# 小结

|  |
| --- |
| 1、栈：先进后出FILO  队列：先进先出FIFO  栈的插入和删除操作只允许在表的尾端进行（在栈中成为“栈顶”），满足“FIFO：First In Last Out”；  队列只允许在表尾插入数据元素，在表头删除数据元素,满足“First In First Out”。  2、堆和栈的区别  （1）内存中的堆和栈  Java把内存划分成两种：一种是栈内存，一种是堆内存。  在函数中定义的一些基本类型的变量和对象的引用变量都在函数的栈内存中分配。  堆内存用来存放由new创建的对象和数组。在堆中分配的内存，由Java虚拟机的自动垃圾回收器来管理。  （2）数据结构中的堆和栈  数据结构方面的堆和栈，这些都是不同的概念。这里的堆实际上指的就是（满足堆性质的）优先队列的一种数据结构，第1个元素有最高的优先权；栈实际上就是满足先进后出的性质的数据结构。  通常我们所说的堆的数据结构，是指二叉堆。  堆的特点是根结点的值最小（或最大），且根结点的两个子树也是一个堆。  二叉堆是一种特殊的堆，二叉堆是完全二元树（二叉树）或者是近似完全二元树（二叉树）。二叉堆有两种：最大堆和最小堆。最大堆：父结点的键值总是大于或等于任何一个子节点的键值；最小堆：父结点的键值总是小于或等于任何一个子节点的键值。  3、HashMap的实现原理：数组 + 单向链表（红黑树）实现。  1）利用key的hashCode重新hash计算出当前对象的元素在数组中的下标。  2）存储时，如果出现hash值相同的key，此时有两种情况。1)如果key相同，则覆盖原始值；2)如果key不同（出现冲突），则将当前的key-value放入链表中。  3）获取时，直接找到hash值对应的下标，在进一步判断key是否相同，从而找到对应值。  4）理解了以上过程就不难明白HashMap是如何解决hash冲突的问题，核心就是使用了数组的存储方式，然后将冲突的key的对象放入链表中，一旦发现冲突就在链表中做进一步的对比。  5）最消耗性能的点就出现了：原数组中的数据必须重新计算其在新数组中的位置，并放进去，这就是resize。新版本的HashMap对扩容进行优化。  4、ArrayList：Object数组 + 可扩容；  LinkedList：双向链表；  HashMap: Entry数组 + 链表（红黑树） <---- HashSet  LinkedHashMap: HashMap + 双向链表记录插入顺序 <---- LinkedHashSet  Treemap:红黑树(自平衡的二叉搜素树) <---- TreeSet  线程安全的集合常见的三个：  Vector：功能同ArrayList，另外是线程安全的；  HashTable 是线程安全的；HashTable 内部的方法基本都经过synchronized 修饰，HashtableEntry数组 + 链表。  ConcurrentHashMap是HashMap的线程安全版（自JDK1.5引入），提供比Hashtable更高效的并发性能。采用分离锁，高效。  Vector和HashTable都是很少使用了，过时。    Hashtable 在进行读写操作时会锁住整个Entry数组，这就导致数据越多性能越差。  而ConcurrentHashMap使用分离锁的思路解决并发性能，其将 Entry数组拆分至16个Segment中，以哈希算法决定Entry应该存储在哪个Segment。这样就可以实现在写操作时只对一个Segment 加锁，大幅提升了并发写的性能。  5、Set接口继承Collection，用于存储不含重复元素的集合。  几乎所有的Set实现都是基于同类型Map的，简单地说，Set是阉割版的Map。每一个Set内都有一个同类型的Map实例（CopyOnWriteArraySet除外，它内置的是CopyOnWriteArrayList实例），Set把元素作为key存储在自己的Map实例中，value则是一个空的Object。  Set的常用实现包括 HashSet、TreeSet、ConcurrentSkipListSet等，原理和对应的Map实现完全一致。 |

ArrayList：Object数组 + 可扩容；

Vector：同ArrayList，另外是线程安全的；

LinkedList：双向链表；

HashMap: Entry数组 + 链表（红黑树） <---- HashSet

LinkedHashMap: HashMap + 双向链表记录插入顺序 <---- LinkedHashSet

Treemap:红黑树(自平衡的二叉搜素树) <---- TreeSet

List , Set继承至Collection接口，Map为独立接口

List接口继承自Collection，用于定义以线性表形式存储的集合

List下有ArrayList，LinkedList，Vector

Set下有HashSet，LinkedHashSet，TreeSet

Map接口在Collection的基础上，为其中的每个对象指定了一个key，并使用Entry保存每个key-value对，以实现通过key快速定位到对象(value)。

Map下有HashMap，LinkedHashMap，TreeMap，Hashtable

# Collection接口和Map接口

Java提供的众多集合类由两大接口衍生而来：Collection接口和Map接口

## Collection接口

Collection接口定义了一个包含一批对象的集合。接口的主要方法包括：

size() - 集合内的对象数量

add(E)/addAll(Collection) - 向集合内添加单个/批量对象

remove(Object)/removeAll(Collection) - 从集合内删除单个/批量对象

contains(Object)/containsAll(Collection) - 判断集合中是否存在某个/某些对象

toArray() - 返回包含集合内所有对象的数组等

## Map接口

Map接口在Collection的基础上，为其中的每个对象指定了一个key，并使用Entry保存每个key-value对，以实现通过key快速定位到对象(value)。Map接口的主要方法包括：

size() - 集合内的对象数量

put(K,V)/putAll(Map) - 向Map内添加单个/批量对象

get(K) - 返回Key对应的对象

remove(K) - 删除Key对应的对象

keySet() - 返回包含Map中所有key的Set

values() - 返回包含Map中所有value的Collection

entrySet() - 返回包含Map中所有key-value对的EntrySet

containsKey(K)/containsValue(V) - 判断Map中是否存在指定key/value

# List类集合

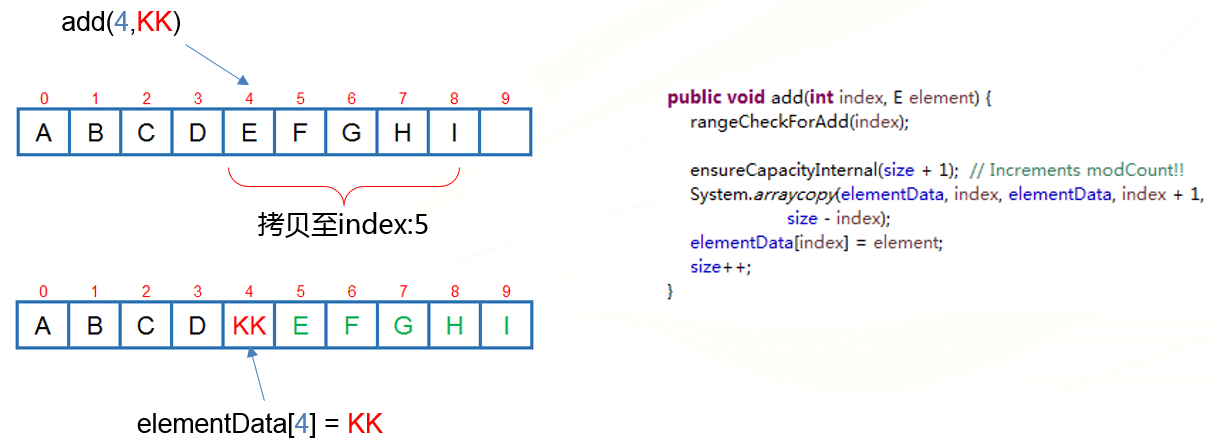
List接口继承自Collection，用于定义以线性表形式存储的集合，List接口为集合中的每个对象分配了一个索引(index)，标记该对象在List中的位置，并可以通过index定位到指定位置的对象。

## 2.1 ArrayList

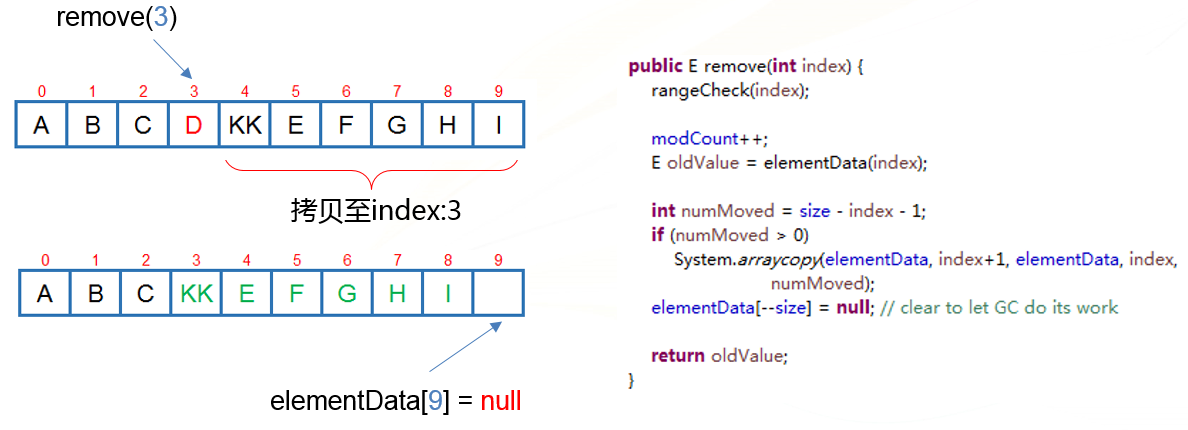
ArrayList基于数组来实现集合的功能，其内部维护了一个可变长的对象数组，集合内所有对象存储于这个数组中，并实现该数组长度的动态伸缩。

ArrayList使用数组拷贝来实现指定位置的插入和删除：

插入：



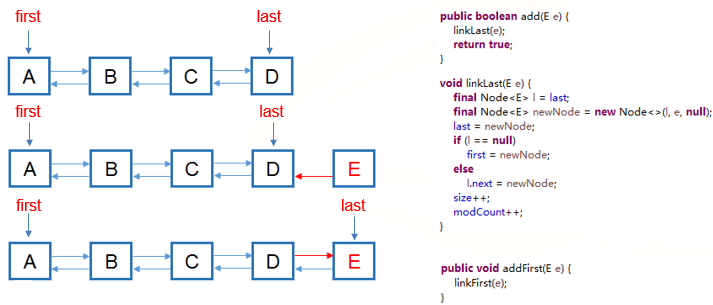
删除：



## 2.2 LinkedList

LinkedList基于链表来实现集合的功能，其实现了静态类Node，集合中的每个对象都由一个Node保存，每个Node都拥有到自己的前一个和后一个Node的引用。

LinkedList追加元素的过程示例：



## 2.3 ArrayList与LinkedList

ArrayList的随机访问更高，基于数组实现的ArrayList可直接定位到目标对象，而LinkedList需要从头Node或尾Node开始向后/向前遍历若干次才能定位到目标对象。

LinkedList在头/尾节点执行插入/删除操作的效率比ArrayList要高。

由于ArrayList每次扩容的容量是当前的1.5倍，所以LinkedList所占的内存空间要更小一些。

二者的遍历效率接近，但需要注意，遍历LinkedList时应用iterator方式，不要用get(int)方式，否则效率会很低。

## 2.4 Vector与ArrayList

Vector和ArrayList很像，都是基于数组实现的集合，它和ArrayList的主要区别在于

Vector是线程安全的，而ArrayList不是线程安全的。

由于Vector中的方法基本都是synchronized的，其性能低于ArrayList。

Vector可以定义数组长度扩容的因子，ArrayList不能。

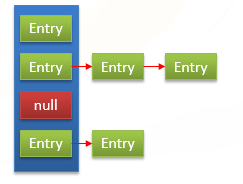
# Map类集合

Map将key和value封装至一个叫做Entry的对象中，Map中存储的元素实际是Entry。只有在keySet()和values()方法被调用时，Map才会将keySet和values对象实例化。

每一个Map根据其自身特点，都有不同的Entry实现，以对应Map的内部类形式出现。

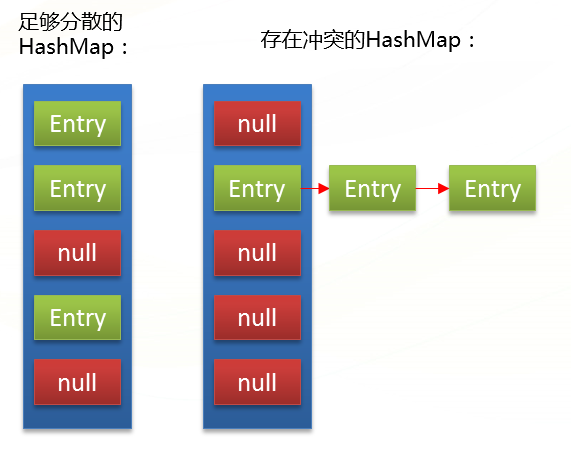
## 3.1 HashMap

HashMap将Entry对象存储在一个数组中，并通过哈希表来实现对Entry的快速访问：



由每个Entry中的key的哈希值决定该Entry在数组中的位置。以这种特性能够实现通过key快速查找到Entry，从而获得该key对应的value。在不发生哈希冲突的前提下，查找的时间复杂度是O(1)。

如果两个不同的key计算出的index是一样的，就会发生两个不同的key都对应到数组中同一个位置的情况，也就是所谓的哈希冲突。HashMap处理哈希冲突的方法是拉链法，也就是说数组中每个位置保存的实际是一个Entry链表，链表中每个Entry都拥有指向链表中后一个Entry的引用。在发生哈希冲突时，将冲突的Entry追加至链表的头部。当HashMap在寻址时发现某个key对应的数组index上有多个Entry，便会遍历该位置上的 Entry链表，直到找到目标的Entry。



HashMap的Entry类：

static class Entry<K,V> implements Map.Entry<K,V> {

final K key;

V value;

Entry<K,V> next;

int hash;

}

HashMap由于其快速寻址的特点，可以说是最经常被使用的Map实现类

## 3.2 Hashtable

Hashtable 可以说是HashMap的前身（Hashtable自JDK1.0就存在，而HashMap乃至整个Map接口都是JDK1.2引入的新特性），其实现思路与HashMap几乎完全一样，都是通过数组存储Entry，以key的哈希值计算Entry在数组中的index，用拉链法解决哈希冲突。二者最大的不同在于，Hashtable是线程安全的，其提供的方法几乎都是同步的。

## 3.3 ConcurrentHashMap

ConcurrentHashMap是HashMap的线程安全版（自JDK1.5引入），提供比Hashtable更高效的并发性能。

Hashtable 在进行读写操作时会锁住整个Entry数组，这就导致数据越多性能越差。而ConcurrentHashMap使用分离锁的思路解决并发性能，其将 Entry数组拆分至16个Segment中，以哈希算法决定Entry应该存储在哪个Segment。这样就可以实现在写操作时只对一个Segment 加锁，大幅提升了并发写的性能。

在进行读操作时，ConcurrentHashMap在绝大部分情况下都不需要加锁，其Entry中的value是volatile的，这保证了value被修改时的线程可见性，无需加锁便能实现线程安全的读操作。

ConcurrentHashMap的HashEntry类：

static final class HashEntry<K,V> {

final int hash;

final K key;

volatile V value;

volatile HashEntry<K,V> next;

}

但是鱼与熊掌不可兼得，ConcurrentHashMap的高性能是有代价的（否则Hashtable就没有存在价值了），那就是它不能保证读操作的绝对一致性。ConcurrentHashMap保证读操作能获取到已存在Entry的value的最新值，同时也能保证读操作可获取到已完成的写操作的内容，但如果写操作是在创建一个新的Entry，那么在写操作没有完成时，读操作是有可能获取不到这个Entry的。

## 3.4 HashMap vs Hashtable vs ConcurrentHashMap

三者在数据存储层面的机制原理基本一致。

HashMap不是线程安全的，多线程环境下除了不能保证数据一致性之外，还有可能在rehash阶段引发Entry链表成环，导致死循环。

Hashtable是线程安全的，能保证绝对的数据一致性，但性能是问题，并发线程越多，性能越差。

ConcurrentHashMap 也是线程安全的，使用分离锁和volatile等方法极大地提升了读写性能，同时也能保证在绝大部分情况下的数据一致性。但其不能保证绝对的数据一致性， 在一个线程向Map中加入Entry的操作没有完全完成之前，其他线程有可能读不到新加入的Entry。

## 3.5 LinkedHashMap

LinkedHashMap与HashMap非常类似，唯一的不同在于前者的Entry在HashMap.Entry的基础上增加了到前一个插入和后一个插入的Entry的引用，以实现能够按Entry的插入顺序进行遍历。

通过改变Entry节点，使得有链表的功能。

## 3.6 TreeMap

TreeMap是基于红黑树实现的Map结构，其Entry类拥有到左/右叶子节点和父节点的引用，同时还记录了自己的颜色。

通过特殊的节点、特殊的插入与删除规则实现，自平衡的二叉树。

static final class Entry<K,V> implements Map.Entry<K,V> {

K key;

V value;

Entry<K,V> left = null;

Entry<K,V> right = null;

Entry<K,V> parent;

boolean color = BLACK;

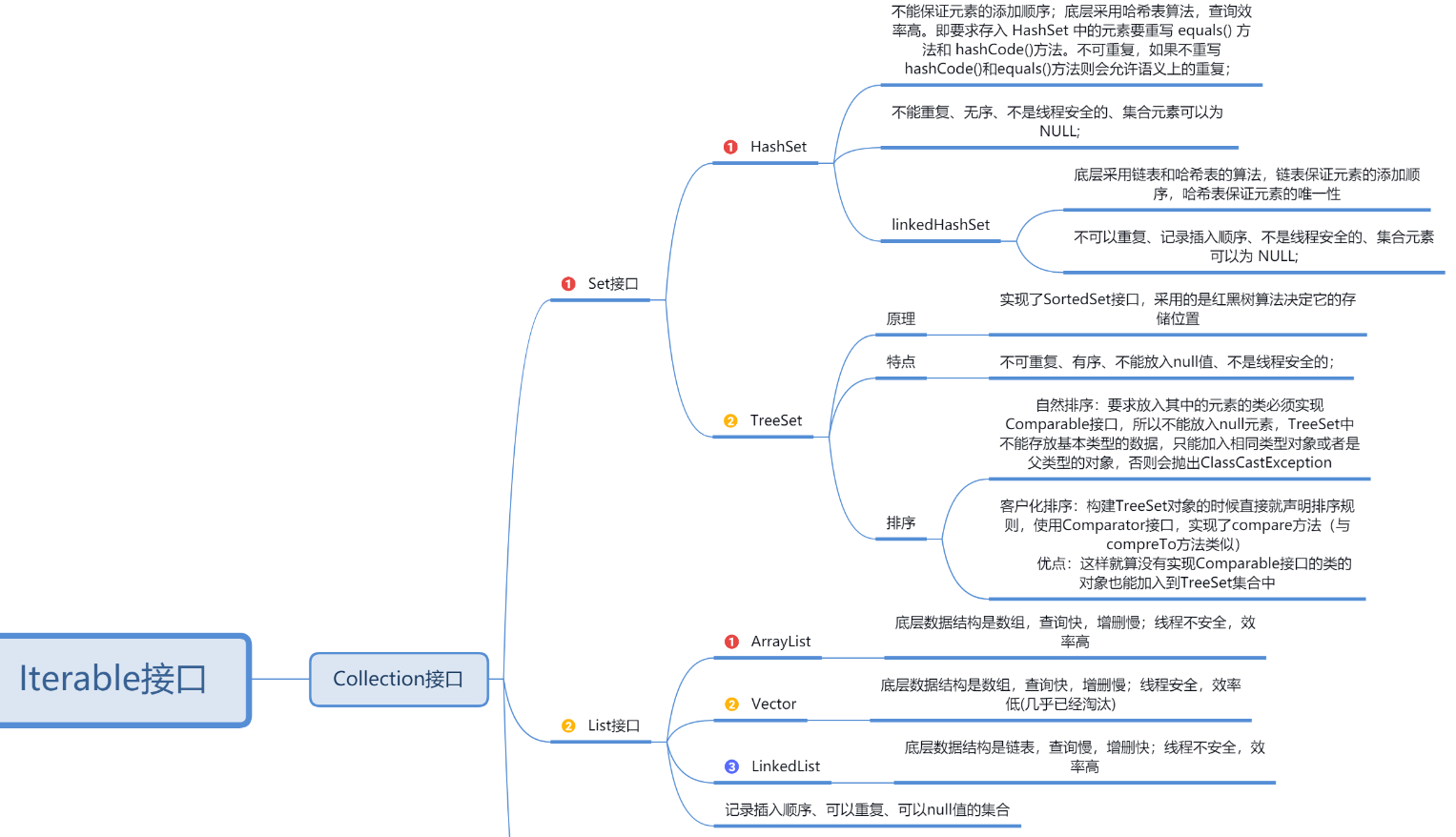
}

红黑树实际是一种算法复杂但高效的平衡二叉树，具备二叉树的基本性质，即任何节点的值大于其左叶子节点，小于其右叶子节点，利用这种特性，TreeMap能够实现Entry的排序和快速查找。

TreeMap的Entry是有序的，所以提供了一系列方便的功能，比如获取以升序或降序排列的KeySet(EntrySet)、获取在指定key(Entry)之前/之后的key(Entry)等等。适合需要对key进行有序操作的场景。

# Set类集合

Set 接口继承Collection，用于存储不含重复元素的集合。几乎所有的Set实现都是基于同类型Map的，简单地说，Set是阉割版的Map。每一个Set内都有一个同类型的Map实例（CopyOnWriteArraySet除外，它内置的是CopyOnWriteArrayList实例），Set把元素作为key存储在自己的Map实例中，value则是一个空的Object。Set的常用实现也包括 HashSet、TreeSet、ConcurrentSkipListSet等，原理和对应的Map实现完全一致。



# List 有序,可重复

（注：有序，就是有顺序，存在索引。）

## 1.1 ArrayList:

优点: 底层数据结构是数组，查询快，增删慢。（可以自动动态扩容的数组。）

缺点: 线程不安全，效率高。（方法没有同步）

## 1.2 Vector:

优点: 底层数据结构是数组，查询快，增删慢。

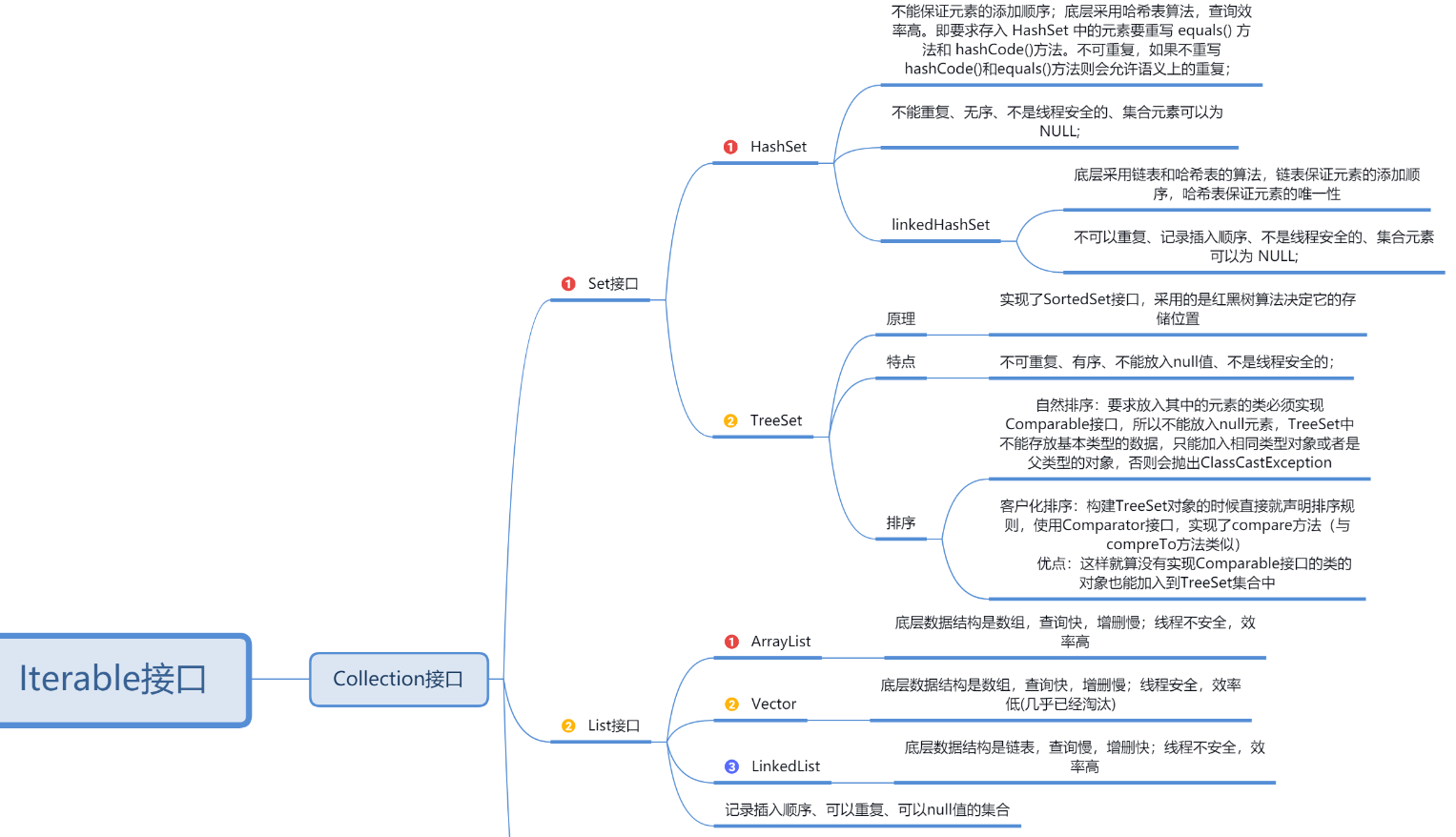
缺点: 线程安全，效率低 （跟ArrayList很像，差别就是线程安全）

## 1.3 LinkedList:

优点: 底层数据结构是链表，查询慢，增删快。

（实际上增删也是不快，其操作前还是要遍历查询，只是单纯的增删哪一步比较快。 我们比较ArrayList和LinkedList时，一般情况下，增删的复杂度是一样的。ArrayList的增删重在移动元素，LinkedList的重在遍历查询。）

缺点: 线程不安全，效率高



# 2.Set 无序,唯一

## 2.1 HashSet：

说一下 HashSet 的实现原理？

底层数据结构是哈希表。(无序,唯一)

如何来保证元素唯一性?

1.依赖两个方法：hashCode()和equals()

HashSet底层数据结构采用哈希表实现，元素无序且唯一，线程不安全，效率高，可以存储null元素，元素的唯一性是靠所存储元素类型是否重写hashCode()和equals()方法来保证的，如果没有重写这两个方法，则无法保证元素的唯一性。

具体实现唯一性的比较过程:

1.存储元素时首先会使用hash()算法函数生成一个int类型hashCode散列值，然后已经的所存储的元素的hashCode值比较，如果hashCode不相等，肯定是不同的对象。

2.hashCode值相同，再比较equals方法。

3.equals相同，对象相同。（则无需储存）

HashSet内部持有一个HashMap,所有的操作都是对这个map对象的操作。Set把元素作为key存储在自己的Map实例中，value则是一个空的Object。

public class HashSet<E>{

private transient HashMap<E,Object> map;

// Dummy value to associate with an Object in the backing Map

private static final Object PRESENT = new Object();

public HashSet() {

map = new HashMap<>();

}

public boolean add(E e) {

return map.put(e, PRESENT)==null;

}

......

}

HashSet是基于HashMap实现的，默认构造函数是构建一个初始容量为16，负载因子为0.75 的HashMap。封装了一个 HashMap 对象来存储所有的集合元素，所有放入 HashSet 中的集合元素实际上由 HashMap 的 key 来保存，而 HashMap 的 value 则存储了一个 PRESENT，它是一个静态的 Object 对象。

当我们试图把某个类的对象当成 HashMap的 key，或试图将这个类的对象放入 HashSet 中保存时，重写该类的equals(Object obj)方法和 hashCode() 方法很重要，而且这两个方法的返回值必须保持一致：当该类的两个的 hashCode() 返回值相同时，它们通过 equals() 方法比较也应该返回 true。通常来说，所有参与计算 hashCode() 返回值的关键属性，都应该用于作为 equals() 比较的标准。

HashSet的其他操作都是基于HashMap的。

## 2.2 LinkedHashSet：

底层数据结构是链表和哈希表。(FIFO插入有序,唯一)

1.由链表保证元素有序

2.由哈希表保证元素唯一

LinkedHashSet底层数据结构采用链表和哈希表共同实现，链表保证了元素的顺序与存储顺序一致，哈希表保证了元素的唯一性。线程不安全，效率高。

|  |
| --- |
| LinkedHashSet 的实现  对于 LinkedHashSet 而言，它继承与 HashSet、又基于 LinkedHashMap 来实现的。  LinkedHashSet 底层使用 LinkedHashMap 来保存所有元素，它继承与 HashSet，其所有的方法操作上又与 HashSet 相同，因此 LinkedHashSet 的实现上非常简单，只提供了四个构造方法，并通过传递一个标识参数，调用父类的构造器，底层构造一个 LinkedHashMap 来实现，在相关操作上与父类 HashSet 的操作相同，直接调用父类 HashSet 的方法即可。LinkedHashSet 的源代码如下：  public class LinkedHashSet<E>  extends HashSet<E>  implements Set<E>, Cloneable, java.io.Serializable {  private static final long serialVersionUID = -2851667679971038690L;  /\*\*  \* 构造一个带有指定初始容量和加载因子的新空链接哈希set。  \*  \* 底层会调用父类的构造方法，构造一个有指定初始容量和加载因子的LinkedHashMap实例。  \* @param initialCapacity 初始容量。  \* @param loadFactor 加载因子。  \*/  public LinkedHashSet(int initialCapacity, float loadFactor) {  super(initialCapacity, loadFactor, true);  }  /\*\*  \* 构造一个带指定初始容量和默认加载因子0.75的新空链接哈希set。  \*  \* 底层会调用父类的构造方法，构造一个带指定初始容量和默认加载因子0.75的LinkedHashMap实例。  \* @param initialCapacity 初始容量。  \*/  public LinkedHashSet(int initialCapacity) {  super(initialCapacity, .75f, true);  }  /\*\*  \* 构造一个带默认初始容量16和加载因子0.75的新空链接哈希set。  \*  \* 底层会调用父类的构造方法，构造一个带默认初始容量16和加载因子0.75的LinkedHashMap实例。  \*/  public LinkedHashSet() {  super(16, .75f, true);  }  /\*\*  \* 构造一个与指定collection中的元素相同的新链接哈希set。  \*  \* 底层会调用父类的构造方法，构造一个足以包含指定collection  \* 中所有元素的初始容量和加载因子为0.75的LinkedHashMap实例。  \* @param c 其中的元素将存放在此set中的collection。  \*/  public LinkedHashSet(Collection<? extends E> c) {  super(Math.max(2\*c.size(), 11), .75f, true);  addAll(c);  }  }  以上几乎就是 LinkedHashSet 的全部代码了，那么读者可能就会怀疑了，不是说 LinkedHashSet 是基于 LinkedHashMap 实现的吗？那我为什么在源码中甚至都没有看到出现过 LinkedHashMap。不要着急，我们可以看到在 LinkedHashSet 的构造方法中，其调用了父类的构造方法。我们可以进去看一下：  /\*\*  \* 以指定的initialCapacity和loadFactor构造一个新的空链接哈希集合。  \* 此构造函数为包访问权限，不对外公开，实际只是是对LinkedHashSet的支持。  \*  \* 实际底层会以指定的参数构造一个空LinkedHashMap实例来实现。  \* @param initialCapacity 初始容量。  \* @param loadFactor 加载因子。  \* @param dummy 标记。  \*/  HashSet(int initialCapacity, float loadFactor, boolean dummy) {  map = new LinkedHashMap<E,Object>(initialCapacity, loadFactor);  }  在父类 HashSet 中，专为 LinkedHashSet 提供的构造方法如下，该方法为包访问权限，并未对外公开。  由上述源代码可见，LinkedHashSet 通过继承 HashSet，底层使用 LinkedHashMap，以很简单明了的方式来实现了其自身的所有功能。  总结  以上就是关于 LinkedHashSet 的内容，我们只是从概述上以及构造方法这几个方面介绍了，并不是我们不想去深入其读取或者写入方法，而是其本身没有实现，只是继承于父类 HashSet 的方法。  所以我们需要注意的点是：  •LinkedHashSet 是 Set 的一个具体实现，其维护着一个运行于所有条目的双重链接列表。此链接列表定义了迭代顺序，该迭代顺序可为插入顺序或是访问顺序。 （双向链表维持插入、访问顺序。）  •LinkedHashSet 继承与 HashSet，并且其内部是通过 LinkedHashMap 来实现的。有点类似于我们之前说的LinkedHashMap 其内部是基于 Hashmap 实现一样，不过还是有一点点区别的（具体的区别大家可以自己去思考一下）。  •如果我们需要迭代的顺序为插入顺序或者访问顺序，那么 LinkedHashSet 是需要你首先考虑的。 |

## 3.3 TreeSet：

底层数据结构是红黑树。(唯一，有序)

1. 如何保证元素排序的呢?

自然排序

比较器排序

2.如何保证元素唯一性的呢?

根据比较的返回值是否是0来决定

TreeSet底层数据结构采用红黑树来实现，元素唯一且已经排好序；唯一性同样需要重写hashCode和equals()方法，二叉树结构保证了元素的有序性。根据构造方法不同，分为自然排序（无参构造）和比较器排序（有参构造），自然排序要求元素必须实现Compareable接口，并重写里面的compareTo()方法，元素通过比较返回的int值来判断排序序列，返回0说明两个对象相同，不需要存储；比较器排需要在TreeSet初始化是时候传入一个实现Comparator接口的比较器对象，或者采用匿名内部类的方式new一个Comparator对象，重写里面的compare()方法；

在学习红黑树之前，咱们需要先来理解下二叉查找树（BST）。

二叉查找树

要想了解二叉查找树，我们首先看下二叉查找树有哪些特性呢？

1， 左子树上所有的节点的值均小于或等于他的根节点的值

2， 右子数上所有的节点的值均大于或等于他的根节点的值

3， 左右子树也一定分别为二叉排序树

我们来看下图的这棵树，他就是典型的二叉查找树

红黑树

红黑树就是一种平衡的二叉查找树，说他平衡的意思是他不会变成“瘸子”，左腿特别长或者右腿特别长。除了符合二叉查找树的特性之外，还具体下列的特性：

1. 节点是红色或者黑色

2. 根节点是黑色

3. 每个叶子的节点都是黑色的空节点（NULL）

4. 每个红色节点的两个子节点都是黑色的。

5. 从任意节点到其每个叶子的所有路径都包含相同的黑色节点。

public TreeSet(Comparator<? super E> comparator) {

this(new TreeMap<>(comparator));

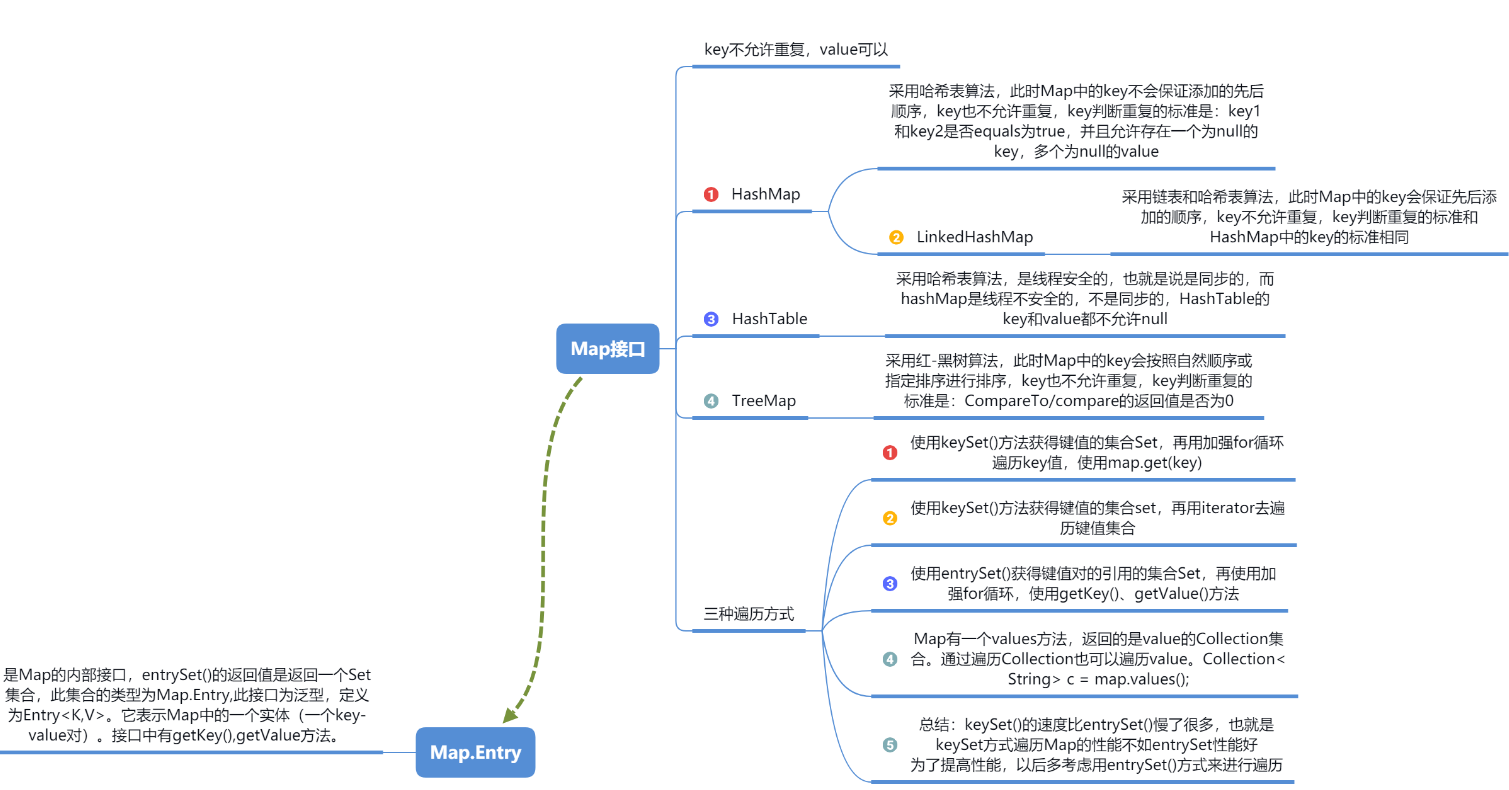
}

TreeSet实现了SortedSet接口，它是一个有序的集合类，TreeSet的底层是通过TreeMap实现的。TreeSet并不是根据插入的顺序来排序，而是根据实际的值的大小来排序。TreeSet也支持两种排序方式：自然排序、自定义排序。

TreeSet的底层是用TreeMap实现的。在构造方法中会创建一个TreeMap实例，用于存放元素，另外TreeSet是有序的，也提供了制定比较器的构造函数，如果没有提供比较器，则采用key的自然顺序进行比较大小，如果指定的比较器，则采用指定的比较器，进行key值大小的比较。

add()方法和remove()方法都比较的简单都是调用TreeMap的方法进行实现。

|  |
| --- |
| public class TreeSet<E> extends AbstractSet<E>  implements NavigableSet<E>, Cloneable, java.io.Serializable  {  //存放元素的map对象  private transient NavigableMap<E,Object> m;  //key-value ,不同的键都会对象相同的value, value = PRESENT  private static final Object PRESENT = new Object();  //指定的map对象  TreeSet(NavigableMap<E,Object> m) {  this.m = m;  }  //无参构造方法，初始化一个TreeMap对象  public TreeSet() {  this(new TreeMap<E,Object>());  }  //构造方法，指定比较器  public TreeSet(Comparator<? super E> comparator) {  this(new TreeMap<>(comparator));  }  //将集合中的元素转化为TreeSet存储  public TreeSet(Collection<? extends E> c) {  this();  addAll(c);  }  //构造方法，SortedSet转化为TreeSet存储，并使用SortedSet的比较器  public TreeSet(SortedSet<E> s) {  this(s.comparator());  addAll(s);  }  //遍历方法，返回m.keyset集合  public Iterator<E> iterator() {  return m.navigableKeySet().iterator();  }  //逆序排序的迭代器  public Iterator<E> descendingIterator() {  return m.descendingKeySet().iterator();  }  /\*\*  \* @since 1.6  \*/  public NavigableSet<E> descendingSet() {  return new TreeSet<>(m.descendingMap());  }  //返回 m 包含的键值对的数量  public int size() {  return m.size();  }  //是否为空  public boolean isEmpty() {  return m.isEmpty();  }  //是否包含指定的key  public boolean contains(Object o) {  return m.containsKey(o);  }  //添加元素，调用m.put方法实现  public boolean add(E e) {  return m.put(e, PRESENT)==null;  }  //删除方法，调用m.remove()方法实现  public boolean remove(Object o) {  return m.remove(o)==PRESENT;  }  //清除集合  public void clear() {  m.clear();  }  //将一个集合中的所有元素添加到TreeSet中  public boolean addAll(Collection<? extends E> c) {  // Use linear-time version if applicable  if (m.size()==0 && c.size() > 0 &&  c instanceof SortedSet &&  m instanceof TreeMap) {  SortedSet<? extends E> set = (SortedSet<? extends E>) c;  TreeMap<E,Object> map = (TreeMap<E, Object>) m;  Comparator<? super E> cc = (Comparator<? super E>) set.comparator();  Comparator<? super E> mc = map.comparator();  if (cc==mc || (cc != null && cc.equals(mc))) {  map.addAllForTreeSet(set, PRESENT);  return true;  }  }  return super.addAll(c);  }  //返回子集合，通过 m.subMap()方法实现  public NavigableSet<E> subSet(E fromElement, boolean fromInclusive,  E toElement, boolean toInclusive) {  return new TreeSet<>(m.subMap(fromElement, fromInclusive,  toElement, toInclusive));  }  //返回set的头部  public NavigableSet<E> headSet(E toElement, boolean inclusive) {  return new TreeSet<>(m.headMap(toElement, inclusive));  }  //返回尾部  public NavigableSet<E> tailSet(E fromElement, boolean inclusive) {  return new TreeSet<>(m.tailMap(fromElement, inclusive));  }  //返回子Set  public SortedSet<E> subSet(E fromElement, E toElement) {  return subSet(fromElement, true, toElement, false);  }  //返回set的头部  public SortedSet<E> headSet(E toElement) {  return headSet(toElement, false);  }  //返回set的尾部  public SortedSet<E> tailSet(E fromElement) {  return tailSet(fromElement, true);  }  //返回m使用的比较器  public Comparator<? super E> comparator() {  return m.comparator();  }  //返回第一个元素  public E first() {  return m.firstKey();  }  //返回最后一个元素  public E last() {  return m.lastKey();  }  //返回set中小于e的最大的元素  public E lower(E e) {  return m.lowerKey(e);  }  //返回set中小于/等于e的最大元素  public E floor(E e) {  return m.floorKey(e);  }  //返回set中大于/等于e的最大元素  public E ceiling(E e) {  return m.ceilingKey(e);  }  //返回set中大于e的最小元素  public E higher(E e) {  return m.higherKey(e);  }  //获取TreeSet中第一个元素，并从Set中删除该元素  public E pollFirst() {  Map.Entry<E,?> e = m.pollFirstEntry();  return (e == null) ? null : e.getKey();  }  //获取TreeSet中最后一个元素，并从Set中删除该元素  public E pollLast() {  Map.Entry<E,?> e = m.pollLastEntry();  return (e == null) ? null : e.getKey();  }  //克隆方法  public Object clone() {  TreeSet<E> clone = null;  try {  clone = (TreeSet<E>) super.clone();  } catch (CloneNotSupportedException e) {  throw new InternalError();  }  clone.m = new TreeMap<>(m);  return clone;  }  //将对象写入到输出流中。  private void writeObject(java.io.ObjectOutputStream s)  throws java.io.IOException {  // Write out any hidden stuff  s.defaultWriteObject();  // Write out Comparator  s.writeObject(m.comparator());  // Write out size  s.writeInt(m.size());  // Write out all elements in the proper order.  for (E e : m.keySet())  s.writeObject(e);  }  //从输入流中读取对象的信息  private void readObject(java.io.ObjectInputStream s)  throws java.io.IOException, ClassNotFoundException {  // Read in any hidden stuff  s.defaultReadObject();  // Read in Comparator  Comparator<? super E> c = (Comparator<? super E>) s.readObject();  // Create backing TreeMap  TreeMap<E,Object> tm;  if (c==null)  tm = new TreeMap<>();  else  tm = new TreeMap<>(c);  m = tm;  // Read in size  int size = s.readInt();  tm.readTreeSet(size, s, PRESENT);  }  //序列化版本号  private static final long serialVersionUID = -2479143000061671589L;  } |



# 3.Map接口:

Map用于保存具有映射关系的数据，Map里保存着两组数据：key和value，它们都可以使任何引用类型的数据，但key不能重复。所以通过指定的key就可以取出对应的value。

Map接口有四个比较重要的实现类，分别是HashMap、LinkedHashMap、TreeMap和HashTable。

TreeMap是有序的，HashMap和HashTable是无序的。

Hashtable的方法是同步的，HashMap的方法不是同步的。这是两者最主要的区别。

## 3.1 HashMap

Map 主要用于存储键(key)值(value)对，根据键得到值，因此键不允许重复,但允许值重复。

HashMap 是一个最常用的Map,它根据键的HashCode 值存储数据,根据键可以直接获取它的值，具有很快的访问速度。

HashMap最多只允许一条记录的键为Null;允许多条记录的值为 Null;

HashMap不支持线程的同步，即任一时刻可以有多个线程同时写HashMap;可能会导致数据的不一致。如果需要同步，可以用 Collections的synchronizedMap方法使HashMap具有同步的能力，或者使用ConcurrentHashMap。

HashMap基于哈希表结构实现的 ，当一个对象被当作键时，必须重写hasCode和equals方法。

## 3.2 LinkedHashMap

LinkedHashMap继承自HashMap，它主要是用链表实现来扩展HashMap类，HashMap中条目是没有顺序的，但是在LinkedHashMap中元素既可以按照它们插入图的顺序排序，也可以按它们最后一次被访问的顺序排序。

1.LinkedHashMap是继承于HashMap，是基于HashMap和双向链表来实现的。

2.HashMap无序；LinkedHashMap有序，可分为插入顺序和访问顺序两种。如果是访问顺序，那put和get操作已存在的Entry时，都会把Entry移动到双向链表的表尾(其实是先删除再插入)。

3.LinkedHashMap存取数据，还是跟HashMap一样使用的Entry[]的方式，双向链表只是为了保证顺序。

4.LinkedHashMap是线程不安全的。

## 3.3 TreeMap

TreeMap红黑树实现的key->value融合，可排序，红黑树是一种自平衡二叉查找树。

TreeMap基于红黑树数据结构的实现，键值可以使用Comparable或Comparator接口来排序。TreeMap继承自AbstractMap，同时实现了接口NavigableMap，而接口NavigableMap则继承自SortedMap。SortedMap是Map的子接口，使用它可以确保图中的条目是排好序的。

在实际使用中，如果更新图时不需要保持图中元素的顺序，就使用HashMap，如果需要保持图中元素的插入顺序或者访问顺序，就使用LinkedHashMap，如果需要使图按照键值排序，就使用TreeMap。

1.TreeMap存储K-V键值对，通过红黑树（R-B tree）实现；

2.TreeMap继承了NavigableMap接口，NavigableMap接口继承了SortedMap接口，可支持一系列的导航定位以及导航操作的方法，当然只是提供了接口，需要TreeMap自己去实现；

3.TreeMap实现了Cloneable接口，可被克隆，实现了Serializable接口，可序列化；

4.TreeMap因为是通过红黑树实现，红黑树结构天然支持排序，默认情况下通过Key值的自然顺序进行排序；

## 3.4 Hashtable

Hashtable和前面介绍的HashMap很类似，它也是一个散列表，存储的内容是键值对映射，不同之处在于，Hashtable是继承自Dictionary的，Hashtable中的函数都是同步的，这意味着它也是线程安全的，另外，Hashtable中key和value都不可以为null。

HashtableEntry数组 + 链表， 所有方法都是线程安全的。

# 顺序存储和链式存储

数据就是一些或某部分有关系的内容的组合。

数据结构是数据的存储方式。从不同的角度来讨论，分类如下:

按逻辑结构分：线性结构、集合结构、树形结构、图形结构；

按物理结构分：顺序存储结构、链式存储结构。

•逻辑结构，即按照人们的思维逻辑对其分类。

•物理结构就是数据在磁盘上的存储方式，可以是一整块存储区域，也可以是不同的存储块（但是它们之前有关系，所以就划分为一组数据）。

\*\*物理结构分法的分类

顺序存储：就是在磁盘上连续存储的，在Java中就是数组，它从第一个索引开始，所有的数据都是紧跟其后的。

链式存储:这些数据（称为数据元素，是不可再分隔的）在磁盘上是分开存储的，只是因为它们之间有一些关系，所以我们就将其联系到了一起，组成了一种数据结构————链表。

# 5.什么是链表？

（1）链表是一种物理存储结构上非连续、非顺序的存储结构，数据元素的逻辑顺序是通过链表中的指针链接次序实现的。在数据结构中，a1里面的指针存储着a2的地址，这样一个链接一个，就形成了链表。

◦相邻元素之间通过指针链接

◦最后一个元素的后继指针为NULL

◦在程序执行过程中，链表的长度可以增加或缩小

◦链表的空间能够按需分配

◦没有内存空间的浪费

（2）链表的优缺点？

•优点：

◦插入和删除时不需移动其他元素, 只需改变指针,效率高。

◦链表各个节点在内存中空间不要求连续，空间利用率高。

◦大小没有固定，拓展很灵活。

•缺点：

◦查找数据时效率低，因为不具有随机访问性。

（3）链表的种类？

•有单链表、双向链表、循环单链表、循环双链表等等。

（4）双向链表（java.util中的LinkedList就是双链的一种实现）

双向链表(双链表)是链表的一种。和单链表一样，双链表也是由节点组成，它的每个数据结点中都有两个指针，分别指向直接后继和直接前驱。所以，从双向链表中的任意一个结点开始，都可以很方便地访问它的前驱结点和后继结点。一般我们都构造双向循环链表。

•优点：对于链表中一个给的的结点，可以从两个方向进行操，双向链表相对单链表更适合元素的查询工作。

•缺点： ◦每个结点需要再添加一个额外的指针，因此需要更多的空间开销。

◦结点的插入或者删除更加费时。

# 6.@ ArrayList与LinkedList区别

0.从线程安全的方面来说，这两个都不能线程安全的；

1.从数据结构来说，ArrayList 的底层使用的是object数组，而LinkedList底层是双向链表数据结构。

2.对于随机访问，ArrayList优于LinkedList。由于ArrayList是底层是数组，所以可以快速随机访问，通过坐标就可以快速定位，而LinkedList 则不可以。

3. 对于插入和删除操作，LinkedList优于ArrayList。在添加和删除方面，ArrayList 是数组结构，如果添加/删除的是第i个数据，那么数组中的i之后的所有数据都会向前后移动。而链表结构则不会。所以LinkedList做添加删除快速。

4.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。

LinkedList中的每一个元素中还存储了它的前一个元素的索引和后一个元素的索引。

ArrayList会在尾部预留一定的容量空间，因为如果做添加操作的时候，超出当前数组最大值时，会添加50%的空间。LinkedList的空间花费则体现在它的每一个元素都需要消耗比ArrayList更多的空间（因为要存放直接后继和直接前驱以及数据ArrayList使用一个内置的数组来存储元素，这个数组的起始容量是10，当数组需要增长时，新的容量按如下公式获得：新容量 = 旧容量\*1.5 + 1，也就是说每一次容量大概会增长50%

5.无论ArrayList还是LinkedList，遍历建议使用foreach，尤其是数据量较大时LinkedList避免使用get遍历。LinkedList是链表结构，使用迭代器遍历快，使用for循环慢很多。

6.List使用首选ArrayList。对于个别插入删除非常多的可以使用LinkedList。

# Java实现栈、队列

java中通过数组、链表，或者LinkedList都可以实现栈的功能。

JDK的官方文档不建议使用Stack实现栈的功能，转而使用Deque接口的ArrayDeque实现栈的功能。

当初 JDK1.0 在开发时，可能为了快速的推出一些基本的数据结构操作，所以推出了一些比较粗糙的类。比如，Vector、Stack、Hashtable等。这些类中的一些方法加上了 synchronized 关键字，容易给一些初级程序员在使用上造成一些误解！而且在之前的几个版本中，性能还不怎么好。

基于 Vector 实现的栈 Stack。底层实际上还是数组，所以还是存在需要扩容。Vector 是由数组实现的集合类，它包含了大量集合处理的方法。而 Stack 之所以继承 Vector，是为了复用 Vector 中的方法，来实现进栈（push）、出栈(pop)等操作。这里就是 Stack 设计不好的地方，既然只是为了实现栈，不用链表来单独实现，而是为了复用简单的方法而迫使它继承 Vector，Stack 和 Vector 本来是毫无关系的。这使得 Stack 在基于数组实现上效率受影响，另外因为继承 Vector 类，Stack 可以复用 Vector 大量方法，这使得 Stack 在设计上不严谨。

Java 提供了 Deuqe。Deque 是继承自 Queue，而 Stack 是继承自 Vector。Java 中的 Deuqe，即“double ended queue”的缩写，是 Java 中的双端队列集合类型。Deque 具备普通队列 FIFO 的功能，同时它也具备了 Stack 的 LIFO 功能，并且保留了 push 和 pop 函数，所以使用起来应该是一点障碍都没有。

ArrayDeque 是 Deque 接口的一种具体实现，是依赖于可变数组来实现的。ArrayDeque 没有容量限制，可根据需求自动进行扩容。ArrayDeque 可以作为栈来使用，效率要高于 Stack。ArrayDeque 也可以作为队列来使用，效率相较于基于双向链表的 LinkedList 也要更好一些。注意，ArrayDeque 不支持为 null 的元素。

# 说说List,Set,Map三者的区别？

•List(对付顺序的好帮手)： List接口存储一组不唯一（可以有多个元素引用相同的对象），有序的对象

•Set(注重独一无二的性质): 不允许重复的集合。不会有多个元素引用相同的对象。

•Map(用Key来搜索的专家): 使用键值对存储。Map会维护与Key有关联的值。两个Key可以引用相同的对象，但Key不能重复，典型的Key是String类型，但也可以是任何对象。

# @ Arraylist 与 LinkedList 区别?

•1. 是否保证线程安全： ArrayList 和 LinkedList 都是不同步的，也就是不保证线程安全；

•2. 底层数据结构： Arraylist 底层使用的是 Object 数组；LinkedList 底层使用的是 双向链表 数据结构（JDK1.6之前为循环链表，JDK1.7取消了循环。注意双向链表和双向循环链表的区别，下面有介绍到！）

•3. 插入和删除是否受元素位置的影响： ① ArrayList 采用数组存储，所以插入和删除元素的时间复杂度受元素位置的影响。 比如：执行add(E e) 方法的时候， ArrayList 会默认在将指定的元素追加到此列表的末尾，这种情况时间复杂度就是O(1)。但是如果要在指定位置 i 插入和删除元素的话（add(int index, E element) ）时间复杂度就为 O(n-i)。因为在进行上述操作的时候集合中第 i 和第 i 个元素之后的(n-i)个元素都要执行向后位/向前移一位的操作。 ② LinkedList 采用链表存储，所以插入，删除元素时间复杂度不受元素位置的影响，都是近似 O（1）而数组为近似 O（n）。

•4. 是否支持快速随机访问： LinkedList 不支持高效的随机元素访问，而 ArrayList 支持。快速随机访问就是通过元素的序号快速获取元素对象(对应于get(int index) 方法)。

•5. 内存空间占用： ArrayList的空 间浪费主要体现在在list列表的结尾会预留一定的容量空间，而LinkedList的空间花费则体现在它的每一个元素都需要消耗比ArrayList更多的空间（因为要存放直接后继和直接前驱以及数据）。

　　1.ArrayList是实现了基于动态数组的数据结构，LinkedList基于链表的数据结构。

　　2.对于随机访问get和set，ArrayList觉得优于LinkedList，因为LinkedList要移动指针。

　　3.对于新增和删除操作add和remove，LinedList比较占优势，因为ArrayList要移动数据。

尽量避免同时遍历和删除集合。因为这会改变集合的大小；

# ArrayList 与 Vector 区别呢?为什么要用Arraylist取代Vector呢？

Vector类的所有方法都是同步的。可以由两个线程安全地访问一个Vector对象、但是一个线程访问Vector的话代码要在同步操作上耗费大量的时间。

Arraylist不是同步的，所以在不需要保证线程安全时建议使用Arraylist。

# @ HashSet与TreeSet与LinkedHashSet对比

HashSet不能保证元素的排列顺序，顺序有可能发生变化，不是同步的，集合元素可以是null,但只能放入一个null

TreeSet是SortedSet接口的唯一实现类，TreeSet可以确保集合元素处于排序状态。TreeSet支持两种排序方式，自然排序 和定制排序，其中自然排序为默认的排序方式。向 TreeSet中加入的应该是同一个类的对象。

TreeSet判断两个对象不相等的方式是两个对象通过equals方法返回false，或者通过CompareTo方法比较没有返回0

自然排序

自然排序使用要排序元素的CompareTo（Object obj）方法来比较元素之间大小关系，然后将元素按照升序排列。

定制排序

自然排序是根据集合元素的大小，以升序排列，如果要定制排序，应该使用Comparator接口，实现 int compare(To1,To2)方法

LinkedHashSet集合同样是根据元素的hashCode值来决定元素的存储位置，但是它同时使用链表维护元素的次序。这样使得元素看起 来像是以插入顺 序保存的，也就是说，当遍历该集合时候，LinkedHashSet将会以元素的添加顺序访问集合的元素。

LinkedHashSet在迭代访问Set中的全部元素时，性能比HashSet好，但是插入时性能稍微逊色于HashSet。

# @ LinkedHashMap和HashMap，TreeMap对比

Hashtable与 HashMap类似,它继承自Dictionary类，不同的是:它不允许记录的键或者值为空;它支持线程的同步，即任一时刻只有一个线程能写Hashtable,因此也导致了 Hashtable在写入时会比较慢。

Hashmap 是一个最常用的Map,它根据键的HashCode 值存储数据,根据键可以直接获取它的值，具有很快的访问速度，遍历时，取得数据的顺序是完全随机的。

LinkedHashMap保存了记录的插入顺序，在用Iterator遍历LinkedHashMap时，先得到的记录肯定是先插入的.也可以在构造时用带参数，按照应用次数排序。在遍历的时候会比HashMap慢，不过有种情况例外，当HashMap容量很大，实际数据较少时，遍历起来可能会比LinkedHashMap慢，因为LinkedHashMap的遍历速度只和实际数据有关，和容量无关，而HashMap的遍历速度和他的容量有关。

TreeMap实现SortMap接口，能够把它保存的记录根据键排序,默认是按键值的升序排序，也可以指定排序的比较器，当用Iterator 遍历TreeMap时，得到的记录是排过序的。

我们用的最多的是HashMap,HashMap里面存入的键值对在取出的时候是随机的,在Map 中插入、删除和定位元素，HashMap 是最好的选择。

TreeMap取出来的是排序后的键值对。但如果您要按自然顺序或自定义顺序遍历键，那么TreeMap会更好。

LinkedHashMap 是HashMap的一个子类，如果需要输出的顺序和输入的相同,那么用LinkedHashMap可以实现,它还可以按读取顺序来排列，像连接池中可以应用。

# HashMap 和 Hashtable 的区别

1.线程是否安全： HashMap 是非线程安全的，HashTable 是线程安全的；HashTable 内部的方法基本都经过synchronized 修饰。（如果你要保证线程安全的话就使用 ConcurrentHashMap 吧！）；

2.效率： 因为线程安全的问题，HashMap 要比 HashTable 效率高一点。另外，HashTable 基本被淘汰，不要在代码中使用它；

3.对Null key 和Null value的支持： HashMap 中，null 可以作为键，这样的键只有一个，可以有一个或多个键所对应的值为 null。。但是在 HashTable 中 put 进的键值只要有一个 null，直接抛出 NullPointerException。

4.初始容量大小和每次扩充容量大小的不同 ： ①创建时如果不指定容量初始值，Hashtable 默认的初始大小为11，之后每次扩充，容量变为原来的2n+1。HashMap 默认的初始化大小为16。之后每次扩充，容量变为原来的2倍。②创建时如果给定了容量初始值，那么 Hashtable 会直接使用你给定的大小，而 HashMap 会将其扩充为2的幂次方大小（HashMap 中的tableSizeFor()方法保证，下面给出了源代码）。也就是说 HashMap 总是使用2的幂作为哈希表的大小,后面会介绍到为什么是2的幂次方。

5.底层数据结构： JDK1.8 以后的 HashMap 在解决哈希冲突时有了较大的变化，当链表长度大于阈值（默认为8）时，将链表转化为红黑树，以减少搜索时间。Hashtable 没有这样的机制。

# @ HashMap 和 HashSet区别

如果你看过 HashSet 源码的话就应该知道：HashSet 底层就是基于 HashMap 实现的。（HashSet 的源码非常非常少，因为除了 clone() 、writeObject()、readObject()是 HashSet 自己不得不实现之外，其他方法都是直接调用 HashMap 中的方法。

HashSet类内部持有一个HashMap对象，对元素的操作其实都是调用HashMap的接口，元素作为Key，对应的Value是一个固定的Object，即map.put(key, PRESENT)新增。

private transient HashMap<E,Object> map;

// Dummy value to associate with an Object in the backing Map

private static final Object PRESENT = new Object();



# HashSet如何检查重复

当你把对象加入HashSet时，HashSet会先计算对象的hashcode值来判断对象加入的位置，同时也会与其他加入的对象的hashcode值作比较，如果没有相符的hashcode，HashSet会假设对象没有重复出现。但是如果发现有相同hashcode值的对象，这时会调用equals（）方法来检查hashcode相等的对象是否真的相同。如果两者相同，HashSet就不会让加入操作成功。（摘自我的Java启蒙书《Head fist java》第二版）

hashCode（）与equals（）的相关规定：

1.如果两个对象相等，则hashcode一定也是相同的

2.两个对象相等,对两个equals方法返回true

3.两个对象有相同的hashcode值，它们也不一定是相等的

4.综上，equals方法被覆盖过，则hashCode方法也必须被覆盖

5.hashCode()的默认行为是对堆上的对象产生独特值。如果没有重写hashCode()，则该class的两个对象无论如何都不会相等（即使这两个对象指向相同的数据）。

# comparable 和 Comparator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，当我们需要对某一个集合实现两种排序方式，比如一个song对象中的歌名和歌手名分别采用一种排序方法的话，我们可以重写compareTo()方法和使用自制的Comparator方法或者以两个Comparator来实现歌名排序和歌星名排序，第二种代表我们只能使用两个参数版的 Collections.sort().