集合框架

为什么使用集合

如果并不知道程序运行时会需要多少对象，或者需要更复杂的方式存储对象，那我们就可以使用Java的集合框架！

【重中之重】

Collection 接口存储一组不唯一，无序的对象

List 接口存储一组不唯一，有序的对象。

Set 接口存储一组唯一，无序的对象

Map 接口存储一组键值对象，提供key到value的映射

ArrayList实现了长度可变的数组，在内存中分配连续的空间。遍历元素和随机访问元素的效率比较 高

LinkedList采用链表存储方式。插入、删除元素时效率比较高

HashSet:采用哈希算法实现的Set

HashSet的底层是用HashMap实现的，因此查询效率较高，由于采用hashCode算法直接确定 元素的内存地址，增删效率也挺高的。

List

1、ArrayList概述

1. ArrayList是可以动态增长和缩减的索引序列，它是基于数组实现的List类。
2. 该类封装了一个动态再分配的Object[]数组，每一个类对象都有一个capacity【容量】属性，表示它们所封装的Object[]数组的长度，当向ArrayList中添加元素时，该属性值会自动增加。如果想ArrayList中添加大量元素，可使用ensureCapacity方法一次性增加capacity，可以减少增加重分配的次数提高性能。
3. ArrayList的用法和Vector向类似，但是Vector是一个较老的集合，具有很多缺点，不建议使用。

另外，ArrayList和Vector的区别是：ArrayList是线程不安全的，当多条线程访问同一个ArrayList集合时，程序需要手动保证该集合的同步性，而Vector则是线程安全的。

2、ArrayList的数据结构

底层的数据结构就是数组，数组元素类型为Object类型，即可以存放所有类型数据。我们对 ArrayList类的实例的所有的操作底层都是基于数组的。

3、ArrayList源码分析

1、ArrayList实现了哪些接口

RandomAccess接口：

这个是一个标记性接口，通过查看api文档，它的作用就是用来快速随机存取， 有关效率的问题，在实现了该接口的话，那么使用普通的for循环来遍历，性能更高，例如ArrayList。而没有实现该接口的话，使用Iterator来迭代，这样性能更高，例如linkedList。所以这个标记性只是为了让我们知道我们用什么样的方式去获取数据性能更好。

Cloneable接口：实现了该接口，就可以使用Object.Clone()方法了。

Serializable接口：实现该序列化接口，表明该类可以被序列化，什么是序列化？简单的说，就是能够从类变成字节流传输，然后还能从字节流变成原来的类。

2、类中的属性

3、构造方法

1. 无参构造方法

```
/*
Constructs an empty list with an initial capacity of ten.
这里就说明了默认会给10的大小，所以说一开始arrayList的容量是10.
*/
//ArrayList中储存数据的其实就是一个数组，这个数组就是elementData.
public ArrayList() {
    super(); //调用父类中的无参构造方法，父类中的是个空的构造方法
    this.elementData = EMPTY_ELEMENTDATA;
//EMPTY_ELEMENTDATA：是个空的Object[]， 将elementData初始化，elementData也是个
Object[]类型。空的Object[]会给默认大小10，等会会解释什么时候赋值的。
}
```

2. 有参构造方法 1

```
/*
Constructs an empty list with the specified initial capacity.
构造具有指定初始容量的空列表。
@param initialCapacity the initial capacity of the list
初始容量列表的初始容量
@throws IllegalArgumentException if the specified initial capacity is
negative
如果指定的初始容量为负，则为IllegalArgumentException
*/
public ArrayList(int initialCapacity) {
if (initialCapacity > 0) {
    ////将自定义的容量大小当成初始化 initialCapacity 的大小
    this.elementData = new Object[initialCapacity];
    } else if (initialCapacity == 0) {
        this.elementData = EMPTY_ELEMENTDATA; //等同于无参构造方法
        } else {
        ////判断如果自定义大小的容量小于0，则报下面这个非法数据异常
        throw new IllegalArgumentException("Illegal Capacity:
"+initialCapacity);
        }
}
```

3. 有参构造方法 2

```
/*
Constructs a list containing the elements of the specified collection,
in the order they are returned by the collection's iterator.
按照集合迭代器返回元素的顺序构造包含指定集合的元素的列表。
@param c the collection whose elements are to be placed into this list
@throws NullPointerException if the specified collection is null
```

```
*/
public ArrayList(Collection<? extends E> c) {
    elementData = c.toArray(); //转换为数组
    //每个集合的toarray()的实现方法不一样，所以需要判断一下，如果不是Object[].class类型，那么久需要使用ArrayList中的方法去改造一下。
    if ((size = elementData.length) != 0) {
    // c.toArray might (incorrectly) not return Object[] (see 6260652)
        if (elementData.getClass() != Object[].class)
            elementData = Arrays.copyOf(elementData, size, Object[].class);
    } else {
    // replace with empty array.
    this.elementData = EMPTY_ELEMENTDATA;
    }
}
```

【总结】

ArrayList的构造方法就做一件事情，就是初始化一下储存数据的容器，其实本质上就是一个数组，在其中就叫elementData

4、核心方法-add

1、boolean add(E)

```
/**
* Appends the specified element to the end of this list.
* 添加一个特定的元素到list的末尾。
* @param e element to be appended to this list
* @return <tt>true</tt> (as specified by {@link Collection#add})
*/
public boolean add(E e) {
//确定内部容量是否够了，size是数组中数据的个数，因为要添加一个元素，所以size+1，先判断size+1的这个个数数组能否放得下，就在这个方法中去判断是否数组.length是否够用了。
    ensureCapacityInternal(size + 1); // Increments modCount!!
    elementData[size++] = e; //在数据中正确的位置上放上元素e，并且size++
    return true;
}

【分析：ensureCapacityInternal(xxx); 确定内部容量的方法】

private void ensureCapacityInternal(int minCapacity) {
    ensureExplicitCapacity(calculateCapacity(elementData, minCapacity));
}

private static int calculateCapacity(Object[] elementData, int minCapacity)
{
    //看，判断初始化的elementData是不是空的数组，也就是没有长度
    if (elementData == DEFAULTCAPACITY_EMPTY_ELEMENTDATA) {
    //因为如果是空的话，minCapacity=size+1；其实就是等于1，空的数组没有长度就存放不了，所以就将minCapacity变成10，也就是默认大小，但是在这里，还没有真正的初始化这个elementData的大小。
    return Math.max(DEFAULT_CAPACITY, minCapacity);
}
    //确认实际的容量，上面只是将minCapacity=10，这个方法就是真正的判断elementData是否够用
    return minCapacity;
}
```

```
private void ensureExplicitCapacity(int minCapacity) {
    modCount++;
// overflow-conscious code
//minCapacity如果大于了实际elementData的长度，那么就说明elementData数组的长度不够用，不够用那么就要增加elementData的length。这里有的同学就会模糊minCapacity到底是什么呢，这里给你们分析一下

/*第一种情况：由于elementData初始化时是空的数组，那么第一次add的时候，minCapacity=size+1；也就minCapacity=1，在上一个方法(确定内部容量ensureCapacityInternal)就会判断出是空的数组，就会给将minCapacity=10，到这一步为止，还没有改变elementData的大小。

第二种情况：elementData不是空的数组了，那么在add的时候，minCapacity=size+1；也就是minCapacity代表着elementData中增加之后的实际数据个数，拿着它判断elementData的length是否够用，如果length不够用，那么肯定要扩大容量，不然增加的这个元素就会溢出。*/

    if (minCapacity - elementData.length > 0)
    grow(minCapacity);
}


//arrayList核心的方法，能扩展数组大小的真正秘密。
private void grow(int minCapacity) {
    // overflow-conscious code
    //将扩充前的elementData大小给oldCapacity
    int oldCapacity = elementData.length;
    //newCapacity就是1.5倍的oldCapacity
    int newCapacity = oldCapacity + (oldCapacity >> 1);
    //这句话就是适应于elementData就空数组的时候，length=0，那么oldCapacity=0，newCapacity=0，所以这个判断成立，在这里就是真正的初始化elementData的大小了，就是为10.前面的工作都是准备工作。
    if (newCapacity - minCapacity < 0)
        newCapacity = minCapacity;
    //如果newCapacity超过了最大的容量限制，就调用hugeCapacity，也就是将能给的最大值给newCapacity
    if (newCapacity - MAX_ARRAY_SIZE > 0)
        newCapacity = hugeCapacity(minCapacity);
    // minCapacity is usually close to size, so this is a win:
    //新的容量大小已经确定好了，就copy数组，改变容量大小咯。
    elementData = Arrays.copyOf(elementData, newCapacity);
}


//这个就是上面用到的方法，很简单，就是用来赋最大值。
private static int hugeCapacity(int minCapacity) {
    if (minCapacity < 0) // overflow
    throw new OutOfMemoryError();
    //如果minCapacity都大于MAX_ARRAY_SIZE，那么就Integer.MAX_VALUE返回，反之将MAX_ARRAY_SIZE返回。因为maxCapacity是三倍的minCapacity，可能扩充的太大了，就用minCapacity来判断了。
    //Integer.MAX_VALUE:2147483647 MAX_ARRAY_SIZE：2147483639 也就是说最大也就能给到第一个数值。还是超过了这个限制，就要溢出了。相当于arraylist给了两层防护。
return (minCapacity > MAX_ARRAY_SIZE) ?Integer.MAX_VALUE :MAX_ARRAY_SIZE;
}
```

2、void add(int，E)

```
public void add(int index, E element) {
    //检查index也就是插入的位置是否合理。
    rangeCheckForAdd(index);
    ensureCapacityInternal(size + 1); // Increments modCount!!
    //这个方法就是用来在插入元素之后，要将index之后的元素都往后移一位，
    System.arraycopy(elementData, index, elementData, index + 1,size - index);
    //在目标位置上存放元素
    elementData[index] = element;
    size++;
}
```

【分析：rangeCheckForAdd(index)】

```
private void rangeCheckForAdd(int index) {
    //插入的位置肯定不能大于size 和小于0
    if (index > size || index < 0)
    //如果是，就报这个越界异常
    throw new IndexOutOfBoundsException(outOfBoundsMsg(index));
}
```

【System.arraycopy(...)：就是将elementData在插入位置后的所有元素，往后面移一位.】

```
public static void arraycopy(Object src,
int srcPos,
Object dest,
int destPos,
int length)
```

src：源对象
srcPos：源对象对象的起始位置
dest：目标对象
destPost：目标对象的起始位置
length：从起始位置往后复制的长度。

【总结】

正常情况下会扩容1.5倍，特殊情况下（新扩展数组大小已经达到了最大值）则只取最大值。

5、核心方法-remove

其实这几个删除方法都是类似的。我们选择几个讲，其中fastRemove(int)方法是private的，是提供给remove(Object)这个方法用的。

1、remove(int):

通过删除指定位置上的元素

```
public E remove(int index) {
    rangeCheck(index);//检查index的合理性
    modCount++;//这个作用很多，比如用来检测快速失败的一种标志。
    E oldValue = elementData(index);//通过索引直接找到该元素
    int numMoved = size - index - 1;//计算要移动的位数。
    if (numMoved > 0)
    //这个方法也已经解释过了，就是用来移动元素的。
    System.arraycopy(elementData, index+1, elementData, index,numMoved);
    //将--size上的位置赋值为null，让gc(垃圾回收机制)更快的回收它。
    elementData[--size] = null; // clear to let GC do its work
    //返回删除的元素。
    return oldValue;
}
```

2、remove(Object):

这个方法可以看出来，arrayList是可以存放null值得。

```
//感觉这个不怎么要分析吧，都看得懂，就是通过元素来删除该元素，就依次遍历，如果有这个元素，就将该元素的索引传给fastRemove(index)，使用这个方法来删除该元素，
//fastRemove(index)方法的内部跟remove(index)的实现几乎一样，这里最主要是知道arrayList可以存储null值
public boolean remove(Object o) {
    if (o == null) {
        for (int index = 0; index < size; index++)
            if (elementData[index] == null) {
                fastRemove(index);
                return true;
            }
    } else {
        for (int index = 0; index < size; index++)
            if (o.equals(elementData[index])) {
                fastRemove(index);
                return true;
            }
    }
    return false;
}
```

3、clear():

将elementData中每个元素都赋值为null，等待垃圾回收将这个给回收掉，所以叫clear

```
public void clear() {
    modCount++;

    // clear to let GC do its work
    for (int i = 0; i < size; i++)
        elementData[i] = null;

    size = 0;
}
```

4、removeAll(collection c)

```
public boolean removeAll(Collection<?> c) {
    return batchRemove(c, false);//批量删除
}
```

5、batchRemove(xx,xx):

用于两个方法，一个removeAll()：它只清楚指定集合中的元素，retainAll() 用来测试两个集合是否有交集。

```
//这个方法，用于两处地方，如果complement为false，则用于removeAll如果为true，则给
retainAll()用，retainAll（）是用来检测两个集合是否有交集的。
private boolean batchRemove(Collection<?> c, boolean complement) {
    final Object[] elementData = this.elementData; //将原集合，记名为A
    int r = 0, w = 0; //r用来控制循环，w是记录有多少个交集
    boolean modified = false;
    try {
        for (; r < size; r++)
            //参数中的集合C一次检测集合A中的元素是否有，
            if (c.contains(elementData[r]) == complement)
                //有的话，就给集合A
                elementData[w++] = elementData[r];
    } finally {
        // Preserve behavioral compatibility with AbstractCollection,
        // even if c.contains() throws.
        //如果contains方法使用过程报异常
        if (r != size) {
            //将剩下的元素都赋值给集合A，
            System.arraycopy(elementData, r,
                             elementData, w,
                             size - r);
            w += size - r;
        }
        if (w != size) {
            //这里有两个用途，在removeAll()时，w一直为0，就直接跟clear一样，全是为null。
            //retainAll()：没有一个交集返回true，有交集但不全交也返回true，而两个集合相等
的时候，返回false，所以不能根据返回值来确认两个集合是否有交集，而是通过原集合的大小是否发生改变
来判断，如果原集合中还有元素，则代表有交集，而元集合没有元素了，说明两个集合没有交集。
            // clear to let GC do its work
            for (int i = w; i < size; i++)
                elementData[i] = null;
                modCount += size - w;
                size = w;
                modified = true;
            }
        }
        return modified;
}
```

总结：remove函数，用户移除指定下标的元素，此时会把指定下标到数组末尾的元素向前移动一个单位，并且会把数组最后一个元素设置为null，这样是为了方便之后将整个数组不被使用时，会被GC，可以作为小的技巧使用。

6、其他方法

【set()方法】 说明：设定指定下标索引的元素值

```
public E set(int index, E element) {
    // 检验索引是否合法
    rangeCheck(index);
    // 旧值
    E oldValue = elementData(index);
    // 赋新值
    elementData[index] = element;
    // 返回旧值
    return oldValue;
}
```

【indexOf()方法】

说明：从头开始查找与指定元素相等的元素，注意，是可以查找null元素的，意味着ArrayList中可以存放null元素的。与此函数对应的lastIndexOf，表示从尾部开始查找。

```
// 从首开始查找数组里面是否存在指定元素
public int indexOf(Object o) {
    if (o == null) { // 查找的元素为空
        for (int i = 0; i < size; i++) // 遍历数组，找到第一个为空的元素，返回下标
            if (elementData[i]==null)
            return i;
        } else { // 查找的元素不为空
            for (int i = 0; i < size; i++) // 遍历数组，找到第一个和指定元素相等的元素，
返回下标
                if (o.equals(elementData[i]))
                return i;
        }
        // 没有找到，返回空
        return -1;
}
```

【get()方法】

```
public E get(int index) {
// 检验索引是否合法
rangeCheck(index);
return elementData(index);
}
```

说明：get函数会检查索引值是否合法（只检查是否大于size，而没有检查是否小于0），值得注意的是，在get函数中存在element函数，element函数用于返回具体的元素，具体函数如下：

```
E elementData(int index) {
return (E) elementData[index];
}
```

说明：返回的值都经过了向下转型（Object -> E），这些是对我们应用程序屏蔽的小细节。

ArrayList总结

1）arrayList可以存放null。

2）arrayList**本质**上就是一个elementData**数组。**

3）arrayList区别于数组的地方在于能够自动扩展大小，其中关键的方法就是grow()方法。

4）arrayList中removeAll(collection c)和clear()的区别就是removeAll可以删除批量指定的元素，而clear是全是删除集合中的元素。

5）arrayList由于本质是数组，所以它在数据的**查询**方面会很**快**，而在插入删除这些方面，性能下降很多，有移动很多数据才能达到应有的效果

6）arrayList实现了RandomAccess，所以在**遍历它的时候推荐使用for循环。**

LinkedList实践

1、LinkedList概述

插入，删除操作频繁时，可使用LinkedList来提高效率。

LinkedList提供对头部和尾部元素进行添加和删除操作的方法！

LinkedList是一种可以在**任何位置**进行**高效地插入和移除操作**的**有序序列**，它是基于**双向链表**实现 的。

LinkedList 是一个继承于AbstractSequentialList的双向链表。它也可以被当作堆栈、队列或双端队 列进行操作。

LinkedList 实现 List 接口，能对它进行队列操作。

LinkedList 实现 Deque 接口，即能将LinkedList当作双端队列使用。

LinkedList 实现了Cloneable接口，即覆盖了函数clone()，能克隆。

LinkedList 实现java.io.Serializable接口，这意味着LinkedList支持序列化，能通过序列化去传输。

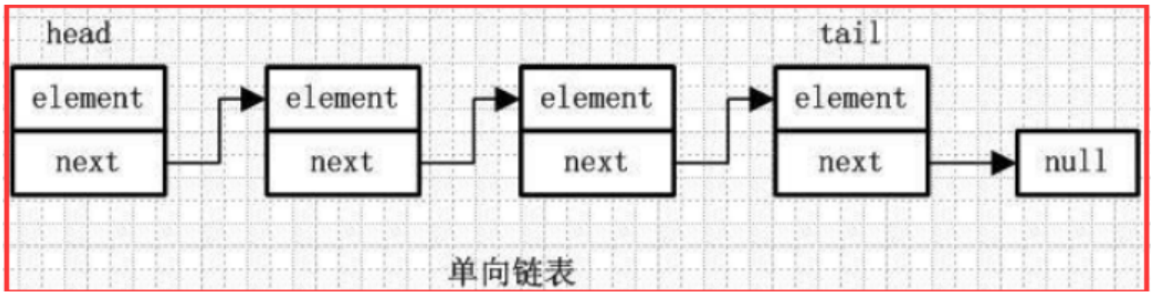
LinkedList 是非同步的。

2、LinkedList的数据结构

单向链表：

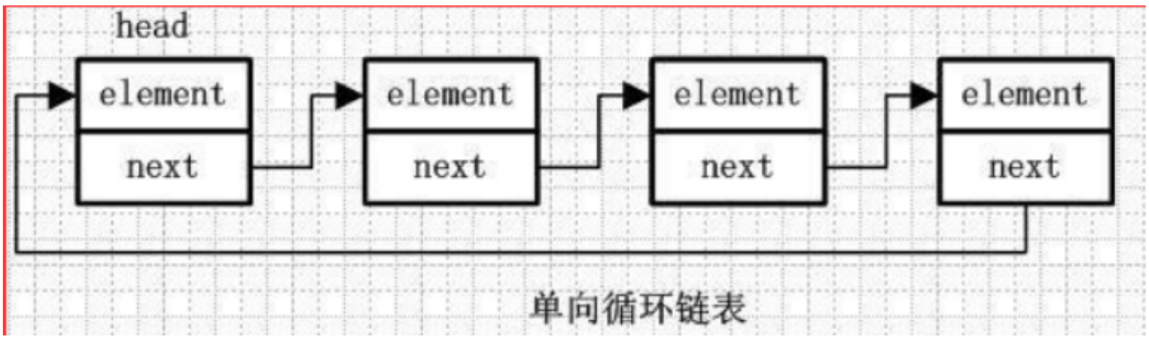
element：用来存放元素

next：用来指向下一个节点元素 通过每个结点的指针指向下一个结点从而链接起来的结构，最后一个节点的next指向null。



单向循环链表：

element、next 跟前面一样 在单向链表的最后一个节点的next会指向头节点，而不是指向null，这样存成一个环



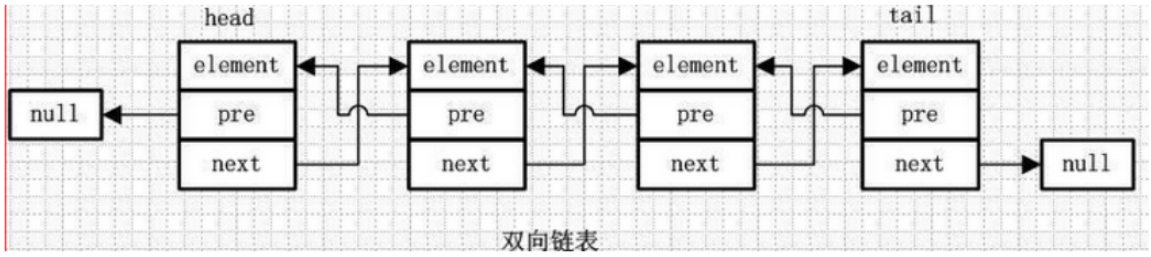
双向链表：

element：存放元素

pre：用来指向前一个元素

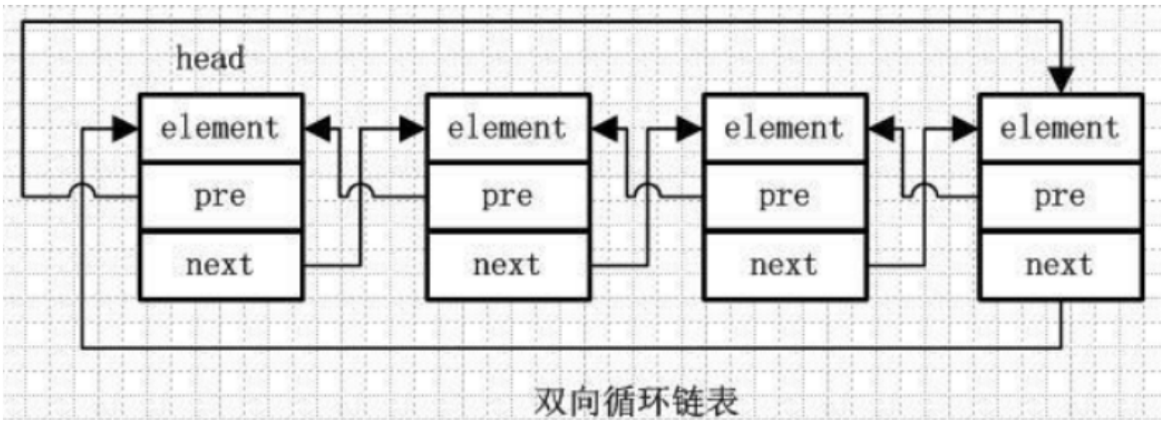
next：指向后一个元素

双向链表是包含两个指针的，pre指向前一个节点，next指向后一个节点，但是第一个节点head的pre指向null，最后一个节点的tail指向null。



双向循环链表：

element、pre、next 跟前面的一样 第一个节点的pre指向最后一个节点，最后一个节点的next指向第一个节点，也形成一个“环”。



如上图所示，LinkedList底层使用的双向链表结构，有一个头结点和一个尾结点，双向链表意味着我们可以从头开始正向遍历，或者是从尾开始逆向遍历，并且可以针对头部和尾部进行相应的操作。

3、LinkedList的特性

1）是通过链表实现的

2）如果在频繁的插入，或者删除数据时，就用linkedList性能会更好。

3）api中接下来讲的一大堆，就是说明linkedList是一个非线程安全的(异步)，其中在操作Interator时，如果改变列表结构(adddelete等)，会发生fail-fast。 通过API再次总结一下LinkedList的特性：

1）异步，也就是非线程安全

2）双向链表。由于实现了list和Deque接口，能够当作队列来使用。 链表：查询效率不高，但是插入和删除这种操作性能好。

3）是顺序存取结构（注意和随机存取结构两个概念搞清楚）

4、继承结构以及层次关系

通过API我们会发现：

1）减少实现顺序存取（例如LinkedList）这种类的工作，通俗的讲就是方便，抽象出类似LinkedList这种类的一些共同的方法

2）既然有了上面这句话，那么以后如果自己想实现顺序存取这种特性的类(就是链表形式)，那么就继承这个AbstractSequentialList抽象类，如果想像数组那样的随机存取的类，那么就去实现AbstracList抽象类。

3）这样的分层，就很符合我们抽象的概念，越在高处的类，就越抽象，往在底层的类，就越有自己独特的个性。自己要慢慢领会这种思想。

4）LinkedList的类继承结构很有意思，我们着重要看是Deque接口，Deque接口表示是一个双端队列，那么也意味着LinkedList是双端队列的一种实现，所以，基于双端队列的操作在LinkedList中全部有效。

1）List接口：列表，add、set、等一些对列表进行操作的方法

2）Deque接口：有队列的各种特性，

3）Cloneable接口：能够复制，使用那个copy方法。

4）Serializable接口：能够序列化。

5）应该注意到没有RandomAccess：那么就推荐使用iterator，在其中就有一个foreach，增强的for循环，其中原理也就是iterator，我们在使用的时候，使用foreach或者iterator都可以。

5、类的属性

```
public class LinkedList<E> extends AbstractSequentialList<E> implements List<E>,
Deque<E>, Cloneable, java.io.Serializable
{
    // 实际元素个数
    transient int size = 0;
    // 头结点
    transient Node<E> first;
    // 尾结点
    transient Node<E> last;
}
```

LinkedList的属性非常简单，一个头结点、一个尾结点、一个表示链表中实际元素个数的变量。注意，头结点、尾结点都有transient关键字修饰，这也意味着在序列化时该域是不会序列化的。

6、构造方法

```
【空参构造函数】

public LinkedList() {
}
```

【有参构造函数】

```
//将集合c中的各个元素构建成LinkedList链表。
public LinkedList(Collection<? extends E> c) {
    // 调用无参构造函数
    this();
    // 添加集合中所有的元素
    addAll(c);
}
```

说明：会调用无参构造函数，并且会把集合中所有的元素添加到LinkedList中。

7、内部类（Node）

```
//根据前面介绍双向链表就知道这个代表什么了，linkedList的奥秘就在这里。
private static class Node<E> {
    E item; // 数据域（当前节点的值）
    Node<E> next; // 后继（指向当前一个节点的后一个节点）
    Node<E> prev; // 前驱（指向当前节点的前一个节点）
    // 构造函数，赋值前驱后继
    Node(Node<E> prev, E element, Node<E> next) {
        this.item = element;
        this.next = next;
        this.prev = prev;
    }
}
```

说明：内部类**Node**就是实际的结点，用于存放实际元素的地方。

8、核心方法

1、【add()方法】

```
public boolean add(E e) {
    // 添加到末尾
    linkLast(e);
    return true;
}
```

说明：add函数用于向LinkedList中添加一个元素，并且添加到链表尾部。具体添加到尾部的逻辑是由linkLast函数完成的。

【LinkLast(XXXXX)】

```
/**
* Links e as last element.
*/
void linkLast(E e) {
    final Node<E> l = last; //临时节点l(L的小写)保存last，也就是l指向了最后一个节点
    final Node<E> newNode = new Node<>(l, e, null);//将e封装为节点，并且e.prev指向了最后一个节点
    last = newNode;//newNode成为了最后一个节点，所以last指向了它
    if (l == null) //判断是不是一开始链表中就什么都没有，如果没有，则newNode就成为了第一个节点，first和last都要指向它
```

```
            first = newNode;
        else //正常的在最后一个节点后追加，那么原先的最后一个节点的next就要指向现在真正的最后一个节点，原先的最后一个节点就变成了倒数第二个节点
            l.next = newNode;
        size++;//添加一个节点，size自增
        modCount++;
    }
```

说明：对于添加一个元素至链表中会调用add方法 -> linkLast方法。

2、【addAll方法】

addAll有两个重载函数，addAll(Collection)型和addAll(int, Collection) 型，我们平时习惯调用的addAll(Collection)型会转化为addAll(int, Collection)型。

```
public boolean addAll(Collection<? extends E> c) {
//继续往下看
return addAll(size, c);
}
```

addAll(size，c)：这个方法，能包含三种情况下的添加，我们这里分析的只是构造方法，空链表的情况，看的时候只需要按照不同的情况分析下去就行了。

```
//真正核心的地方就是这里了，记得我们传过来的是size，c
public boolean addAll(int index, Collection<? extends E> c) {
    //检查index这个是否为合理。这个很简单，自己点进去看下就明白了。
    checkPositionIndex(index);
    //将集合c转换为Object数组 a
    Object[] a = c.toArray();
    //数组a的长度numNew，也就是有多少个元素
    int numNew = a.length;
    if (numNew == 0)
        //集合c是个空的，直接返回false，什么也不做。
        return false;
    //集合c是非空的，定义两个节点(内部类)，每个节点都有三个属性，item、next、prev。注意：不要管这两个什么含义，就是用来做临时存储节点的。这个Node看下面一步的源码分析，Node就是linkedList的最核心的实现，可以直接先跳下一个去看Node的分析
    Node<E> pred, succ;
    //构造方法中传过来的就是index==size
    if (index == size) {
        //linkedList中三个属性：size、first、last。 size：链表中的元素个数。first：头节点 last：尾节点，就两种情况能进来这里
        //情况一、：构造方法创建的一个空的链表，那么size=0，last、和first都为null。linkedList中是空的。什么节点都没有。succ=null、pred=last=null
        //情况二、：链表中有节点，size就不是为0，first和last都分别指向第一个节点，和最后一个节点，在最后一个节点之后追加元素，就得记录一下最后一个节点是什么，所以把last保存到pred临时节点中。
        succ = null;
        pred = last;
    } else {
        //情况三、index! =size，说明不是前面两种情况，而是在链表中间插入元素，那么就得知道index上的节点是谁，保存到succ临时节点中，然后将succ的前一个节点保存到pred中，这样保存了这两个节点，就能够准确的插入节点了
        //举个简单的例子，有2个位置，1、2、如果想插数据到第二个位置，双向链表中，就需要知道第一个位置是谁，原位置也就是第二个位置上是谁，然后才能将自己插到第二个位置上。如果这里还不明白，先看一下文章开头对于各种链表的删除，add操作是怎么实现的。
```

```
            succ = node(index);
            pred = succ.prev;
        }
        //前面的准备工作做完了，将遍历数组a中的元素，封装为一个个节点。
        for (Object o : a) {
            @SuppressWarnings("unchecked") E e = (E) o;
            //pred就是之前所构建好的，可能为null、也可能不为null，为null的话就是属于情况一、不为null则可能是情况二、或者情况三
            Node<E> newNode = new Node<>(pred, e, null);
            //如果pred==null，说明是情况一，构造方法，是刚创建的一个空链表，此时的newNode就当作第一个节点，所以把newNode给first头节点
            if (pred == null)
                first = newNode;
            else
            //如果pred! =null，说明可能是情况2或者情况3，如果是情况2，pred就是last，那么在最后一个节点之后追加到newNode，如果是情况3，在中间插入，pred为原index节点之前的一个节点，将它的next指向插入的节点，也是对的
                pred.next = newNode;
            //然后将pred换成newNode，注意，这个不在else之中，请看清楚了。
            pred = newNode;
        }
        if (succ == null) {
/*如果succ==null，说明是情况一或者情况二，情况一、构造方法，也就是刚创建的一个空链表，pred已经是newNode了，last=newNode，所以linkedList的first、last都指向第一个节点。情况二、在最后节后之后追加节点，那么原先的last就应该指向现在的最后一个节点了，就是newNode。*/
            last = pred;
        } else {
//如果succ! =null,说明可能是情况三、在中间插入节点,举例说明这几个参数的意义,有1、2两个节点,现在想在第二个位置插入节点newNode,根据前面的代码,pred=newNode,succ=2,并且1.next=newNode,已经构建好了，pred.next=succ，相当于在newNode.next =2; succ.prev = pred，相当于 2.prev = newNode; 这样一来，这种指向关系就完成了。first和last不用变，因为头节点和尾节点没变
                pred.next = succ;
            //。。
            succ.prev = pred;
        }
        //增加了几个元素，就把 size = size +numNew 就可以了
        size += numNew;
        modCount++;
        return true;
    }
```

说明：参数中的index表示在索引下标为index的结点（实际上是第index + 1个结点）的前面插 入。

在addAll函数中，addAll函数中还会调用到node函数，get函数也会调用到node函数，此函数是根据索引下标找到该结点并返回，具体代码如下：

```
Node<E> node(int index) {
    // 判断插入的位置在链表前半段或者是后半段
    if (index < (size >> 1)) { // 插入位置在前半段
        Node<E> x = first;
        for (int i = 0; i < index; i++) // 从头结点开始正向遍历
            x = x.next;
        return x; // 返回该结点
    } else { // 插入位置在后半段
        Node<E> x = last;
        for (int i = size - 1; i > index; i--) // 从尾结点开始反向遍历
            x = x.prev;
```

```
            return x; // 返回该结点
        }
    }
```

说明：在根据索引查找结点时，会有一个小优化，结点在前半段则从头开始遍历，在后半段则从尾开始遍历，这样就保证了只需要遍历最多一半结点就可以找到指定索引的结点。

addAll()中的一个问题：

在addAll函数中，传入一个集合参数和插入位置，然后将集合转化为数组，然后再遍历数组，挨个添加数组的元素，但是问题来了，为什么要先转化为数组再进行遍历，而不是直接遍历集合呢？

从效果上两者是完全等价的，都可以达到遍历的效果。关于为什么要转化为数组的问题，我的思考如下：

1. 如果直接遍历集合的话，那么在遍历过程中需要插入元素，在堆上分配内存空间，修改指针域，这个过程中就会一直占用着这个集合，考虑正确同步的话，其他线程只能一直等待。
2. 2. 如果转化为数组，只需要遍历集合，而遍历集合过程中不需要额外的操作，所以占用的时间相对是 较短的，这样就利于其他线程尽快的使用这个集合。说白了，就是有利于提高多线程访问该集合的 效率，尽可能短时间的阻塞。

3、remove(Object o)

```
/**
* Removes the first occurrence of the specified element from this list,
* if it is present. If this list does not contain the element, it is
* unchanged. More formally, removes the element with the lowest index
* {@code i} such that
* <tt>(o==null ? get(i)==null : o.equals(get(i)))
</tt>
* (if such an element exists). Returns {@code true} if this list
* contained the specified element (or equivalently, if this list
* changed as a result of the call).
*
* @param o element to be removed from this list, if present
* @return {@code true} if this list contained the specified element
*/
//首先通过看上面的注释，我们可以知道，如果我们要移除的值在链表中存在多个一样的值，那么我们会移除index最小的那个，也就是最先找到的那个值，如果不存在这个值，那么什么也不做。
public boolean remove(Object o) {
    //这里可以看到，linkedList也能存储null
    if (o == null) {
        //循环遍历链表，直到找到null值，然后使用unlink移除该值。下面的这个else中也一样
        for (Node<E> x = first; x != null; x = x.next) {
            if (x.item == null) {
                unlink(x);
                return true;
            }
        }
    } else {
        for (Node<E> x = first; x != null; x = x.next) {
            if (o.equals(x.item)) {
                unlink(x);
                return true;
            }
        }
    }
```

```
    return false;
}
```

【unlink(xxxx)】

```
/**
* Unlinks non-null node x.
*/
//不能传一个null值过，注意，看之前要注意之前的next、prev这些都是谁。
E unlink(Node<E> x) {
    // assert x != null;
    //拿到节点x的三个属性
    final E element = x.item;
    final Node<E> next = x.next;
    final Node<E> prev = x.prev;
    //这里开始往下就进行移除该元素之后的操作，也就是把指向哪个节点搞定。
    if (prev == null) {
        //说明移除的节点是头节点，则first头节点应该指向下一个节点
        first = next;
    } else {
        //不是头节点，prev.next=next：有1、2、3，将1.next指向3
        prev.next = next;
        //然后解除x节点的前指向。
        x.prev = null;
    }
    if (next == null) {
        //说明移除的节点是尾节点
        last = prev;
    } else {
        //不是尾节点，有1、2、3，将3.prev指向1. 然后将2.next=解除指向。
        next.prev = prev;
        x.next = null;
    }
    //x的前后指向都为null了，也把item为null，让gc回收它
    x.item = null;
    size--; //移除一个节点，size自减
    modCount++;
    return element; //由于一开始已经保存了x的值到element，所以返回。
}
```

4、get(index)

```
/**
* Returns the element at the specified position in this list.
*
* @param index index of the element to return
* @return the element at the specified position in this list
* @throws IndexOutOfBoundsException {@inheritDoc}
*/
//这里没有什么，重点还是在node(index)中
public E get(int index) {
    checkElementIndex(index);
    return node(index).item;
}
```

【node(index)】

```
/**
* Returns the (non-null) Node at the specified element index.
*/
//这里查询使用的是先从中间分一半查找
Node<E> node(int index) {
    // assert isElementIndex(index);
    //"<<":*2的几次方 “>>”:/2的几次方，例如：size<<1：size*2的1次方，
    //这个if中就是查询前半部分
    if (index < (size >> 1)) {//index<size/2
        Node<E> x = first;
            for (int i = 0; i < index; i++)
                x = x.next;
            return x;
    } else {//前半部分没找到，所以找后半部分
        Node<E> x = last;
            for (int i = size - 1; i > index; i--)
            x = x.prev;
        return x;
    }
}
```

5、indexOf(Object o)

```
//这个很简单，就是通过实体元素来查找到该元素在链表中的位置。跟remove中的代码类似，只是返回类型不一样。
public int indexOf(Object o) {
    int index = 0;
    if (o == null) {
        for (Node<E> x = first; x != null; x = x.next) {
            if (x.item == null)
                return index;
            index++;
        }
    } else {
        for (Node<E> x = first; x != null; x = x.next) {
            if (o.equals(x.item))
                return index;
            index++;
        }
    }
    return -1;
}
```

9、LinkedList的迭代器

【ListItr内部类】

```
private class ListItr implements ListIterator<E> {
}
```

【DescendingIterator内部类】

```
private class DescendingIterator implements Iterator<E> {
//看一下这个类，还是调用的ListItr，作用是封装一下Itr中几个方法，让使用者以正常的思维去写代码，
例如，在从后往前遍历的时候，也是跟从前往后遍历一样，使用next等操作，而不用使用特殊的previous。
private final ListItr itr = new ListItr(size());
public boolean hasNext() {
    return itr.hasPrevious();
}
public E next() {
    return itr.previous();
}
public void remove() {
    itr.remove();
}
}
```

LinkedList总结

1. linkedList本质上是一个双向链表，通过一个Node内部类实现的这种链表结构。
2. 能存储null值
3. 跟ArrayList相比较，就真正的知道了，LinkedList在删除和增加等操作上性能好，而ArrayList在查询的性能上好
4. 从源码中看，它不存在容量不足的情况
5. LinkedList不光能够向前迭代，还能像后迭代，并且在迭代的过程中，可以修改值、添加值、还能移除值。
6. LinkedList不光能当链表，还能**当队列**使用，这个就是因为实现了Deque接口。

Vector和Stack

注意在学习这一篇之前，需要有多线程的知识：

1）锁机制：对象锁、方法锁、类锁

对象锁就是方法锁：就是在一个**类中的方法上加上synchronized**关键字，这就是给这个方法加锁了。

类锁：锁的是整个类，当有多个线程来声明这个类的对象的时候将会被阻塞，直到拥有这个类锁的对象被销毁或者主动释放了类锁。这个时候在被阻塞住的线程被挑选出一个占有该类锁，声明该类的对象。其他线程继续被阻塞住。例如：在类A上有关键字synchronized，那么就是给类A加了类锁，线程1第一个声明此类的实例，则线程1拿到了该类锁，线程2在想声明类A的对象，就会被阻塞。

2）在本文中，使用的是方法锁。

3）每个对象只有一把锁，有线程A，线程B，还有一个集合C类，线程A操作C拿到了集合中的锁(在集合C中有用synchronized关键字修饰的)，并且还没有执行完，那么线程A就不会释放锁，当轮到线程B去操作集合C中的方法时 ，发现锁被人拿走了，所以线程B只能等待那个拿到锁的线程使用完，然后才能 拿到锁进行相应的操作。

1、Vector概述

1. Vector是一个可变化长度的数组
2. Vector增加长度通过的是capacity和capacityIncrement这两个变量，目前还不知道如何实现自动扩增的，等会源码分析
3. Vector也可以获得iterator和listIterator这两个迭代器，并且他们发生的是fail-fast，而不是fail□safe，注意这里，不要觉得这个vector是线程安全就搞错了，具体分析在下面会说
4. Vector是一个线程安全的类，如果使用需要线程安全就使用Vector，如果不需要，就使用arrayList

5. Vector和ArrayList很类似，就少许的不一样，从它继承的类和实现的接口来看，跟arrayList一模一样。

   注意：现在的版本已经是jdk1.7，还有更高的jdk1.8了，在开发中，建议不用vector，原因在文章的 结束会有解释，如果需要线程安全的集合类直接用java.util.concurrent包下的类。

2、Vector源码分析

```
public class Vector<E>
extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, java.io.Serializable
{
}
```

【构造方法】

构造方法作用：

1. 初始化存储元素的容器，也就是数组，elementData，
2. 初始化capacityIncrement的大小，默认是0，这个的作用就是扩展数组的时候，增长的大小，为0则每次扩展2倍

【Vector()：空构造】

```
/**
* Constructs an empty vector so that its internal data array
* has size {@code 10} and its standard capacity increment is
* zero.
*/
//看注释，这个是一个空的Vector构造方法，所以让他使用内置的数组，这里还不知道什么是内置的数组，看它调用了自身另外一个带一个参数的构造器
public Vector() {
    this(10);
}
```

【Vector(int)】

```
/**
* Constructs an empty vector with the specified initial capacity and
* with its capacity increment equal to zero.
*
* @param initialCapacity the initial capacity of the vector
* @throws IllegalArgumentException if the specified initial capacity
* is negative
*/
//注释说，给空的vector构造器用和带有一个特定初始化容量用的，并且又调用了另外一个带两个参数的构造器，并且给容量增长值(capacityIncrement=0)为0，查看vector中的变量可以发现capacityIncrement是一个成员变量
public Vector(int initialCapacity) {
    this(initialCapacity, 0);
}
```

【vector(int，int)】

```
/**
* Constructs an empty vector with the specified initial capacity and
* capacity increment.
*
* @param initialCapacity the initial capacity of the vector
* @param capacityIncrement the amount by which the capacity is
* increased when the vector overflows
* @throws IllegalArgumentException if the specified initial capacity
* is negative
*/
//构建一个有特定的初始化容量和容量增长值的空的Vector，
public Vector(int initialCapacity, int capacityIncrement) {
    super();//调用父类的构造，是个空构造
    if (initialCapacity < 0)//小于0，会报非法参数异常：不合法的容量
        throw new IllegalArgumentException("Illegal Capacity:
"+initialCapacity);
    //elementData是一个成员变量数组，初始化它，并给它初始化长度。默认就是10，除非自己给值。
    this.elementData = new Object[initialCapacity];
//capacityIncrement的意思是如果要扩增数组，每次增长该值，如果该值为0，那数组就变为两倍的原长
度，这个之后会分析到
this.capacityIncrement = capacityIncrement;
}
```

【Vector(Collection c)】

```
/**
* Constructs a vector containing the elements of the specified
* collection, in the order they are returned by the collection's
* iterator.
*
* @param c the collection whose elements are to be placed into this
* vector
* @throws NullPointerException if the specified collection is null
* @since 1.2
*/
//将集合c变为Vector，返回Vector的迭代器。
public Vector(Collection<? extends E> c) {
    elementData = c.toArray();
    elementCount = elementData.length;
    // c.toArray might (incorrectly) not return Object[] (see 6260652)
    if (elementData.getClass() != Object[].class)
        elementData = Arrays.copyOf(elementData, elementCount,Object[].class);
}
```

3、核心方法

【add()方法】

```
/**
* Appends the specified element to the end of this Vector.
*
* @param e element to be appended to this Vector
```

```
* @return {@code true} (as specified by {@link Collection#add})
* @since 1.2
*/
//就是在vector中的末尾追加元素。但是看方法，synchronized，明白了为什么vector是线程安全的，因为在方法前面加了synchronized关键字，给该方法加锁了，哪个线程先调用它，其它线程就得等着，如果不清楚的就去看看多线程的知识，到后面我也会一一总结的。
public synchronized boolean add(E e) {
    modCount++;
    //通过arrayList的源码分析经验，这个方法应该是在增加元素前，检查容量是否够用
    ensureCapacityHelper(elementCount + 1);
    elementData[elementCount++] = e;
    return true;
}
```

【ensureCapacityHelper(int)】

```
/**
* This implements the unsynchronized semantics of ensureCapacity.
* Synchronized methods in this class can internally call this
* method for ensuring capacity without incurring the cost of an
* extra synchronization.
*
* @see #ensureCapacity(int)
*/
//这里注释解释，这个方法是异步(也就是能被多个线程同时访问)的，原因是为了让同步方法都能调用到这个检测容量的方法，比如add的同时，另一个线程调用了add的重载方法，那么两个都需要同时查询容量够不够，所以这个就不需要用synchronized修饰了。因为不会发生线程不安全的问题
private void ensureCapacityHelper(int minCapacity) {
    // overflow-conscious code
    if (minCapacity - elementData.length > 0)
    //容量不够，就扩增，核心方法
    grow(minCapacity);
}
```

【grow(int)】

```
//看一下这个方法，其实跟arrayList一样，唯一的不同就是在扩增数组的方式不一样，如果capacityIncrement不为0，那么增长的长度就是capacityIncrement，如果为0，那么扩增为2倍的原容量
private void grow(int minCapacity) {
    // overflow-conscious code
    int oldCapacity = elementData.length;
    int newCapacity = oldCapacity + ((capacityIncrement > 0) ?capacityIncrement
: oldCapacity);
    if (newCapacity - minCapacity < 0)
        newCapacity = minCapacity;
    if (newCapacity - MAX_ARRAY_SIZE > 0)
        newCapacity = hugeCapacity(minCapacity);
    elementData = Arrays.copyOf(elementData, newCapacity);
}

public synchronized E get(int index) {
    if (index >= elementCount)
        throw new ArrayIndexOutOfBoundsException(index);
    return elementData(index);
}
```

4、 Stack

就几个操作，出栈，入栈等，构造方法 也是空的，用的还是数组，父类中的构造，跟父类一样的扩增方式，并且它的方法也是同步的，所以也 是线程安全。

```
class Stack<E> extends Vector<E> {}
```

5、 总结Vector和Stack

【Vector总结（通过源码分析）】

1. Vector线程安全是因为它的方法都加了synchronized关键字
2. Vector的本质是一个数组，特点能是能够自动扩增，扩增的方式跟capacityIncrement的值有关
3. 它也会fail-fast，还有一个fail-safe两个的区别在下面的list总结中会讲到。

【Stack的总结】

1. 对栈的一些操作，先进后出
2. 底层也是用数组实现的，因为继承了Vector
3. 也是线程安全的

6、 List总结

【ArrayList和LinkedList区别】

ArrayList底层是用数组实现的顺序表，是随机存取类型，可自动扩增，并且在初始化时，数组的长度是0，只有在增加元素时，长度才会增加。默认是10，不能无限扩增，有上限，在查询操作的时候性能更好

LinkedList底层是用链表来实现的，是一个双向链表，注意这里不是双向循环链表,顺序存取类型。 在源码中，似乎没有元素个数的限制。应该能无限增加下去，直到内存满了在进行删除，增加操作时性能更好。 两个都是线程不安全的，在iterator时，会发生fail-fast：快速失效。

【arrayList和Vector的区别】

arrayList线程不安全，在用iterator，会发生fail-fast

Vector线程安全，因为在方法前加了Synchronized关键字。也会发生fail-fast

【fail-fast和fail-safe区别和什么情况下会发生】

简单的来说：在java.util下的集合都是发生fail-fast，而在java.util.concurrent下的发生的都是fail-safe。

1）fail-fast快速失败，例如在arrayList中使用迭代器遍历时，有另外的线程对arrayList的存储数组进行了改变，比如add、delete、等使之发生了结构上的改变，所以Iterator就会快速报一个java.util.ConcurrentModificationException 异常（并发修改异常），这就是快速失败。

2）fail-safe安全失败，在java.util.concurrent下的类，都是线程安全的类，他们在迭代的过程中，如果有线程进行结构的改变，不会报异常，而是正常遍历，这就是安全失败。

3）为什么在java.util.concurrent包下对集合有结构的改变，却不会报异常？ 在concurrent下的集合类增加元素的时候使用Arrays.copyOf()来拷贝副本，在副本上增加元素，如果有其他线程在此改变了集合的结构，那也是**在副本上的改变，而不是影响到原集合，迭代器还是照常遍历，遍历完之后，改变原引用指向副本**，所以总的一句话就是如果在此包下的类进行增加删除，就会出现一个副本。所以能防止fail-fast，这种机制并不会出错，所以我们叫这种现象为fail-safe。

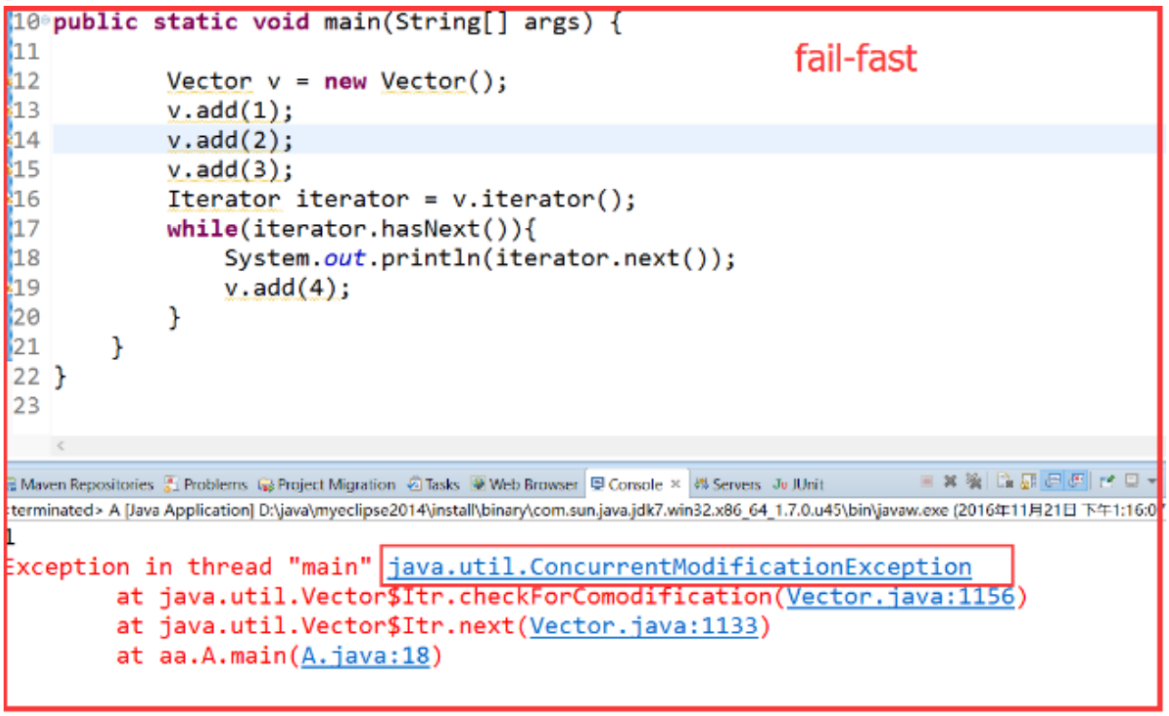
4）vector也是线程安全的，为什么是fail-fast呢？ 这里搞清楚一个问题，并不是说线程安全的集合就不会报fail-fast，而是报fail-safe，你得搞清楚前面所说答案的原理，出现fail-safe是因为他们在实现增删的底层机制不一样，就像上面说的，会有一个副本，而像arrayList、linekdList、verctor等，他们**底层就是对着真正的引用进行操作，所以才会发生异常。**

5）既然是线程安全的，为什么在迭代的时候，还会有别的线程来改变其集合的结构呢(也就是对其删除和增加等操作)？ 首先，我们迭代的时候，根本就没用到集合中的删除、增加，查询的操作，就拿vector来说，我们都没有用那些加锁的方法，也就是方法锁放在那没人拿，在迭代的过程中，有人拿了那把锁，我们也没有办法，因为那把锁就放在那边。

【举例说明fail-fast和fail-safe的区别】

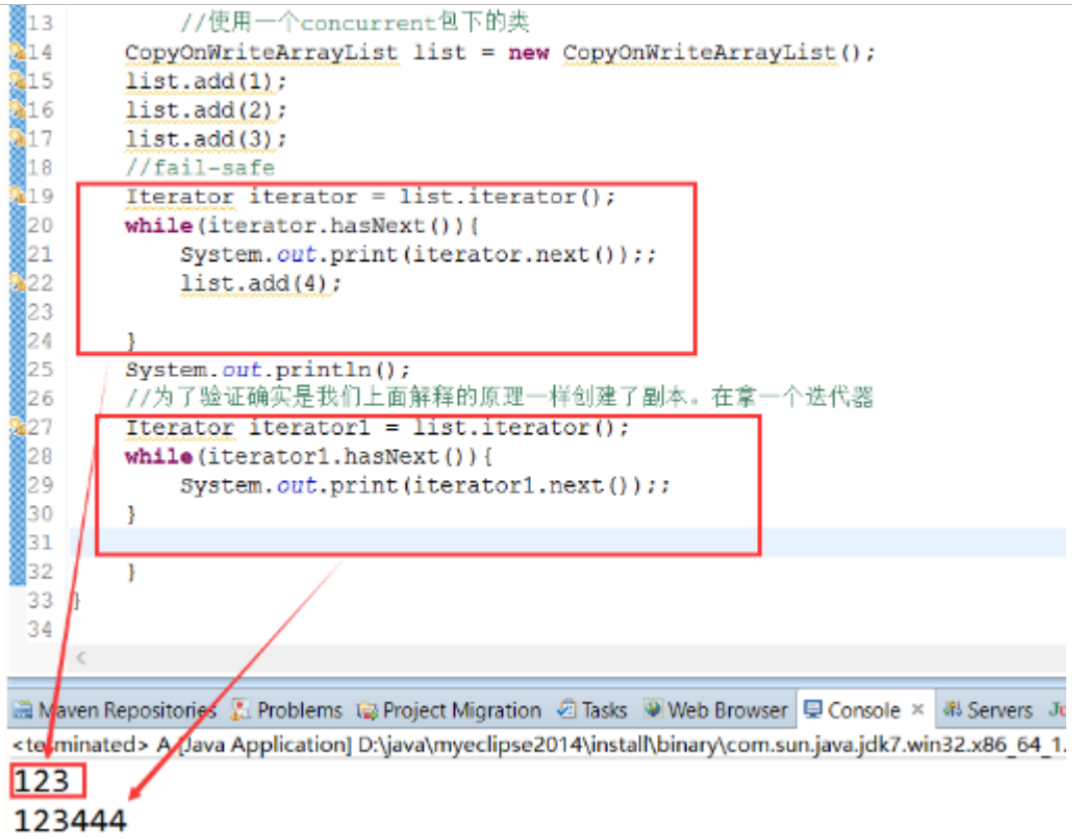
1. fail-fast

-



2. fail-safe

通过CopyOnWriteArrayList这个类来做实验，不用管这个类的作用，但是他确实没有报异常， 并且还通过第二次打印，来验证了上面我们说创建了副本的事情。 原理是在添加操作时会创建副本，在副本上进行添加操作，等迭代器遍历结束后，会将原引用改为副本引用，所以我们在创建了一个list的迭代器，结果打印的就是123444了，证明了确实改变成为了副本引用，后面为什么是三个4，原因是我们循环了3次，不久添加了3个4吗。如果还感觉不爽的话，看下add的源码。



【为什么现在都不提倡使用vector了】

1）vector实现线程安全的方法是在每个操作方法上加锁，这些锁并不是必须要的，在实际开发中，一般都是通过锁一系列的操作来实现线程安全，也就是说将需要同步的资源放一起加锁来保证线程安全。

2）如果多个Thread并发执行一个已经加锁的方法，但是在该方法中，又有vector的存在，vector本身实现中已经加锁了，那么相当于锁上又加锁，会造成额外的开销。

3）就如上面第三个问题所说的，vector还有fail-fast的问题，也就是说它也无法保证遍历安全，在遍历时又得额外加锁，又是额外的开销，还不如直接用arrayList，然后再加锁呢。

总结：Vector在你不需要进行线程安全的时候，也会给你加锁，也就导致了额外开销，所以在 jdk1.5之后就被弃用了，现在如果要用到线程安全的集合，都是从java.util.concurrent包下去拿相应的类。

HashMap

Map接口专门处理键值映射数据的存储，可以根据键实现对值的操作。 最常用的实现类是HashMap。

HashMap数据结构

1、HashMap概述

HashMap是基于哈希表的Map接口实现的，它存储的是内容是键值对映射。此类不保证映射的顺序，假定哈希函数将元素适当的分布在各桶之间，可为基本操作(get和put)提供稳定的性能。

2、HashMap在JDK1.8以前数据结构和存储原理

【链表散列】

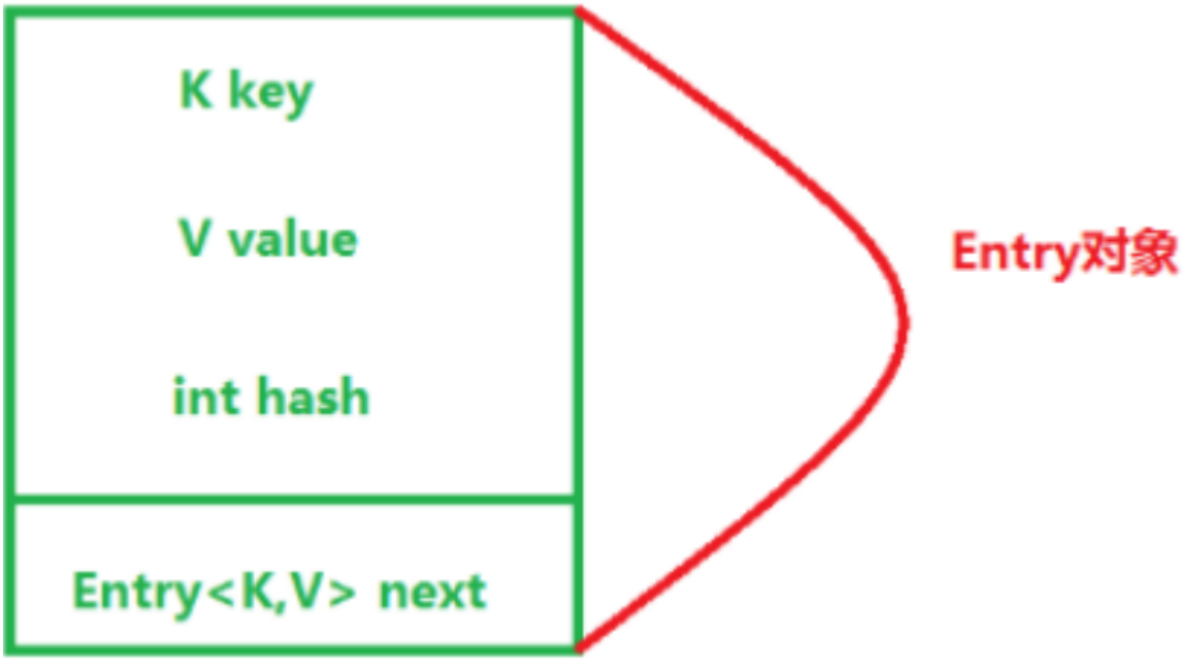
首先我们要知道什么是链表散列？通过数组和链表结合在一起使用，就叫做链表散列。这其实就是hashmap存储的原理图。

【HashMap的数据结构和存储原理】

HashMap的数据结构就是用的链表散列。那HashMap底层是怎么样使用这个数据结构进行数据存取的呢?

分成两个部分： 第一步：HashMap内部有一个entry的内部类，其中有四个属性，我们要存储一个值，则需要一个key 和一个value，存到map中就会先将key和value保存在这个Entry类创建的对象中。

```
static class Entry<K,V> implements Map.Entry<K,V> {
    final K key; //就是我们说的map的key
    V value; //value值，这两个都不陌生
    Entry<K,V> next;//指向下一个entry对象
    int hash;//通过key算过来的你hashcode值。
}
```



第二步：构造好了entry对象，然后将该对象放入数组中，如何存放就是这hashMap的精华所在了。

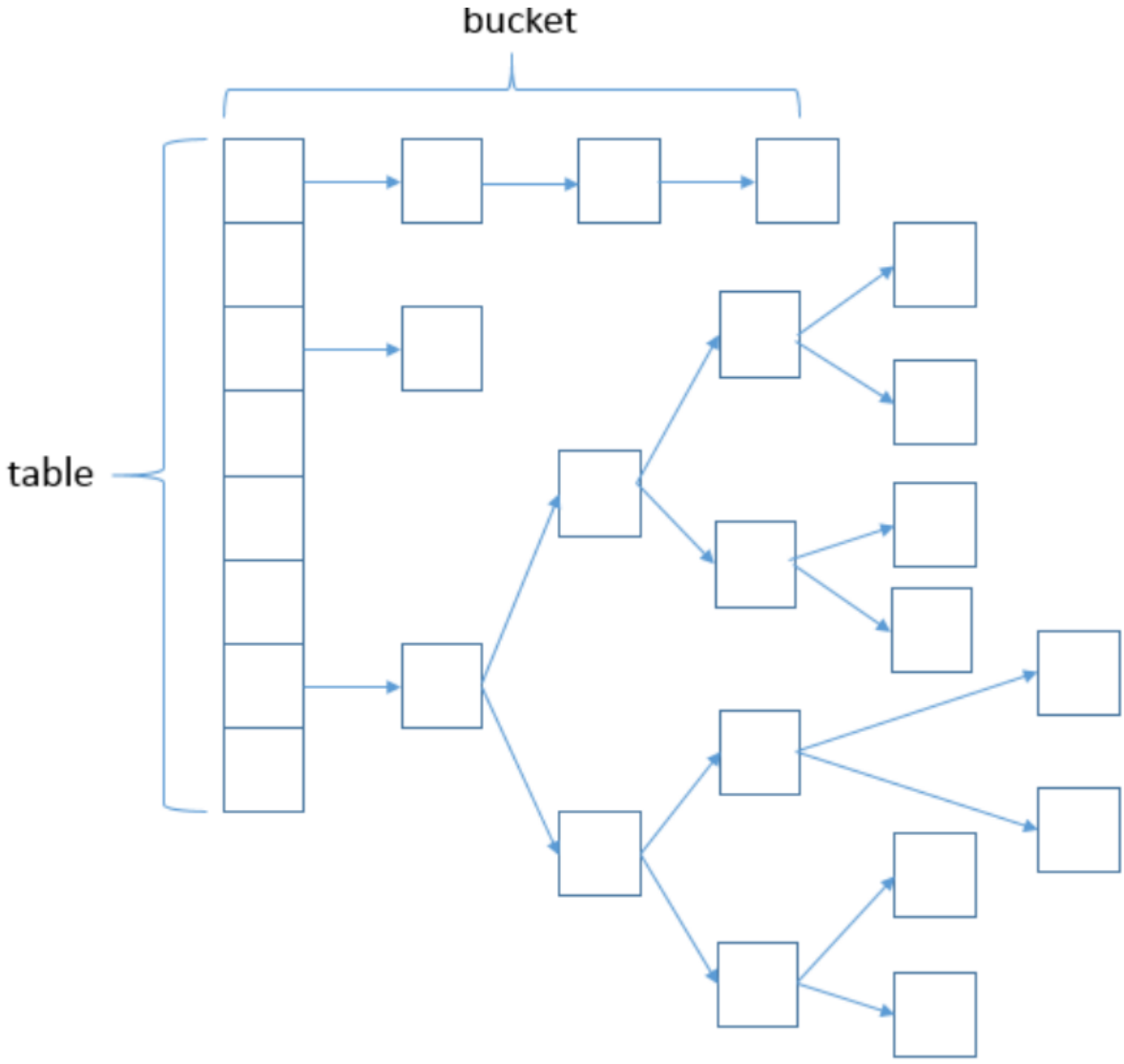
大概的一个存放过程是：通过entry对象中的hash值来确定将该对象存放在数组中的哪个位置上，如果在这个位置上还有其他元素，则通过链表来存储这个元素。

【Hash存放元素的过程】

通过key、value封装成一个entry对象，然后通过key的值来计算该entry的hash值，通过entry的hash值和数组的长度length来计算出entry放在数组中的哪个位置上面，每次存放都是将entry放在第一个位置。

在这个过程中，就是通过hash值来确定将该对象存放在数组中的哪个位置上。

3、JDK1.8后HashMap的数据结构



上图很形象的展示了HashMap的数据结构（数组+链表+红黑树），桶中的结构可能是链表，也可能是红黑树，红黑树的引入是为了提高效率。

4、HashMap的属性

HashMap的实例有两个参数影响其性能。

初始容量：哈希表中桶的数量

加载因子：哈希表在其容量自动增加之前可以达到多满，的一种尺度

当哈希表中条目数超出了当前容量*加载因子(其实就是HashMap的**实际容量**)时，则对该哈希表进行rehash操作，将哈希表扩充至两倍的桶数。 Java中默认初始容量为16，加载因子为0.75。

```
static final int DEFAULT_INITIAL_CAPACITY = 1 << 4; // aka 16
static final float DEFAULT_LOAD_FACTOR = 0.75f;
```

【loadFactor加载因子】

定义：loadFactor译为装载因子。装载因子用来衡量HashMap满的程度。loadFactor的默认值为0.75f。计算HashMap的实时装载因子的方法为：size/capacity，而不是占用桶的数量去除以capacity。

loadFactor加载因子是控制数组存放数据的**疏密**程度，loadFactor越**趋近于1**，那么数组中存放的数据(entry)也就越多，也就越**密**，也就是会让链表的长度增加，loadFactor越小，也就是**趋近于0**，那么数组中存放的数据也就越**稀**，也就是可能数组中每个位置上就放一个元素。

那有人说，就把loadFactor变为1 最好吗，存的数据很多，但是这样会有一个问题，就是我们在通过key拿到我们的value时，是先通过key的hashcode值，找到对应数组中的位置，如果该位置中有很多元素，则需要通过equals来依次比较链表中的元素，拿到我们的value值，这样花费的性能就很高，如果能让数组上的每个位置尽量只有一个元素最好，我们就能直接得到value值了，所以有人又会说，那把loadFactor变得很小不就好了，但是如果变得太小，在数组中的位置就会太稀，也就是分散的太开，浪费很多空间，这样也不好，所以在hashMap 中loadFactor的初始值就是0.75，一般情况下不需要更改它。

【桶】

根据前面画的HashMap存储的数据结构图，你这样想，数组中每一个位置上都放有一个桶，每个桶里就是装一个链表，链表中可以有很多个元素(entry)，这就是桶的意思。也就相当于把元素都放在桶中。

【capacity】

capacity译为容量代表的数组的容量，也就是数组的长度，同时也是HashMap中桶的个数。默认值是16。

一般第一次扩容时会扩容到64，之后好像是2倍。总之，容量都是2的幂。

【size的含义】

size就是在该HashMap的实例中实际存储的元素的个数

【threshold的作用】

```
int threshold;//这个的意思就是衡量数组是否需要扩增的一个标准
```

5、HashMap的源码分析

【实现接口】

```
public class HashMap<K,V> extends AbstractMap<K,V>
implements Map<K,V>, Cloneable, Serializable {
}
```

6、HashMap类的属性

```
public class HashMap<K,V> extends AbstractMap<K,V> implements
Map<K,V>,Cloneable, Serializable {
// 序列号
private static final long serialVersionUID = 362498820763181265L;
// 默认的初始容量是16
static final int DEFAULT_INITIAL_CAPACITY = 1 << 4;
// 最大容量
static final int MAXIMUM_CAPACITY = 1 << 30;
// 默认的填充因子
static final float DEFAULT_LOAD_FACTOR = 0.75f;
// 当桶(bucket)上的结点数大于这个值时会转成红黑树
static final int TREEIFY_THRESHOLD = 8;
// 当桶(bucket)上的结点数小于这个值时树转链表
static final int UNTREEIFY_THRESHOLD = 6;
// 桶中结构转化为红黑树对应的table的最小大小
static final int MIN_TREEIFY_CAPACITY = 64;
// 存储元素的数组，总是2的幂次倍
transient Node<k,v>[] table;
// 存放具体元素的集
```

```
transient Set<map.entry<k,v>> entrySet;
// 存放元素的个数，注意这个不等于数组的长度。
transient int size;
// 每次扩容和更改map结构的计数器
transient int modCount;
// 临界值 当实际大小(容量*填充因子)超过临界值时，会进行扩容
int threshold;
// 填充因子
final float loadFactor;
}
```

7、HashMap的构造方法

【HashMap()】

```
//看上面的注释就已经知道，DEFAULT_INITIAL_CAPACITY=16，DEFAULT_LOAD_FACTOR=0.75
//初始化容量：也就是初始化数组的大小
//加载因子：数组上的存放数据疏密程度。
public HashMap() {
    this(DEFAULT_INITIAL_CAPACITY, DEFAULT_LOAD_FACTOR);
}
```

【HashMap(int)】

```
public HashMap(int initialCapacity) {
    this(initialCapacity, DEFAULT_LOAD_FACTOR);
}
```

【HashMap(int,float)】

```
public HashMap(int initialCapacity, float loadFactor) {
    // 初始容量不能小于0，否则报错
    if (initialCapacity < 0)
    throw new IllegalArgumentException("Illegal initial capacity: "
+initialCapacity);
    // 初始容量不能大于最大值，否则为最大值
    if (initialCapacity > MAXIMUM_CAPACITY)
        initialCapacity = MAXIMUM_CAPACITY;
    // 填充因子不能小于或等于0，不能为非数字
    if (loadFactor <= 0 || Float.isNaN(loadFactor))
        throw new IllegalArgumentException("Illegal load factor: " +loadFactor);
    // 初始化填充因子
    this.loadFactor = loadFactor;
    // 初始化threshold大小
    this.threshold = tableSizeFor(initialCapacity);
}
```

【HashMap(Map<? extends K,? extends V> m)】

```
public HashMap(Map<? extends K, ? extends V> m) {
    // 初始化填充因子
    this.loadFactor = DEFAULT_LOAD_FACTOR;
    // 将m中的所有元素添加至HashMap中
    putMapEntries(m, false);
}
```

【putMapEntries(Map<? extends K,? extends V> m, boolean evict)函数将m的所有元素存入本 HashMap实例中】

```
final void putMapEntries(Map<? extends K, ? extends V> m, boolean evict) {
    int s = m.size();
    if (s > 0) {
        // 判断table是否已经初始化
        if (table == null) { // pre-size
            // 未初始化，s为m的实际元素个数
            float ft = ((float)s / loadFactor) + 1.0F;
            int t = ((ft < (float)MAXIMUM_CAPACITY) ?
                    (int)ft : MAXIMUM_CAPACITY);
            // 计算得到的t大于阈值，则初始化阈值
            if (t > threshold)
                threshold = tableSizeFor(t);
    }
    // 已初始化，并且m元素个数大于阈值，进行扩容处理
    else if (s > threshold)
        resize();
    // 将m中的所有元素添加至HashMap中
    for (Map.Entry<? extends K, ? extends V> e : m.entrySet()) {
        K key = e.getKey();
        V value = e.getValue();
        putVal(hash(key), key, value, false, evict);
    }
}
}
```

8、常用方法

【put(K key,V value)】

```
public V put(K key, V value) {
    return putVal(hash(key), key, value, false, true);
}
```

【putVal(int hash, K key, V value, boolean onlyIfAbsent,boolean evict)】

```
final V putVal(int hash, K key, V value, boolean onlyIfAbsent,boolean evict) {
    Node<K,V>[] tab; Node<K,V> p; int n, i;
    // table未初始化或者长度为0，进行扩容
    if ((tab = table) == null || (n = tab.length) == 0)
        n = (tab = resize()).length;
    // (n - 1) & hash 确定元素存放在哪个桶中，桶为空，新生成结点放入桶中(此时，这个结点是放在数组中)
```

```
    if ((p = tab[i = (n - 1) & hash]) == null)
        tab[i] = newNode(hash, key, value, null);
    // 桶中已经存在元素
    else {
        Node<K,V> e; K k;
        // 比较桶中第一个元素(数组中的结点)的hash值相等，key相等
        if (p.hash == hash &&((k = p.key) == key || (key != null &&
key.equals(k))))
            // 将第一个元素赋值给e，用e来记录
            e = p;
        // hash值不相等，即key不相等；为红黑树结点
        else if (p instanceof TreeNode)
            // 放入树中
            e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);
        // 为链表结点
        else {
            // 在链表最末插入结点
            for (int binCount = 0; ; ++binCount) {
                // 到达链表的尾部
                if ((e = p.next) == null) {
                    // 在尾部插入新结点
                    p.next = newNode(hash, key, value, null);
                    // 结点数量达到阈值，转化为红黑树
                    if (binCount >= TREEIFY_THRESHOLD - 1) // -1 for 1st
                        treeifyBin(tab, hash);
                    // 跳出循环
                    break;
                }
                // 判断链表中结点的key值与插入的元素的key值是否相等
                if (e.hash == hash &&
                    ((k = e.key) == key || (key != null && key.equals(k))))
                    // 相等，跳出循环
                    break;
                // 用于遍历桶中的链表，与前面的e = p.next组合，可以遍历链表
                p = e;
            }
        }
        // 表示在桶中找到key值、hash值与插入元素相等的结点
        if (e != null) {
            // 记录e的value
            V oldValue = e.value;
            // onlyIfAbsent为false或者旧值为null
            if (!onlyIfAbsent || oldValue == null)
                //用新值替换旧值
                e.value = value;
            // 访问后回调
            afterNodeAccess(e);
            // 返回旧值
            return oldValue;
        }
    }
    // 结构性修改
    ++modCount;
    // 实际大小大于阈值则扩容
    if (++size > threshold)
        resize();
    // 插入后回调
    afterNodeInsertion(evict);
```

```
    return null;
}
```

【get(Object key)】

```
public V get(Object key) {
    Node<K,V> e;
    return (e = getNode(hash(key), key)) == null ? null : e.value;
}
```

【getNode(int hash,Object key)】

```
final Node<K,V> getNode(int hash, Object key) {
    Node<K,V>[] tab; Node<K,V> first, e; int n; K k;
    // table已经初始化，长度大于0，根据hash寻找table中的项也不为空
    if ((tab = table) != null && (n = tab.length) > 0 &&
        (first = tab[(n - 1) & hash]) != null) {
        // 桶中第一项(数组元素)相等
        if (first.hash == hash && // always check first node
            ((k = first.key) == key || (key != null && key.equals(k))))
            return first;
        // 桶中不止一个结点
        if ((e = first.next) != null) {
            // 为红黑树结点
            if (first instanceof TreeNode)
                // 在红黑树中查找
                return ((TreeNode<K,V>)first).getTreeNode(hash, key);
            // 否则，在链表中查找
            do {
                if (e.hash == hash &&
                    ((k = e.key) == key || (key != null && key.equals(k))))
                    return e;
            } while ((e = e.next) != null);
        }
    }
    return null;
}
```

【resize方法】

```
final Node<K,V>[] resize() {
        // 当前table保存
        Node<K,V>[] oldTab = table;
        // 保存table大小
        int oldCap = (oldTab == null) ? 0 : oldTab.length;
        // 保存当前阈值
        int oldThr = threshold;
        int newCap, newThr = 0;
        // 之前table大小大于0
        if (oldCap > 0) {
            // 之前table大于最大容量
            if (oldCap >= MAXIMUM_CAPACITY) {
                // 阈值为最大整型
                threshold = Integer.MAX_VALUE;
```

```
                return oldTab;
            }
            // 容量翻倍，使用左移，效率更高
            else if ((newCap = oldCap << 1) < MAXIMUM_CAPACITY &&
                     oldCap >= DEFAULT_INITIAL_CAPACITY)
                // 阈值翻倍
                newThr = oldThr << 1; // double threshold
        }
        // 之前阈值大于0
        else if (oldThr > 0) // initial capacity was placed in threshold
            newCap = oldThr;
        // oldCap = 0并且oldThr = 0，使用缺省值（如使用HashMap()构造函数，之后再插入一个
元素会调用resize函数，会进入这一步）
        else {               // zero initial threshold signifies using defaults
            newCap = DEFAULT_INITIAL_CAPACITY;
            newThr = (int)(DEFAULT_LOAD_FACTOR * DEFAULT_INITIAL_CAPACITY);
        }
        // 新阈值为0
        if (newThr == 0) {
            float ft = (float)newCap * loadFactor;
            newThr = (newCap < MAXIMUM_CAPACITY && ft < (float)MAXIMUM_CAPACITY
?
                      (int)ft : Integer.MAX_VALUE);
        }
        threshold = newThr;
        @SuppressWarnings({"rawtypes","unchecked"})
        // 初始化table
        Node<K,V>[] newTab = (Node<K,V>[])new Node[newCap];
        table = newTab;
        // 之前的table已经初始化过
        if (oldTab != null) {
            // 复制元素，重新进行hash
            for (int j = 0; j < oldCap; ++j) {
                Node<K,V> e;
                if ((e = oldTab[j]) != null) {
                    oldTab[j] = null;
                    if (e.next == null)
                        newTab[e.hash & (newCap - 1)] = e;
                    else if (e instanceof TreeNode)
                        ((TreeNode<K,V>)e).split(this, newTab, j, oldCap);
                    else { // preserve order
                        Node<K,V> loHead = null, loTail = null;
                        Node<K,V> hiHead = null, hiTail = null;
                        Node<K,V> next;
                        // 将同一桶中的元素根据(e.hash & oldCap)是否为0进行分割，分成两
个不同的链表，完成rehash
                        do {
                            next = e.next;
                            if ((e.hash & oldCap) == 0) {
                                if (loTail == null)
                                    loHead = e;
                                else
                                    loTail.next = e;
                                loTail = e;
                            }
                            else {
                                if (hiTail == null)
                                    hiHead = e;
```

```
                                else
                                    hiTail.next = e;
                                hiTail = e;
                            }
                        } while ((e = next) != null);
                        if (loTail != null) {
                            loTail.next = null;
                            newTab[j] = loHead;
                        }
                        if (hiTail != null) {
                            hiTail.next = null;
                            newTab[j + oldCap] = hiHead;
                        }
                    }
                }
            }
        }
        return newTab;
    }
```

进行扩容，会伴随着一次重新hash分配，并且会遍历hash表中所有的元素，是非常耗时的。在编写程序中，要尽量避免resize。

9、 总结

【关于数组扩容】

从putVal源代码中我们可以知道，当插入一个元素的时候size就加1，若size大于threshold的时候，就会进行扩容。

假设我们的capacity大小为32，loadFator为0.75,则threshold为24 = 32 * 0.75， 此时，插入了25个元素，并且插入的这25个元素都在同一个桶中，桶中的数据结构为红黑树，则还有31个桶是空的，也会进行扩容处理，其实，此时，还有31个桶是空的，好像似乎不需要进行扩容处理，但是是需要扩容处理的，因为此时我们的capacity大小可能不适当。我们前面知道，扩容处理会遍历所有的元素，时间复杂度很高；前面我们还知道，经过一次扩容处理后，元素会更加均匀的分布在各 个桶中，会提升访问效率。所以，说尽量避免进行扩容处理，也就意味着，遍历元素所带来的坏处大于元素在桶中均匀分布所带来的好处。

【总结】

1. 要知道hashMap在JDK1.8以前是一个链表散列这样一个数据结构，而在JDK1.8以后是一个数组加链表加红黑树的数据结构。
2. 通过源码的学习，hashMap是一个能快速通过key获取到value值得一个集合，原因是内部使用的是hash查找值得方法。

迭代器

所有实现了Collection接口的容器类都有一个iterator方法用以返回一个实现Iterator接口的对象

Iterator对象称作为迭代器，用以方便的对容器内元素的遍历操作，Iterator接口定义了如下方法：

boolean hashNext();//判断是否有元素没有被遍历

Object next();//返回游标当前位置的元素并将游标移动到下一个位置

void remove();//删除游标左边的元素，在执行完next之后该操作只能执行一次

问题：何遍历Map集合呢?

方法1：通过迭代器Iterator实现遍历

获取Iterator ： Collection 接口的iterator()方法

Iterator的方法：

boolean hasNext(): 判断是否存在另一个可访问的元素

Object next(): 返回要访问的下一个元素

```
Set keys=dogMap.keySet(); //取出所有key的集合
Iterator it=keys.iterator(); //获取Iterator对象
while(it.hasNext()){
String key=(String)it.next(); //取出key
Dog dog=(Dog)dogMap.get(key); //根据key取出对应的值
System.out.println(key+"\t"+dog.getStrain());
}
```

方法2：增强for循环

```
for(元素类型t 元素变量x : 数组或集合对象){
    引用了x的java语句
}
```

泛型

Java 泛型（generics）是 JDK 5 中引入的一个新特性, 泛型提供了编译时类型安全检测机制，该机制允许程序员在编译时检测到非法的类型。

泛型的本质是参数化类型，也就是说所操作的数据类型被指定为一个参数。

如何解决以下强制类型转换时容易出现的异常问题?

List的get(int index)方法获取元素

Map的get(Object key)方法获取元素

Iterator的next()方法获取元素

分析：通过泛型， JDK1.5使用泛型改写了集合框架中的所有接口和类

Collections工具类

Java提供了一个操作Set、List和Map等集合的工具类：Collections，该工具类提供了大量方法对集合进行排序、查询和修改等操作，还提供了将集合对象置为不可变、对集合对象实现同步控制等方法。 这个类不需要创建对象，内部提供的都是静态方法。

1、Collectios概述

此类完全由在 collection 上进行操作或返回 collection 的静态方法组成。它包含在 collection 上操作的多态算法，即“包装器”，包装器返回由指定 collection 支持的新 collection，以及少数其他内容。如果为此类的方法所提供的 collection 或类对象为 null，则这些方法都将抛出 NullPointerException 。

2、排序操作

【方法】

```
1）static void reverse(List<?> list):
//反转列表中元素的顺序。
2）static void shuffle(List<?> list) :
//对List集合元素进行随机排序。
3）  static void sort(List<T> list)
//根据元素的自然顺序 对指定列表按升序进行排序
4）static <T> void sort(List<T> list, Comparator<? super T> c) :
//根据指定比较器产生的顺序对指定列表进行排序。
5）static void swap(List<?> list, int i, int j)
//在指定List的指定位置i,j处交换元素。
6）static void rotate(List<?> list, int distance)
//当distance为正数时，将List集合的后distance个元素“整体”移到前面；当distance为负数时，将list集合的前distance个元素“整体”移到后边。该方法不会改变集合的长度。
```

【演示】

```
import java.util.ArrayList;
import java.util.Collections;
public class CollectionsTest {
    public static void main(String[] args) {
        ArrayList list = new ArrayList();
            list.add(3);
            list.add(-2);
            list.add(9);
            list.add(5);
            list.add(-1);
            list.add(6);
            //输出：[3, -2, 9, 5, -1, 6]
            System.out.println(list);
            //集合元素的次序反转
            Collections.reverse(list);
            //输出：[6, -1, 5, 9, -2, 3]
            System.out.println(list);
            //排序：按照升序排序
            Collections.sort(list);
            //[-2, -1, 3, 5, 6, 9]
            System.out.println(list);
            //根据下标进行交换
            Collections.swap(list, 2, 5);
            //输出：[-2, -1, 9, 5, 6, 3]
            System.out.println(list);
            /*//随机排序
            Collections.shuffle(list);
            //每次输出的次序不固定
            System.out.println(list);*/
            //后两个整体移动到前边
            Collections.rotate(list, 2);
            //输出：[6, 9, -2, -1, 3, 5]
            System.out.println(list);
        }
}
```

3、查找、替换操作

```
1） static <T> int binarySearch(List<? extends Comparable<? super T>>list, T key)
//使用二分搜索法搜索指定列表，以获得指定对象在List集合中的索引。
//注意：此前必须保证List集合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collection 的最大元素。
3）static Object max(Collection coll,Comparator comp):
//根据指定比较器产生的顺序，返回给定 collection 的最大元素。
4）static Object min(Collection coll):
//根据元素的自然顺序，返回给定collection 的最小元素。
5）static Object min(Collection coll,Comparator comp):
//根据指定比较器产生的顺序，返回给定 collection 的最小元素。
6） static <T> void fill(List<? super T> list, T obj) :
//使用指定元素替换指定列表中的所有元素。
7）static int frequency(Collection<?> c, Object o)
//返回指定 collection 中等于指定对象的出现次数。
8）static int indexOfSubList(List<?> source, List<?> target) :
//返回指定源列表中第一次出现指定目标列表的起始位置；如果没有出现这样的列表，则返回-1。
9）static int lastIndexOfSubList(List<?> source, List<?> target)
//返回指定源列表中最后一次出现指定目标列表的起始位置；如果没有出现这样的列表，则返回-1。
10）static <T> boolean replaceAll(List<T> list, T oldVal, T newVal)
//使用一个新值替换List对象的所有旧值oldVal
```

【演示：实例使用查找、替换操作】

```
import java.util.ArrayList;
import java.util.Collections;
public class CollectionsTest1 {
    public static void main(String[] args) {
        ArrayList list = new ArrayList();
        list.add(3);
        list.add(-2);
        list.add(9);
        list.add(5);
        list.add(-1);
        list.add(6);
        //[3, -2, 9, 5, -1, 6]
        System.out.println(list);
        //输出最大元素9
        System.out.println(Collections.max(list));
        //输出最小元素：-2
        System.out.println(Collections.min(list));
        //将list中的-2用1来代替
        System.out.println(Collections.replaceAll(list, -2, 1));
        //[3, 1, 9, 5, -1, 6]
        System.out.println(list);
        list.add(9);
        //判断9在集合中出现的次数，返回2
        System.out.println(Collections.frequency(list, 9));
        //对集合进行排序
        Collections.sort(list);
        //[-1, 1, 3, 5, 6, 9, 9]
        System.out.println(list);
        //只有排序后的List集合才可用二分法查询，输出2
        System.out.println(Collections.binarySearch(list, 3));
    }
}
```

4、同步控制

Collectons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，从而解决多线程并发访问集合时的线程安全问题。

正如前面介绍的HashSet，TreeSet，arrayList，LinkedList，HashMap，TreeMap都是线程不安全的。 Collections提供了多个静态方法可以把他们包装成线程同步的集合。

【方法】

```
1）static <T> Collection<T> synchronizedCollection(Collection<T> c)
//返回指定 collection 支持的同步（线程安全的）collection。
2）static <T> List<T> synchronizedList(List<T> list)
//返回指定列表支持的同步（线程安全的）列表。
3）static <K,V> Map<K,V> synchronizedMap(Map<K,V> m)
//返回由指定映射支持的同步（线程安全的）映射。
4）static <T> Set<T> synchronizedSet(Set<T> s)
//返回指定 set 支持的同步（线程安全的）set。
```

【实例】

```
import java.util.*;
public class TestSynchronized
{
    public static void main(String[] args)
    {
        //下面程序创建了四个同步的集合对象
        Collection c = Collections.synchronizedCollection(new ArrayList());
        List list = Collections.synchronizedList(new ArrayList());
        Set s = Collections.synchronizedSet(new HashSet());
        Map m = Collections.synchronizedMap(new HashMap());
    }
}
```

5、Collesction设置不可变集合

```
1）emptyXxx()
//返回一个空的、不可变的集合对象，此处的集合既可以是List，也可以是Set，还可以是Map。
2）singletonXxx():
//返回一个只包含指定对象（只有一个或一个元素）的不可变的集合对象，此处的集合可以是：List，Set，
Map。
3）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。

【实例】

```
import java.util.*;
public class TestUnmodifiable
{
    public static void main(String[] args)
    {
```

```
        //创建一个空的、不可改变的List对象
        List<String> unmodifiableList = Collections.emptyList();
        //unmodifiableList.add("java");
        //添加出现异常：java.lang.UnsupportedOperationException
        System.out.println(unmodifiableList);// []
        //创建一个只有一个元素，且不可改变的Set对象
        Set unmodifiableSet = Collections.singleton("Struts2权威指南");
        //[Struts2权威指南]
        System.out.println(unmodifiableSet);
        //创建一个普通Map对象
        Map scores = new HashMap();
        scores.put("语文" , 80);
        scores.put("Java" , 82);
        //返回普通Map对象对应的不可变版本
        Map unmodifiableMap = Collections.unmodifiableMap(scores);
        //下面任意一行代码都将引发UnsupportedOperationException异常
        unmodifiableList.add("测试元素");
        unmodifiableSet.add("测试元素");
        unmodifiableMap.put("语文",90);
    }
}
```