源码笔记

1 ArrayList

1.1 ArrayList 源码结论：

- ArrayList 中维护了一个Object 类型的数组elementData.

> transient Object[] elementData; //表示该属性不会被序列化

- 当创建ArrayList对象时，如果使用的是无参构造器，则初始elementData 容量为0.第一次添加，则扩容elementData 为10，如果需要再次扩容，则扩容elementData为1.5倍。
- 如果使用的是指定大小的构造器，则初始elementData 容量为指定大小，如果需要扩容，则直接扩容elementData为1.5倍。
- 容量最大为 $2^{31}-1$
- 移除元素，数组并不缩容。

```
public static void main(String[] args) {
    ArrayList list = new ArrayList(8);
    ArrayList list = new ArrayList();


    for (int i = 0; i <= 15; i++) {
        list.add(i);
    }
    list.add(100);
    list.add(200);
    list.add(null);

    for (Object o : list) {
        System.out.println(o);
    }
}
```

对上述代码进行debugger 测试，追踪源码。

```
modCount 结构上修改 列表大小的次数。
```

1.2 空参构造器

- 构造器源码

```
//源码注释，显示创建一个容量大小为10的空列表，
public ArrayList() {
    this.elementData = DEFAULTCAPACITY_EMPTY_ELEMENTDATA;
}
```

但是，实际扩容是在第一次添加元素时候扩容为10.

- 追踪add()

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

- 追踪 ensureCapacityInternal（）其作用确定列表容量够用，

size 列表包含的元素数。执行add（）,意味着元素数要加1.

```
private void ensureCapacityInternal(int minCapacity) {
    ensureExplicitCapacity(calculateCapacity(elementData, minCapacity));
}
```

此时传递的minCapacity为1，

```
private static final int DEFAULT_CAPACITY = 10;
```

- 将minCapacity 和DEFAULT_CAPACITY 比较，返回最大值，10

```
private static int calculateCapacity(Object[] elementData, int minCapacity) {
    if (elementData == DEFAULTCAPACITY_EMPTY_ELEMENTDATA) {
        return Math.max(DEFAULT_CAPACITY, minCapacity);
    }
    return minCapacity;
}
```

- 此时minCapacity 值为10，elementData.length为0。不执行grow(),也就是扩容。

```
private void ensureExplicitCapacity(int minCapacity) {
    //表结构改变+1
    modCount++;

    // overflow-conscious code
    if (minCapacity - elementData.length > 0)
        grow(minCapacity);
}
```

1.3 ArrayList 扩容

- 当 minCapacity 为11时，上述过程走到

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

满足条件执行grow（）

- 追踪grow()

```
private void grow(int minCapacity) {
    //获取之前，list 容量
    int oldCapacity = elementData.length;
    //新容量变为原来的1.5倍
    int newCapacity = oldCapacity + (oldCapacity >> 1);
    //不满足
    if (newCapacity - minCapacity < 0)
        newCapacity = minCapacity;
    //不满足
    if (newCapacity - MAX_ARRAY_SIZE > 0)
        newCapacity = hugeCapacity(minCapacity);
    //扩容关键代码
    elementData = Arrays.copyOf(elementData, newCapacity);
}
```

```
private static final int MAX_ARRAY_SIZE = Integer.MAX_VALUE - 8;
```

- 下面代码可以看出数组容量最大值为 2^{31}-1

```
private static int hugeCapacity(int minCapacity) {
    if (minCapacity < 0) // overflow
        throw new OutOfMemoryError();
    return (minCapacity > MAX_ARRAY_SIZE) ?
        Integer.MAX_VALUE :
        MAX_ARRAY_SIZE;
}
```

- 之后是执行扩容的代码

Arrays.copy() 参数一，要复制的数组，参数二，新数组的容量

返回值： 一个新数组，增加的部分值为null.

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

1.4 有参构造器

ArrayList list = new ArrayList(8);

追踪源码

```
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 Capacity: "+
                                           initialCapacity);
    }
}
```

扩容机制，同样和上面一样，这里就不在详细描述。

1.5 ArrayList remove 移除数据并不缩容。

想到添加元素扩容，那么移除元素缩容吗?

查看源码

```
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;
    }
```

```
    private void rangeCheck(int index) {
        if (index >= size)
            throw new IndexOutOfBoundsException(outOfBoundsMsg(index));
    }
```

```
    @SuppressWarnings("unchecked")
    E elementData(int index) {
        return (E) elementData[index];
    }
```

本地方法

```
    public static native void arraycopy(Object src,  int  srcPos,
                                         Object dest, int destPos,
                                         int length);
```

代码比较简单，就不细说了，可以看出，并不缩容。

---

2 Vector源码

2.1 基本介绍

- Vector 类定义说明

  Class Vector

  - java.lang.Object
  - - java.util.AbstractCollection
    - - java.util.AbstractList
      - - java.util.Vector

    - **Type Parameters:**

      `E` - Type of component elements

      - All Implemented Interfaces:

        `Serializable`, `Cloneable`, `Iterable<E>`, `Collection<E>`, `List<E>`, `RandomAccess`

      - Direct Known Subclasses:

        `Stack`

- 底层是一个可变对象数组。
- 默认容量10,也可以定制容量大小。扩容机制2倍。最大容量长度2^{31}-1
- 线程同步，即线程安全，如果开发中考虑线程安全可以使用Vector.

2.2 源码

2.3 级别标题

```
public static void main(String[] args) {
    Vector vector = new Vector();

    for (int i = 0; i < 15;i++) {
        vector.add(i);
    }
}
```

```
//默认容量10,
public Vector() {
    this(10);
}
```

```
public Vector(int initialCapacity) {
    this(initialCapacity, 0);
}
```

```
public Vector(int initialCapacity, int capacityIncrement) {
    super();
    if (initialCapacity < 0)
        throw new IllegalArgumentException("Illegal Capacity: "+
                                           initialCapacity);
    this.elementData = new Object[initialCapacity];
    this.capacityIncrement = capacityIncrement;
}
```

2.4 扩容源码

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

```
private void ensureCapacityHelper(int minCapacity) {
    // overflow-conscious code
    if (minCapacity - elementData.length > 0)
        grow(minCapacity);
}
```

```
protected int capacityIncrement;
```

```
private void grow(int minCapacity) {
    // overflow-conscious code
    int oldCapacity = elementData.length;
    //原来容量的2倍
    int newCapacity = oldCapacity + ((capacityIncrement > 0) ?
                                     capacityIncrement : oldCapacity);
    if (newCapacity - minCapacity < 0)
        newCapacity = minCapacity;
    if (newCapacity - MAX_ARRAY_SIZE > 0)
        newCapacity = hugeCapacity(minCapacity);
    elementData = Arrays.copyOf(elementData, newCapacity);
}
```

---

3 LinkedList

3.1 基本介绍

- LinkedList 底层维护的是一个双向链表
- LinkedList 维护了2个属性first 和 last 分别指向首节点和尾节点
- 每个节点（Node 对象），里面又维护了prev、next、item三个属性，其中通过prev 指向前一个节点，通过next 指向后一个节点，最终实现双向链表。

- LinkedList 元素的添加和删除不是通过数组完成的，相对效率较高。

3.2 检测源码

```
public static void main(String[] args) {
    LinkedList linkedList = new LinkedList();
    linkedList.add(1);
    linkedList.add(2);
    linkedList.add(3);
    System.out.println("linkedList=="+linkedList);
    Object o = linkedList.get(1);
    linkedList.remove();
    linkedList.remove(1);
    System.out.println("linkedList=="+linkedList);


}
```

3.3 添加元素

构造器

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

```
public boolean add(E e) {
    linkLast(e);
    return true;
}
```

```
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++;
}
```

```
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;
    }
}
```

3.4 删除元素

```
public E remove() {
    return removeFirst();
}
```

```
public E removeFirst() {
    final Node<E> f = first;
    if (f == null)
        throw new NoSuchElementException();
    return unlinkFirst(f);
}
```

```
private E unlinkFirst(Node<E> f) {
    // assert f == first && f != null;
    final E element = f.item;
    final Node<E> next = f.next;
    f.item = null;
    f.next = null; // help GC
    first = next;
    if (next == null)
        last = null;
    else
        next.prev = null;
    size--;
    modCount++;
    return element;
}
```

3.5 查找元素

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

```
private void checkElementIndex(int index) {
    if (!isElementIndex(index))
        throw new IndexOutOfBoundsException(outOfBoundsMsg(index));
}
```

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

```
Node<E> node(int index) {
    // assert isElementIndex(index);
    //index 和 size /2 比较。
    if (index < (size >> 1)) {
        //index 在前半部分，从前往后查找
        Node<E> x = first;
        for (int i = 0; i < index; i++)
            x = x.next;
        return x;
    } else {
        //index 在后半部分，从后往往前查找
        Node<E> x = last;
        for (int i = size - 1; i > index; i--)
            x = x.prev;
        return x;
    }
}
```

4 HashSet

4.1 基本介绍

- HashSet 的全面书名
- HashSet 实现了Set 接口
- HashSet 实际上是HashMap
- 可以存放null ,但是只能有一个
- 不保证元素是有序的，即存入的顺序和遍历输出的顺序一直。
- 不能有重复的对象

4.2 概述

HashSet 底层 是 HasHMap

添加一个元素是，先得到hash值，会转成索引值，

- 找到存储数据表table，Table 默认大小16,看到这个索引位置是否已经存放的有元素,
- 如果没有直接加入
- 如果使用equlas（） 方法比较是否相等，equals 需要重写。相等就放弃添加，如果不相同添加到最后。
- 一个链表的元素个数到达8个，并且 table 大小大于等于，MIN_TREEIFY_CAPACITY(默认64)时，转为红黑树。移除元素少于6转回链表。

4.3 源码解读

初始容量默认16，负载因子0/75

```
public HashSet() {
    map = new HashMap<>();
}
```

```
private transient HashMap<E,Object> map;
```

所以说HashSet 底层是HashMap。

HashSet 添加元素

```
public boolean add(E e) {
    return map.put(e, PRESENT)==null;
}
```

```
private static final Object PRESENT = new Object();
```

```
public V put(K key, V value) {
    return putVal(hash(key), key, value, false, true);
}
```

```
static final int hash(Object key) {
    int h;
    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
}
```

Object 类的本地方法hashCode()

```
public native int hashCode();
```

具体实现由Key 的 类型来实现。如这里key 是String 类型，String 类中的 hashCode 代码如下。

```
public int hashCode() {
    int h = hash;
    if (h == 0 && value.length > 0) {
        char val[] = value;

        for (int i = 0; i < value.length; i++) {
            h = 31 * h + val[i];
        }
        hash = h;
    }
    return h;
}
```

接下里查看重点方法putVal)

```
final V putVal(int hash, K key, V value, boolean onlyIfAbsent,
               boolean evict) {
    //定义辅助变量
    Node<K,V>[] tab; Node<K,V> p; int n, i;
    if ((tab = table) == null || (n = tab.length) == 0)
        n = (tab = resize()).length;
    if ((p = tab[i = (n - 1) & hash]) == null)
        tab[i] = newNode(hash, key, value, null);
    else {
        Node<K,V> e; K k;
        if (p.hash == hash &&
            ((k = p.key) == key || (key != null && key.equals(k))))
            e = p;
```

```
            else if (p instanceof TreeNode)
                e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);
            else {
                for (int binCount = 0; ; ++binCount) {
                    if ((e = p.next) == null) {
                        p.next = newNode(hash, key, value, null);
                        if (binCount >= TREEIFY_THRESHOLD - 1) // -1 for 1st
                            treeifyBin(tab, hash);
                        break;
                    }
                    if (e.hash == hash &&
                        ((k = e.key) == key || (key != null && key.equals(k))))
                        break;
                    p = e;
                }
            }
            if (e != null) { // existing mapping for key
                V oldValue = e.value;
                if (!onlyIfAbsent || oldValue == null)
                    e.value = value;
                afterNodeAccess(e);
                return oldValue;
            }
        }
        ++modCount;
        if (++size > threshold)
            resize();
        afterNodeInsertion(evict);
        return null;
    }
```

HashMap 静态内部Node

```
    static class Node<K,V> implements Map.Entry<K,V> {
        final int hash;
        final K key;
        V value;
        Node<K,V> next;

        Node(int hash, K key, V value, Node<K,V> next) {
            this.hash = hash;
            this.key = key;
            this.value = value;
            this.next = next;
        }

        public final K getKey()        { return key; }
        public final V getValue()      { return value; }
```

```
16      public final String toString() { return key + "=" + value; }
17
18      public final int hashCode() {
19          return Objects.hashCode(key) ^ Objects.hashCode(value);
20      }
21
22      public final V setValue(V newValue) {
23          V oldValue = value;
24          value = newValue;
25          return oldValue;
26      }
27
28      public final boolean equals(Object o) {
29          if (o == this)
30              return true;
31          if (o instanceof Map.Entry) {
32              Map.Entry<?,?> e = (Map.Entry<?,?>)o;
33              if (Objects.equals(key, e.getKey()) &&
34                  Objects.equals(value, e.getValue()))
35                  return true;
36          }
37          return false;
38      }
39  }
```

```
1   transient Node<K,V>[] table;
```