线性表-数组

数组介绍

> **线性表**是最基本、最简单、也是最常用的一种数据结构，一个线性表是n个具有相同特性的数据元素的有序数列。其中数组就是一种顺序表，顺序表存储是将数据元素放到一块连续的内存存储空间，相邻数据元素的存放地址也相邻。 生活中的线性表：火车，糖葫芦

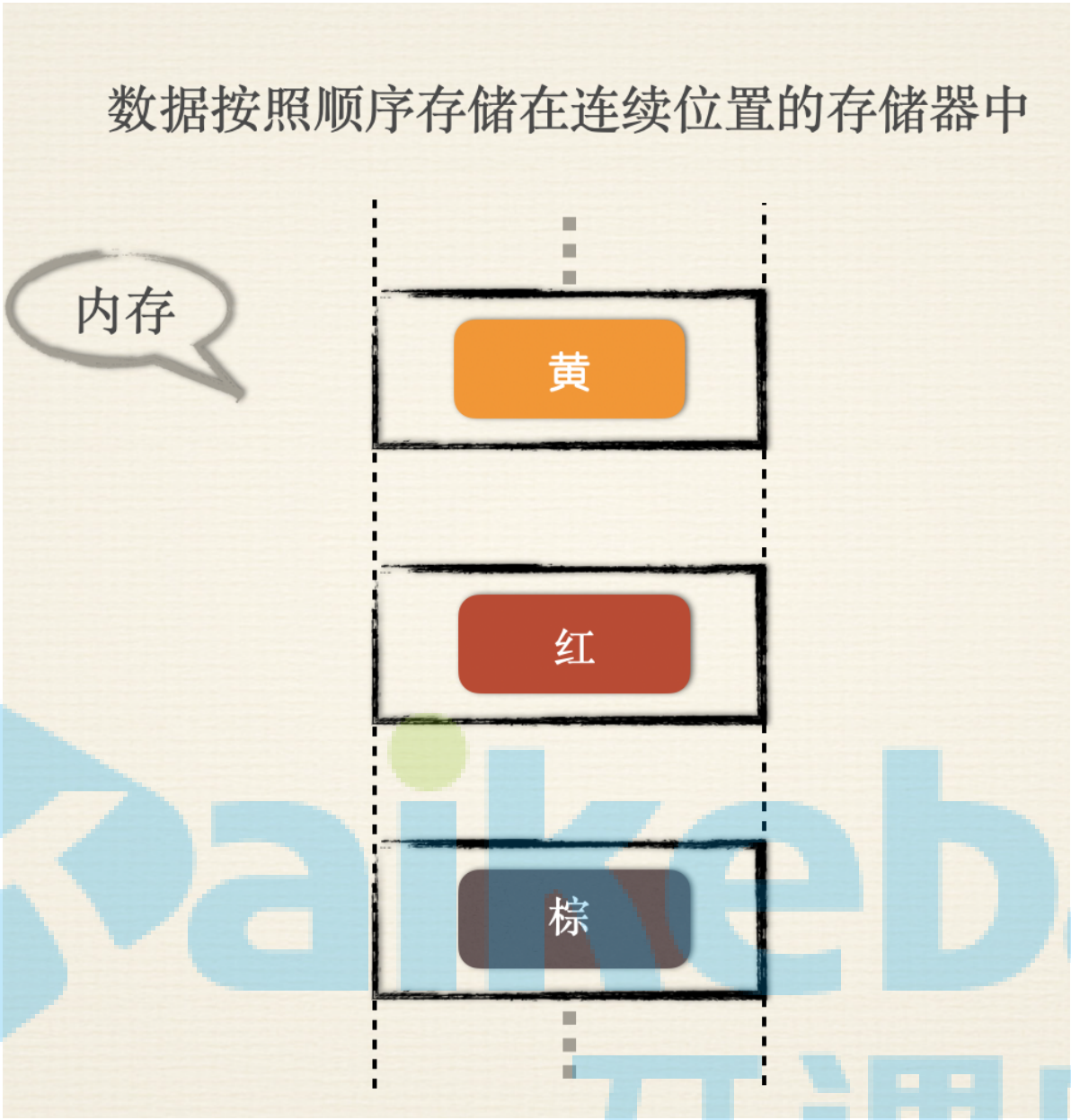
数组是一种顺序存储的线性表，可以存储多个值，每个元素可以通过索引进行访问，**所有元素的内存地址是连续的。**

优点:

- 空间利用率高。
- 查询速度高效，通过下标来直接存取。

缺点:

- 插入和删除比较慢，比如：插入或者删除一个元素时，整个表需要遍历移动元素来重新排一次顺序。
- 不可以增长长度，有空间限制,当需要存取的元素个数可能多于顺序表的元素个数时,会出现"溢出"问题.当元素个数远少于预先分配的空间时,空间浪费巨大



java中数组的3种声明

1、数组类型 [] 数组名 = {em1,em2,em3,...,emN};//声明数组的时候初始化，一共N个元素，例如：

```
int[] array = {3,5,4,8,12,5};//一共六个元素
```

2、数组类型[] 数组名 = new 数组类型[N]//用new关键字声明数组的同时指定数组长度，例如：

```
String[] str = new String[6]; 数组长度为6，即数组有六个元素
```

3、数组类型[] 数组名 = new 数组类型[] {em1,em2,em3,...,emN};用new关键字声明数组的同时初始化数组，例如：**

```
int[] array = new int[] {2,4,5,6,8,9};array数组一共五个元素。
```

//数组一旦声明，数组长度就已经确定。每个数组都有一个length属性，不可改变。可以改变数组元素。

数组的优缺点

优点：

- 空间利用率高。
- 查找速度高效，通过下标来直接查找。

缺点：

- 插入和删除比较慢，比如：插入或者删除一个元素时，整个表需要遍历移动元素来重新排一次顺序。
- 不可以增长长度，有空间限制,当需要存取的元素个数可能多于顺序表的元素个数时,会出现"溢出"问题.当元素个数远少于预先分配的空间时,空间浪费巨大。

ArrayList自实现（动态数组）

需求：底层采用数组 实现动态扩容数组（ArrayList）

目的：

1. 体会线性结构
2. 练习常用算法
3. 强化数组
4. 练习泛型

面对的问题：

- 数组一旦创建，容量不可变
- 扩容的条件

动态数组接口设计

■intsize();*// 元素的数量*

■booleanisEmpty();*// 是否为空*

■booleancontains(E element);*// 是否包含某个元素*

■voidadd(E element);*// 添加元素到最后面*

■ Eget(intindex);*// 返回index位置对应的元素*

■ Eset(intindex,E element);*// 设置index位置的元素*

■voidadd(intindex,E element);*// 往index位置添加元素*

■ Eremove(intindex);*// 删除index位置对应的元素*

■intindexOf(E element);*// 查看元素的位置*

■voidclear();*// 清除所有元素*

具体实现：

```
package com.kkb;
/**
 * 动态可变数组   自动扩容
 */
public class DyTest<E> {
    private int size = 0;//保存当前元素长度
    //定义默认初始化容量
    private final int DEFAULT_CAPACITY = 10;
    //查找失败返回值
    private final int ELEMENT_NOt_FOUND = -1;
```

```
    //用于保存数组元素
    private E[] elements = (E[]) new Object[DEFAULT_CAPACITY];
    /**
     * 检查索引越界
     *
     * @param index 当前访问索引
     */
    private void checkIndex(int index) {
        if (index < 0 || index >= size) {
            throw new IndexOutOfBoundsException("索引越界" + "允许范围 size：0 => " + (size - 1) + " 当前索引：" + index);
        }
    }
    /**
     * 检查添加索引越界
     *
     * @param index 添加位置的索引
     */
    private void checkAddIndex(int index) {
        if (index < 0 || index > size) {
            throw new IndexOutOfBoundsException("索引越界" + "允许范围 size：0 => " + (size) + " 当前索引：" + index);
        }
    }
    /**
     * 确保数组容量够用
     */
    private void ensureCapacity() {
        //扩容1.5倍
        E[] newElements = (E[]) new Object[elements.length + (elements.length >> 1)];
        for (int i = 0; i < size; i++) {
            newElements[i] = elements[i];
        }
        elements = newElements;
    }
    //无参构造方法
    public DyTest() {
    }
    /**
     * 带参初始化
     *
     * @param capacity 初始化容量
     */
    public DyTest(int capacity) {
        if (capacity < 10) {
            elements = (E[]) new Object[DEFAULT_CAPACITY];
        } else {
            elements = (E[]) new Object[capacity];
        }
    }
    /**
     * 返回当前元素的数量
     *
     * @return 当前元素的个数
     */
    public int size() {
        return size;
```

```
    }
    /**
     * 当前数组是否为空
     * 空：true
     * 非空：false
     *
     * @return 返回true | false
     */
    public boolean isEmpty() {
        return size == 0;
    }
    /**
     * 是否包含某个元素
     *
     * @param element
     * @return 返回true | false
     */
    public boolean contains(E element) {
        if (element == null) {
            for (int i = 0; i < size; i++) {
                if (elements[i] == null) return true;
            }
        } else {
            for (int i = 0; i < size; i++) {
                if (element.equals(elements[i])) return true;
            }
        }
        return false;
    }
    /**
     * 添加元素到尾部
     *
     * @param element 待添加的元素
     */
    public void add(E element) {
        if (size > elements.length - 1) {
            ensureCapacity();
        }
        elements[size++] = element;
    }
    /**
     * 返回对应索引的值 不存在返回-1
     *
     * @param index 元素的索引
     * @return 对应值 | -1
     */
    public E get(int index) {
        checkIndex(index);
        return elements[index];
    }
    /**
     * 设置index位置元素的值
     *
     * @param index   需要设置的位置索引
     * @param element 设置的值
     * @return 返回原先的值
     */
    public E set(int index, E element) {
```

```
        checkIndex(index);//检查索引越界
        E old = elements[index];
        //
        elements[index] = element;
        return old;
    }

    /**
     * 向index位置添加元素
     *
     * @param index   插入位置的索引
     * @param element 插入的元素
     */
    public void add(int index, E element) {
        checkAddIndex(index);//检查索引越界
        for (int i = size; i > index; i--) {
            elements[i] = elements[i - 1];//把元素右移
        }
        elements[index] = element;
        size++;
    }
    /**
     * 移除index位置元素
     *
     * @param index 被移除元素的索引
     * @return 返回原先值
     */
    public E remove(int index) {
        checkIndex(index);
        E old = elements[index];
        for (int i = index; i < size; i++) {
            elements[i] = elements[i + 1];
        }
        elements[--size] = null;//清空最后一个元素
        return old;
    }
    /**
     * 查找元素
     *
     * @param element 需要查找的元素(注意)可能为null
     * @return 返回该元素索引 | -1
     */
    public int indexOf(E element) {
        if (element == null) {
            for (int i = 0; i < size; i++) {
                if (elements[i] == null) {
                    return i;
                }
            }
        } else {
            for (int i = 0; i < size; i++) {
                if (element.equals(elements[i])) {
                    return i;
                }
            }
        }
        return ELEMENT_NOt_FOUND;
    }
```

```
    /**
     * 清空所有元素
     */
    public void clear() {
        for (int i = 0; i < size; i++) {
            elements[i] = null;
        }
    }
    /**
     * 返回元素集合size:5, [1, 3, 4 ,5 ,7 ]
     *
     * @return
     */
    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder("size:" + size + " => [");
        for (int i = 0; i < size; i++) {
            if (i != 0) {
                sb.append(" ,");
            }
            sb.append(elements[i]);
        }
        sb.append("]");
        return sb.toString();
    }
}
```

测试

```
package com.kkb;
/**
 * 测试动态数组
 */
public class Test {
    public static void main(String[] args) {
        //int[] array=new int[]{23,56,7,45};
        //DynamicArray<Integer> list = new DynamicArray<Integer>();
        DyTest<Integer> list=new DyTest<>();
        list.add(1);
        list.add(2);
        list.add(3);
        System.out.println(list.toString());
        list.remove(2);
        System.out.println(list.toString());
        System.out.println(list.indexOf(2));
    }
}
```

ArrayList源码分析（基于JDK1.8）

ArrayList简介

ArrayList 是一个数组队列，相当于 动态数组。与Java中的数组相比，它的容量能动态增长。它继承于AbstractList，实现了List, RandomAccess, Cloneable, java.io.Serializable这些接口。

ArrayList属性

ArrayList属性主要就是当前数组长度size，以及存放数组的对象elementData数组，除此之外还有一个经常用到的属性就是从AbstractList继承过来的modCount属性，代表ArrayList集合的修改次数。

```
public class ArrayList<E> extends AbstractList<E> implements List<E>, RandomAccess, Cloneable, Serializable {
    // 序列化id
    private static final long serialVersionUID = 8683452581122892189L;
    // 默认初始的容量
    private static final int DEFAULT_CAPACITY = 10;
    // 一个空对象
    private static final Object[] EMPTY_ELEMENTDATA = new Object[0];
    // 一个空对象，如果使用默认构造函数创建，则默认对象内容默认是该值
    private static final Object[] DEFAULTCAPACITY_EMPTY_ELEMENTDATA = new Object[0];
    // 当前数据对象存放地方，当前对象不参与序列化
    transient Object[] elementData;
    // 当前数组长度
    private int size;
    // 数组最大长度
    private static final int MAX_ARRAY_SIZE = 2147483639;
    // 省略方法。。
}
```

ArrayList构造函数

无参构造函数

如果不传入参数，则使用默认无参构建方法创建ArrayList对象，如下：

```
    public ArrayList() {
        this.elementData = DEFAULTCAPACITY_EMPTY_ELEMENTDATA;
    }
```

注意：此时我们创建的ArrayList对象中的elementData中的长度是1，size是0,当进行第一次add的时候，elementData将会变成默认的长度：10.

带int类型的构造函数

如果传入参数，则代表指定ArrayList的初始数组长度，传入参数如果是大于等于0，则使用用户的参数初始化，如果用户传入的参数小于0，则抛出异常，构造方法如下：

```
    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);
        }
    }
```

带Collection对象的构造函数

1）将collection对象转换成数组，然后将数组的地址的赋给elementData。

2）更新size的值，同时判断size的大小，如果是size等于0，直接将空对象EMPTY_ELEMENTDATA的地址赋给elementData

3）如果size的值大于0，则执行Arrays.copy方法，把collection对象的内容（可以理解为深拷贝）copy到elementData中。

注意：this.elementData = arg0.toArray(); 这里执行的简单赋值时浅拷贝，所以要执行Arrays,copy 做深拷贝

```

    public ArrayList(Collection<? extends E> c) {
        elementData = c.toArray();
        if ((size = elementData.length) != 0) {
            // c.toArray might (incorrectly) not return Object[] (see
6260652)
            if (elementData.getClass() != Object[].class)
                elementData = Arrays.copyOf(elementData, size,
Object[].class);
        } else {
            // replace with empty array.
            this.elementData = EMPTY_ELEMENTDATA;
        }
    }
```

add方法

add的方法有两个，一个是带一个参数的，一个是带两个参数的。

add(E e) 方法

add主要的执行逻辑如下：

1）确保数组已使用长度（size）加1之后足够存下一个数据

2）修改次数modCount 标识自增1，如果当前数组已使用长度（size）加1后的大于当前的数组长度，则调用grow方法，增长数组，grow方法会将当前数组的长度变为原来容量的1.5倍。

3）确保新增的数据有地方存储之后，则将新元素添加到位于size的位置上。

4）返回添加成功布尔值。

添加元素方法入口：

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

确保添加的元素有地方存储，当第一次添加元素的时候this.size+1 的值是1，所以第一次添加的时候会将当前elementData数组的长度变为10：

```
    private void ensureCapacityInternal(int minCapacity) {
        if (elementData == DEFAULTCAPACITY_EMPTY_ELEMENTDATA) {
            minCapacity = Math.max(DEFAULT_CAPACITY, minCapacity);
        }

        ensureExplicitCapacity(minCapacity);
    }
```

将修改次数（modCount）自增1，判断是否需要扩充数组长度,判断条件就是用当前所需的数组最小长度与数组的长度对比，如果大于0，则增长数组长度。

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

如果当前的数组已使用空间（size）加1之后 大于数组长度，则增大数组容量，扩大为原来的1.5倍。

```
     private void grow(int arg0) {
         int arg1 = this.elementData.length;=
         int arg2 = arg1 + (arg1 >> 1);
         if (arg2 - arg0 < 0) {
             arg2 = arg0;
         }
         if (arg2 - 2147483639 > 0) {
             arg2 = hugeCapacity(arg0);
         }
         this.elementData = Arrays.copyOf(this.elementData, arg2);
     }
```

add(int index, E element)方法

这个方法其实和上面的add类似，该方法可以按照元素的位置，指定位置插入元素，具体的执行逻辑如下：

1）确保数插入的位置小于等于当前数组长度，并且不小于0，否则抛出异常

2）确保数组已使用长度（size）加1之后足够存下 下一个数据

3）修改次数（modCount）标识自增1，如果当前数组已使用长度（size）加1后的大于当前的数组长度，则调用grow方法，增长数组

4）grow方法会将当前数组的长度变为原来容量的1.5倍。

5）确保有足够的容量之后，使用System.arraycopy 将需要插入的位置（index）后面的元素统统往后移动一位。

6）将新的数据内容存放到数组的指定位置（index）上

注意：使用该方法的话将导致指定位置后面的数组元素全部重新移动，即往后移动一位。

```
public void add(int index, E element) {
        rangeCheckForAdd(index);
        ensureCapacityInternal(size + 1);  // Increments modCount!!
        System.arraycopy(elementData, index, elementData, index + 1,
                         size - index);
        elementData[index] = element;
        size++;
    }
```

get方法

返回指定位置上的元素，

```
        public E get(int index) {
            rangeCheck(index);
            checkForComodification();
            return ArrayList.this.elementData(offset + index);
        }
```

set方法

确保set的位置小于当前数组的长度（size）并且大于0，获取指定位置（index）元素，然后放到oldValue存放，将需要设置的元素放到指定的位置（index）上，然后将原来位置上的元素oldValue返回给用户。

```
 public E set(int index, E element) {
        rangeCheck(index);
        E oldValue = elementData(index);
        elementData[index] = element;
        return oldValue;
    }
```

contains方法

调用indexOf方法，遍历数组中的每一个元素作对比，如果找到对于的元素，则返回true，没有找到则返回false。

```
    public boolean contains(Object o) {
        return indexOf(o) >= 0;
    }
```

```
public int indexOf(Object o) {
        if (o == null) {
            for (int i = 0; i < size; i++)
                if (elementData[i]==null)
                    return i;
        } else {
            for (int i = 0; i < size; i++)
                if (o.equals(elementData[i]))
                    return i;
        }
        return -1;
    }
```

remove方法

根据索引remove

1）判断索引有没有越界

2）自增修改次数

3）将指定位置（index）上的元素保存到oldValue

4）将指定位置（index）上的元素都往前移动一位

5）将最后面的一个元素置空，好让垃圾回收器回收

6）将原来的值oldValue返回

注意：调用这个方法不会缩减数组的长度，只是将最后一个数组元素置空而已。

```
 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;
        return oldValue;
    }
```

根据对象remove

循环遍历所有对象，得到对象所在索引位置，然后调用fastRemove方法，执行remove操作

```
    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;
        }
```

定位到需要remove的元素索引，先将index后面的元素往前面移动一位（调用System.arraycooy实现），然后将最后一个元素置空。

```
private void fastRemove(int index) {
        modCount++;
        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
    }
```

clear方法

添加操作次数（modCount），将数组内的元素都置空，等待垃圾收集器收集，不减小数组容量。

```
    public void clear() {
        modCount++;
        for (int i = 0; i < size; i++)
            elementData[i] = null;
        size = 0;
    }
```

iterator方法

interator方法返回的是一个内部类，由于内部类的创建默认含有外部的this指针，所以这个内部类可以调用到外部类的属性。

```
    public Iterator<E> iterator() {
        return new Itr();
    }
```

trimToSize方法

1）修改次数加1

2）将elementData中空余的空间（包括null值）去除，例如：数组长度为10，其中只有前三个元素有值，其他为空，那么调用该方法之后，数组的长度变为3.

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

sublist方法

我们看到代码中是创建了一个ArrayList 类里面的一个内部类SubList对象，传入的值中第一个参数时this参数，其实可以理解为返回当前list的部分视图，真实指向的存放数据内容的地方还是同一个地方，如果修改了sublist返回的内容的话，那么原来的list也会变动。

```
    public List<E> subList(int arg0, int arg1) {
        subListRangeCheck(arg0, arg1, this.size);
        return new ArrayList.SubList(this, 0, arg0, arg1);
    }
```

System.arraycopy 方法

参数	说明
src	原数组
srcPos	原数组
dest	目标数组
destPos	目标数组的起始位置
length	要复制的数组元素的数目

Arrays.copyOf方法

original - 要复制的数组

newLength - 要返回的副本的长度

newType - 要返回的副本的类型

其实Arrays.copyOf底层也是调用System.arraycopy实现的。