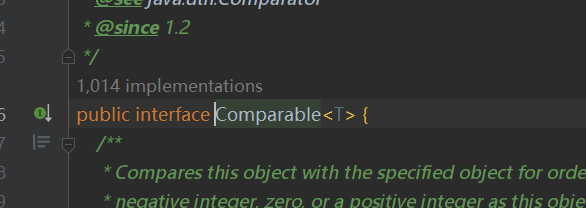
这种方式使得你能够轻松地根据不同的需求对对象进行排序，而无需改变对象本身的结构。

# Comparable接口

## 基本概念

Java提供了一个Comparable接口，该接口里定义了一个compareTo(Object obj)方法，该方法返回一个数据值，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，当一个对象调用该方法与另一个对象进行比较时，例如：obj.compareTo(obj)，如果该方法返回0，则表明这两个对象相等，如果该方法返回一个正整数，则表明obj1大于obj2，如果该方法返回一个负整数，则表明obj1小于obj2。java的一些常用类已经实现了Comparable接口，并提供了比较大小的标准。

## 源码



## 方法

### int compareTo(T o)

比较此对象与指定对象的大小。如果该方法返回0，则表明这两个对象相等，如果该方法返回一个正整数，则表明obj1大于obj2，如果该方法返回一个负整数，则表明obj1小于obj2。

只有这一个方法。

## 实现Comparable接口的常用类

（1）BigDecimal、BigInteger以及所有的数字型对应的包装类，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。

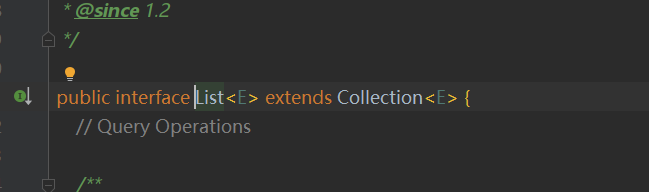
# Comparator接口和Comparable接口的区别

# List接口

## 基本概念

List接口代表一个有序、可重复的集合，集合中每个元素都有其对应的顺序索引，可以通过索引访问指定位置的集合元素，List集合默认按元素的添加顺序设置元素的索引。List集合的实现类：ArrayList、LinkedList、Vector。

## 源码



## 主要方法

### void add(int index,Object obj)

向指定位置添加元素

### boolean add(Object obj)

添加元素

### boolean addAll(Collection c)

添加指定 collection 中的所有元素到此列表的结尾

### boolean addAll(int index,Collection c)

将指定 collection 中的所有元素都插入到列表中的指定位置

### Object get(int index)

查找列表中指定位置的元素

### int indexOf(Object obj)

返回集合中第一次出现指定元素的索引，如果此列表不包含该元素，则返回-1。

### int lastindexOf(Object obj)

返回集合中最后出现指定元素的索引，如果集合中不包含此元素，则返回-1.

### ListIterator listIterator()

返回集合元素的迭代器

### E remove(int index);

参数：要删除元素的索引

返回：先前位于指定位置的元素。

### boolean remove(Object o);

### boolean removeAll(Collection<?> c);

用来移除List中包含在指定集合c中的所有元素。

### default boolean removeIf(Predicate<? super E> filter)

用于根据指定的条件（Predicate）删除符合条件的元素。这个方法是在Java 8中引入的，它接受一个Predicate对象作为参数，该Predicate对象描述了要删除的条件。对于List中的每个元素，都会应用Predicate的test方法进行判断，如果返回true，则该元素将被删除。

### Object set(int index,Object obj)

用指定元素替换列表中指定位置的元素

### List subList(int fromIndex,int toIndex)

返回列表中指定的 fromIndex（包括 ）和 toIndex（不包括）之间的部分列表。

说明：所有的List实现类都可以调用这些方法来操作集合元素，与Set集合元素相比，List集合增加了根据索引来插入、替换、删除集合元素的方法。List判断两个对象相等只要通过equal()方法比较返回true即可。与Set提供了一个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，提供了专门操作List的方法，ListIterator接口在Iterator接口的基础上，添加了如下方法：

（1）boolean hasPrevious()

（2）Object previous()

（3）void add()

### default void sort(Comparator<? super E> c)

在Java中，List接口中的default void sort(Comparator<? super E> c)方法是一个默认方法，允许在List接口实现中对列表进行排序。这个方法是Java 8引入的，目的是简化排序操作。

示例：

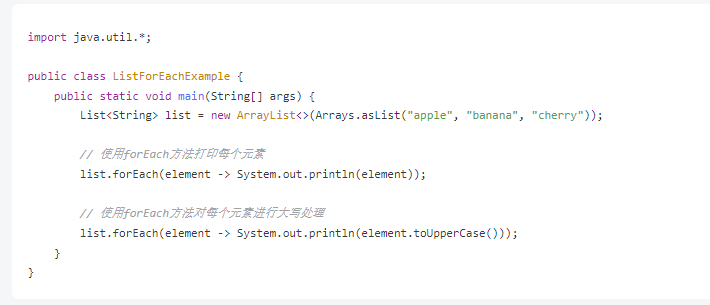


在这个例子中，list.sort((a, b) -> Integer.compare(a.length(), b.length()))按照字符串的长度对列表进行排序。

### default void forEach(Consumer<? super T> action)

forEach方法对List中的每个元素应用提供的Consumer操作。这个操作会对每个元素执行指定的处理逻辑。

代码示例：



在这个示例中，list.forEach(element -> System.out.println(element))打印了列表中的每个元素。list.forEach(element -> System.out.println(element.toUpperCase()))对每个元素进行大写转换并打印。

List接口中的default void forEach(Consumer<? super T> action)方法提供了一种简单的方式来对List中的每个元素执行操作。它利用了Java 8引入的流式编程风格，使得操作集合中的元素变得更直观和简洁。通过传递一个Consumer实例，你可以灵活地定义对每个元素的处理逻辑。

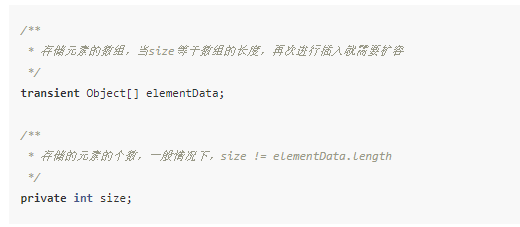
# List接口的实现类

ArrayList和Vector类都是基于数组实现的List实现类，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参数来设置数组的长度，当向ArrayList或Vector类中添加元素超出数组的长度时，它们的initialCapacity参数会自动增加。如果开始就知道ArrayList或Vector类中保存多少个元素，则可以在创建它们时就指定initialCapacity大小，如果创建空的ArrayList或Vector集合时不指定initialCapacity参数，则Object[]数组的长度默认为10。

# ArrayList类

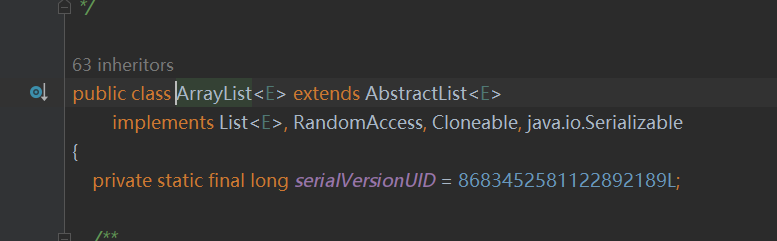
## 基本概念

ArrayList利用一个数组存储元素，它的实现很简单，基本属性只有两个。



ArrayList是有序可重复的。

## 源码



## 构造函数

### public ArrayList()

### public ArrayList(int initialCapacity)

### public ArrayList(Collection<? extends E> c)

## 方法

## 应用场景

### 1、****快速随机访问****

当需要频繁进行元素访问（按索引）且对插入和删除操作的性能要求不高时，ArrayList 是一个理想的选择。例如，处理需要频繁访问数据的缓存系统。

### **2、**数据存储****

ArrayList 可以作为简单的内存数据存储容器，适合用于保存一组元素并且可以方便地进行检索。例如，存储用户输入的数据列表或处理基本的数据存储需求。

### **3、**需要动态调整大小的数组****

当需要一个可扩展的数组以便在运行时添加或删除元素，而不需要提前定义数组大小时，ArrayList 提供了灵活的解决方案。例如，动态生成的列表项数量不确定时，如处理网络请求返回的结果列表。

### **4、**排序和查找****

ArrayList 提供了有效的排序和查找操作。例如，使用 Collections.sort() 方法对 ArrayList 进行排序，并且可以利用二分查找方法对排序后的列表进行快速查找。

### **5、**实现非线程安全的应用****

在单线程应用或对线程安全性要求不高的场景中，ArrayList 是一个高效且易于使用的选择。例如，处理内部数据存储、构建临时数据结构等。

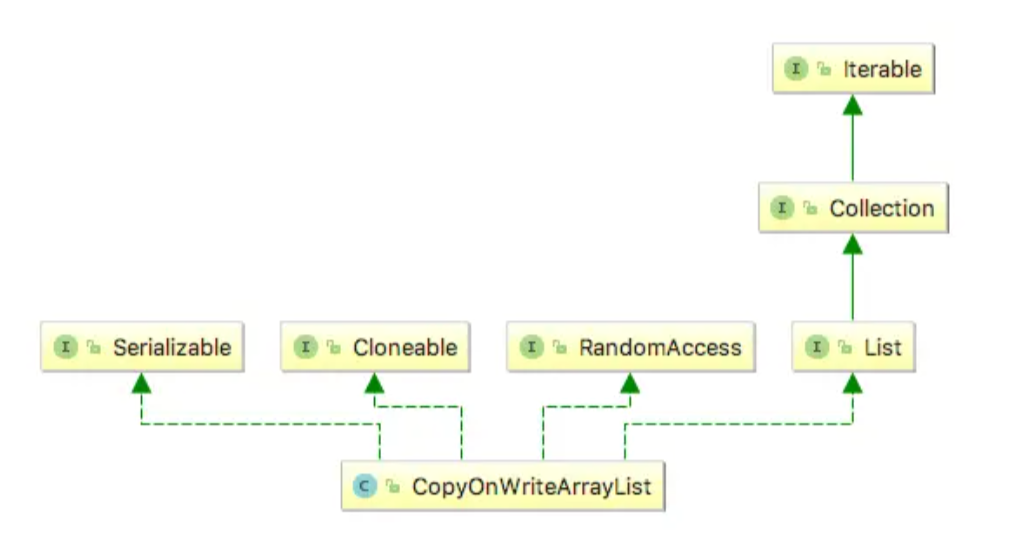
### **6、**与其他集合类的互操作****

ArrayList 可以方便地与其他集合类（如 HashSet, TreeSet 等）进行转换和交互。你可以通过构造函数或 addAll 方法将 ArrayList 作为参数传递给其他集合类进行操作。

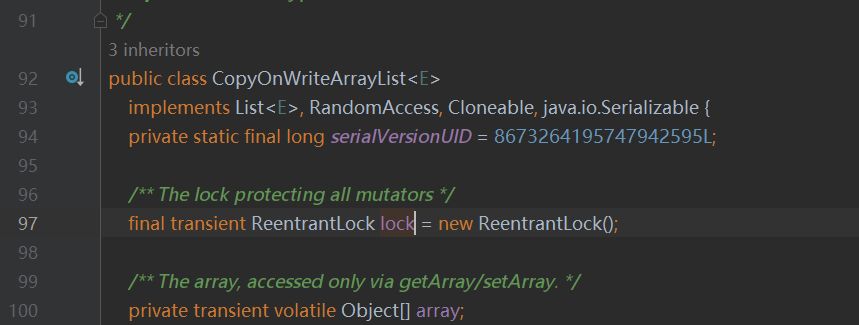
# CopyOnWriteArrayList

## 基本概念

CopyOnWriteArrayList是ArrayList的线程安全版本，从他的名字可以推测，CopyOnWriteArrayList是在有写操作的时候会copy一份数据，然后写完再设置成新的数据。CopyOnWriteArrayList适用于读多写少的并发场景。



## 源码



## 构造函数

### public CopyOnWriteArrayList()

### public CopyOnWriteArrayList(Collection<? extends E> c)

### public CopyOnWriteArrayList(E[] toCopyIn)

## 方法

和List接口相同

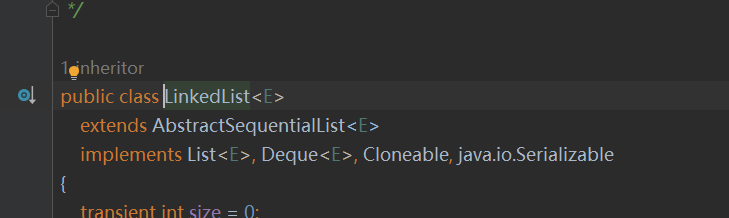
# LinkedList

## 基本概念

LinkedList是List接口的实现类，可以根据索引随机访问集合中的元素，LinkedList实现了Deque接口，因此它可以被当成双端队列来使用，也可以被当成栈来使用。LinkedList不是同步的，如果多个线程同时访问一个链接列表，而其中至少一个线程从结构上修改了该列表，则必须保持它同步。

LinkedList是有序可重复的。

## 源码



## 特有的方法

### boolean offer(E e)

将指定元素添加到此列表的末尾（最后一个元素）

### void push(E e)

将元素插入到列表的前面

### boolean offerFirst(E e)

在此列表的开头插入指定的元素

### boolean offerLast(E e)

在此列表末尾插入指定的元素

### E peek()

获取但不移除此列表的第一个元素

## 代码示例

LinkedList li = new LinkedList();

li.offer("a");//将字符串元素加入队列的尾部

li.push("b");//将字符串元素加入栈的顶部

li.offerFirst("c");//将字符串元素加入栈的顶部

System.out.println(li);//输出集合

System.out.println(li.peek());//输出集合中的栈的顶部

System.out.println(li.peekFirst());//输出集合中的栈的顶部

System.out.println(li.peekLast());//输出集合中的栈的底部

## 应用场景

### 1、****频繁插入和删除操作****

当你需要在列表的开头、中间或结尾进行频繁的插入和删除操作时，LinkedList 是一个合适的选择。例如，处理实现队列的任务调度系统，或实现一个待处理的任务列表。

### **2、**双端队列操作****

LinkedList 可以作为双端队列（deque），支持从两端进行插入和删除操作。这在需要处理双向队列的应用中非常有用，如实现浏览器的历史记录、任务调度系统等。

### ****3、栈和队列实现****

### **4、**实现复杂数据结构****

### ****5、避免内存碎片****

# ArrayList和Vector类区别

ArrayList类是线程不安全的，当多个线程访问同一个ArrayList时，如果有超过一个线程修改了ArrayList集合，则程序必须手动保证该集合的同步性，但是Vector集合无须程序保证该集合的同步性，因为Vector是线程安全的，所以Vector的性能比ArrayList的性能要低，实际上即使需要保证List集合线程安全，也同样不推荐使用Vector实现类，使用Collections工具类可以将一个ArrayList变成线程安全的。

# ArrayList与LinkedList的区别

LinkedList与ArrayList的实现机制不同，ArrayList内部以数组的形式保存集合中的元素，因此随机访问集合元素时有较好的性能，而LinkedList内部以链表的形式来保存集合中的元素，因此随机访问集合元素的性能较差，但在插入、删除元素时性能非常出色（只需改变指针所指的地址即可），需要指出的是，虽然Vector也是以数组的形式来存储集合元素，但是因为它实现了线程同步的功能，所以各方面性能有所下降。对于所有内部基于数组的集合实现，例如ArrayList，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。

# 各种线性表的性能分析

Java提供的List是一个线性表接口，而ArrayList、LinkedList又是线性表的两种典型实现：基于数组的线性表和基于链的线性表，Queue代表了队列，Deque代表了双端队列（既可以作为队列使用，也可以作为栈来使用）。

数组以一块连续内存区来保存所有的数组元素，所以数组在随机访问时性能最好，所有的内部以数组作为底层实现的集合在随机访问时性能较好。而内部以链表作为底层实现的集合在执行插入、删除操作时有很好的性能；进行迭代操作时，以链表作为底层实现的集合比以数组作为底层实现的集合性能好。

# 关于使用List集合的建议

如果需要遍历List集合元素，对于ArrayList、Vector集合，应该使用随机访问方法（get）来遍历集合元素，这样性能更好。对于LinkedList集合，则应该采用迭代器（Iterator）来遍历集合元素。

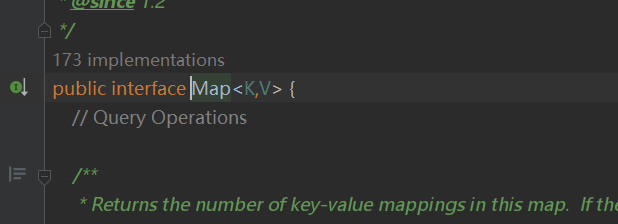
如果需要经常执行插入、删除操作来改变List集合的大小，则应该采用LinkedList集合，而不是ArrayList。使用ArrayList、Vector集合需要经常重新分配内部数组的大小，其时间开销常常是使用LinkedList的时间开销的几十倍，效果很差。如果有多个线程需要同时访问List集合中的元素，开发者可考虑使用Collections将集合包装成线程安全的集合。

# Map接口

## 基本概念

用于保存具有映射关系的数据，因此Map集合里保存着两组值，一组值用于保存Map里的key，另一组值用于保存Map里的value，key和value都可以是任何引用类型的数据。Map的key不允许重复，即同一个Map对象的任何两个key通过equal方法比较总是返回false。key和value之间存在单向的一对一关系，即通过指定的key，总能找到唯一的、确定的value。从Map中取出数据时，只要给出指定的key，就可以取出对应的value。

## 源码



## 常用方法

### void clear()

移除Map中的所有元素

### boolean containsKey(Object key)

包含指定的键，返回true

### boolean containsValue(Object value)

包含指定的值，返回true

### Set<Map.Entry<K,V>> entrySet()

返回此映射中包含的映射关系的set集合，这个方法用于循环遍历Map集合中的数据。

### Object get(Object key)

返回指定键的值

### boolean isEmpty()

Map集合为空，返回true

### Set<K> keySet()

返回映射中包含键的Set集合

### Object put(Object key,Object value)

添加数据

### void putAll(Map m)

添加Map集合

### Object remove(Object key)

删除指定key的数据

### default boolean remove(Object key, Object value)

### int size()

返回集合的大小

### Collection values()

返回Map集合中包含值的 [Collection](mk:@MSITStore:C:\Users\jinshihui\Desktop\我的编程知识总结\Java\JDK_API_1_6_zh_CN.CHM::/java/util/Collection.html)对象

### default V replace(K key, V value)

这个方法的作用是在 Map 中用新的值替换指定键的旧值。如果指定键存在于 Map 中，那么它会用新值替换掉原来的值，并返回旧值；如果指定键不存在，则返回 null。

参数：

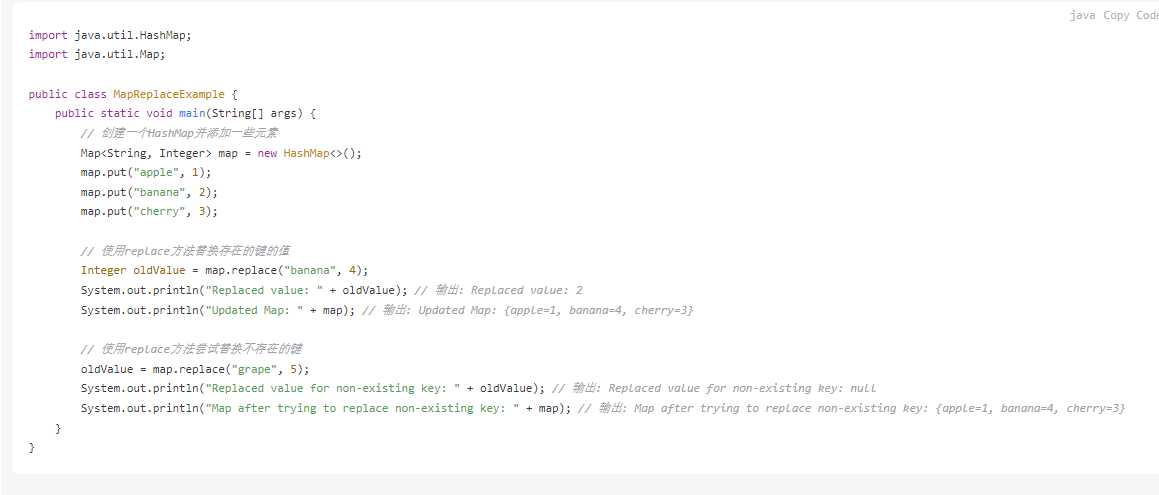
K key：要替换的键。

V value：用于替换的值。

返回值：

如果 Map 中存在指定的 key，方法将返回键对应的旧值，并将键对应的值更新为新值。如果 Map 中不存在指定的 key，方法将返回 null，并且 Map 中的内容不会发生变化。

代码示例：



### default boolean replace(K key, V oldValue, V newValue)

它提供了一种原子操作来替换 Map 中某个键的值。这种方法允许你在替换值之前检查当前值是否匹配某个指定的旧值。如果当前值匹配，那么它将用新值替换旧值，并返回 true；如果当前值不匹配，则不进行任何替换操作，并返回 false。

参数：

K key：要替换值的键。

V oldValue：期望当前键对应的旧值。

V newValue：要替换成的新值。

返回值：

如果 Map 中存在指定的 key 且对应的值与 oldValue 相匹配，则用 newValue 替换旧值，并返回 true。如果 Map 中存在指定的 key 但对应的值与 oldValue 不匹配，或者 key 不存在，则不进行任何操作，返回 false。

#### 关键点

##### 1、条件替换

这个方法不仅用于替换值，还用于确保只有在当前值与指定的旧值匹配时才进行替换操作。这种条件检查使得操作变得原子性和安全，特别是在并发环境中，避免了不必要的覆盖。

##### 2、原子操作

方法的实现确保了替换操作的原子性，即在检查和替换值的过程中，Map 不会被其他线程修改，从而避免了竞争条件。

##### 3、返回值的意义

true 表示替换成功，即当前值匹配了 oldValue，并且已被替换为 newValue。

false 表示替换失败，即当前值不匹配 oldValue 或者 key 不存在，导致没有进行任何替换操作。

### default void replaceAll(BiFunction<? super K, ? super V, ? extends V> function)

### default V getOrDefault(Object key, V defaultValue)

用于根据指定的键获取对应的值。如果键不存在于 Map 中，则返回指定的默认值 defaultValue。

key：要查找的键。

defaultValue：如果键不存在时返回的默认值。

返回值：

如果 Map 中包含指定的键，则返回该键对应的值。

如果 Map 中不包含该键，则返回 defaultValue

#### 示例代码



#### 使用场景

1、当你希望从 Map 中获取值，但又不想在键不存在时抛出异常或进行额外的空值检查时，可以使用 getOrDefault 方法。

2、这种方法特别适用于处理可能缺失的键值对，避免手动检查键是否存在。

#### 注意事项

1、defaultValue 可以是任意类型的值，包括 null。

2、如果 Map 中允许存储 null 值，且键对应的值为 null，getOrDefault 方法也会返回 null，而不是 defaultValue。

#### 总结

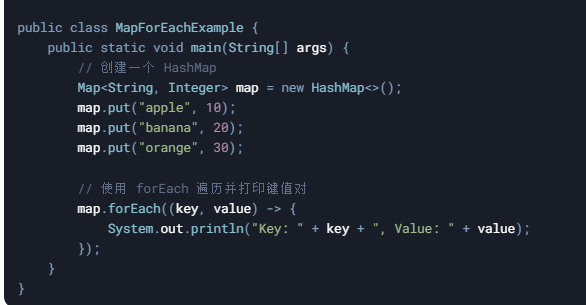
getOrDefault 方法提供了一种简洁的方式来处理 Map 中可能缺失的键，避免了繁琐的空值检查和异常处理，使代码更加简洁和易读。

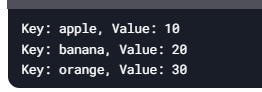
### default void forEach(BiConsumer<? super K, ? super V> action)

用于遍历 Map 中的每一个键值对，并对每个键值对执行指定的操作。它提供了一种简洁的方式来遍历 Map，而不需要显式地使用迭代器或 entrySet()。

#### 代码示例

##### 1、遍历 Map 并打印键值对





##### 2、对 Map 中的值进行操作





#### 使用场景

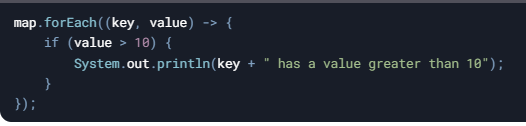
##### 1、****遍历并打印****Map****内容****



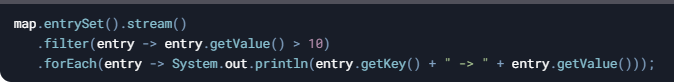
##### **2、**对**Map**中的值进行批量操作****



##### **3、**过滤或处理特定键值对****



##### **4、**与 Stream API 结合使用****



#### 注意事项

##### 1、并发修改

（1）在 forEach 中修改 Map 可能导致 ConcurrentModificationException。

（2）如果需要修改，建议使用迭代器或 ConcurrentHashMap。

##### 2、空值处理

如果 Map 中包含 null 键或值，forEach 仍然会正常遍历。

##### 3、性能

对于非常大的 Map，forEach 的性能与传统的 for-each 循环相当，但代码更简洁。

### default V putIfAbsent(K key, V value)

用于在指定的键不存在时将其与给定的值关联。如果键已经存在，则不会修改 Map 中的值，并返回当前与键关联的值。

key：要插入的键

value：要与键关联的值（仅在键不存在时插入）

返回值：

如果键已经存在于 Map 中，则返回当前与键关联的值。

如果键不存在，则将键值对插入 Map，并返回 null。

#### 示例代码



#### 使用场景

1、当你希望在键不存在时才插入键值对时，可以使用 putIfAbsent 方法。

2、这种方法特别适用于避免覆盖已存在的值，或者在并发环境中确保原子性操作。

#### 注意事项

1、如果 Map 允许存储 null 值，且键对应的值为 null，putIfAbsent 方法仍然会返回 null，而不是插入新值。

2、在并发环境中，putIfAbsent 方法的行为取决于 Map 的实现。例如，ConcurrentHashMap 提供了线程安全的原子操作。

#### 与 put 方法的区别

1、put 方法会直接插入或覆盖键值对，无论键是否已存在。

2、putIfAbsent 方法仅在键不存在时插入键值对，否则不进行任何操作

### default V compute(K key,BiFunction<? super K, ? super V, ? extends V> remappingFunction)

用于根据指定的键和重新映射函数来计算新的值。它提供了一种灵活的方式来更新 Map 中的键值对，特别适合需要根据当前值动态计算新值的场景。

参数：

key：要操作的键。

remappingFunction：一个 BiFunction 函数式接口，用于根据键和当前值计算新值。

返回值：

返回与键关联的新值。如果新值为 null，则删除该键值对。

#### 示例代码

##### 1、更新 Map 中的值



##### 2、插入新的键值对



##### 3、删除键值对



#### 注意事项

##### 1、空值处理

如果 remappingFunction 返回 null，则删除该键值对。

如果 Map 允许存储 null 值，且键对应的值为 null，remappingFunction 仍然会接收 null 作为参数。

##### 2、并发修改

在并发环境中，compute 方法的行为取决于 Map 的实现。例如，ConcurrentHashMap 提供了线程安全的原子操作。

##### 3、性能

对于频繁更新的场景，compute 方法比传统的 get 和 put 组合更高效，因为它避免了多次查找。

### default V computeIfAbsent(K key,Function<? super K, ? extends V> mappingFunction)

参数：

key：要查找或计算的键。

mappingFunction：一个函数，用于在键不存在时计算值。

返回值：

如果键存在且值不为 null，则返回当前值。

如果键不存在或值为 null，则返回通过 mappingFunction 计算的新值，并将其放入 Map 中。

#### 作用

##### 1、检查键是否存在

如果 key 在 Map 中不存在，或者对应的值为 null，则调用 mappingFunction 计算新值。

##### 2、放入新值

将计算得到的新值放入 Map 中，并返回该值。

##### 3、避免重复计算

如果键已经存在且值不为 null，则直接返回当前值，不会调用 mappingFunction。

#### 使用场景

##### 1、延迟初始化

在需要时动态计算并缓存值。

##### 2、缓存机制

避免重复计算相同的值。

##### 3、简化代码

减少手动检查键是否存在和放入值的代码。

#### 示例代码

##### 1、缓存计算结果



##### 2、避免重复计算



#### 方法的行为

1、键存在且值不为 null：直接返回当前值，不会调用 mappingFunction。

2、键不存在或值为 null：调用 mappingFunction 计算新值，并将其放入 Map 中。

3、mappingFunction 返回 null：如果 mappingFunction 返回 null，则不会将 null 放入 Map 中。

#### 注意事项

1、线程安全：computeIfAbsent 不是线程安全的。如果多个线程同时调用此方法，可能会导致竞态条件。

2、mappingFunction 的副作用：mappingFunction 不应修改 Map，否则可能导致不可预测的行为。

3、性能：mappingFunction 的计算可能会影响性能，尤其是在高并发场景下。

### default V computeIfPresent(K key,BiFunction<? super K, ? super V, ? extends V> remappingFunction)

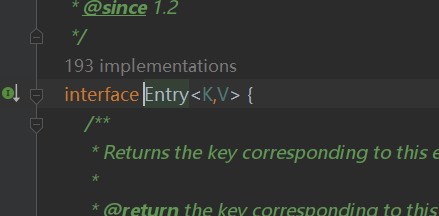
### default V merge(K key, V value,BiFunction<? super V, ? super V, ? extends V> remappingFunction)

# Map.Entry

## 基本概念

Map中有一个内部类Entry，该类封装了一个key-value对。

## 源码



## 方法

### （1）Object getKey()

返回此项对应的键

### （2）Object getValue()

返回此项对应的值

### （3）Object setValue(V value)

设置值

# Map接口的实现类

## 基本概念

HashMap、Hashtable、LinkedHashMap、TreeMap

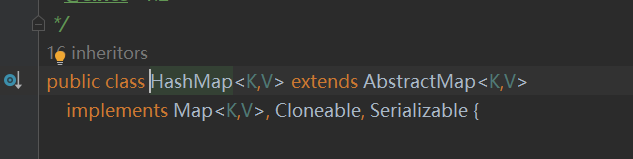
HashMap和Hashtable是Map接口的典型实现类，它们之间的关系完全类似于ArrayList和Vector的关系，HashMap和Hashtable存在两点典型区别：

（1）Hashtable是一个线性安全的Map实现，HashMap是线程不安全的实现，所以HashMap比Hashtable的性能高一点，但如果有多个线程访问同一个Map对象，使用Hashtable实现类会更好。

（2）Hashtable不允许使用null作为key和value，如果试图把null值放进Hashtable中，将会引发空指针异常，但是HashMap可以使用null作为key和value。由于HashMap里的key不能重复，所以HashMap里最多只有一个key为null，但是可以有多个value为null。

## HashMap

### 源码



### 重要变量

1、DEFAULT\_INITIAL\_CAPACITY：Table数组的初始化长度：2^4=16

2、MAXIMUM\_CAPACITY： Table数组的最大长度： 1<<30 2^30=1073741824

3、DEFAULT\_LOAD\_FACTOR 负载因子：默认值为0.75。 当元素的总个数>当前数组的长度 \* 负载因子（16\*0.75），数组会进行扩容，扩容为原来的两倍

4、TREEIFY\_THRESHOLD 链表树化阙值：默认值为 8 ，表示在一个node（Table）节点下的值的个数大于8时，会将链表转换成为红黑树。

5、UNTREEIFY\_THRESHOLD 红黑树链化阙值： 默认值为 6 。 表示在进行扩容期间，单个Node节点下的红黑树节点的个数小于6时候，会将红黑树转化成为链表。

6、MIN\_TREEIFY\_CAPACITY = 64 最小树化阈值，当Table所有元素超过改值，才会进行树化（为了防止前期阶段频繁扩容和树化过程冲突）。

### Hash的基本概念

Hash叫做”散列表“，就是把任意长度的输入，通过散列算法变成固定长度输出，该输出结果是散列值。其实这种转换是一种压缩映射，散列表的空间通常小于输入的空间，不同的输入可能会散列成相同的输出，所以不能从散列表来唯一的确定输入值，这就出现了Hash冲突。

### Hash冲突

根据key（键）经过一个函数f(key)得到的结果作为地址去存放当前key value键值对(这个是hashmap的存值方式)，但是却发现算出来的地址上已经被占用了，这就是所谓的hash冲突。

### 解决hash冲突的方法

#### 1、开放定址法

该方法也叫做再散列法，其基本原理是：当关键字key的哈希地址p=H（key）出现冲突时，以p为基础，产生另一个哈希地址p1，如果p1仍然冲突，再以p为基础，产生另一个哈希地址p2，…，直到找出一个不冲突的哈希地址pi 。

#### 2、再Hash法

这种方法就是同时构造多个不同的哈希函数： Hi=RH1（key）  i=1，2，…，k。当哈希地址Hi=RH1（key）发生冲突时，再计算Hi=RH2（key）……，直到冲突不再产生。这种方法不易产生聚集，但增加了计算时间。

#### 3、链地址法（也就是拉链法，Java就是采用这种方法）

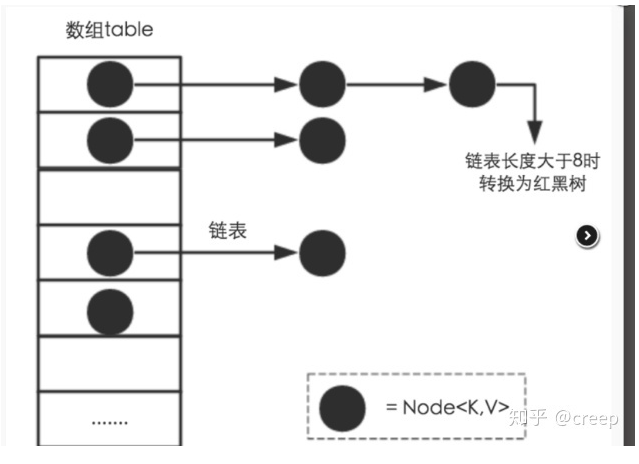
其基本思想: 将所有哈希地址为i的元素构成一个称为同义词链的单链表，并将单链表的头指针存在哈希表的第i个单元中，因而查找、插入和删除主要在同义词链中进行。链地址法适用于经常进行插入和删除的情况。

#### 4、建立公共溢出区

这种方法的基本思想是：将哈希表分为基本表和溢出表两部分，凡是和基本表发生冲突的元素，一律填入溢出表。

### 实现原理

我们都知道，在HashMap中，采用数组+链表的方式来实现对数据的储存。



HashMap采⽤Entry数组来存储key-value对，每⼀个键值对组成了⼀个Entry实体，Entry类实际上是⼀个单向的链表结构，它具有Next指针，可以连接下⼀个Entry实体。只是在JDK1.8中，链表⻓度⼤于8的时候，链表会转成红⿊树。

### 数据结构

JDK7 是数组+链表

在JDK 1.8中，HashMap的底层数据结构是“数组+链表+红黑树”，即在链表的长度超过阈值8时转化为红黑树结构，这样大大减少了查找时间。

### 为什么使用数组+链表

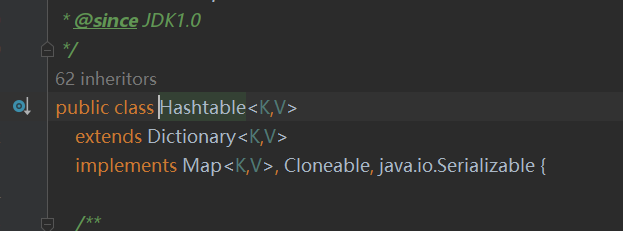
由于数组的值是限制死的，我们在对key值进行散列取到下标以后，放入到数组中时，难免出现两个key值不同，但是却放入到下标相同的格子中，此时我们就可以使用链表来对其进行链式的存放。

### 我⽤LinkedList代替数组结构可以吗

可以，为什么有偏偏使用到数组呢，因为⽤数组效率最⾼。在HashMap中，定位节点的位置是利⽤元素的key的哈希值对数组⻓度取模得到。此时，我们已得到节点的位置。显然数组的查 找效率⽐LinkedList⼤（底层是链表结构）。那ArrayList，底层也是数组，查找也快啊，为啥不⽤ArrayList? 因为采⽤基本数组结构，扩容机制可以⾃⼰定义，HashMap中数组扩容刚好是2的次幂，在做取模运算的效率⾼。 ⽽ArrayList的扩容机制是1.5倍扩容（这一点我相信学习过的都应该清楚）

## HashTable

### 源码



### 为什么HashTable慢

Hashtable之所以效率低下主要是因为其实现使用了synchronized关键字对put等操作进行加锁，而synchronized关键字加锁是对整个对象进行加锁，也就是说在进行put等修改Hash表的操作时，锁住了整个Hash表，从而使得其表现的效率低下。

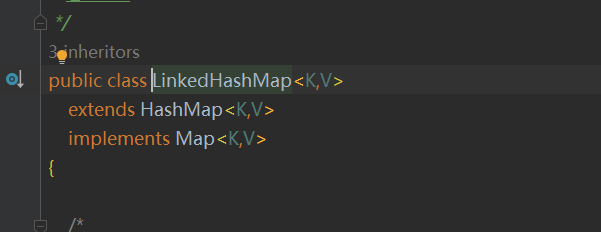
## LinkedHashMap

### 基本概念

LinkedHashMap是HashMap的子类，LinkedHashMap不是线程安全的，LinkedHashMap使用双向链表来维护key-value对的次序，该链表负责维护Map的迭代顺序，迭代顺序与key-value对的插入顺序保持一致。

HashMap存储元素的顺序是无序的，LinkedHashMap可以避免对HashMap、Hashtable里的key--value对进行排序（只要插入key-value对时保持顺序即可），同时又可避免使用TreeMap所增加的成本。LinkedHashMap需要维护元素的插入顺序，因此性能略低于HahMap的性能，但因为它以链表来维护内部顺序，所以迭代访问Map里的全部元素时将有较好的性能。

### 源码



### 代码示例

LinkedHashMap lhm = new LinkedHashMap();

lhm.put("语文", 80);

lhm.put("数学", 90);

lhm.put("英语", 80);

System.out.println(lhm);

for(Object str:lhm.keySet()){//使用增强for循环遍历Map集合

System.out.println(str+""+lhm.get(str));

}

## HashMap与LinkedHashMap的区别

### 1、存储顺序

HashMap：

HashMap 不保证元素的顺序。它根据键的 hashCode() 进行存储，因此元素的顺序可能会随时间变化。即使是相同的键值对，遍历 HashMap 时的顺序也不一定是插入顺序。

LinkedHashMap：

LinkedHashMap 维护了一个链表来保证元素的顺序。这个链表可以是插入顺序（默认）或者访问顺序（如果在构造时指定了）。在插入或访问元素时，它会根据链表的顺序来保证元素的遍历顺序。

### 2、性能

HashMap：

插入、删除和查找操作的平均时间复杂度是 O(1)。由于 HashMap 只是基于哈希表的实现，所有操作都是通过哈希值来直接访问桶的位置，因此性能非常快。

LinkedHashMap：

插入、删除和查找操作的平均时间复杂度也是 O(1)，但由于维护了链表，因此可能会略微慢一些。链表的维护会带来一定的额外开销。在遍历时，LinkedHashMap 比 HashMap 更快，因为它可以按插入顺序或访问顺序直接遍历链表。

### 3、内存使用

HashMap：

内存使用相对较少，因为它只需要存储哈希表的桶和链表节点。由于不维护额外的链表，HashMap 的内存开销相对较低。

LinkedHashMap：

内存使用相对较高，因为它需要额外存储链表节点的引用。每个节点不仅包含键值对，还需要指向前后节点的引用。

### 4、特殊功能

HashMap：

HashMap 不提供按插入顺序或访问顺序遍历的功能。它主要用于快速存储和检索数据。

LinkedHashMap：

LinkedHashMap 提供了按插入顺序或访问顺序遍历的功能。这在需要维护元素顺序的情况下非常有用，例如实现缓存机制时可以使用访问顺序来移除最少使用的元素。

LinkedHashMap 具有 accessOrder 参数，允许构造函数根据访问顺序来维护元素顺序。这可以用于实现 LRU（Least Recently Used）缓存策略。

### 5、默认构造函数

HashMap：

默认构造函数会创建一个空的 HashMap，初始容量是 16，负载因子是 0.75。

LinkedHashMap：

默认构造函数也会创建一个空的 LinkedHashMap，初始容量是 16，负载因子是 0.75。可以通过构造函数来指定插入顺序或访问顺序。

### 示例代码



### 总结

1、HashMap：不保证顺序，性能较高，适用于不需要顺序维护的场景。

2、LinkedHashMap：保证顺序（插入顺序或访问顺序），性能稍低，但提供了顺序维护的功能，适用于需要元素顺序的场景。

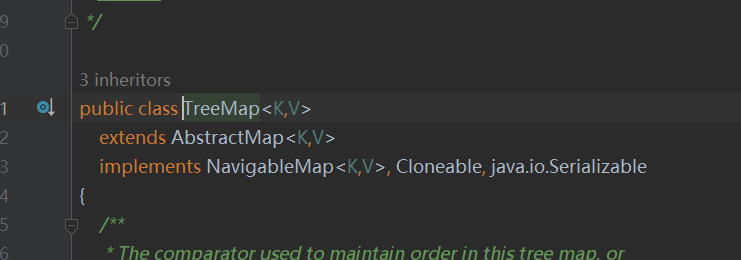
## TreeMap

### 基本概念

SortedMap接口有一个TreeMap实现类，TreeMap就是一个红黑树数据结构，每个key-value对作为红黑树的一个节点，TreeMap存储key-value对时，需要根据key-value对节点进行排序，TreeMap可以保证所有的key-value对处于有序状态。TreeMap默认按键的值升序排序，可以定制。

TreeMap中判断两个key相等的标准是两个key通过compareTo()方法返回0，TreeMap即认为这两个key是相等的。再次强调：Set和Map的关系十分密切，Java源码就是先实现了HashMap、TreeMap等集合，然后通过包装一个所有的value都为null的Map集合实现了Set集合类。与TreeSet类似，TreeMap中提供了一系列根据key顺序访问key-value对的方法。

### 源码



### 应用场景

#### 1、****有序数据存储和检索****

#### ****2、范围查询操作****

TreeMap 提供了一些方法，如 subMap(), headMap(), 和 tailMap()，用于执行范围查询，这对于需要按一定范围检索数据的应用特别有用。

（1）日期范围查询：在时间序列数据中，查找某个时间段内的所有记录。

（2）价格范围查询：在价格排序的商品列表中，查找某个价格区间内的所有商品。

示例代码：



#### ****3、快速查找和顺序操作****

#### ****4、自定义排序****

#### ****5、避免重复元素****

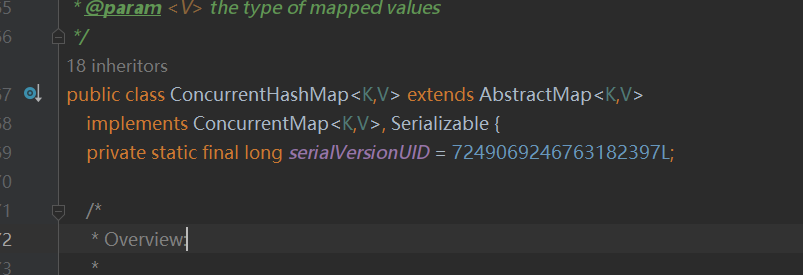
## ConcurrentHashMap

### 基本概念

在JDK1.5~1.7版本，Java使用了分段锁机制实现ConcurrentHashMap。简而言之ConcurrentHashMap在对象中保存了一个Segment数组，即将整个Hash表划分为多个分段；而每个Segment元素，即每个分段则类似于一个Hashtable；这样在执行put操作时首先根据hash算法定位到元素属于哪个Segment，然后对该Segment加锁即可。因此ConcurrentHashMap在多线程并发编程中实现了多线程put操作。

JDK1.8则使用数组+链表+红黑树数据结构和CAS原子操作实现ConcurrentHashMap。

### 源码



### 为什么要用ConcurrentHashMap

#### 1、线程不安全的HashMap

在多线程环境下，使用HashMap的put操作会引起死循环，原因是多线程会导致HashMap的Entry链表形成环形数据结构，导致Entry的next节点永远不为空，就会产生死循环获取Entry。

#### 2、效率低下的HashTable

HashTable容器使用sychronized来保证线程安全，采取锁住整个表结构来达到同步目的，在线程竞争激烈的情况下，当一个线程访问HashTable的同步方法，其他线程也访问同步方法时，会进入阻塞或轮询状态；如线程1使用put方法时，其他线程既不能使用put方法，也不能使用get方法，效率非常低下。

#### 3、ConcurrentHashMap的锁分段技术可提升并发访问效率

首先将数据分成一段一段地存储，然后给每一段数据配一把锁，当一个线程占用锁访问其中一个段数据的时候，其他段的数据也能被其他线程访问。

### ConcurrentHashMap - JDK 1.7

#### 基本说明

在JDK1.5~1.7版本，Java使用了分段锁机制实现ConcurrentHashMap.

简而言之，ConcurrentHashMap在对象中保存了一个Segment数组，即将整个Hash表划分为多个分段；而每个Segment元素，即每个分段则类似于一个Hashtable；这样，在执行put操作时首先根据hash算法定位到元素属于哪个Segment，然后对该Segment加锁即可。因此，ConcurrentHashMap在多线程并发编程中可是实现多线程put操作。接下来分析JDK1.7版本中ConcurrentHashMap的实现原理。

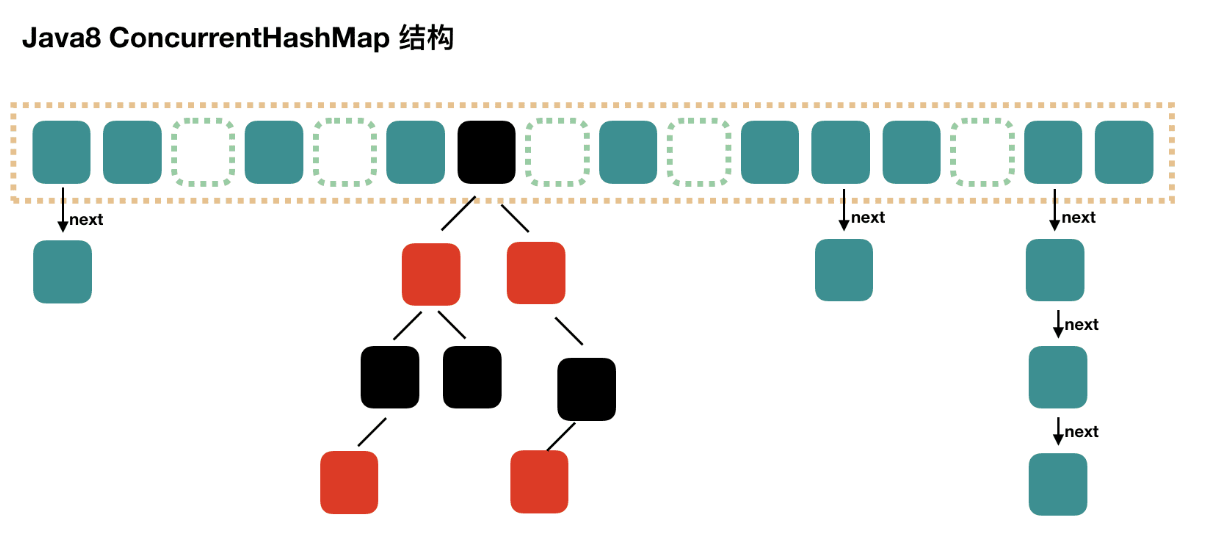
#### 数据结构

### ConcurrentHashMap - JDK 1.8

#### 基本说明

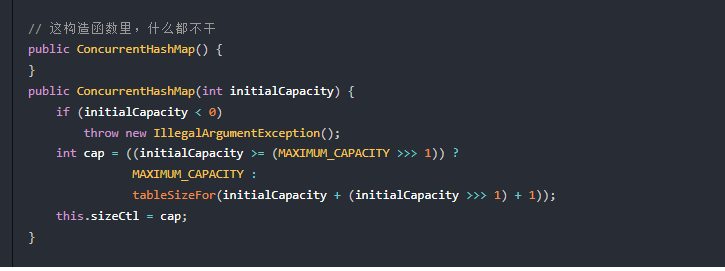
在JDK1.7之前，ConcurrentHashMap是通过分段锁机制来实现的，所以其最大并发度受Segment的个数限制。因此，在JDK1.8中，ConcurrentHashMap的实现原理摒弃了这种设计，而是选择了与HashMap类似的数组+链表+红黑树的方式实现，而加锁则采用CAS和synchronized实现。

#### 数据结构



结构上和 Java8 的 HashMap 基本上一样，不过它要保证线程安全性，所以在源码上确实要复杂一些。

#### 初始化



这个初始化方法有点意思，通过提供初始容量，计算了 sizeCtl，sizeCtl = 【 (1.5 \* initialCapacity + 1)，然后向上取最近的 2 的 n 次方】。如 initialCapacity 为 10，那么得到 sizeCtl 为 16，如果 initialCapacity 为 11，得到 sizeCtl 为 32。

### ConcurrentHashMap实现高并发的总结

#### 读操作的高效率

在实际的应用中，散列表一般的应用场景是：除了少数插入操作和删除操作外，绝大多数都是读取操作，而且读操作在大多数时候都是成功的。正是基于这个前提，ConcurrentHashMap针对读操作做了大量的优化。通过**HashEntry对象的不变性**和用**volatile型变量协调线程间的内存可见性**，使得大多数时候，**读操作不需要加锁**就可以正确获得值。

#### 比HashTable和HashMap拥有更高并发性

相比于HashTable和用同步包装器包装的HashMap



ConcurrentHashMap拥有更高的并发性。在HashTable和由同步包装器包装的HashMap中，使用一个**全局的锁**来同步不同线程间的并发访问。同一时间点，只能有一个线程持有锁，也就是说在同一时间点，只能有一个线程能访问容器。这虽然保证多线程间的安全并发访问，但同时也导致**对容器的访问变成串行化**的了。

#### ConcurrentHashMap的高并发性主要来自于三个方面

1、用分离锁实现多个线程间的更深层次的共享访问。

2、用HashEntry对象的不变性来降低执行读操作的线程在遍历链表期间对加锁的需求。

3、通过对同一个volatile变量的写/读访问，协调不同线程间读/写操作的内存可见性。

# 各Map实现类的性能分析

对于Map的常用实现类而言，HashMap和Hashtable的效率大致相同，因为它们的实现机制几乎完全一样，但是HashMap通常比Hashtable要快一点，因为Hashtable需要额外的线程同步控制。TreeMap通常比HashMap、hashtable要慢（尤其在插入、删除key-value对时更慢），因为TreeMap底层采用红黑树来管理key-value对（红黑树的每个节点就是一个key-value对）。LinkedHashMap比HashMap慢一点，因为它需要维护链表来保持Map中key-value对的添加顺序。

对于一般的应用场景，程序应该多考虑使用HashMap，因为HashMap正是为快速查询设计的（HashMap底层其实也是采用数组来存储key-value对）。但如果程序需要一个总是排好序的Map时则可以考虑使用TreeMap。

# Map集合中判断key和value相等的标准

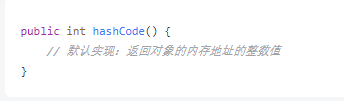
为了成功的在HashMap、Hashtable中存储、获取对象，用作key的对象必须实现hashCode()方法和equal()方法。HashMap、Hashtable判断两个key相等的标准是：两个key通过equals()方法比较返回true，两个key的hashCode值也相等。HashMap、Hashtable判断两个value相等的标准更简单，只要两个对象通过equals()方法比较返回true即可。

# hashCode与equals方法之间的关系

## hashCode() 方法

**定义**：hashCode() 方法返回一个对象的哈希码（一个整数），用于在哈希表中对对象进行快速检索。

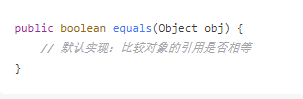
作用：哈希码用于确定对象在哈希表中的位置。哈希码的好坏直接影响哈希表的性能。



## equals() 方法

定义：equals(Object obj) 方法用于比较当前对象与另一个对象是否相等。

作用：用于检查两个对象是否具有相同的值或语义上的等价性。



## hashCode() 和 equals() 之间的关系

（1）如果两个对象使用 equals() 方法比较相等，那么它们的 hashCode() 方法必须返回相同的整数值。

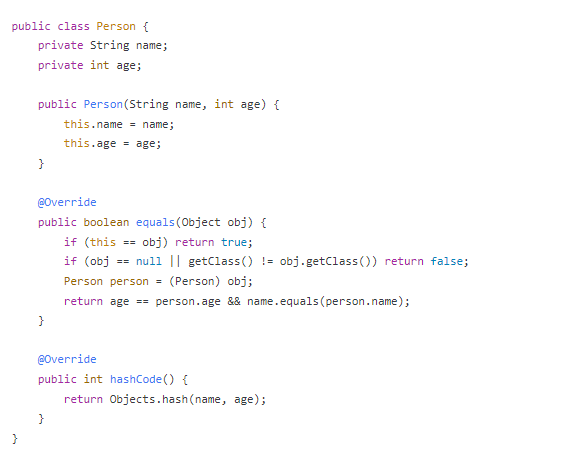
解释：在哈希表中，如果两个对象是相等的（即 obj1.equals(obj2) 返回 true），则它们的哈希码必须相同。这是为了确保哈希表中的键值对能够正确匹配和检索。

（2）如果两个对象的 hashCode() 方法返回相同的值，它们不一定需要通过 equals() 方法比较相等。

解释：哈希码相同的对象不一定是相等的，这意味着不同的对象可能会具有相同的哈希码（哈希冲突）。这不会影响哈希表的工作，只会导致哈希冲突，需要额外的处理。

### 代码示例

为了遵守这个契约，如果你重写了 equals() 方法，你通常也需要重写 hashCode() 方法。以下是一个示例，展示如何正确实现这两个方法。



在这个示例中：

equals() 方法比较两个 Person 对象是否具有相同的 name 和 age。

hashCode() 方法根据 name 和 age 计算哈希码，确保了相同的 name 和 age 的对象具有相同的哈希码。

两个对象 equals的时候，hashCode必须相等，但hashCode相等，对象不一定equals。

## 总结

1、hashCode() 方法用于确定对象在哈希表中的位置，确保哈希表的高效检索。

2、equals() 方法用于比较对象的等价性，确定两个对象是否相等。

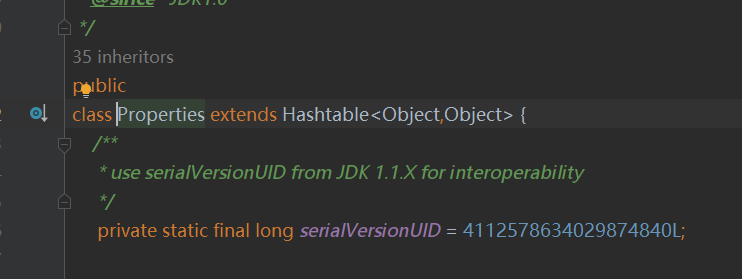
3、它们之间的契约确保了在使用基于哈希的集合时的一致性和正确性。如果两个对象相等，它们的 hashCode() 必须相同。如果 hashCode() 相同，它们不一定相等，但在相等的情况下，hashCode() 必须相同。

# 使用Properties类读写属性文件

## 基本概念

Properties类是Hashtable类的子类，该对象在处理属性文件时特别方便，Properties类可以把Map对象和属性文件关联起来，从而可以把Map对象中的key-value对写入属性文件中，也可以把属性文件中的“属性名=属性值”加载到Map对象中，由于属性文件里的属性名、属性值只能是字符串类型，所以Properties里的key、value都是字符串类型，该类提供了如下三个方法来修改Properties里的key、value值。

## 源码



## 方法

### String getProperty(String key)

用指定的键在此属性列表中搜索属性

### String getProperty(String key,String defaultValue)

用指定的键在属性列表中搜索属性

### Object setProperty(String key,String value)

设置键和值

### void load(InputStream inStream)

从属性文件中读取key-value对，把加载到的key-value对追加到Properties类里。（Properties是hashtable的子类，它不保证key-value对之间的顺序）

### void store(OutputStream out,String comments)

将Properties类中的key-value对写入到指定的属性文件（以输出流表示）中。

### void storeXML(OutputStream out,String comments)

将Properties中的key-value对写入到指定的XML文件（以输出流表示）中。

### loadFromXML(InputStream inStream)

从XML文件中读取key-value对，把加载到的key-value对追加到Properties里。

## 代码示例

把Properties中的键值对输出到属性文件中

Properties pro = new Properties();

pro.put("语文", "90");

pro.put("数学", "80");

System.out.println(pro);

pro.setProperty("英语", "90");

System.out.println(pro);

//把pro对象存放到属性文件中

pro.store(new FileOutputStream("src\\userinfo.properties"), "note");

Properties pro1 = new Properties();

pro1.setProperty("历史", "10");

//从属性文件中读取键值对并加载到properties集合中

pro1.load(new FileInputStream("src\\userinfo.properties"));

System.out.println(pro1);

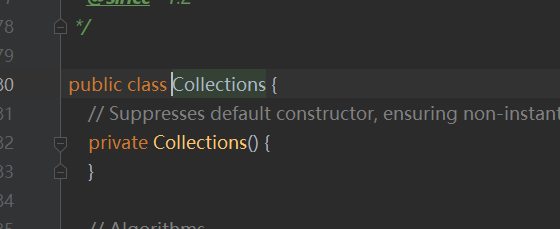
说明：Properties可以把key-value对以XML文件的形式保存起来，也可以从XML文件中加载key-value对。

# Collections类

## 基本概念

Java提供了一个操作Set、List和Map等集合的工具类Collections，这个类在java.util包中，该工具类里提供了大量方法对集合元素进行排序、查询和修改等操作，还提供了将集合对象设置为不可变、对集合对象实现同步控制等方法。

## 源码



## List集合排序方法

### （1）static void reverse(List list)

反转集合中的元素

### （2）static void shuffle(List list)

对集合中的元素进行随机排序

### （3）static void sort(List list)

根据元素的自然顺序对集合中的元素按升序进行排序

### （4）static void sort(List list,Comparator c)

根据指定Comparator产生的顺序对集合元素进行排序。

### （5）static void swap(List list,int i,int j)

将集合中的i处元素和j处元素进行交换。

### （6）static void rotate(List list,int distance)

当distance为正数时，将集合的后distance个元素整体移到前面，反之distance为负数时，将集合前的distance个元素整体移到后面。

### 代码示例

ArrayList al = new ArrayList();

al.add(1);

al.add(2);

al.add(-3);

al.add(4);

al.add(-10);

System.out.println(al);//输出集合中的元素

//将集合中的元素进行反转

Collections.reverse(al);

System.out.println(al);

//将集合中的元素按自然顺序进行升序排序

Collections.sort(al);

System.out.println(al);

//将集合中的元素进行随机排序

Collections.shuffle(al);

## Collections提供了查找、替换集合元素的常用方法

### （1）static int binarySearch(List list,Object key)

使用二分搜索法搜索指定的List集合，以获得指定对象在List集合中的索引，如果要使该方法可以正常工作，则必须保证List中的元素已经处于有序状态。

### （2）static Object max(Collection coll)

根据元素的自然顺序，返回给定集合中的最大元素。

### （3）static Object min(Collection coll)

根据元素的自然顺序，返回给定集合中的最小元素。

### （4）static void fill(List list,Object obj)

使用指定元素obj替换指定List集合中的所有元素。

### （5）static int frequency(Collection c,Object o)

返回指定集合中指定元素的出现次数。

### （6）static int indexOfSubList(List source,List target)

返回子List对象在父List对象中第一次出现的位置索引，如果父List中没有出现这样的子List，则返回-1。

### （7）static int latIndexOfSubList(List source,List target)

返回子List对象在父List对象中最后一次出现的位置索引，如果父List中没有出现这样的子List，则返回-1。

### （8）static boolean replaceAll(List list,Object oldVal,Objct newVal)

使用一个新值newVal替换List对象的所有旧值oldVal。

### 代码示例

ArrayList al = new ArrayList();

al.add(2);

al.add(-5);

al.add(0);

al.add(3);

al.add(2);

System.out.println(al);

//输出集合中最大的元素

//输出集合中最小的元素

//将集合中的0用1去代替

//判断集合中元素2出现的次数

//对集合进行排序，从小到大

//搜搜集合中某个元素的索引位置

int num = Collections.max(al);

System.out.println("集合中的最大元素："+num);

int min = Collections.min(al);

System.out.println("集合中的最小元素："+min);

boolean a = Collections.replaceAll(al, 0, 1);

System.out.println(a);

System.out.println(al);

int num1 = Collections.frequency(al, 2);

System.out.println(num1);

Collections.sort(al);

System.out.println(al);

Collections.reverse(al);

System.out.println(al);

Collections.sort(al);

System.out.println(al);

int index = Collections.binarySearch(al,-5);

System.out.println(index);

## 同步控制

### 基本说明

Collections类中提供了多个synchronizedXxx()方法，该方法可以将指定集合包装成线程同步的集合，从而可以解决多线程并发访问集合时的线程安全问题。Java中常用的集合框架中的实现类HashSet、TreeSet、ArrayList、LinkedList、HashMap和TreeMap都是线程不安全的，如果有多个线程访问它们，而且有超过一个线程试图修改它们，则可能出现错误。Collections提供了多个静态方法可以把它们包装成线程同步的集合。

Collection coll = Collections.synchronizedCollection(new ArrayList());

List list = Collections.synchronizedList(new ArrayList());

Set set = Collections.synchronizedSet(new HashSet());

Map map = Collections.synchronizedMap(new HashMap());

### 设置不可变对象

Collections提供了三个方法可以返回一个不可变的集合。

#### （1）static List emptyXxx()

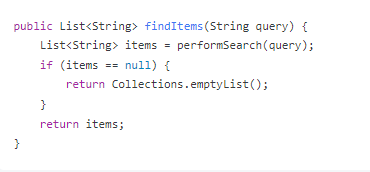
返回一个空的、不可变的集合对象，此处的集合可以是List、Set、Map。

##### 应用场景

###### 1、****初始化不可变空列表****

当你需要返回一个空的 List 并且不希望它被修改时，可以使用 Collections.emptyList()。这在某些情况下，比如返回值为空的 API 或方法中很有用。

例如，假设你有一个方法根据某些条件返回一个 List，如果没有符合条件的元素，可以返回空列表：



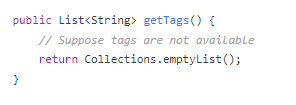
###### 2、性能优化

Collections.emptyList() 返回的空列表是共享的，并且是不可变的，因此创建它的开销非常小。这与每次都创建新的空列表相比，能够减少内存使用和提高性能。

比如，当方法需要返回空列表时，使用 Collections.emptyList() 比创建一个新的空 ArrayList 更高效。

###### 3、作为默认值

在设计类或方法时，可能需要提供默认的返回值。如果没有特定的值可返回，使用 Collections.emptyList() 可以作为默认值返回。

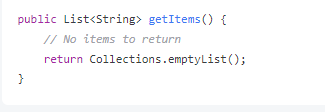


###### 4、简化代码

使用 Collections.emptyList() 可以简化代码，使其更清晰且减少了对额外条件判断的需求。

###### 5、避免 NullPointerException

返回 Collections.emptyList() 可以避免在调用者代码中进行 null 检查，从而减少了 NullPointerException 的风险。



##### 注意事项

###### 1、不可变性

Collections.emptyList() 返回的列表是不可变的。如果尝试修改该列表（例如调用 add、remove 方法），会抛出 UnsupportedOperationException。

例如：



###### 2、类型安全

Collections.emptyList() 是类型安全的，并且根据返回类型自动推断。例如，Collections.<String>emptyList() 返回 List<String> 类型。

通过使用 Collections.emptyList()，可以使代码更加简洁且高效，避免了对空列表的重复创建，同时提供了明确的不可变性保障。

#### （2）static List singletonXxx()

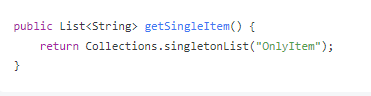
返回一个只包含指定对象的、不可变的集合对象，此处的集合可以是List、Set、Map。

##### 应用场景

###### 1、返回单一元素的列表

当你需要一个只包含单个元素的列表时，可以使用 Collections.singletonList()。这种情况通常在需要返回特定的默认值或唯一值时非常有用。

例如，某个方法如果只有一个特定的元素可以返回，使用 Collections.singletonList() 可以简化代码：



###### 2、避免创建不必要的集合

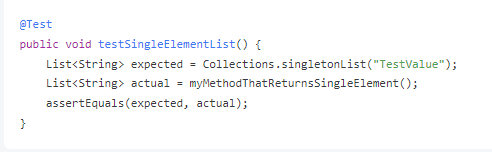
（1）Collections.singletonList() 是一个高效的方式来创建包含单个元素的列表，避免了手动创建和初始化 ArrayList 或其他集合类型。

（2）使用 Collections.singletonList() 可以减少内存开销和提升性能，因为它返回的是一个共享的、不可变的列表实例。

###### 3、在测试中使用

在编写单元测试时，可能需要创建包含单一元素的列表作为测试数据。使用 Collections.singletonList() 可以简化测试代码并提高可读性。

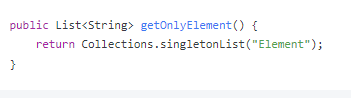
例如：



###### 4、简化代码

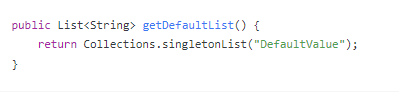
使用 Collections.singletonList() 可以使代码更简洁。例如，在某些情况下，你可能需要返回包含单个元素的列表而不是创建一个新的列表并添加元素。

例如：



###### 5、避免null值

如果你的方法在某些情况下只需要返回一个默认的非空值，可以使用 Collections.singletonList() 避免返回 null。这样可以减少 NullPointerException 的风险。例如：

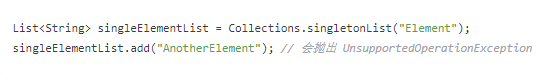


##### 注意事项

###### 1、不可变性

Collections.singletonList() 返回的列表是不可变的。如果尝试对其进行修改（例如调用 add、remove 方法），将会抛出 UnsupportedOperationException。

例如：



###### 2、类型安全

Collections.singletonList() 是类型安全的，返回的 List 类型将根据传入的元素类型进行推断。

###### 3、性能

Collections.singletonList() 返回的列表是共享的，并且它是不可变的，因此创建它的开销非常小。与每次都创建新的 ArrayList 相比，这能够提高性能和减少内存使用。

通过使用 Collections.singletonList()，你可以方便地创建包含单一元素的列表，使代码更简洁且性能更佳。这种方法适合在需要返回单一元素的列表时使用，并且有助于避免不必要的集合创建和维护。

#### （3）static List unmodifiableXxx()

返回指定集合对象的不可变视图，此处的集合可以是List、Set、Map。

上面三类方法的参数是原有的集合对象，返回值是该集合的“只读”版本，通过Collections提供的三类方法，可以产生“只读”的Collection或Map集合。

##### 应用场景

# 集合循环遍历的方法

## List集合遍历的方法

List list = new ArrayList();

list.add(3);

list.add(2);

list.add(10);

list.add(4);

list.add(70);

for(int i=0;i<list.size();i++){

int j = (Integer)list.get(i);

System.out.println(j);

}

说明：遍历List集合中的元素，使用for循环去遍历获取。

## Set集合循环遍历的方法

Set<String> set = new HashSet<String>();

Iterator<String> it = set.iterator();

while (it.hasNext()) {

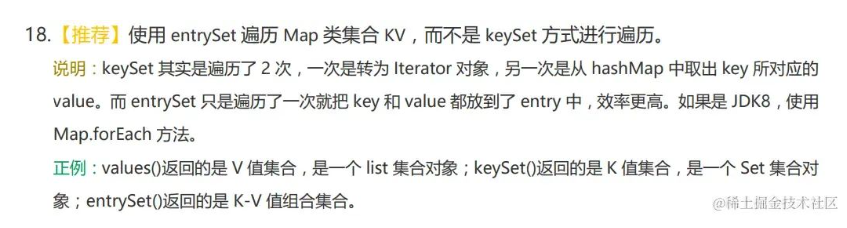
String str = it.next();

System.out.println(str);

}

说明：遍历Set集合中的元素，使用iterator()方法去遍历。

## Map集合循环遍历的方法



### 使用entrySet()方法（使用这种方法）

（1）Iterator iterator = map.entrySet().iterator();

while(iterator.hasNext()){

Map.Entry mapEntry = (Map.Entry)iterator.next();

Object keyObject = mapEntry.getKey();

Object vaObject = mapEntry.getValue();

System.out.println(keyObject+""+vaObject);

}

### 使用keySet()方法（不推荐使用这个方法）

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

map.put("aa", 11);

map.put("bb", 11);

map.put("cc", 11);

System.out.println(map);

Iterator<String> it = map.keySet().iterator();

while(it.hasNext()){

String key = it.next();

int value = (int) map.get(key);

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

}

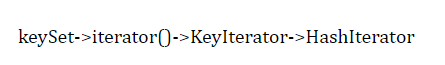
总结：

1、使用keySet遍历，其实内部是使用了对应的iterator()方法。

2、iterator()方法是创建了一个KeyIterator对象

3、KeyIterator对象extendHashIterator对象。

4、HashIterator对象的构造方法中，会遍历找到第一个不为空的entry。



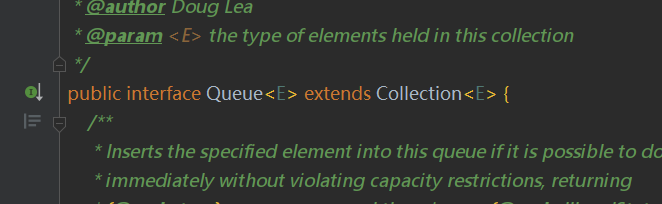
# Queue接口

## 基本概念

队列是一种特殊的线性表，遵循先入先出、后入后出的基本原则，一般来说，它只允许在表的前端进行删除操作，而在表的后端进行插入操作，但是java的某些队列运行在任何地方插入删除；比如我们常用的 LinkedList 集合，它实现了Queue 接口，因此我们可以理解为 LinkedList 就是一个队列。



## 源码



## 实现类

LinkedList

ArrayDeque

PriorityQueue

## 方法

### boolean add(E e);

将指定的元素插入队列。如果任务成功，则add()返回true，否则将引发异常。

### boolean offer(E e);

将指定的元素插入队列。如果任务成功，则offer()返回true，否则返回false。

### E remove();

返回并删除队列的头部。如果队列为空，则引发异常

### E poll();

返回并删除队列的开头。 如果队列为空，则返回null。

### E element();

返回队列的开头。如果队列为空，则引发异常。

### E peek();

返回队列的开头。 如果队列为空，则返回null。

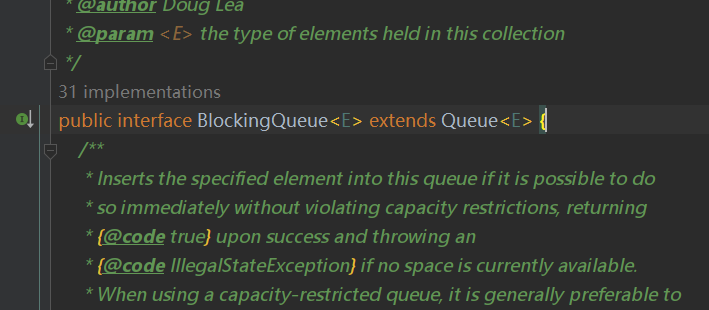
## 分类

队列主要分为阻塞和非阻塞，有界和无界、单向链表和[双向链表](https://so.csdn.net/so/search?q=%E5%8F%8C%E5%90%91%E9%93%BE%E8%A1%A8&spm=1001.2101.3001.7020)之分。

# BlockingQueue接口

## 基本概念

## 源码

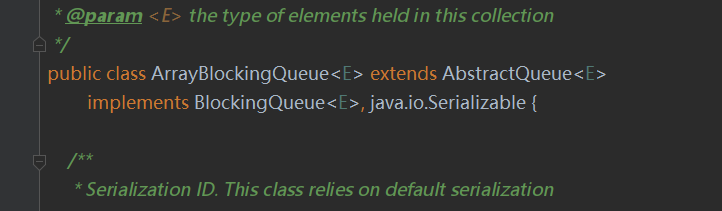


## 方法

# ArrayBlockingQueue类

## 基本概念

## 源码



## 方法

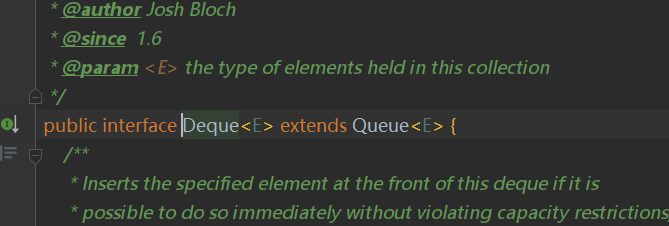
# Deque接口

## 基本概念

Deque接口提供了双端队列(Deque)的功能，它继承了Queue接口。在常规队列中，元素是从后面添加的，而从前面删除的。但是在双端队列中，我们可以**从前后插入和删除元素**。



## 源码



## 实现类

LinkedList

ArrayDeque

## 方法

### void addFirst(E e);

在双端队列的开头添加指定的元素。如果双端队列已满，则引发异常。

### void addLast(E e);

在双端队列的末尾添加指定的元素。如果双端队列已满，则引发异常。

### offerFirst()

在双端队列的开头添加指定的元素。如果双端队列已满，则返回false。

### offerLast()

在双端队列的末尾添加指定的元素。如果双端队列已满，则返回false。

### getFirst()

返回双端队列的第一个元素。如果双端队列为空，则引发异常。

### getLast()

返回双端队列的最后一个元素。如果双端队列为空，则引发异常。

### peekFirst()

返回双端队列的第一个元素。如果双端队列为空，则返回null。

### peekLast()

返回双端队列的最后一个元素。如果双端队列为空，则返回null。

### removeFirst()

返回并删除双端队列的第一个元素。如果双端队列为空，则引发异常。

### removeLast()

返回并删除双端队列的最后一个元素。如果双端队列为空，则引发异常。

### pollFirst()

返回并删除双端队列的第一个元素。如果双端队列为空，则返回null。

### pollLast()

返回并删除双端队列的最后一个元素。如果双端队列为空，则返回null。

### push()

在双端队列的开头添加元素

### pop()

从双端队列的开头删除元素

### peek()

从双端队列的开头返回一个元素