让数组动起来

今日目标：

1：能够说出线性表的概念

2：能够说出数组的存储结构

3：能够说出数组查询，插入，删除操作的特点及对应的时间复杂度

4：能够完成动态数组ArrayList的代码编写

1、线性表

在数据结构中，数据的逻辑结构分为**线性结构**和**非线性结构**。

线性结构：n个数据元素的有序（次序）集合。其特征如下：

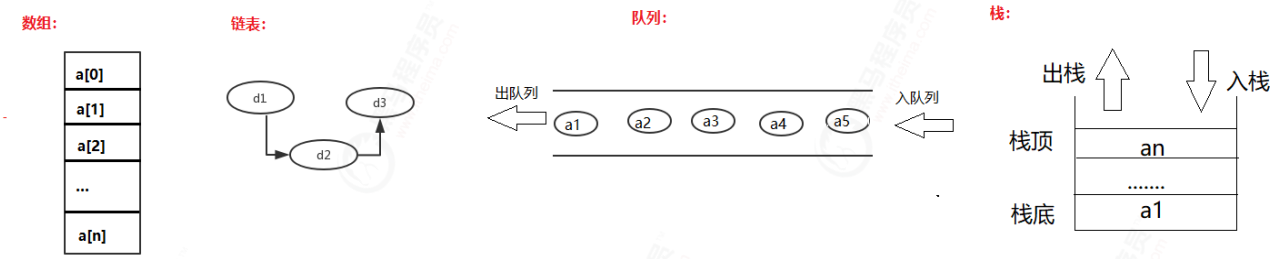
1．集合中必存在唯一的一个"第一个元素"；

2．集合中必存在唯一的一个"最后的元素"；

3．除最后元素之外，其它数据元素均有唯一的"后继"；

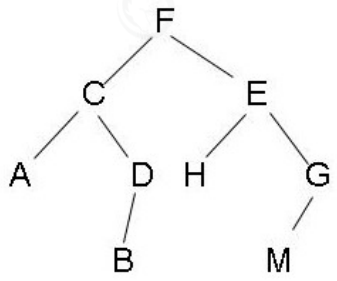
4．除第一元素之外，其它数据元素均有唯一的"前驱"。

数据结构中线性结构指的是数据元素之间存在着“一对一”的线性关系的数据结构。当然这个线性并不是说一定是直线，常见的线性数据结构有：数组(一维)，链表，栈，队列；它们也是我们后续学习其他数据结构的基础，表现形式如下：

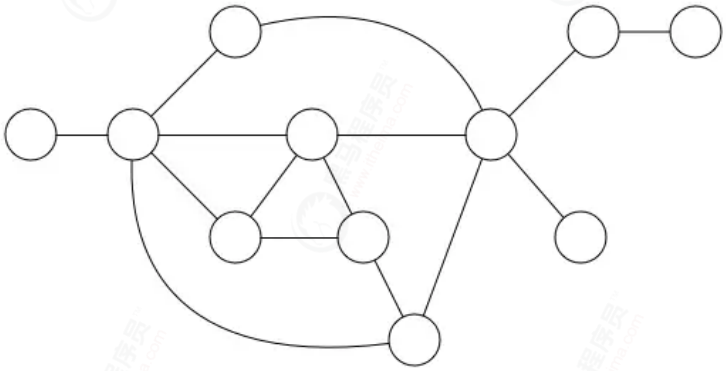


相对应于线性结构，非线性结构的逻辑特征是一个结点元素可能对应多个直接前驱和多个后继，比如后续要学习的：树，图，堆等，如下图

树



图

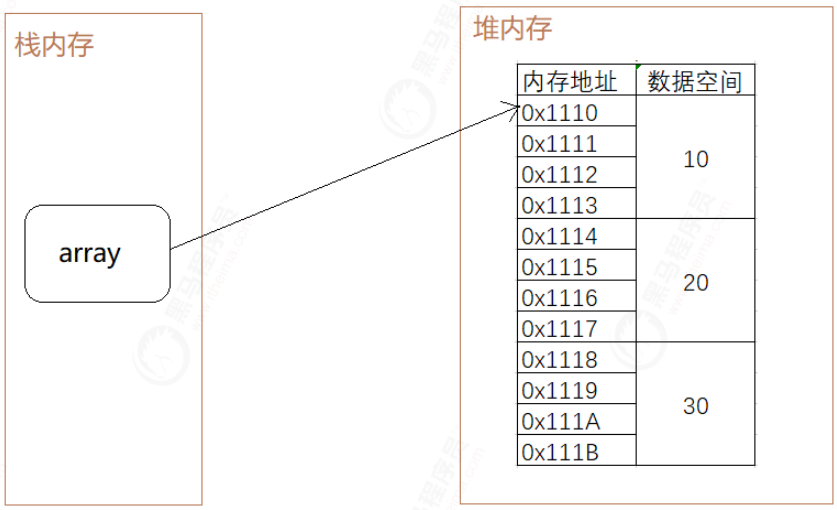


2、数组基础

2.1、概念和结构

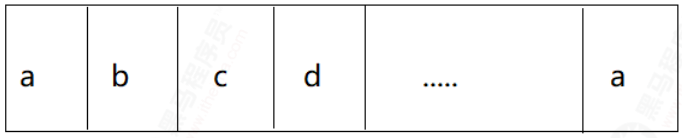
数组（Array）是一种用**连续的内存空间**存储**相同数据类型**数据的线性数据结构。

```
int[] array = new int[]{10,20,30}
```



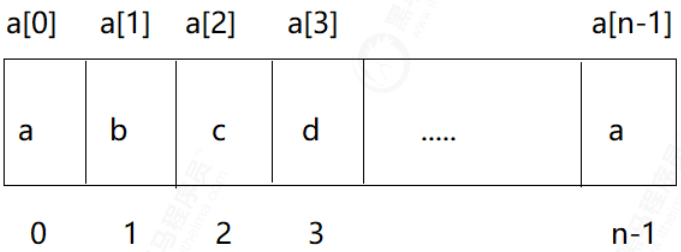
数组的表示方式：使用**下标**来获取数组元素数据

现假设有一个字符数组 a，元素个数是n

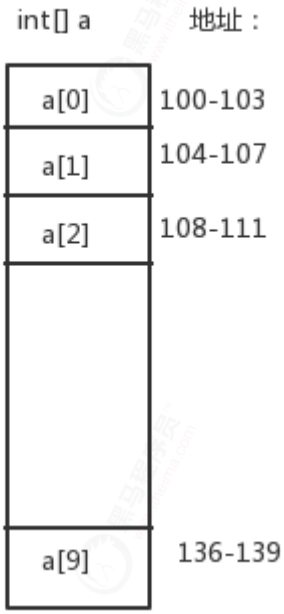


如何来表示或者说获取数组中的元素呢?

通过下标来表示和获取，数组下标从0开始，到数组长度-1



思考一下操作平台是如何根据下标来找到对应元素的内存地址的呢？

> 我们拿一个长度为10的数组来举例，int [] a= new int[10]，在下面的图中，计算机给数组分配了一块连续的空间，100-139，其中内存的起始地址为baseAddress=100
>
> 已知：长度为10的int数组a,数组a的起始地址为100
>
> 
>
>
> 我们知道，计算机给每个内存单元都分配了一个地址，通过地址来访问其数据，因此要访问数组中的某个元素时，首先要经过一个寻址公式计算要访问的元素在内存中的地址：

```
a[i] = baseAddress + i * dataTypeSize
```

其中dataTypeSize代表数组中元素类型的大小，在这个例子中，存储的是int型的数据，因此dataTypeSize=4个字节

下标为什么从**0**开始而不是**1**呢？

从数组存储的内存模型上来看，“下标”最确切的定义应该是“偏移（offset）”。前面也讲到，如果用 array 来表示数组的首地址，array[0] 就是偏移为 0 的位置，也就是首地址，array[k] 就表示偏移 k 个type_size 的位置，所以计算 array[k] 的内存地址只需要用这个公式:

```
array[k]_address = base_address + k * type_size
```

但是如果下标从1开始，那么计算array[k]的内存地址会变成：

```
array[k]_address = base_address + (k-1)*type_size
```

对比两个公式，不难发现从数组下标从1开始如果根据下标去访问数组元素，对于CPU来说，就多了一次减法指令。

当然另一方面也是由于历史原因，c语言设计者使用0开始作为数组的下标，后来的高级语言沿用了这一设计。

2.2、数组的特点

2.2.1、查询O(1)

数组元素的访问是通过下标来访问的，计算机通过数组的首地址和寻址公式能够很快速的找到想要访问的元素

```
public int test01(int[] a,int i){
    return a[i];
}
```

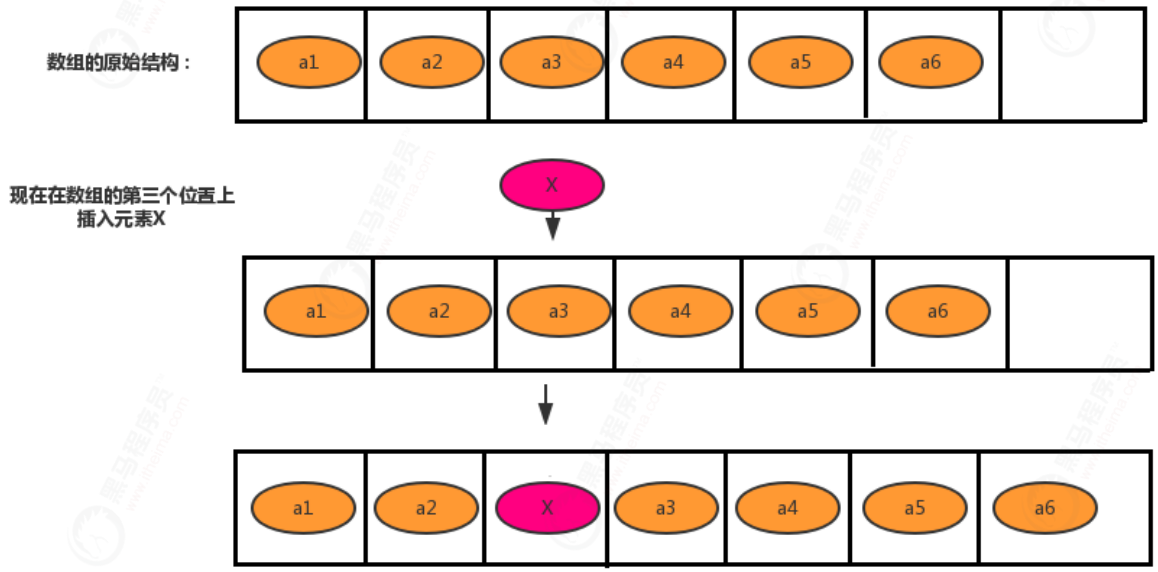
代码的执行次数并不会随着数组的数据规模大小变化而变化，是常数级的，所以查询数据操作的时间复杂度是O(1)

2.2.2、插入删除O(n)

数组是一段连续的内存空间，因此为了保证数组的连续性会使得数组的插入和删除的效率变的很低。

数据插入：

假设数组的长度为 n，现在如果我们需要将一个数据插入到数组中的第 k 个位置。为了把第 k 个位置腾出来给新来的数据，我们需要将第 k～n 这部分的元素都顺序地往后挪一位。如下图所示：



插入操作对应的时间复杂度是多少呢？

> 最好情况下是O(1)的，最坏情况下是O(n)的，平均情况下的时间复杂度是O(n)。

数据删除：

同理可得：如果我们要删除第 k 个位置的数据，为了内存的连续性，也需要搬移数据，不然中间就会出现空洞，内存就不连续了，时间复杂度仍然是O(n)。

思考一下在什么场景下用什么办法能提高数据删除的效率呢？

> 实际上，在某些特殊场景下，我们并不一定非得追求**数组中数据的连续性**。如果我们将多次删除操作集中在一起执行，删除的效率是不是会提高很多呢？举个例子
>
> 数组 a[6] 中存储了 6 个元素：a1,a2,a3,a4,a5,a6。现在，我们要依次删除 a1,a2 这两个元素。
>
> 
>
>
> 为了避免 a3,a4,a5,a6这几个数据会被搬移两次，我们可以先记录下已经删除的数据。每次的删除操作并不是真正地搬移数据，只是记录数据已经被删除。当数组没有更多空间存储数据时，我们再触发执行一次真正的删除操作，这样就大大减少了删除操作导致的数据搬移。
>
> 对于这种思想，不就是 JVM 标记清除垃圾回收算法的核心思想吗？
>
> 这其实就是告诉我们，**我们并不需要去死记硬背某个数据结构或者算法，而是理解和学习它背后的思想和处理技巧。**

3、面试实战

3.1、11：盛最多水的容器

华为，腾讯最近面试题：11：盛最多水的容器

1：暴力破解，朴素解法

```
class Solution {
    //暴力破解法 枚举出所有坐标的组合,求出每个组合
    public int maxArea(int[] height) {
        //定义容纳水的最大值
        int max = 0 ;
        for (int i=0; i<height.length;i++) {
            for (int j=i+1;j<height.length;j++) {
                int area = (j-i) * Math.min(height[i], height[j]);
                max = Math.max(max, area);
            }
        }
        return max;
    }
}
```

2：双指针（左右指针）（双指针夹逼）

```
class Solution {
    public int maxArea(int[] height) {
        //定义最大水的值
        int res = 0;
        //定义双指针
        int i=0;
        int j= height.length -1;
        while (i!=j) {
            int rest = Math.min(height[i],height[j]) * (j - i);
            if ( height[i] < height[j] ) {
                i++;
            }else {
                j--;
            }
            res = res > rest ? res:rest;
        }
        return res;
    }
}
```

3.2、283：移动零

字节跳动，滴滴打车最近面试题：283：移动零

1：双指针（快慢指针），一维数组的下标变换操作思想

```
class Solution {
    // 双指针移动 时间复杂度O(n) 空间复杂度O(1)
    public void moveZeroes(int[] nums) {
        if (nums == null || nums.length < 2) {
            return;
        }
        //从前到后非零元素的指针
        int p1 = 0;
        for (int i =0; i < nums.length; i++) {
            if (nums[i] != 0) {
                nums[p1] = nums[i];
                p1++;
            }
        }
        while (p1 < nums.length) {
            nums[p1] = 0;
            p1++;
        }
    }
}
```

注意查看精选题解，还有一次循环解决问题的解法

4、动态数组

4.1、需求

设计一个基于数组的集合容器，满足以下条件：

1：java中直接操作数组虽然获取数据比较方便，但是插入删除比较麻烦；因此容器要屏蔽（封装）底层的数组操作细节，支持数据的查询，插入，删除操作

2：java中的数组类型一旦申请之后大小是不可变的，而容器需要支持动态扩容

4.2、实现

基于java语言的特性，我们先定义接口，面向接口编程

（1）定义容器接口`com.itheima.array.inf.List`

```java
package com.itheima.array.inf;

/**
 * Created by 传智播客*黑马程序员.
 */
public interface List {

    /**
     * 返回容器中元素的个数
     * @return
     */
    int size();

    /**
     * 判断容器是否为空
     * @return
     */
    boolean isEmpty();

    /**
     * 查询元素在容器中的索引下标
     * @param o 元素对象
     * @return  在容器中的下标 不存在则返回-1
     */
    int indexOf(int o);

    /**
     * 判断容器是否包含某个特定的元素
     * @param e
     * @return
     */
    boolean contains(int e);

    /**
     * 将元素添加到容器结尾
     * @param e 要添加的元素
     * @return  是否添加成功
     */
    boolean add(int e);

    /**
     * 向指定位置添加元素，该位置及以后的元素后移
     * @param index     位置下标
     * @param element   元素对象
     */
    void add(int index, int element);

    /**
     * 用指定的元素替换指定位置的数据
     * @param index     指定的位置索引下标
     * @param element    新的元素
     * @return           原始的元素

     */
```

```
    int set(int index, int element);

    /**
     * 获取指定位置的元素
     * @param index   索引下标
     * @return        该位置的元素
     */
    int get(int index);

    /**
     * 移除指定位置的元素
     * @param index 索引下标
     * @return      被移除的元素
     */
    int remove(int index);

    /**
     * 清除容器中所有元素
     */
    void clear();

}
```

（2）创建接口的实现类：`com.itheima.array.ArrayList`并且实现`List`接口。

（3）我们的集合容器底层要基于数组存储数据，因此定义成员数组，另外还需要返回集合容器中元素的个数，因此定义一个代表元素个数的成员变量

```
//存储元素的数组
int[] elementData;

//容器中元素个数
private int size;
```

（4）定义构造方法，在容器初始化时初始化数组，并定义一个初始化容量大小

```
//容器默认容量大小
private final int DEFAULT_CAPACITY = 10;

public ArrayList() {
    this.elementData = new int[DEFAULT_CAPACITY];
}
```

并且还可以重载构造函数，初始化时可以指定数组容量

```
public ArrayList(int initCapaticy) {
    if (initCapaticy > 0) {
        this.elementData = new int[initCapaticy];
    }else {
        throw new IllegalArgumentException("initCapaticy error"+initCapaticy);
    }
}
```

（5）完成`size,isEmpty,indexOf,contains`等方法

```
@Override
public int size() {
    return size;
}

@Override
public boolean isEmpty() {
    return size == 0;
}

@Override
public int indexOf(int o) {
    for (int i=0; i < size; i++) {
        if ( elementData[i] == o) {
            return i;
        }
    }
    return -1;
}

@Override
public boolean contains(int e) {
    return indexOf(e) >=0;
}
```

（6）完成添加元素到容器尾部的方法`add`，这个过程中涉及到容器的动态扩容

```java
@Override
public boolean add(int e) {
    //添加之前先确保容量是否够
    ensureCapacity(size+1);
    this.elementData[size++] = e;
    return true;
}

private void ensureCapacity(int minCapacity) {
    if (minCapacity > this.elementData.length) {
        grow(minCapacity);
    }
}

private void grow(int minCapacity) {
    int oldCapacity = this.elementData.length;
    int newCapacity = oldCapacity + (oldCapacity >> 1);
    if (newCapacity < minCapacity) {
        newCapacity = minCapacity;
    }
    //用新的容量大小创建新的数组,并将原数组中的数据拷贝到新数组中，并使得this.elementData指向新数组
    copyOf(newCapacity);
}

private void copyOf(int newCapacity) {
    int[] newarray = new int[newCapacity];
    System.arraycopy(this.elementData,0,newarray,0,size);
    this.elementData = newarray;
}
```

（7）完成`add(int index, int element),set(int index, int element),get(int index)`方法，方法中先检查索引位置是否合理，然后检查是否需要扩容，最后在指定索引位置插入元素

```
@Override
public void add(int index, int element) {
    rangeCheck(index);
    //因为要加一个元素，之前的元素不会被覆盖，只会向后移动，所以可能在元素后移之前要先扩容
    ensureCapacity(size+1);

    //扩容完成后先将从index处的元素依次后移
    System.arraycopy(this.elementData,index,this.elementData,index+1,size-index);

    //在index位置存入数据
    this.elementData[index] = element;
    size++;
}

@Override
public int set(int index, int element) {
    rangeCheck(index);
    int oldElement = this.elementData[index];
    this.elementData[index] = element;
    return oldElement;
}

private void rangeCheck(int index) {
    if (index < 0 || index >size) {
        throw new IndexOutOfBoundsException("index:"+index+",size="+this.size);
    }
}

@Override
public int get(int index) {
    rangeCheck(index);
    return this.elementData[index];
}
```

（8）完成`remove(int index),clear`

```
@Override
public int remove(int index) {
    rangeCheck(index);
    int oldValue = this.elementData[index];
    //要移动元素的个数
    int moveCount = size - index -1;
    System.arraycopy(this.elementData,index+1,this.elementData,index,moveCount);

    //清理最后一个位置的元素
    this.elementData[size-1] = 0;
    //容器中元素个数-1
    size--;
    return oldValue;
}



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

（9）重写一下ArrayList的toString方法，方便打印输出

```
@Override
public String toString() {
    //按照[1,2,3,5,6]的格式输出数组中的元素
    StringBuilder sb = new StringBuilder("[");
    for (int i=0 ;i < size; i++) {
        sb.append(this.elementData[i]).append(",");
    }
    sb.append("]");
    return sb.toString();
}
```

（10）编写测试类完成测试：`com.itheima.array.ArrayListTest`

```java
package com.itheima.array;

import com.itheima.array.inf.List;

/**
 * Created by 传智播客*黑马程序员.
 */
public class ArrayListTest {

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

        //添加数据
        list.add(1);
        list.add(2);
        list.add(3);
        list.add(4);
        list.add(5);
        System.out.println("索引为4的元素:"+list.get(4));
        list.add(6);
        list.add(7);
        list.add(8);
        list.add(9);
        list.add(10);
        //再添就要扩容了
        list.add(11);
        System.out.println("size="+list.size()+"--"+list);
        System.out.println("是否包含8:"+list.contains(8)+",集合容器是否为空:"+list.isEmpty());
        list.add(12);
        list.add(13);
        list.add(14);
        list.add(15);
        //既要扩容，元素又要后移
        list.add(13,141);
        System.out.println("size="+list.size()+"--"+list);
        int set = list.set(13, 142);
        System.out.println("set方法返回的值:"+set+"--完成后的list:"+list);
        int remove = list.remove(13);
        System.out.println("remove方法返回的值:"+remove+"--remove后的list:"+list);
        list.clear();
        System.out.println("clear之后:"+list);
    }
}
```

思考一下课后完成：将ArrayList改造成能存储任意jiava类型数据的容器，应该如何完成？

提示：将int类型的数组变换成Object类型的数组，然后改造相应的接口方法，添加泛型

作为高级语言编程者，是不是数组就无用武之地了呢？

当然不是，有些时候，用数组会更合适些，我总结了几点自己的经验。

1.Java ArrayList 无法存储基本类型，比如 int、long，需要封装为 Integer、Long 类，而 Autoboxing、Unboxing 则有一定的性能消耗，所以如果特别关注性能，或者希望使用基本类型，就可以选用数组。

2.如果数据大小事先已知，并且对数据的操作非常简单，用不到 ArrayList 提供的大部分方法，也可以直接使用数组。

总结一下就是说：对于业务开发，直接使用容器就足够了，省时省力。毕竟损耗一丢丢性能，完全不会影响到系统整体的性能。但如果你是做一些非常底层的开发，比如开发网络框架，性能的优化需要做到极致，这个时候数组就会优于容器，成为首选。
