**ArrayList**

**数据结构：数组**

**特点：查询操作快，删除操作慢（涉及数组拷贝，如果数据量大且删除第一个就很慢），添加较快，不过可能需要扩容。再指定位置添加的时候涉及数组拷贝。**

构造器：

默认：初始化容量为0的数组（在使用时，在进行扩容）

Int cap:构造容量为cap的数组，小于0时为0；

Collection<? Extend e> 将旧数组拷贝到当前数组中

1:添加

add(object e)当容量满了，进行扩容（容量\*1.5如果新增后超过这个容量，则容量为新增后所需的最小容量,1\*1.5=1,但是需要2），然后在数组末添加

if (s == elementData.length) {  
 elementData = this.grow();  
}  
elementData[s] = e;  
this.size = s + 1;

int newCapacity = oldCapacity + (oldCapacity >> 1);

add(int index,object e)：扩容判断相同，插入方式为数据拷贝，然后替换指定位置数据

this.rangeCheckForAdd(Index);  
++this.modCount;  
int s;  
Object[] elementData;  
if ((s = this.size) == (elementData = this.elementData).length) {  
 elementData = this.grow();  
}

//数据拷贝  
System.arraycopy(elementData, index, elementData, index + 1, s - index);

//替换指定位置数据  
elementData[index] = element;  
this.size = s + 1;

addAll(Collection e):数据拷贝

2.删除

remove(object o) :先对数组进行遍历，找到指定下标，然后进行数组拷贝，覆盖指定位置内容,然后将末尾置空

remove(object o):数据拷贝，覆盖内容，然后将末尾置空

removeRange(int from,int end):类似上面

3.查询

Get(int index)首先判断是否越界，直接返回指定下标的数据

**4.size和modCount的区别**

**Size:ArrayList实际的大小**

**modCount:**

**元素数目被修改的次数，list.set(i,item)不会改变modeCount**

**Fail-fast机制使用到，在迭代器中，如果其他线程修改了此集合，导致对象的modeCount和迭代器中的不一样会抛出异常。**

fail-safe任何对集合结构的修改都会在一个复制的集合上进行修改，因此不会抛出ConcurrentModificationException，CopyOnWriteArrayList,  
ConcurrentHashMap是fail-safe机制

fail-safe机制有两个问题

（1）需要复制集合，产生大量的无效对象，开销大

（2）无法保证读取的数据是目前原始数据结构中的数据。

**LinkedList**

**数据结构：双向链表，一个first结点和一个Last结点维护，结点中维护者next结点和pre结点。**

**特点：添加快，比Array更快，因为不需要扩容，添加指定位置为O（2/n），涉及遍历查找。删除指定结点为0(n/2)，要找到指定结点再替换。**

1. 添加

Addfist/ offerFirst（）:在表头添加，通过linkFirst私有方法

LinkedList.Node<E> f = this.first;

//在表头新建一个结点，pre为空，next为旧表头  
LinkedList.Node<E> newNode = new LinkedList.Node((LinkedList.Node)null, e, f);

//将表头移向新表头  
this.first = newNode;

//头尾为同一个  
if (f == null) {  
 this.last = newNode;  
} else {  
 f.prev = newNode;  
}

Add/offer():默认为尾插，与addLast/offerLast()相同，多了返回boolean值，插入是否成功，调用linkLast。与头插类似

LinkedList.Node<E> l = this.last;  
LinkedList.Node<E> newNode = new LinkedList.Node(l, e, (LinkedList.Node)null);  
this.last = newNode;  
if (l == null) {  
 this.first = newNode;  
} else {  
 l.next = newNode;  
}

Add(int idx,E item):指定位置添加，首先遍历找到插入位置(idx 和size >>1比较)的原元素（元素在左半部分从头遍历，在右半部分从右遍历）。然后进行替换

LinkedList.Node<E> pred = succ.prev;  
LinkedList.Node<E> newNode = new LinkedList.Node(pred, e, succ);  
succ.prev = newNode;  
if (pred == null) {  
 this.first = newNode;  
} else {  
 pred.next = newNode;  
}

1. 删除

RemoveFirst(),RemoveLast()直接调用对应unlink函数

Remove():分为删除结点的item为空和非空，遍历找到对应的item，然后调用unlink()。

1. 查询

Get(int index)：遍历链表0(n/2)

提供getFirst,getLast,O(1)

**Vetor**

**数据结构：数组**

**特点：线程安全，使用了sync关键字**

**增删改查基本和ArrayList相同，扩容默认为2倍，或者自己设置扩容大小。而ArrayList不能设置扩容大小，只能是1-5倍。**