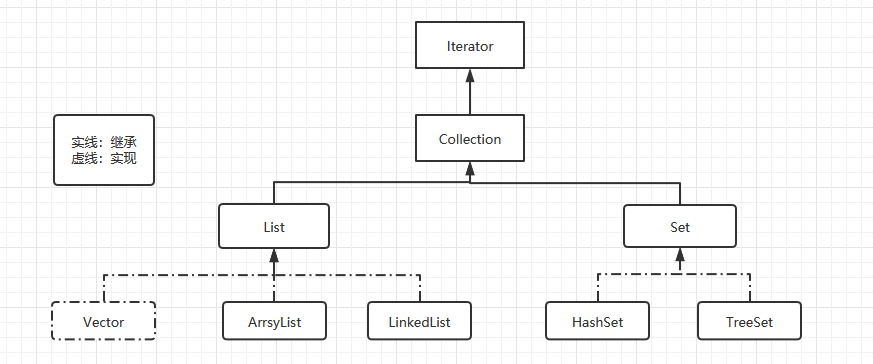
集合分析；

集合分为两类：

Collection(接口) 继承 Iterator(接口)



凡是继承了iterator接口的，都能通过iterator()方法获取迭代器，进而能通过迭代器获取数据。

Iterator iterator = arrayList.iterator();  
while (iterator.hasNext()) {  
 Object next = iterator.next();  
}

List:

1. ArrayList:
   1. 元素有序且可以重复，会记录添加的顺序。
   2. 底层使用的是动态数组，所以相应的查询和修改会比较快。
   3. 在初始化 new ArrayList();的时候，其内是个空数组，当添加第一个元素是，数组会进行扩容为10。

int newCapacity = oldCapacity + (oldCapacity >> 1);//扩容算法

除去第一次，后续扩容后的容量是上一次容量的1.5倍

**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 实现了RandmoAccess 接口，所以有随机访问功能。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)。

实现了该接口的list集合，在使用for循环获取数据时，会优于使用迭代器获取数据。

1. LinkedList:

2.1、元素有序且可以重复。

2.2、底层使用的是链表，所以增加和删除会比较快。

private static class Node<E> { //链表内的节点对象  
 E item;  
 Node<E> next; //存放上一个节点的地址  
 Node<E> prev; //存放下一个节点的地址

1. Vector:

3.1、元素有序且可以重复。

3.2、底层使用的是动态数组，与ArrayList类似。

3.3、在初始化的时候，调用this(10),调用自身带参数的构造器，创建长度为10的Object数组。 capacityIncrement:指定扩容大小，默认是0

int newCapacity = oldCapacity + ((capacityIncrement > 0) ?  
capacityIncrement : oldCapacity); //扩容后的容量是上一次容量的2倍

3.4、其内方法大都是使用synchronized修饰的，使用该集合是是线程安全的。所以相应的性能偏低。

public synchronized boolean add(E e) {  
 modCount++;  
 ensureCapacityHelper(elementCount + 1);  
 elementData[elementCount++] = e;  
 return true;  
 }

list集合的选择：

ArrayList:当不需要线程安全，且数据多以查询或者修改为主，则适合用ArrayList进行存放。

LinkedList:当不需要线程安全，且数据多以增删为主，则适合用LinkedList进行存放。

Vector:当需要线程安全，则选他。

学完线程后，对比下：Collections.synchronizedList(); 和 vector 哪个效率更高。

set:

1. HashSet:
   1. 元素无序且不能重复，不会记录添加的位置。
   2. 初始化时，实际是调用的HashMap的方法。

public HashSet() {  
 map = new HashMap<>();  
}

1. LinkedHashSet: 继承 HashSet

底层调用的是：map = new LinkedHashMap<>(initialCapacity, loadFactor);

LinkedHashMap extends HashMap,所以底层机制都一样

static final int ***DEFAULT\_INITIAL\_CAPACITY*** = 1 << 4 //16

初始化后容量则为为：16

相应的扩容机制也一样，2倍扩容， 扩容因子0.75

2.1、底层是数组+双向链表，数组里面存放的应该是Node对象，实际存放的是Entry

static class Entry<K,V> extends HashMap.Node<K,V> {  
 Entry<K,V> before, after;  
 Entry(int hash, K key, V value, Node<K,V> next) {  
 super(hash, key, value, next);  
 }  
}

linkedHashMap.put("1", 2);  
 */\*  
 LinkedHashMap.Entry<K,V> p = new LinkedHashMap.Entry<K,V>(hash, key, value, e);  
 //所以linkedListMap在hashMap$Node数组中存放的是LinkedHashMap$Entry对象  
 linkNodeLast(p);  
 private void linkNodeLast(LinkedHashMap.Entry<K,V> p) {  
 LinkedHashMap.Entry<K,V> last = tail;  
 tail = p;  
 if (last == null)  
 head = p;  
 else {  
 p.before = last;  
 last.after = p;  
 }  
}*

1. TreeSet:

3.1、 new TreeSet(); 实际是调用TreeMap();

public TreeSet() {  
 this(new TreeMap<E,Object>());  
}

3.2、treeSet添加元素

public class TreeSetSource {  
 public static void main(String[] args) {  
 TreeSet treeSet = new TreeSet();

/\*

TreeSet存入的数据是有顺序的，那么顺序什么呢？其实底层添加元素时，是通过比较器内的比较规则进行比较的

\*/  
 treeSet.add("1");   
 treeSet.add("2"); //第二次添加的时候就是调动的String的compareTo()方法  
 Person p1 = new Person("小红");  
 Person p2 = new Person("小红");  
 treeSet.add(p1); // 由于p1对象没有实现Comparable接口  
 treeSet.add(p2);  
 System.***out***.println(treeSet);  
 }  
}  
class Person{  
 String name;  
 public Person(String name) {  
 this.name = name;  
 }  
}



treeSet.add(p1);

底层在执行 add 方法时，都会将数据进行比较，实际会将数据强转成

(Comparable<? super K>)k1

Comparable类型的数据，如果没有实现该接口，则会报错。

TreeSet的有序与list的有序不一样:

TreeSet:对于添加的数据会调用比较器Comparable/ ComapraTO方法，按照比较器中的比较结果进行存储，例如Integer 实现了Comparable接口，所以：‘’

TreeSet.add(1), TreeSet.add(3), TreeSet.add(2) ，打印该TreeSet，输出结果为：123

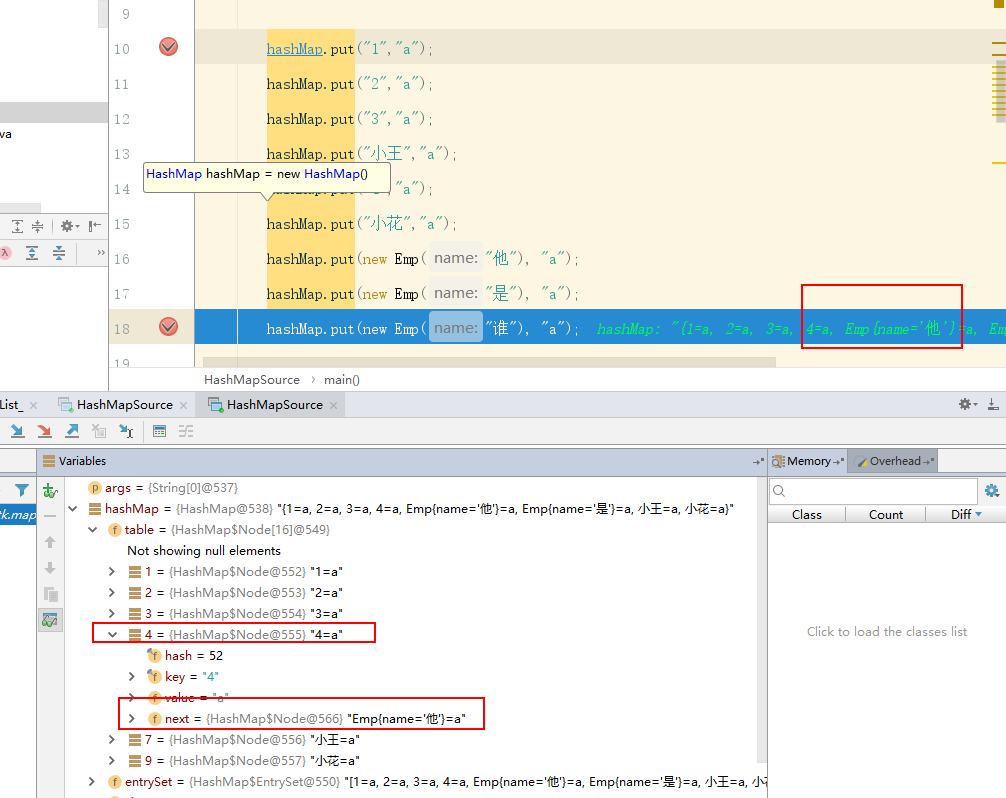
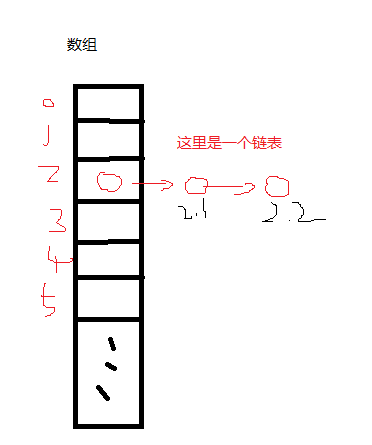
List: 的有序是记录添加顺序， 插入顺序为1 、3、 2 ，则输出的结果也为1、 3、 2

Map:

1. hashMap
   1. 元素存放方式：key-value(键值对)，其中key不能重复，存放是无序的。存放的顺序是通过计算存放的key的hash码通过计算得到的。虽然存放的顺序是计算的到的，但是数据的取出是有序的。从数组开始取，如果数组的该节点下有链表，则继续取链表中的数据，再取下一个索引的数据。

如下图，打印map时的取数顺序则是：0，1，2，2.1，2.2，3，4，5

e.hash & (n-1); //n：数组的长度



HashMap底层是通过：：数组+链表+红黑树实现的。初始化（new HashMap()）时，底层是构建的空数组，在put第一个元素时，会进行扩容，初始容量为16，扩容因子为0.75。

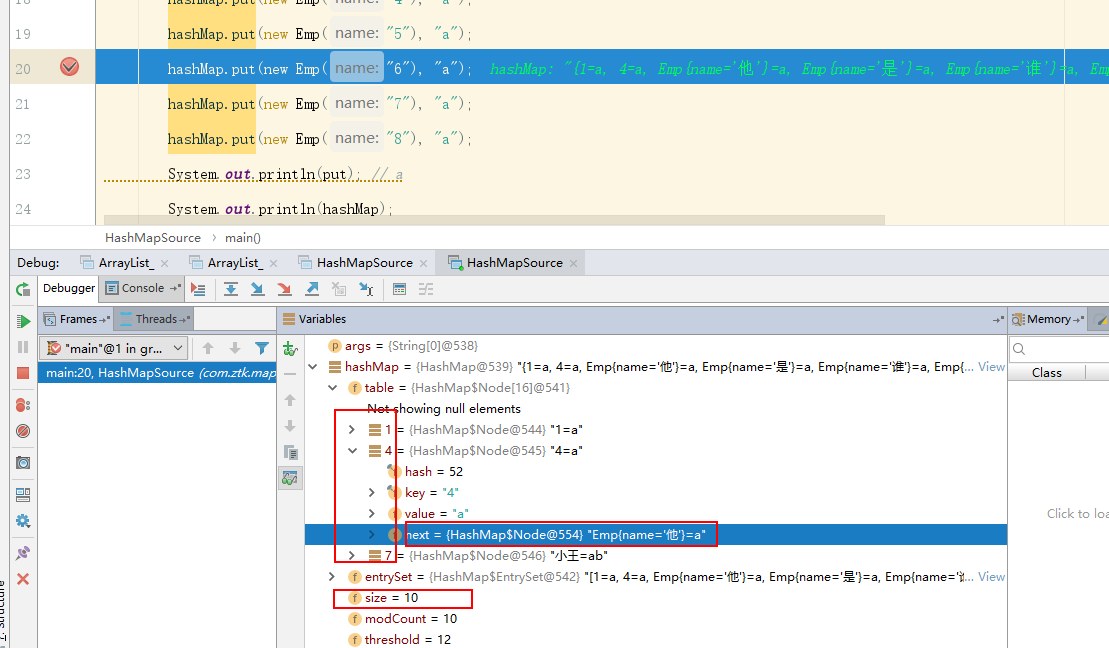
HashMap扩容的条件（以第一次put数据后的数组长16为例）：

1. Map中元素达到16\*0.75 = 12个，则会自动扩容，扩容后的长度是上次容量的2倍。（注：元素达到12个，是指put进去的所有元素，包含数组和链表中的元素）

扩容的算法： <<

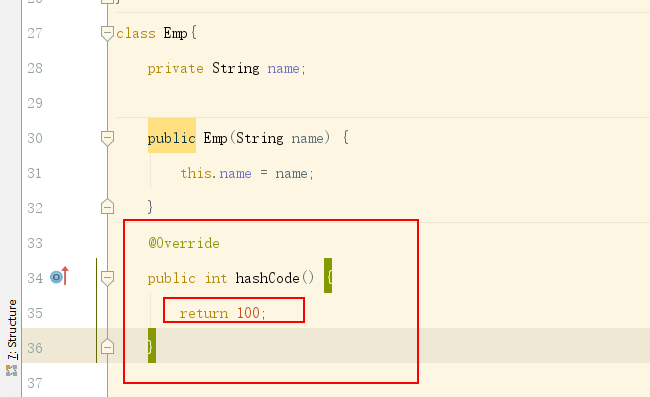
else if ((newCap = oldCap << 1) < ***MAXIMUM\_CAPACITY*** &&  
 oldCap >= ***DEFAULT\_INITIAL\_CAPACITY***)  
 newThr = oldThr << 1; *// double threshold*

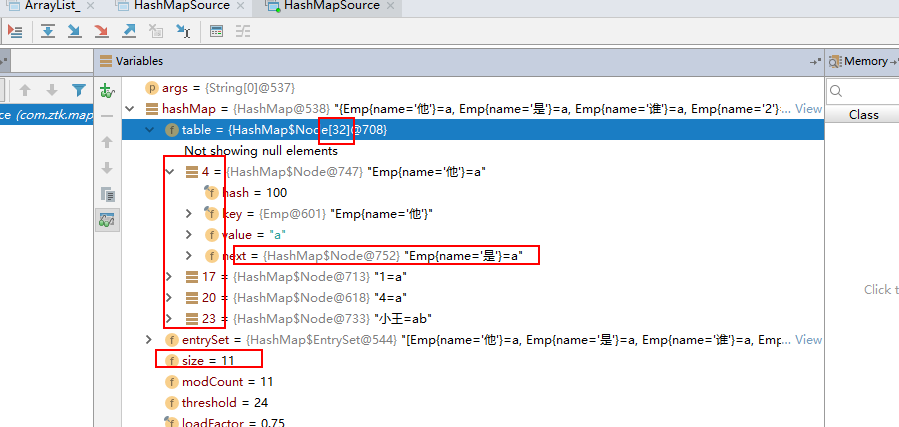
1. 当单条链表长度超过8，且数组长度不到64，会触发扩容。（当单条链表的长度达到8时，数组的长度还只有16，再往该链表后添加一个元素，则数组的长度就会变成32）



此时map中元素达到了10个，但是数组中被占用的索引只有3个，分别是1，4，7，可以看到4号索引位的next有存下一个节点的地址值，可以判断该链表的长度应该为10-2=8，此时再往该链表后面添加一个元素（

注：该条链表中存入的Emp类中重写了hashcode方法，所有的该对象返回的都是同一个hashcode,所以在添加时，才会在同一条链表上）





如上图，继续添加一个元素后，数组的长度变成了32，并且旧数组中存放的数据的索引值与新数组中存放的索引值并不一致。

是因为添加元素时，元素存放的索引值是通过如下计算公式计算出来的：

newTab[e.hash & (newCap - 1)]

e.hash: 元素的hash值

newCap: 数组的长度

旧数组得到索引的计算方法：

e.hash & (16-1)

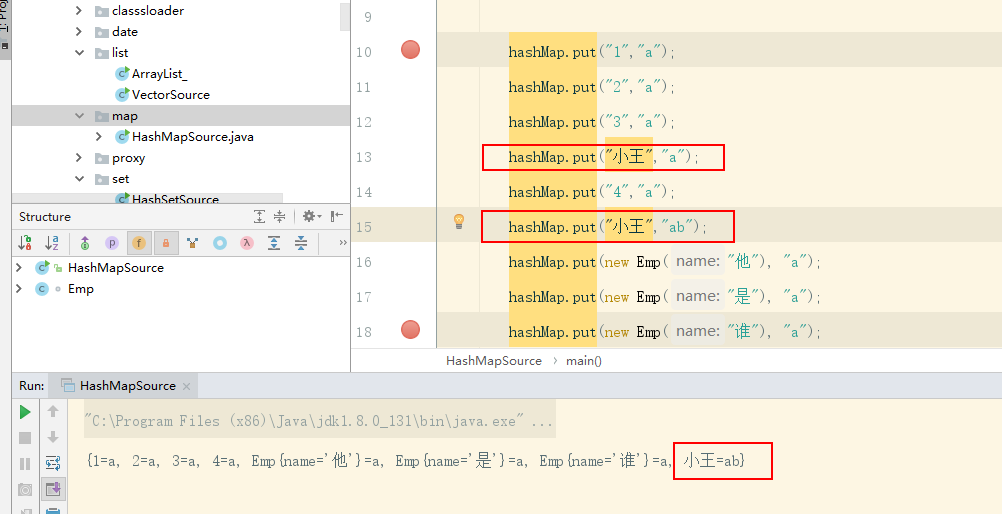
新数组得到索引的计算方法：

e.hasn & (32-1)

所以导致新数组存放的数据的索引值有旧数组的不一致

3、当单条链表的长度超过8，且数组的长度达到64，链表则会转为红黑树的结构，进行数据的存储。

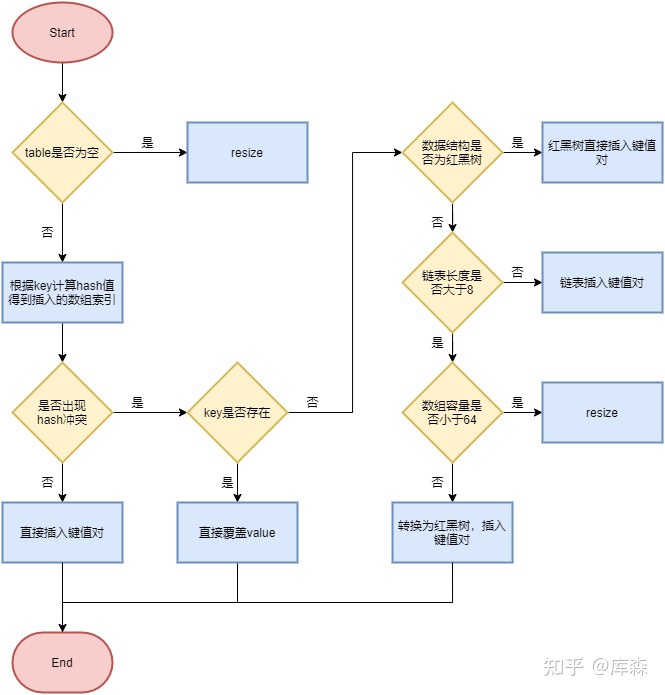
* 1. key有且只有一个为null,value存值无所谓，当key重复存值时，后一个key的value会替换掉前一个value的值。



**HashMap 的put方法流程？**

以JDK1.8为例，简要流程如下：

1. 首先根据 key 的值计算 hash 值，找到该元素在数组中存储的下标；
2. 如果数组是空的，则调用 resize 进行初始化；
3. 如果没有哈希冲突直接放在对应的数组下标里；
4. 如果冲突了，且 key 已经存在，就覆盖掉 value；
5. 如果冲突后，发现该节点是红黑树，就将这个节点挂在树上；
6. 如果冲突后是链表，判断该链表是否大于 8 ，如果大于 8 并且数组容量小于 64，就进行扩容；如果链表节点大于 8 并且数组的容量大于 64，则将这个结构转换为红黑树；否则，链表插入键值对，若 key 存在，就覆盖掉 value。



HashTable:

Key 和 value都不能为null

线程安全

初始容量是11，扩容是\*2+1

数组 + 链表

HashTable是较为远古的使用Hash算法的容器结构了，现在基本已被淘汰，单线程转为使用HashMap，多线程使用ConcurrentHashMap。

HashTable: 很多方法都是用synchronized修饰，但同时因为加锁导致并发效率低下，单线程环境效率也十分低

线程不安全

java.util.HashMap、java.util.ArrayList、java.util.LinkedList、java.util.HashSet等集合实现类都是线程不安全的，在多线程环境下使用的话，将会得到无法预期的结果。

遍历不安全

java.util.Hashtable、java.util.Vector等集合类在多线程环境下，如果只是调用put、get、remove等方法的话是能保证线程安全的，但如果进行遍历的话就无法保证线程安全了。这种情况也叫做“条件线程安全”。

线程安全

java.util.concurrent.ConcurrentHashMap、java.util.concurrent.CopyOnWriteArrayList、java.util.concurrent.ConcurrentArraySet等是线程安全的。

线程安全问题：

java类库里面提供了以下三个轮子可以实现线程安全的List，它们是 - Vector - Collections.synchronizedList - CopyOnWriteArrayList

本文简要的分析了下它们线程安全的实现机制并对它们的读，写，迭代性能进行了对比。

## Vector 从JDK1.0开始，`Vector`便存在JDK中，`Vector`是一个线程安全的列表，底层采用数组实现。其线程安全的实现方式非常粗暴：`Vector`大部分方法和`ArrayList`都是相同的，只是加上了`synchronized`关键字，这种方式严重影响效率，因此，不再推荐使用`Vector`了

Collections.synchronizedList 因为ArrayList不是线程安全的，JDK提供了一个`Collections.synchronizedList`静态方法将一个非线程安全的List(并不仅限ArrayList)包装为线程安全的List。使用方式如下： ```java List list = Collections.synchronizedList(new ArrayList()); ``` 根据文档，转换包装后的list可以实现add，remove，get等操作的线程安全性，但是对于迭代操作，`Collections.synchronizedList`并没有提供相关机制，所以迭代时需要对包装后的list（敲黑板，必须对包装后的list进行加锁，锁其他的不行）进行手动加锁，使用方式如下： ```java List list = Collections.synchronizedList(new ArrayList()); //必须对list进行加锁 synchronized (list) { Iterator i = list.iterator(); while (i.hasNext()) foo(i.next()); } ``` 这个地方要注意两个地方： 1. 迭代操作必须加锁，可以使用`synchronized`关键字修饰; 2. synchronized持有的监视器对象必须是`synchronized (list)`,即包装后的list,使用其他对象如`synchronized (new Object())`会使`add`,`remove`等方法与迭代方法使用的锁不一致，无法实现完全的线程安全性。 通过源码可知`Collections.synchronizedList`生成了特定同步的`SynchronizedCollection`，生成的集合每个同步操作都是持有`mutex`这个锁，所以再进行操作时就是线程安全的集合了。