ArrayList、Vector、LinkedList底层原理

ArrayList

ArrayList 实现于 List、RandomAccess 接口。可以插入空数据，也支持随机访问。

ArrayList相当于动态数据，其中最重要的两个属性分别是: elementData 数组，以及 size 大小。 在调用 add() 方法的时候：

```
public boolean add(E e) {
ensureCapacityInternal(size + 1); // Increments modCount!!
elementData[size++] = e;
return true;
}
```

- 首先进行扩容校验。
- 将插入的值放到尾部，并将 size + 1 。

如果是调用 add(index,e) 在指定位置添加的话：

```
public void add(int index, E element) {
rangeCheckForAdd(index);

ensureCapacityInternal(size + 1); // Increments modCount!!
//复制，向后移动
System.arraycopy(elementData, index, elementData, index + 1,
size - index);
elementData[index] = element;
size++;
}
```

- 也是首先扩容校验。
- 接着对数据进行复制，目的是把 index 位置空出来放本次插入的数据，并将后面的数据向后移动一个位置。

其实扩容最终调用的代码:

```
private void grow(int minCapacity) {
// overflow-conscious code
int oldCapacity = elementData.length;
int newCapacity = oldCapacity + (oldCapacity >> 1);
if (newCapacity - minCapacity < 0)
newCapacity = minCapacity;
if (newCapacity - MAX_ARRAY_SIZE > 0)
newCapacity = hugeCapacity(minCapacity);
// minCapacity is usually close to size, so this is a win:
elementData = Arrays.copyOf(elementData, newCapacity);
}
```

也是一个数组复制的过程。

由此可见 ArrayList 的主要消耗是数组扩容以及在指定位置添加数据，在日常使用时最好是指定大小，尽量减少扩容。更要减少在指定位置插入数据的操作。

Vector

Vector 也是实现于 List 接口，底层数据结构和 ArrayList 类似,也是一个动态数组存放数据。不过是在 add() 方法的时候使用 synchronized 进行同步写数据，但是开销较大，所以 Vector 是一个同步容器并不是一个并发容器。

以下是 add() 方法：

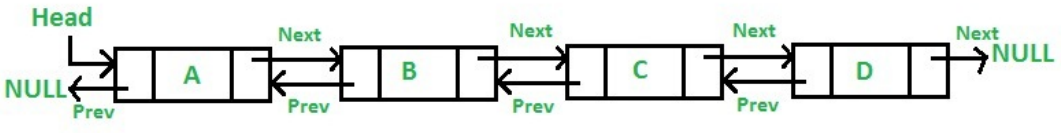
```
public synchronized boolean add(E e) {
modCount++;
ensureCapacityHelper(elementCount + 1);
elementData[elementCount++] = e;
return true;
}
```

以及指定位置插入数据:

```
public void add(int index, E element) {
insertElementAt(element, index);
}
public synchronized void insertElementAt(E obj, int index) {
```

```
modCount++;
if (index > elementCount) {
throw new ArrayIndexOutOfBoundsException(index
+ " > " + elementCount);
}
ensureCapacityHelper(elementCount + 1);
System.arraycopy(elementData, index, elementData, index + 1,
elementCount - index);
elementData[index] = obj;
elementCount++;
}
```

LinkedList 底层分析



如图所示 LinkedList 底层是基于双向链表实现的，也是实现了 List 接口，所以也拥有 List 的一些特点(JDK1.7/8 之后取消了循环，修改为双向链表)。

新增方法

```
public boolean add(E e) {
linkLast(e);
return true;
}
/** * Links e as last element. */
void linkLast(E e) {
final Node<E> l = last;
final Node<E> newNode = new Node<>(l, e, null);
last = newNode;
if (l == null)
first = newNode;
```

```
else
l.next = newNode;
size++;
modCount++;
}
```

可见每次插入都是移动指针，和 ArrayList 的拷贝数组来说效率要高上不少。

查询方法

```
public E get(int index) {
checkElementIndex(index);
return node(index).item;
}
Node<E> node(int index) {
// assert isElementIndex(index);

if (index < (size >> 1)) {
Node<E> x = first;
for (int i = 0; i < index; i++)
x = x.next;
return x;
} else {
Node<E> x = last;
for (int i = size - 1; i > index; i--)
x = x.prev;
return x;
}
}
```

由此可以看出是使用二分查找来看 index 离 size 中间距离来判断是从头结点正序查还是从尾节点倒序查。

- node()会以O(n/2)的性能去获取一个结点

- 如果索引值大于链表大小的一半，那么将从尾结点开始遍历

这样的效率是非常低的，特别是当 index 越接近 size 的中间值时。
总结：

- LinkedList 插入，删除都是移动指针效率很高。
- 查找需要进行遍历查询，效率较低。