ArrayList和LinkedList是Java集合框架中用来存储对象引用列表的两个类。ArrayList和LinkedList都实现List接口。先对List做一个简单的了解：

列表（list）是元素的有序集合，也称为序列。它提供了基于元素位置的操

作，有助于快速访问、添加和删除列表中特定索引位置的元素。List接口实现了Collection和Iterable作为父接口。它允许存储重复值和空值，支持通过索引访问元素。

读完这篇文章要搞清楚的问题：ArrayList和LinkedList有什么不同之处？什么时候应该用ArrayList什么时候又该用LinkedList呢？

下面以增加和删除元素为例比较ArrayList和LinkedList的不同之处。

#### 1、增加元素到列表尾端

在ArrayList中增加元素到队列尾端的代码如下：

public boolean add(E e) {

ensureCapacityInternal(size + 1); //确保内部数组有足够的空间  
 elementData[size++] = e; //将元素加入到数组的末尾，完成添加  
 return true;  
}

ArrayList中add()方法的性能决定于ensureCapacity()方法。ensureCapacity()的实现如下：

**public** void **ensureCapacity**(**int** minCapacity){

  modCount++;  
  **int** oldCapacity=elementData.length;  
  **if**(minCapacity>oldCapacity){    *//如果数组容量不足，进行扩容*  
      Object[] oldData=elementData;  
      **int** newCapacity=(oldCapacity\*3)/2+1;  *//扩容到原始容量的1.5倍*  
      **if**(newCapacitty<minCapacity)   *//如果新容量小于最小需要的容量，则使用最小*  
                                                    *//需要的容量大小*  
         newCapacity=minCapacity ;  *//进行扩容的数组复制*  
         elementData=Arrays.copyof(elementData,newCapacity);  
  }  
}

可以看到，只要ArrayList的当前容量足够大，add()操作的效率非常高的。只有当ArrayList对容量的需求超出当前数组大小时，才需要进行扩容。扩容的过程中，会进行大量的数组复制操作。而数组复制时，最终将调用System.arraycopy()方法，因此add()操作的效率还是相当高的。

LinkedList 的add()操作实现如下，它也将任意元素增加到队列的尾端：

**public** **boolean** **add**(E e){

   addBefore(e,header);*//将元素增加到header的前面*  
   **return** **true**;  
}

其中addBefore()的方法实现如下：

**private** Entry<E> **addBefore**(E e,Entry<E> entry){

    Entry<E> newEntry = **new** Entry<E>(e,entry,entry.previous);  
    newEntry.provious.next=newEntry;  
    newEntry.next.previous=newEntry;  
    size++;  
    modCount++;  
    **return** newEntry;  
}

可见，LinkedList由于使用了链表的结构，因此不需要维护容量的大小。从这点上说，它比ArrayList有一定的性能优势，然而，每次的元素增加都需要新建一个Entry对象，并进行更多的复制操作。在频繁的系统调用中，对性能会产生一定的影响。

#### 2、增加元素到列表任意位置

除了提供元素到List的尾端，List接口还提供了在任意位置插入元素的方法：**void add(int index, E element);**

由于实现的不同，ArrayList和LinkedList在这个方法上存在一定的性能差异，由于ArrayList是基于数组实现的，而数组时一块连续的内存空间，如果在数组的任意位置插入元素，必然导致在该位置后的所有元素需要重新排列，因此，其效率相对会比较低。

以下代码是ArrayList中的实现：

public void add(**int** **index**,E element){

**if**(**index**>size||**index**<0)  
       throw new IndexOutOfBoundsException(  
        "Index:"+**index**+",size: "+size);  
   ensureCapacity(size+1);  
   System.arraycopy(elementData,**index**,elementData,**index**+1,size-**index**);  
   elementData[**index**] = element;  
   size++;

}

可以看到每次插入操作，都会进行一次数组复制。而这个操作在增加元素到List尾端的时候是不存在的，大量的数组重组操作会导致系统性能低下。并且插入元素在List中的位置越是靠前，数组重组的开销也越大。

而LinkedList此时显示了优势：

**public** **void** **add**(**int** index,E element){

    addBefore(element,(index==size?header:entry(index)));  
}

可见，对LinkedList来说，在List的尾端插入数据与在任意位置插入数据是一样的，不会因为插入的位置靠前而导致插入的方法性能降低。

#### 3、删除任意位置元素

对于元素的删除，List接口提供了在任意位置删除元素的方法：

**public** E **remove**(**int** index);

对ArrayList来说，remove()方法和add()方法是雷同的。在任意位置移除元素后，都要进行数组的重组。ArrayList的实现如下：

public E remove(**int** **index**){

    RangeCheck(**index**);  
    modCount++;  
    E oldValue=(E) elementData[**index**];  
   **int** numMoved=size-**index**-1;  
   **if**(numMoved>0)  
      System.arraycopy(elementData,**index**+1,elementData,**index**,numMoved);  
    elementData[--size]=null;  
    **return** oldValue;  
}

可以看到，在ArrayList的每一次有效的元素删除操作后，都要进行数组的重组。并且删除的位置越靠前，数组重组时的开销越大。

public E remove(**int** **index**){

**return** remove(entry(**index**));           
}  
private Entry<E> entry(**int** **index**){  
   **if**(**index**<0 || **index**>=size)  
       throw new IndexOutBoundsException("Index:"+**index**+",size:"+size);  
       Entry<E> e= header;  
       **if**(**index**<(size>>1)){//要删除的元素位于前半段  
          **for**(**int** i=0;i<=**index**;i++)  
              e=e.next;  
      }**else**{  
          **for**(**int** i=size;i>**index**;i--)  
              e=e.previous;  
      }  
    **return** e;  
}

在LinkedList的实现中，首先要通过循环找到要删除的元素。如果要删除的位置处于List的前半段，则从前往后找；若其位置处于后半段，则从后往前找。因此无论要删除较为靠前或者靠后的元素都是非常高效的；但要移除List中间的元素却几乎要遍历完半个List，在List拥有大量元素的情况下，效率很低。

#### 4、容量参数

容量参数是ArrayList和Vector等基于数组的List的特有性能参数。它表示初始化的数组大小。当ArrayList所存储的元素数量超过其已有大小时。它便会进行扩容，数组的扩容会导致整个数组进行一次内存复制。因此合理的数组大小有助于减少数组扩容的次数，从而提高系统性能。

**public**  **ArrayList**(){

**this**(10);    
}  
**public** **ArrayList** (**int** initialCapacity){  
    **super**();  
    **if**(initialCapacity<0)  
        **throw** **new** IllegalArgumentException("Illegal Capacity:"+initialCapacity)  
    **this**.elementData=**new** Object[initialCapacity];  
}

ArrayList提供了一个可以指定初始数组大小的构造函数：

**public** **ArrayList**(**int** initialCapacity)

现以构造一个拥有100万元素的List为例，当使用默认初始化大小时，其消耗的相对时间为125ms左右，当直接制定数组大小为100万时，构造相同的ArrayList仅相对耗时16ms。

#### 5、遍历列表

遍历列表操作是最常用的列表操作之一，在JDK 1.5之后，至少有3种常用的列表遍历方式：

* forEach操作
* 迭代器
* for循环

String tmp;

**long** start=System.currentTimeMills();    *//ForEach*  
**for**(String s:list){  
    tmp=s;  
}  
System.out.println("foreach spend:"+(System.currentTimeMills()-start));  
start = System.currentTimeMills();  
**for**(Iterator<String> it=list.iterator();it.hasNext();){      
   tmp=it.next();  
}  
System.out.println("Iterator spend;"+(System.currentTimeMills()-start));  
start=System.currentTimeMills();  
**int** size=;list.size();  
**for**(**int** i=0;i<size;i++){                       
    tmp=list.get(i);  
}  
System.out.println("for spend;"+(System.currentTimeMills()-start));

构造一个拥有100万数据的ArrayList和等价的LinkedList，使用以上代码进行测试，测试结果：



可以看到，最简便的ForEach循环并没有很好的性能表现，综合性能不如普通的迭代器，而是用for循环通过随机访问遍历列表时，ArrayList表项很好，但是LinkedList的表现却无法让人接受，甚至没有办法等待程序的结束。这是因为对LinkedList进行随机访问时，总会进行一次列表的遍历操作。性能非常差，应避免使用。

#### 6、总结

ArrayList和LinkedList在性能上各有优缺点，都有各自所适用的地方，总的说来可以描述如下：

1. 对ArrayList和LinkedList而言，在列表末尾增加一个元素所花的开销都

是固定的。

对ArrayList而言，主要是在内部数组中增加一项，指向所添加的元素，偶尔可能会导致对数组重新进行分配；

而对LinkedList而言，这个开销是统一的，分配一个内部Entry对象。

2．在ArrayList的中间插入或删除一个元素意味着这个列表中剩余的元素都会被移动；而在LinkedList的中间插入或删除一个元素的开销是固定的。

3．LinkedList不支持高效的随机元素访问。

4．ArrayList的空间浪费主要体现在在list列表的结尾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相当的空间

可以这样说：当操作是在一列数据的后面添加数据而不是在前面或中间,并且需要随机地访问其中的元素时,使用ArrayList会有更好的性能；当操作是在一列数据的前面或中间添加或删除数据,并且按照顺序访问其中的元素时,就应该使用LinkedList了。