ArrayList & LinkedList 源码解析 | MrBird

> 本文记录 ArrayList & LinkedList 源码解析，基于 JDK1.8。

ArrayList & LinkedList 源码解析

2020-08-08 | Visit count 1058052

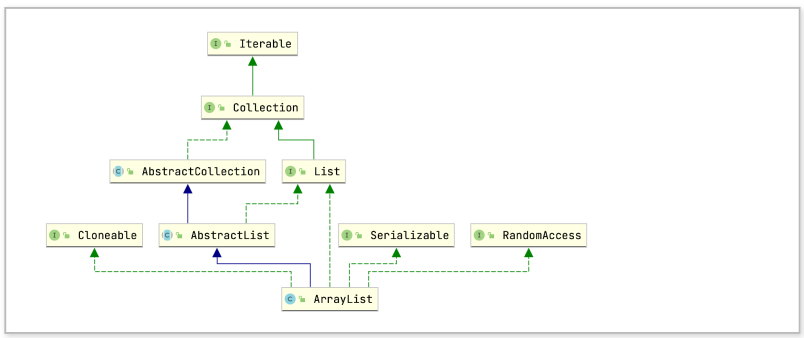
本文记录 ArrayList & LinkedList 源码解析，基于 JDK1.8。

ArrayList

ArrayList 实现了 List 接口的所有方法，可以看成是 “长度可调节的数组”，可以包含任何类型数据（包括 null，可重复）。ArrayList 大体和 Vector 一致，唯一区别是 ArrayList 非线程安全，Vector 线程安全，但 Vector 线程安全的代价较大，推荐使用 CopyOnWriteArrayList，后面文章再做记录。

类结构

ArrayList 类层级关系如下图所示：



ArrayList 额外实现了 RandomAccess 接口，关于 RandomAccess 接口的作用下面再做讨论。

ArrayList 类主要包含如下两个成员变量：

```
public class ArrayList<E> extends AbstractList<E>
        implements List<E>, RandomAccess, Cloneable, j
{

    transient Object[] elementData;
    private int size;
    ......
}
```

elementData 为 Object 类型数组，用于存放 ArrayList 数据；size 表示数组元素个数（并非数组容量）。

ArrayList 类还包含了一些常量：

```
public class ArrayList<E> extends AbstractList<E>
        implements List<E>, RandomAccess, Cloneable, j
{
    // 数组默认初始化容量为10
    private static final int DEFAULT_CAPACITY = 10;
    // 表示空数组
    private static final Object[] EMPTY_ELEMENTDATA =
    // 也是空数组，和EMPTY_ELEMENTDATA区分开
    private static final Object[] DEFAULTCAPACITY_EMPT
}
```

方法解析

知识储备

Arrays 类的 *copyOf(U[] original, int newLength, Class<? extends T[]> newType)* 方法用于复制指定数组 original 到新数组，新数组的长度为 newLength，新数组元素类型为 newType。

1. 如果新数组的长度大于旧数组，那么多出的那部分用 null 填充；
2. 如果新数组的长度小于旧数组，那么少的那部分直接截取掉。

举两个例子：

```
Long[] array1 = new Long[]{1L, 2L, 3L};
Object[] array2 = Arrays.copyOf(array1, 5, Object[].cl
System.out.println(Arrays.toString(array2)); // [1, 2,

Object[] array3 = Arrays.copyOf(array1, 1, Object[].cl
System.out.println(Arrays.toString(array3)); // [1]
```

重载方法 *copyOf(T[] original, int newLength)* 用于复制指定数组 original 到新数组，新数组的长度为 newLength，新数组元素类型和旧数组一致。

copyOf 方法内部调用 System 类的 native 方法 *arraycopy(Object src, int srcPos,Object dest, int destPos, int length)* ：

1. *src* ： 需要被拷贝的旧数组；
2. *srcPos* ： 旧数组开始拷贝的起始位置；
3. *dest* ： 拷贝目标数组；
4. *destPos* ： 目标数组的起始拷贝位置；
5. *length* ： 拷贝的长度。

举例：

```
Long[] array1 = new Long[]{1L, 2L, 3L};
Object[] array2 = new Object[5];
System.arraycopy(array1, 0, array2, 0, 3);
System.out.println(Arrays.toString(array2)); // [1, 2,
```

指定位置插入元素：

```
Long[] array1 = new Long[]{1L, 2L, 3L, null, null, nul
int index = 1;
System.arraycopy(array1, index, array1, index + 1, 3 -
array1[index] = 0L;
System.out.println(Arrays.toString(array1)); // [1, 0,
```

构造函数

public ArrayList(int initialCapacity) ：

```
public ArrayList(int initialCapacity) {
    if (initialCapacity > 0) {
```

```
            this.elementData = new Object[initialCapacity]
        } else if (initialCapacity == 0) {
            this.elementData = EMPTY_ELEMENTDATA;
        } else {
            throw new IllegalArgumentException("Illegal Ca
                                               initialCapa
        }
    }
```

创建容量大小为 initialCapacity 的 ArrayList，如果 initialCapacity 小于 0，则抛出 IllegalArgumentException 异常；如果 initialCapacity 为 0，则 elementData 为 EMPTY_ELEMENTDATA。

public ArrayList() ：

```
    public ArrayList() {
        this.elementData = DEFAULTCAPACITY_EMPTY_ELEMENTDA
    }
```

空参构造函数，elementData 为 DEFAULTCAPACITY_EMPTY_ELEMENTDATA。

public ArrayList(Collection<? extends E> c) ：

```
    public ArrayList(Collection<? extends E> c) {
        elementData = c.toArray();
        if ((size = elementData.length) != 0) {
            // c.toArray might (incorrectly) not return Ob
            if (elementData.getClass() != Object[].class)
                elementData = Arrays.copyOf(elementData, s
        } else {
            // replace with empty array.
            this.elementData = EMPTY_ELEMENTDATA;
        }
    }
```

创建一个包含指定集合 c 数据的 ArrayList。上面为什么要多此一举使用 *Arrays.copyOf(elementData, size, Object[].class)* 复制一遍数组呢？这是因为在某些情况下调用集合的 toArray() 方法返回的类型并不是 Object[].class，比如：

```
    Long[] array1 = {1L, 2L};
    List<Long> list1 = Arrays.asList(array1);
    Object[] array2 = list1.toArray();
    System.out.println(array2.getClass() == Object[].class
```

```
List<Long> list2 = new ArrayList<>();
System.out.println(list2.toArray().getClass() == Objec
```

add(E e)

`add(E e)` 用于尾部添加元素：

```
public boolean add(E e) {
    // 用于确定数组容量
    ensureCapacityInternal(size + 1);
    elementData[size++] = e;
    return true;
}
```

假如现在我们通过如下代码创建了一个 ArrayList 实例：

```
ArrayList<String> list = new ArrayList<>();
list.add("hello");
```

内部过程如下：

```
public boolean add(E e) {
    // 用于确定数组容量，e=hello，size=0
    ensureCapacityInternal(size + 1);
    // 末尾添加元素，然后size递增1
    elementData[size++] = e;
    return true;
}


private void ensureCapacityInternal(int minCapacity) {
    ensureExplicitCapacity(calculateCapacity(elementDat
}

private static int calculateCapacity(Object[] elementDa
    if (elementData == DEFAULTCAPACITY_EMPTY_ELEMENTDAT
        // DEFAULT_CAPACITY=10，minCapacity=1，故返回10
        return Math.max(DEFAULT_CAPACITY, minCapacity);
    }
    return minCapacity;
}

private void ensureExplicitCapacity(int minCapacity) {
    modCount++;

    // minCapacity=10，elementData.length=0，所以调用gro
    if (minCapacity - elementData.length > 0)
        grow(minCapacity);
}

private void grow(int minCapacity) { //minCapacity=10
    // oldCapacity=0
    int oldCapacity = elementData.length;
```

```
        // newCapacity为oldCapacity的1.5倍，这里为0
        int newCapacity = oldCapacity + (oldCapacity >> 1)
        // newCapacity=0，minCapacity=10，所以该条件成立
        if (newCapacity - minCapacity < 0)
            // newCapacity=10
            newCapacity = minCapacity;
        if (newCapacity - MAX_ARRAY_SIZE > 0)
            newCapacity = hugeCapacity(minCapacity);
        // 复制到新数组，数组容量为10
        elementData = Arrays.copyOf(elementData, newCapaci
    }

    private static int hugeCapacity(int minCapacity) {
        if (minCapacity < 0) // overflow
            throw new OutOfMemoryError();
        // MAX_ARRAY_SIZE常量值为Integer.MAX_VALUE - 8，通过
        // 这段逻辑我们可以知道，ArrayList最大容量为Integer.M
        return (minCapacity > MAX_ARRAY_SIZE) ?
            Integer.MAX_VALUE :
            MAX_ARRAY_SIZE;
    }
```

通过上面源码分析我们可以知道：

1. 任何一个空的 ArrayList 在添加第一个元素时，内部数组容量将被扩容为 10；
2. 扩容时，newCapacity 为 oldCapacity 的 1.5 倍；
3. 数组容量最大为 Integer.MAX_VALUE；
4. 尾部添加元素不用移动任何元素，所以速度快。

add(int index, E element)

add(int index, E element) 用于在指定位置添加元素：

```
    public void add(int index, E element) {
        // 下标检查
        rangeCheckForAdd(index);
        // 确定数组容量，和上面add(E e)方法介绍的一致
        ensureCapacityInternal(size + 1);
        // 将原来index后面的所有元素往后面移动一个位置
        System.arraycopy(elementData, index, elementData, 
                         size - index);
        // index处放入新元素
        elementData[index] = element;
        // size递增
        size++;
    }

    private void rangeCheckForAdd(int index) {
        // 下标比size大或者下标小于0，都会抛出下标越界异常
        if (index > size || index < 0)
```

```
            throw new IndexOutOfBoundsException(outOfBounds
    }
```

这里涉及到元素移动，所以速度较慢。

get(int index)

get(int index) 获取指定位置元素：

```
    public E get(int index) {
        // 下标合法性检查
        rangeCheck(index);
        // 直接返回数组指定位置元素
        return elementData(index);
    }

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

get 方法直接返回数组指定下标元素，速度非常快。

set(int index, E element)

set(int index, E element) 设置指定位置元素为指定值：

```
    public E set(int index, E element) {
        // 下标合法性检查
        rangeCheck(index);
        // 根据下标获取旧值
        E oldValue = elementData(index);
        // 设置新值
        elementData[index] = element;
        // 返回旧值
        return oldValue;
    }
```

set 方法不涉及元素移动和遍历，所以速度快。

remove(int index)

remove(int index) 删除指定位置元素：

```
    public E remove(int index) {
        rangeCheck(index);

        modCount++;
        // 获取指定位置元素（需要被删除的元素）
        E oldValue = elementData(index);

        int numMoved = size - index - 1;
```

```
        if (numMoved > 0)
            // 直接将index后面的元素往前移动一位，覆盖index处
            System.arraycopy(elementData, index+1, element
                             numMoved);
        elementData[--size] = null; // clear to let GC do
        // 返回被删除的元
        return oldValue;
    }
```

上述方法涉及到元素移动，所以效率也不高。

remove(Object o)

remove(Object o) 删除指定元素：

```
    // 遍历数组，找到第一个目标元素，然后删除
    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;
    }
    // 逻辑和remove一致，都是将index后面的元素往前移动一位，覆
    private void fastRemove(int index) {
        modCount++;
        int numMoved = size - index - 1;
        if (numMoved > 0)
            System.arraycopy(elementData, index+1, element
                             numMoved);
        elementData[--size] = null; // clear to let GC do
    }
```

方法涉及到数组遍历和元素移动，效率也不高。

trimToSize()

trimToSize() 源码：

```
    public void trimToSize() {
        modCount++;
        if (size < elementData.length) {
            elementData = (size == 0)
              ? EMPTY_ELEMENTDATA
```

```
                : Arrays.copyOf(elementData, size);
        }
    }
```

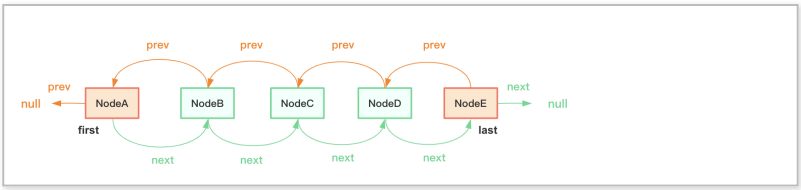
该方法用于将数组容量调整为实际元素个数大小，当一个 ArrayList 元素个数不会发生改变时，可以调用该方法减少内存占用。

> 其他方法可以自己阅读 ArrayList 源码，此外在涉及增删改的方法里，我们都看到了 modCount++ 操作，和之前介绍 HashMap 源码时一致，用于快速失败。

LinkedList

类结构

LinkedList 底层采用双向链表结构存储数据，允许重复数据和 null 值，长度没有限制：



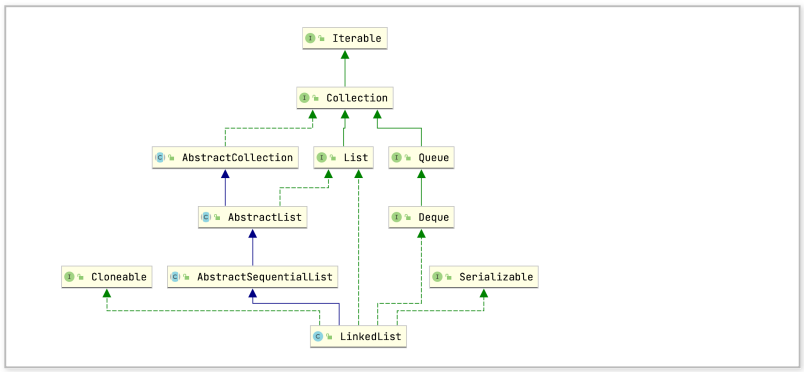
每个节点用内部类 Node 表示：

```
    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;
        }
    }
```

Node 节点包含 item（存储数据），next（后继节点）和 prev（前继节点）。数组内存地址必须连续，而链表就没有这个限制了，Node 可以分布于各个内存地址，它们之间的关系通过 prev 和 next 维护。

LinkedList 类关系图：



可以看到 LinkedList 类并没有实现 RandomAccess 接口，额外实现了 Deque 接口，所以包含一些队列方法。

LinkedList 包含如下成员变量：

```
// 元素个数，默认为0
transient int size = 0;

// 表示第一个节点，第一个节点必须满足(first == null && las
transient Node<E> first;

// 表示最后一个节点，最后一个节点必须满足(first == null &&
transient Node<E> last;
```

方法解析

构造函数

LinkedList() ：

```
public LinkedList() {
}
```

空参构造函数，默认 size 为 0，每次添加新元素都要创建 Node 节点。

LinkedList(Collection<? extends E> c) ：

```
public LinkedList(Collection<? extends E> c) {
    this();
    addAll(c);
}

public boolean addAll(Collection<? extends E> c) {
```

```
    return addAll(size, c);
}

public boolean addAll(int index, Collection<? extends E
    checkPositionIndex(index);

    Object[] a = c.toArray();
    int numNew = a.length;
    if (numNew == 0)
        return false;

    Node<E> pred, succ;
    if (index == size) {
        succ = null;
        pred = last;
    } else {
        succ = node(index);
        pred = succ.prev;
    }
    // 循环创建节点，设置prev，next指向
    for (Object o : a) {
        @SuppressWarnings("unchecked") E e = (E) o;
        Node<E> newNode = new Node<>(pred, e, null);
        if (pred == null)
            first = newNode;
        else
            pred.next = newNode;
        pred = newNode;
    }

    if (succ == null) {
        last = pred;
    } else {
        pred.next = succ;
        succ.prev = pred;
    }

    size += numNew;
    modCount++;
    return true;
}
```

该构造函数用于创建 LinkedList，并往里添加指定集合元素。

add(int index, E element)

add(int index, E element) 指定下标插入元素：

```
public void add(int index, E element) {
    // 下标合法性检查
    checkPositionIndex(index);

    if (index == size)
        // 如果插入下标等于size，说明是在尾部插入，执行尾音
        linkLast(element);
    else
        // 如果不是尾插入，则在指定下标节点前插入
```

```
        linkBefore(element, node(index));
}

private void checkPositionIndex(int index) {
    if (!isPositionIndex(index))
        throw new IndexOutOfBoundsException(outOfBounds
}

private boolean isPositionIndex(int index) {
    return index >= 0 && index <= size;
}

void linkLast(E e) {
    // 获取最后一个节点
    final Node<E> l = last;
    // 创建一个新节点，prev为原链表最后一个节点，next为nul
    final Node<E> newNode = new Node<>(l, e, null);
    // 更新last为新节点
    last = newNode;
    if (l == null)
        // 如果原链表最后一个节点为null，说明原链表没有节点
        first = newNode;
    else
        // 否则更新原链表最后一个节点的next为新节点
        l.next = newNode;
    // size递增
    size++;
    // 模数递增，用于快速失败
    modCount++;
}

void linkBefore(E e, Node<E> succ) {
    // succ为原链表指定index位置的节点，获取其prev节点
    final Node<E> pred = succ.prev;
    // 创建新节点，prev为原链表指定index位置的节点的prev节
    final Node<E> newNode = new Node<>(pred, e, succ);
    // 将原链表指定index位置的节点的prev更新为新节点
    succ.prev = newNode;
    if (pred == null)
        // 如果链表指定index位置的节点的prev为null，说明原
        first = newNode;
    else
        // 否则更新原链表指定index位置的节点的prev的next节
        pred.next = newNode;
    // size递增
    size++;
    // 模数递增，用于快速失败
    modCount++;
}

// 采用二分法遍历每个Node节点，直到找到index位置的节点
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;
        }
    }
```

代码较为简单，无非就是设置节点的 prev 和 next 关系。可以看到，除了头插和尾插外，在链表别的位置插入新节点，涉及到节点遍历操作，所以我们常说的链表插入速度快，指的是插入节点改变前后节点的引用过程很快。

get(int index)

get(int index) 获取指定下标元素：

```
    public E get(int index) {
        checkElementIndex(index);
        return node(index).item;
    }

    // 采用二分法遍历每个Node节点，直到找到index位置的节点
    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;
        }
    }
```

代码较为简单，就是通过 node 函数查找指定 index 下标 Node，然后获取其 item 属性值，节点查找需要遍历。

set(int index, E element)

set(int index, E element) 设置指定下标节点的 item 为指定值：

```
    public E set(int index, E element) {
        // 下标合法性检查
        checkElementIndex(index);
        // 获取index下标节点
        Node<E> x = node(index);
        // 获取旧值
        E oldVal = x.item;
```

```
    // 设置新值
    x.item = element;
    // 返回旧值
    return oldVal;
}

// 采用二分法遍历每个Node节点，直到找到index位置的节点
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;
    }
}
```

可以看到，set 方法也需要通过遍历查找目标节点。

remove(int index)

remove(int index) 删除指定下标节点：

```
public E remove(int index) {
    checkElementIndex(index);
    return unlink(node(index));
}

E unlink(Node<E> x) {
    // assert x != null;
    final E element = x.item;
    final Node<E> next = x.next;
    final Node<E> prev = x.prev;

    if (prev == null) {
        first = next;
    } else {
        prev.next = next;
        x.prev = null;
    }

    if (next == null) {
        last = prev;
    } else {
        next.prev = prev;
        x.next = null;
    }

    x.item = null;
    size--;
    modCount++;
```

```
        return element;
    }
```

remove(int index) 通过 node 方法找到需要删除的节点，然后调用 unlink 方法改变删除节点的 prev 和 next 节点的前继和后继节点。

> 剩下的方法可以自己阅读源码。

RandomAccess 接口

RandomAccess 接口是一个空接口，不包含任何方法，只是作为一个标识：

```
package java.util;

public interface RandomAccess {
}
```

实现该接口的类说明其支持快速随机访问，比如 ArrayList 实现了该接口，说明 ArrayList 支持快速随机访问。所谓快速随机访问指的是通过元素的下标即可快速获取元素对象，无需遍历，而 LinkedList 则没有这个特性，元素获取必须遍历链表。

在 Collections 类的 *binarySearch(List<? extends Comparable<? super T>> list, T key)* 方法中，可以看到 RandomAccess 的应用：

```
public static <T>
int binarySearch(List<? extends Comparable<? super T>>
    if (list instanceof RandomAccess || list.size()<BI
        return Collections.indexedBinarySearch(list, k
    else
        return Collections.iteratorBinarySearch(list, 
}
```

当 list 实现了 RandomAccess 接口时，调用 indexedBinarySearch 方法，否则调用 iteratorBinarySearch。所以当我们遍历集合时，如果集合实现了 RandomAccess 接口，优先选择普通 for 循环，其次 foreach；遍历未实现 RandomAccess 的接口，优先选择 iterator 遍历。

全文完