ArrayList源码分析

ArrayList是一个链表，但是底层是通过动态数组来实现的，能够进行动态扩容。

- ArrayList重要属性

```
// 默认初始化容量。即调用构造函数的时候，不指定任何参数，
// 第一次调用 add() 创建的数组容量大小为10
private static final int DEFAULT_CAPACITY = 10;

// 空数组对象。当调用构造函数时，传入的initCapacity为0时，
// 会执行this.elementData
private static final Object[] EMPTY_ELEMENTDATA = {};    EMPTY_ELEMENTDATA

// 当调用构造函数时，若不指定任何参数，那么会执行
// this.elementData = DEFAULTCAPACITY_EMPTY_ELEMENTDATA;
private static final Object[] DEFAULTCAPACITY_EMPTY_ELEMENTDATA = {};

// 内部数组对象的引用
transient Object[] elementData;

// 链表中元素的个数，初始化元素个数为 0
private int size;
```

- ArrayList的构造方法

```
// 强烈不建议传入initCapacity == 0。如果这样的话，后续添加元素的时候，会
// 造成很多次的扩容操作。影响性能

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

        //根据传入的initCapacity来创建一个Object[] elementData数组对象。
        // 即ArrayList内部实际上是一个数组

        this.elementData = new Object[initialCapacity];
    } else if (initialCapacity == 0) {

        //一个空的数组对象 Object[] EMPTY_ELEMENTDATA = {}
        this.elementData = EMPTY_ELEMENTDATA;
    } else {
        throw new IllegalArgumentException("Illegal Capacity: " +
                initialCapacity);
    }
}
```

```
public ArrayList() {

    // 也是一个空的数组对象，即初始化 ArrayList 在传入 initCapacity
    // 为 0 或者不传入 initCapacity 的情况下，只有在第一次调用 add()
    // 函数时才会创建一个数组对象，然后赋值给 elementData
    this.elementData = DEFAULTCAPACITY_EMPTY_ELEMENTDATA;
}
```

- Notes

    1. private static final Object[] EMPTY_ELEMENTDATA = {}， 而private static final Object[] DEFAULTCAPACITY_EMPTY_ELEMENTDATA = {}。 共同点二者都是一个空数组，但是二者用法不同。
    2. 若指定了initCapacity，那么如果该initCapacity == 0，那么会创建一个空 的elementData，即this.elementData = EMPTY_ELEMENTDATA
    3. 若没有指定initCapacity，即调用构造函数什么也不指定，那么也会创建一个空 的elementData，但是此时是this.elementData = DEFAULTCAPACITY_EMPTY_ELEMENTDATA
    4. important method分析

        1. add()

```
public boolean add(E e) {
        //每次调用add()函数，都会执行ensureCapacityInternal()函数
        // 传入参数为size + 1，表明此时插入的位置为 size，那么必须
        // 确保此时数组的大小为 size + 1
        // size 是当前链表中元素的个数
        ensureCapacityInternal(size + 1);

        // 将该元素添加到末尾。
        elementData[size++] = e;
        return true;
}

//当初始化时，元素个数肯定为0，当第一次执行add操作，此时需要的
// minCapacity 是 size+1，又此时size的值也为0，即此时minCapacity = 1

private void ensureCapacityInternal(int minCapacity) {

        // 说明：
        // 1.当elementData == DEFAULTCAPACITY_EMPTY_ELEMENTDATA时，
在
        // 第一次调用时，calculateCapacity的返回值为10。也就是说此时需
要
        // 的数组的容量应该为 10，而不是 1，并且一直调用 add() 方法直
到
        // size+1 == 11 时，才会之后每次返回 size+1 即此时 需要的数组
```

```
的长
        // 度为 size + 1，即 minCapacity

        // 2.否则，若elementData !=
DEFAULTCAPACITY_EMPTY_ELEMENTDATA，
        // 那么直接返回 minCapacity，即 size+1

        ensureExplicitCapacity(calculateCapacity(elementData,
minCapacity));
}


// 该函数用来计算需要的数组空间的大小。主要是为了elementData ==
// DEFAULTCAPACITY_EMPTY_ELEMENTDATA 服务的。
// 如果不相等，那么直接就返回minCapacity，即 size+1。否则，在相等的
// 情况下，直到minCapacity >= 11，才会返回 minCapacity。

private static int calculateCapacity(Object[] elementData, int
minCapacity) {
    if (elementData == DEFAULTCAPACITY_EMPTY_ELEMENTDATA) {
        //第一次调用时，由于minCapacity = 1，所以此时若条件成立，那么
返回值为DEFAULT_CAPACITY
        return Math.max(DEFAULT_CAPACITY, minCapacity);
    }
    return minCapacity; // 即size + 1
}

// 此函数用来执行扩容操作。根据传入的minCapacity与当前
elementData.length相比，
// 来决定是否需要扩容。真正的扩容函数为 grow() 如果需要的数组的最小的容
量即
// minCapacity 大于数组此时的长度 elementData.length，那么需要进行扩
容
private void ensureExplicitCapacity(int minCapacity) {
    modCount++;

    // 判断此时需不需要扩容操作，判断依据：若minCapacity的值如果比
elementData.length值大，
    // 则需要扩容，且每次扩容长度均为原来的1.5倍。注意：

    // 对于elementData == DEFAULTCAPACITY_EMPTY_ELEMENTDATA和
EMPTY_ELEMENTDATA这两种情况：
    // 第一次扩容，前者直接扩容为长度10的数组。然后只有在 size+1 的大于
10之后才会进行第二次扩容。
    // 而后每一次扩容都会扩容为原来数组长度的 1.5 倍

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

private void grow(int minCapacity) {
    int oldCapacity = elementData.length;

    // 容量增加为原来的1.5倍。注意当oldCapacity的值为0，或者1时，
```

```
    // 会出现newCapacity - minCapacity < 0
    // 首次扩容的情况下：有以下几个特例：

    // 1.当构造函数没有传入initCapacity参数，那么首次扩容会
    // 直接扩容为10。即第一个 if 语句成立。之后只有当
    // size+1 >= 11，且mimCapacity == size + 1 >
elementData>length时
    // 才会再次扩容，此时容量变为原来的1.5倍。

    // 2.当构造函数传入 initCapacity 参数，如果为0。那么第一次添加
    // 元素便会扩容。且扩容后容量为1。再次添加元素，仍会扩容，容量
    // 变为2。再次添加元素 又会扩容，容量变为3。然后 再次添加元素，仍然
会扩容，
    // 容量变为4。再次添加元素，仍然会扩容，容量变为6。....
    // 可见，扩容操作太频繁。很影响性能。


    int newCapacity = oldCapacity + (oldCapacity >> 1);
    if (newCapacity - minCapacity < 0)
        newCapacity = minCapacity;
    if (newCapacity - MAX_ARRAY_SIZE > 0)
        newCapacity = hugeCapacity(minCapacity);

    //容操作需要调用 Arrays.copyOf() 把原数组整个复制到新数组中

    elementData = Arrays.copyOf(elementData, newCapacity);
}
```

2. remove()

```
// 发现 ArrayList 在执行删除操作时，仅仅将待删除元素的后面元素即从索引
// index + 1 开始，将 [index + 1,size -1] 范围内所有的元素都移动到
// [index,size-2] 范围内，并将 elementData[size-1] 位置的元素置为
null
// 从而在下一次 GC 时将其删除

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

    modCount++;
    E oldValue = elementData(index);

    int numMoved = size - index - 1;
    if (numMoved > 0)
        System.arraycopy(elementData, index + 1, elementData,
index,
                numMoved);
    elementData[--size] = null; // clear to let GC do its work
```

```
        return oldValue;
    }
```

Notes：为了不影响性能，强烈建议在构造函数里不传入任何值。 或者就算要传入值，最好传入更比较大一点的数。从而避免频繁扩容操作。影响性能。