**@ClassName** ListTest  
 \***@Description** |----List接口：存储有序的、可重复的数据 --->“动态数组”  
 \* \* |---ArrayList: 作为List主要实现类,线程不安全，效率高：底层使用Object[]存储  
 \* |---LinkedList：对于频繁的插入和删除操作使用此类效率比ArrayList高，底层使用的双向链表存储  
 \* |---Vector：作为List的古老实现类 线程安全，效率低，底层使用Object[]存储  
 \* 1.1 ArrayList的源码分析：jdk7 情况如下  
 \* ArrayList list = new ArrayList();//底层创建了长度是10的Object[]数组elementData.....  
 \* List.add(123);//elementData[0] = new Integer(123);  
 \* ....  
 \* List.add(11);//如果此次的添加导致底层elementData数组容量不够，则扩容。默认情况下扩容为原来容量的1.5倍，  
 \* 同时需要将原有的数组中的数据复制到新的数组中。  
 \* 结论：建议在开发中使用带参的构造器：ArrayList = new ArrayList(int capacity).  
 \* 1.2 在jdk8中ArrayList的变化  
 \* ArrayList list = new ArrayList();//底层创建Object[] elementData初始化为{} ，并没有创建长度为10的数组  
 \* list.add(123);//第一次调用add()时，底层才创建了长度为10的数组，并将数据123添加到elementData  
 \* ....的添加和扩容操作于jdk7相同  
 \* \*  
 \* 后续  
 \* 1.3：小结jdk7中的ArrayList的对象的创建类似于单例的饿汉式，而jdk8中类似于单例中的懒汉式  
 \*  
 \*  
 \* 2.LinkedList的源码分析：  
 \* LinkedList list = new LinkedList();内部声明了Node类型的first和Last属性，默认值为null，当我们通过list.add（123）；  
 \* //将123 封装到Node中，创建了Node对象。  
 \*  
 \* 其中，Node定义为：体现了LinkedList的双向链表的说法  
 \* private static class Node**<E>**{  
 \* E item;  
 \* Node**<E>** next;  
 \* Node**<E>** prev;  
 \*  
 \* Node(Node**<E>** prev, E element,Node**<E>** next){  
 \* this.item = element;  
 \* this.next = next;  
 \* this.prev = prev;  
 \* }  
 \* }  
 \*  
 \*  
 \* 3.Vector的源码分析：jdk7和jdk8中通过Vector()构造器创建对象时，底层都创建了长度为10的数组。  
 \* 在扩容方面，默认扩容为原来的数组长度的2倍。  
 \*  
 \*  
 \*  
 \* 面试题：ArrayList ,LinkedList ,Vector三者的异同  
 \* 同：三个类型都是实现List接口，存储数据的特点相同，存储有序的，可重复的数据  
 \* 不同：ArrayList:见上  
 \***@Author** HuangQingbin  
 \***@Date** 2021/6/6 17:19  
 \***@Version** 1.0  
 \*/  
  
*/\*  
ArrayList:线程不安全的两个原因  
们先来看看 ArrayList 的 add 操作源码。  
 public boolean add(E e) {  
 ensureCapacityInternal(size + 1);  
 elementData[size++] = e;  
 return true;  
 }  
ArrayList 的不安全主要体现在两个方面。  
 其一：  
elementData[size++] = e;  
不是一个原子操作，是分两步执行的。  
elementData[size] = e;  
size++;  
  
单线程执行这段代码完全没问题，可是到多线程环境下可能就有问题了。可能一个线程会覆盖另一个线程的值。  
  
 列表为空 size = 0。  
 线程 A 执行完 elementData[size] = e;之后挂起。A 把 "a" 放在了下标为 0 的位置。此时 size = 0。  
 线程 B 执行 elementData[size] = e; 因为此时 size = 0，所以 B 把 "b" 放在了下标为 0 的位置，于是刚好把 A 的数据给覆盖掉了。  
 线程 B 将 size 的值增加为 1。  
 线程 A 将 size 的值增加为 2。  
  
这样子，当线程 A 和线程 B 都执行完之后理想情况下应该是 "a" 在下标为 0 的位置，"b" 在标为 1 的位置。而实际情况确是下标为 0 的位置为 "b"，下标为 1 的位置啥也没有。  
 其二：  
ArrayList 默认数组大小为 10。假设现在已经添加进去 9 个元素了，size = 9。  
  
 线程 A 执行完 add 函数中的ensureCapacityInternal(size + 1)挂起了。  
 线程 B 开始执行，校验数组容量发现不需要扩容。于是把 "b" 放在了下标为 9 的位置，且 size 自增 1。此时 size = 10。  
 线程 A 接着执行，尝试把 "a" 放在下标为 10 的位置，因为 size = 10。但因为数组还没有扩容，最大的下标才为 9，所以会抛出数组越界异常 ArrayIndexOutOfBoundsException  
  
  
  
如何实现ArrayList线程安全  
1、使用synchronized关键字；  
  
2.使用Collections.synchronizedList();使用方法如下：  
  
假如你创建的代码如下：List<Map<String,Object>> data=new ArrayList<Map<String,Object>>();  
  
那么为了解决这个线程安全问题你可以这么使用Collections.synchronizedList()，如：  
  
List<Map<String,Object>> data=Collections.synchronizedList(new ArrayList<Map<String,Object>>());  
  
其他的都没变，使用的方法也几乎与ArrayList一样，大家可以参考下api文档；  
  
额外说下 ArrayList与LinkedList；这两个都是接口List下的一个实现，用法都一样，但用的场所的有点不同，  
ArrayList适合于进行大量的随机访问的情况下使用，LinkedList适合在表中进行插入、删除时使用，  
二者都是非线程安全，解决方法同上（为了避免线程安全，以上采取的方法，特别是第二种，其实是非常损耗性能的）。*

/\*\*  
 \***@ClassName** MapTest  
 \***@Description** \* \---Map:双列数据，存储key-value对的数据 ---类似于高中学的函数：y = f(x)  
 \* \---HashMap:作为Map的主要实现类不安全，效率高 存储null的Key和value  
 \* \---LinkedHashMap;保证在遍历map元素时，可以按照添加的顺序实现遍历。  
 \* 原因：在原有的HashMap底层结构基础上，添加了一对指针，指向前一个元素和后一个元素  
 \* 对于频繁的遍历操作，此类执行效率高于HashMap。  
 \* \---TreeMap;保证按照添加的key-value对进行排序，实现排序遍历 此时考虑key的自然排序和定制排序  
 \* 底层使用红黑树  
 \* \---Hashtable：作为古老实现类：线程安全，效率低，不能存储null的Key和values  
 \* \---Properites：常用来处理配置文件。key和value都是String类型  
 \*  
 \* HashMap的底层：数组+链表 (jdk7及之前）  
 \* 数组+链表+红黑树（jdk8)  
 \*  
 \* 面试题  
 \* 1.HashMap的底层实现原理？  
 \*  
 \* 2.HashMap 和 Hashtable的异同？  
 \* 3.currentHashMap 与 Hashtable的异同？（展示不讲）  
 \*  
 \* 二、Map结构的理解：  
 \* Map中的Key：无序的、不可重复的。使用Set存储所有的key --->所在类要重写equals()和hashCode()方法(以HashMap()为例)  
 \* Map中的value:无序的可重复的，使用Collection存储所有的value --->需要重写所在类的equals() 查询时候调用;  
 \* 一个键值对：key-value:构成一个Entry对象。  
 \* Map中的entry:无序的，不可重复的，使用Set存储所有的entry  
 \*  
 \* 三、HashMap的底层实现原理？ 以jdk7为例  
 \* HashMap map = new HashMap();  
 \* 在实例化以后，底层创建了长度是16的一维数组Entry[] table.  
 \* map.put(key1,value1);  
 \* 首先，调用key1所在类的hashCode()计算key1哈希值，此哈希值经过某种算法计算以后，得到在Entry数组中存放位置。  
 \* 如果此位置上的数据为空，此时的key1-value1添加成功。 ---情况1  
 \* 如果此位置上的数据不为空，(意味着此位置上存在一个或多个数据(以链表形式存在)),比较key1和已经存在的一个或多个数据  
 \* 的哈希值：  
 \* 如果key1的哈希值与已经存在的数据(key1-value2)哈希值都不相同，此时key1-value1添加成功。---情况2  
 \* 如果key1的哈希值和已经存在的某一个数据的哈希值相同，继续比较：调用key1所在类的equals()方法，比较：  
 \* 如果equals()返回false:此时key1-value1添加成功。---情况3  
 \* 如果equals()返回true：使用value1替换value2.  
 \* 补充：关于情况2和情况3：此时key1-value1和原来的数据以链表的形式存储。  
 \*  
 \* 在不断的添加过程中，会涉及到扩容问题，默认的扩容方式为原来的2倍，并将原来的数据复制到过来。  
 \*  
 \*  
 \* jdk8 相较于jdk7在底层实现方面的不同：  
 \* 1.new HashMap:底层没有创建一个长度为16的数组.  
 \* 2.jdk 8底层数组是：Node[],而非Entry[].  
 \* 3.首次调用put方法时，底层创建长度为16的数组  
 \* 4.jdk7底层结构只有： 数组+链表. jdk 8中底层结构： 数组+链表+红黑树。  
 \* 当数组的某一个索引位置上的元素以链表的形式存在的数据个数 > 8 且当前数据的数组长度 > 64,  
 \* 此时索引位置上的所有的数据改为使用红黑树存储  
 \*  
 \* BEFAULT\_INITAL\_CAPACITY: HashMap的默认值：16  
 \* DEFAULT\_LOAD\_FACTOR: HashMap的默认加载因子：0.75  
 \* \* threshold:扩容的临界值 = 容量 \* 填充因子：16 \* 0.75 => 12  
 \* \* TREEIFA\_THRESHOLD: Bucket中链表的长度大于该默认值，转化为红黑树：8  
 \* \* MIN\_TREEIFA\_CAPACITY: 桶中的Node被树化时最小的hash表的容量：64  
 \* \*  
 \*  
 \* HashMap底层是一个Entry数组，当发生hash冲突的时候，HashMap是采用链表的方式来解决的，  
 \* 在对应的数组位置存放链表的头结点。对链表而言，新加入的节点会从头结点加  
 \*  
 \* HashMap线程不安全：  
 \* ① 2个线程执行put操作： 线程1put()时，记录了头结点为node1，这时时间片用完，线程2put()，且把数据插在了链表的头部，  
 \* 完成put操作。线程1接着完成put()剩余的操作，这时新的头结点已经变了，但是线程1记录的旧的头结点，把数据插入到头结点，  
 \* 覆盖了线程2put的数据，导致线程不安全  
 \* ② A线程执行get获取数据  
 \* modelHashMap.get("1")  
 \* A线程获取的值本来应该是2，但是如果A线程在刚到达获取的动作还没执行的时候，  
 \* 线程执行的机会又跳到线程B，此时线程B又对modelHashMap赋值 如：modelHashMap.put("1","3");  
 \* 然后线程虚拟机又执行线程A，A取到的值为3，这样map中第一个存放的值 就会丢失  
 \*  
 \* ③ addEntry中当加入新的键值对后键值对总数量超过门限值的时候会调用一个resize操作  
 \* 当多个线程同时检测到总数量超过门限值的时候就会同时调用resize操作，各自生成新的数组并rehash后赋给该map底层的数组table，  
 \* 结果最终只有一个线程生成的新数组被赋给table变量，其他线程的均会丢失。而且当某些线程已经完成赋值而其他线程刚开始的时候，  
 \* 就会用已经被赋值的table作为原始数组，这样也会有问题。  
 \*  
 \* 注：当hash表中的负载因子达到负载极限的时候，hash表会自动成倍的增加容量（桶的数量），并将原有的对象  
 \* 重新的分配并加入新的桶内，这称为rehash  
 \*  
 \*  
 \* 如何实现HashMap的同步？  
 \*  
 \* 答：  
 \* 　　第一种方法：  
 \* 　　 直接使用Hashtable，但是当一个线程访问HashTable的同步方法时，其他线程如果也要访问同步方法，会被阻塞住。举个例子，当一个线程使用put方法时，另一个线程不但不可以使用put方法，连get方法都不可以，效率很低，现在基本不会选择它了。  
 \* 第二种方法： HashMap可以通过下面的语句进行同步，  
 \* Collections.synchronizeMap(hashMap);  
 \* 　　 HashMap可以通过Map m = Collections.synchronizedMap(new HashMap())来达到同步的效果。（从源码中看出 synchronizedMap()方法返回一个SynchronizedMap类的对象，  
 \* 而在SynchronizedMap类中使用了synchronized来保证对Map的操作是线程安全的，故效率其实也不高。）  
 \* 　　具体而言，该方法返回一个同步的Map，该Map封装了底层的HashMap的所有方法，使得底层的HashMap即使是在多线程的环境中也是安全的。  
 \* 　　第三种方法：  
 \* 　　 直接使用JDK 5 之后的 ConcurrentHashMap，如果使用Java 5或以上的话，请使用ConcurrentHashMap。  
 \* 　　Collections.synchronizeMap(hashMap); 又是如何保证了HashMap线程安全？  
 \*  
 \*  
 \* 四、LinkedHashMap的底层实现原理(了解)  
 \* LinkedHashMap继承HashMap，所以它是线程不安全的，但是它有序  
 \* LinkedHashMap底层数据结构为双向链表，能保证元素按照插入顺序访问，也能以访问顺序访问，可以用来实现LRU策略缓存。  
 \* LinkedHashMap继承HashMap并实现了HashMap中预留的钩子函数，因此不必重写HashMap的很多方法，设计非常巧妙。  
 \*  
 \* #1.LinkedHashMap默认容量为16，扩容因子默认为0.75，非同步，允许[key,value]为null。  
 \*  
 \* #2.LinkedHashMap底层数据结构为双向链表，可以看成是LinkedList+HashMap。  
 \*  
 \* #3.如果accessOrder为false，则可以按插入元素的顺序遍历元素，如果accessOrder为true，则可以按访问顺序遍历元素  
 \*  
 \* 五、ConcurrentHashMap  
 \* ConcurrentHashMap是HashMap的线程安全版本，内部使用了数组+链表+红黑树的结构来存储数据，相对于同样线程安全的Hashtable来说，  
 \* \* 它在效率方面有很大的提升，因此多线程环境下更多的是使用ConcurrentHashMap，因此有必要对其原理进行分析。  
 \* \* ConcurrentHashMap是HashMap的线程安全版本，底层数据结构为数组+链表+红黑树，默认容量16，线程同步，不允许[key,value]为null  
 \* \* 五、Map中定义的方法  
 \* \*  
 \*  
 \* 总结：常用方法  
 \* 添加：put(Object key,Object value)  
 \* \* 删除：remove(Object key)  
 \* \* 修改：put(Object key, Object value)  
 \* \* 查询：get(Object key)  
 \* \* 长度：size()  
 \* \* 遍历：keySet() / values() / entrySet()  
 \* \* \* \*  
 \*