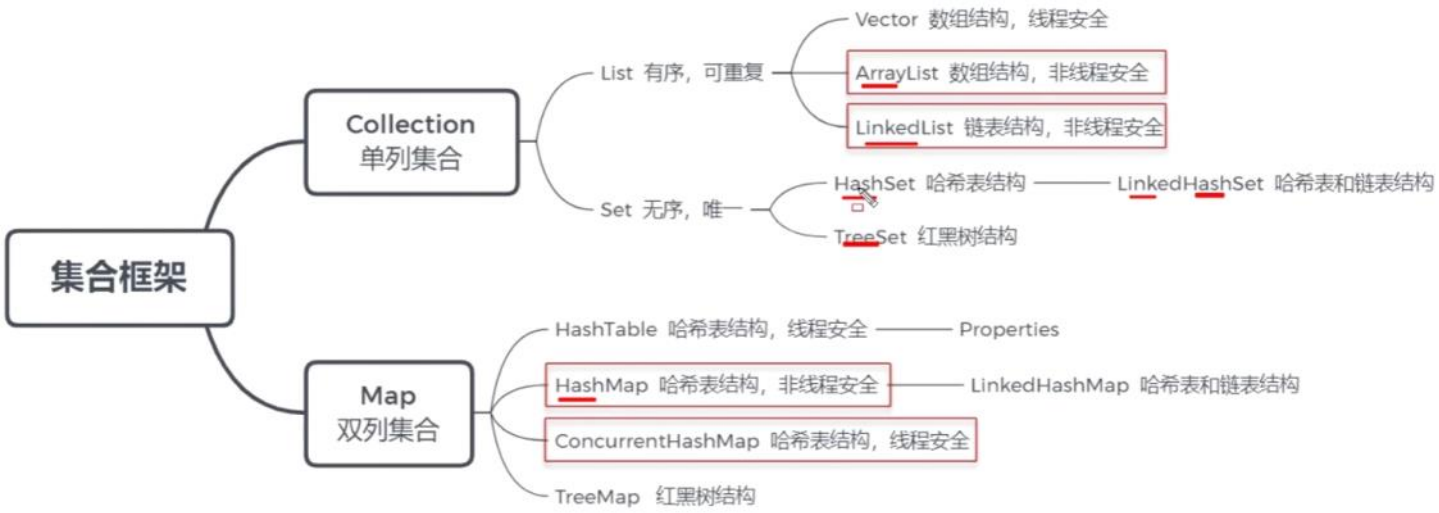
# 常见集合、数组、ArrayList、LinkedList

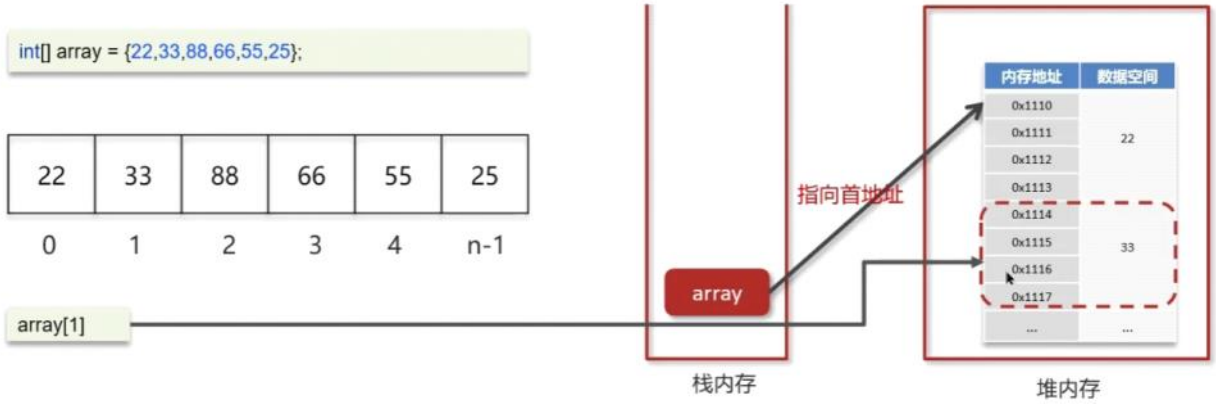
2024年4月2日 11:21

## Java集合框架体系




## 数组

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。



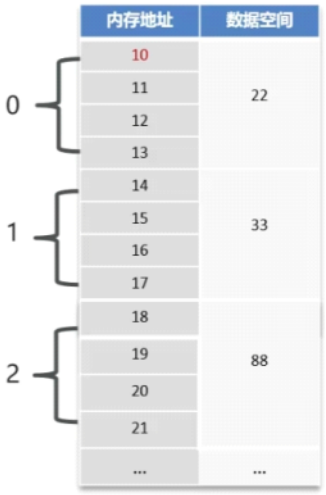
## 数组如何获取其他元素的地址值？

```
int[] array = {22,33,88,66,55,25};
```

| 22 | 33 | 88 | 66 | 55 | 25 |
|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | n-1 |

寻址公式：`a[i] = baseAddress + i * dataTypeSize`

- baseAddress：数组的首地址
- dataTypeSize：代表数组中元素类型的大小，int型的数据，dataTypeSize=4个字节




为什么数组索引从0开始呢？假如从1开始不行吗？

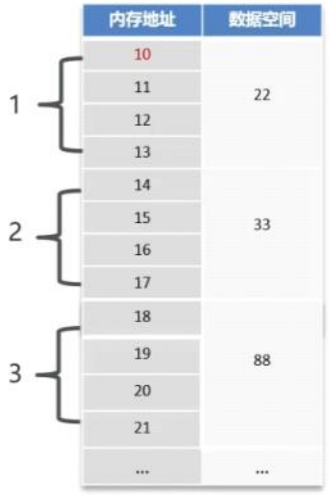
## 为什么数组索引从0开始呢？假如从1开始不行吗？

寻址公式：`a[i] = baseAddress + i * dataTypeSize` 胜出

- baseAddress：数组的首地址
- dataTypeSize：代表数组中元素类型的大小，int型的数据，dataTypeSize=4个字节

寻址公式：`a[i] = baseAddress + (i-1)* dataTypeSize`

对于cpu来说，增加了一个减法指令




- 在根据数组索引获取元素的时候，会用索引和寻址公式来计算内存所对应的元素数据，寻址公式是：数组的首地址+索引乘以存储数据的类型大小
- 如果数组的索引从1开始，寻址公式中，就需要增加一次减法操作，对于CPU来说就多了一次指令，性能不高。

## 操作数组的时间复杂度（查找）

### 1.随机查询(根据索引查询)

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];
    // a[i] = baseAddress + i * dataSize
}
```

O(1)

### 2. 未知索引查询

情况一：查找数组内的元素，查找55号数据

| 9 | 5 | 22 | 33 | 88 | 11 | 55 |
|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |

O(n)

## 操作数组的时间复杂度（插入、删除）

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。


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)。

1. 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。

2. 数组下标为什么从0开始

   寻址公式是：baseAddress+ i * dataTypeSize，计算下标的内存地址效率较高

3. 查找的时间复杂度

   - 随机(通过下标)查询的时间复杂度是O(1)
   - 查找元素（未知下标）的时间复杂度是O(n)
   - 查找元素（未知下标但排序）通过二分查找的时间复杂度是O(logn)

4. 插入和删除时间复杂度

   插入和删除的时候，为了保证数组的内存连续性，需要挪动数组元素，平均时间复杂度为O(n)

## ArrayList源码分析

源码如何分析?

```
List<Integer> list = new ArrayList<Integer>();
list.add(1);
```

## ArrayList源码分析-成员变量

```
/**
 * 默认初始的容量(CAPACITY)
 */
private static final int DEFAULT_CAPACITY = 10;
/**
 * 用于空实例的共享空数组实例
 */
private static final Object[] EMPTY_ELEMENTDATA = {};
/**
 * 用于默认大小的空实例的共享空数组实例。
 * 我们将其与 EMPTY_ELEMENTDATA 区分开来，以了解添加第一个元素时要膨胀多少
 */
private static final Object[] DEFAULTCAPACITY_EMPTY_ELEMENTDATA = {};
/**
 * 存储 ArrayList 元素的数组缓冲区。 ArrayList 的容量就是这个数组缓冲区的长度。
 * 当添加第一个元素时，任何具有 elementData == DEFAULTCAPACITY_EMPTY_ELEMENTDATA 的空 ArrayList
 * 都将扩展为 DEFAULT_CAPACITY
 * 当前对象不参与序列化
 */
transient Object[] elementData; // non-private to simplify nested class access
/**
 * ArrayList 的大小（它包含的元素数量）
 * @serial
 */
private int size;
```

## ArrayList源码分析-构造方法

```
public ArrayList(int initialCapacity) {      带初始化容量的构造函数
    if (initialCapacity > 0) {
        this.elementData = new Object[initialCapacity];
    } else if (initialCapacity == 0) {
        this.elementData = EMPTY_ELEMENTDATA;
    } else {
        throw new IllegalArgumentException("Illegal Capacity: "+
                                           initialCapacity);
    }
}

/**
 * Constructs an empty list with an initial capacity of ten.
 */
public ArrayList() {      无参构造函数，默认创建空集合
    this.elementData = DEFAULTCAPACITY_EMPTY_ELEMENTDATA;
}
```

```
public ArrayList(Collection<? extends E> c) {
    Object[] a = c.toArray();
    if ((size = a.length) != 0) {
        if (c.getClass() == ArrayList.class) {
            elementData = a;
        } else {
            elementData = Arrays.copyOf(a, size, Object[].class);
        }
    } else {
        // replace with empty array.
        elementData = EMPTY_ELEMENTDATA;
    }
}
```

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

## ArrayList源码分析-添加和扩容操作(第1次添加数据)

```
List<Integer> list = new ArrayList<Integer>();
list.add(1);
```

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

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

S

1

确保内部容量

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

计算容量 DEFAULT_CAPACITY=10

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

10

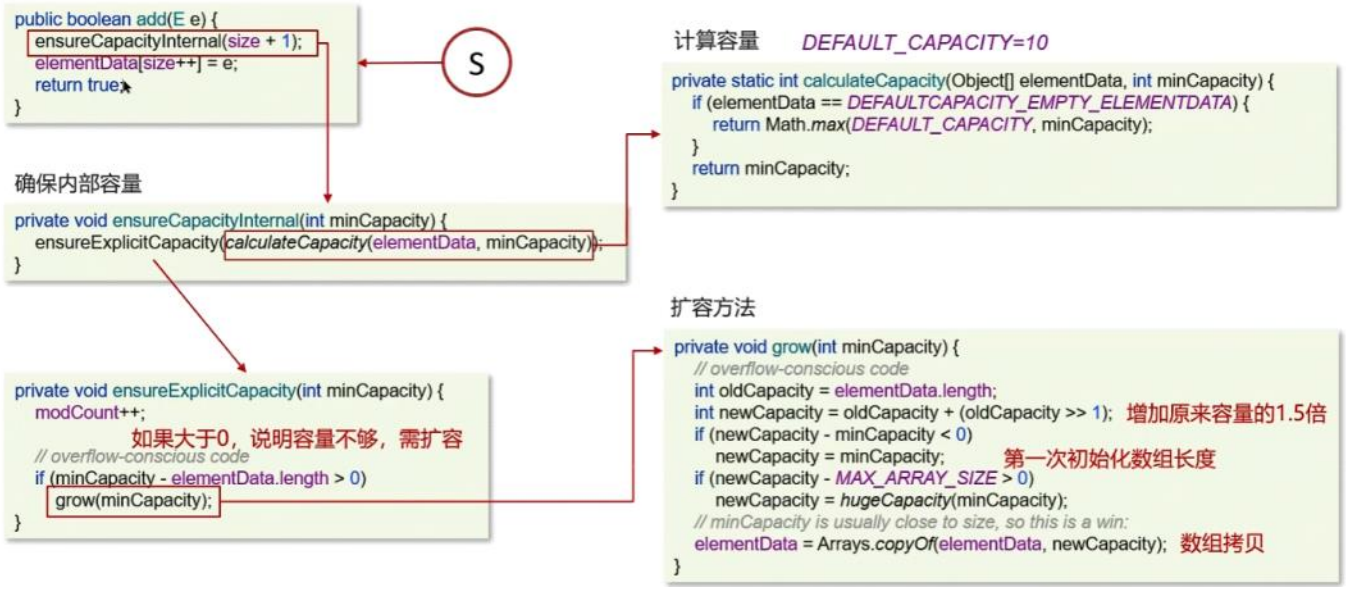
```
private void ensureExplicitCapacity(int minCapacity) {
    modCount++;
    如果大于0，说明容量不够，需扩容
    // overflow-conscious code
    if (minCapacity - elementData.length > 0)
        grow(minCapacity);
}
```

扩容方法

10

```
private void grow(int minCapacity) {
    // overflow-conscious code
    int oldCapacity = elementData.length;
    int newCapacity = oldCapacity + (oldCapacity >> 1);  增加原来容量的1.5倍
    if (newCapacity - minCapacity < 0)
        newCapacity = minCapacity;          第一次初始化数组长度
    if (newCapacity - MAX_ARRAY_SIZE > 0)
        newCapacity = hugeCapacity(minCapacity);
    // minCapacity is usually close to size, so this is a win:
    elementData = Arrays.copyOf(elementData, newCapacity);  数组拷贝
}
```

## ArrayList源码分析-添加和扩容操作(第2至10次添加数据)




## ArrayList底层的实现原理是什么

- ArrayList底层是用动态的数组实现的
- ArrayList初始容量为0，当第一次添加数据的时候才会初始化容量为10
- ArrayList在进行扩容的时候是原来容量的1.5倍，每次扩容都需要拷贝数组
- ArrayList在添加数据的时候
  - 确保数组已使用长度（size）加1之后足够存下下一个数据
  - 计算数组的容量，如果当前数组已使用长度+1后的大于当前的数组长度，则调用grow方法扩容（原来的1.5倍）
  - 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。
  - 返回添加成功布尔值。

## ArrayList list=new ArrayList(10)中的list扩容几次

```
/**
 * 构造一个具有指定初始容量的空列表。
 * 参数：initialCapacity - 列表的初始容量
 * 抛出：IllegalArgumentException – 如果指定的初始容量为负
 */
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);
    }
}
```

参考回答：

该语句只是声明和实例了一个 ArrayList，指定了容量为 10，未扩容

## 如何实现数组和List之间的转换

```java
//数组转List
public static void testArray2List(){
    String[] strs = {"aaa","bbb","ccc"};
    List<String> list = Arrays.asList(strs);
    for (String s : list) {
        System.out.println(s);
    }
}
//List转数组
public static void testList2Array(){
    List<String> list = new ArrayList<String>();
    list.add("aaa");
    list.add("bbb");
    list.add("ccc");
    String[] array = list.toArray(new String[list.size()]);
    for (String s : array) {
        System.out.println(s);
    }
}
```

参考回答：

- 数组转List，使用JDK中java.util.Arrays工具类的asList方法
- List转数组，使用List的toArray方法。无参toArray方法返回Object数组，传入初始化长度的数组对象，返回该对象数组

面试官再问：

- 用Arrays.asList转List后，如果修改了数组内容，list受影响吗
- List用toArray转数组后，如果修改了List内容，数组受影响吗

## 如何实现数组和List之间的转换

```java
//数组转List
public static void testArray2List(){
    String[] strs = {"aaa","bbb","ccc"};
    List<String> list = Arrays.asList(strs);
    for (String s : list) {
        System.out.println(s);
    }
    strs[1]="ddd";
    System.out.println("================");
    for (String s : list) {
        System.out.println(s);
    }
}
//List转数组
public static void testList2Array(){
    List<String> list = new ArrayList<String>();
    list.add("aaa");
    list.add("bbb");
    list.add("ccc");
    String[] array = list.toArray(new String[list.size()]);
    for (String s : array) {
        System.out.println(s);
    }
    list.add("ddd");
    System.out.println("================");
    for (String s : array) {
        System.out.println(s);
    }
}
```

受影响

不受影响

面试官再问：

- 用Arrays.asList转List后，如果修改了数组内容，list受影响吗
- List用toArray转数组后，如果修改了List内容，数组受影响吗

再答：

- Arrays.asList转换list之后，如果修改了数组的内容，list会受影响，因为它的底层使用的Arrays类中的一个内部类ArrayList来构造的集合，在这个集合的构造器中，把我们传入的这个集合进行了包装而已，最终指向的都是同一个内存地址
- list用了toArray转数组后，如果修改了list内容，数组不会影响，当调用了toArray以后，在底层是它是进行了数组的拷贝，跟原来的元素就没啥关系了，所以即使list修改了以后，数组也不受影响

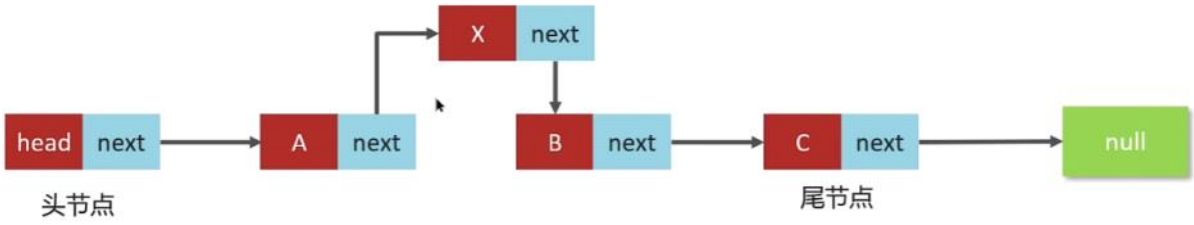
## 单向链表时间复杂度分析

查询操作


- 只有在查询头节点的时候不需要遍历链表，时间复杂度是O(1)
- 查询其他结点需要遍历链表，时间复杂度是O(n)

## 链表时间复杂度分析

**插入\删除操作**



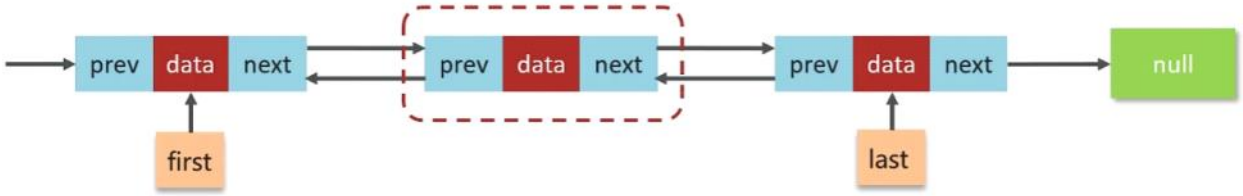

- 只有在添加和删除头节点的时候不需要遍历链表，时间复杂度是O(1)
- 添加或删除其他结点需要遍历链表找到对应节点后，才能完成新增或删除节点，时间复杂度是O(n)

## 双向链表

而双向链表，顾名思义，它支持两个方向

- 每个结点不止有一个后继指针 next 指向后面的结点
- 有一个前驱指针 prev 指向前面的结点

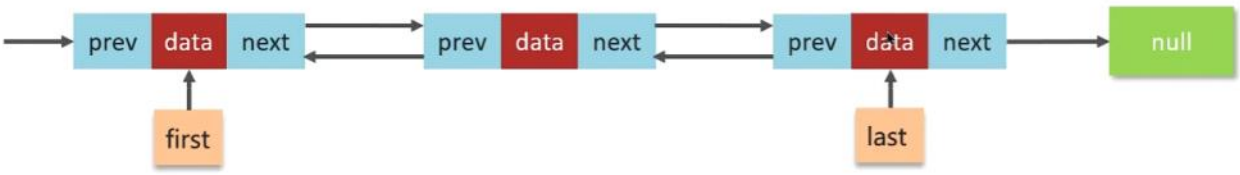
```
private static class Node<E> {
    E item;
    Node<E> next;
    Node<E> prev;

    Node(Node<E> prev, E element, Node<E> next) {
        this.item = element;
        this.next = next;
        this.prev = prev;
    }
}
```




对比单链表：

- 双向链表需要额外的两个空间来存储后继结点和前驱结点的地址
- 支持双向遍历，这样也带来了双向链表操作的灵活性

## 双向链表时间复杂度分析




**查询操作**

查询头尾结点的时间复杂度是O(1)

平均的查询时间复杂度是O(n)

给定节点找前驱节点的时间复杂度为O(1)

**增删操作**

头尾结点增删的时间复杂度为O(1)

其他部分结点增删的时间复杂度是 O(n)

给定节点增删的时间复杂度为O(1)

### 1. 单向链表和双向链表的区别是什么

- 单向链表只有一个方向，结点只有一个后继指针 next。
- 双向链表它支持两个方向，每个结点不止有一个后继指针next指向后面的结点，还有一个前驱指针prev指向前面的结点

### 2. 链表操作数据的时间复杂度是多少

| | 查询 | 新增删除 |
|---|---|---|
| 单向链表 | 头O(1),其他O(n) | |
| 双向链表 | 头尾O(1),其他O(n),给定节点O(1) | |

## ArrayList 和 LinkedList 的区别是什么?

### 1. 底层数据结构

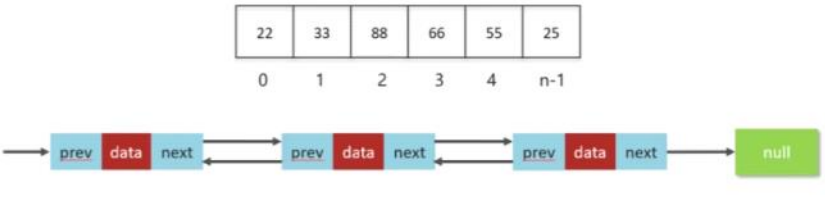

- ArrayList 是动态数组的数据结构实现
- LinkedList 是双向链表的数据结构实现

### 2. 操作数据效率

- ArrayList按照下标查询的时间复杂度O(1)【内存是连续的，根据寻址公式】， LinkedList不支持下标查询
- 查找（未知索引）： ArrayList需要遍历，链表也需要链表，时间复杂度都是O(n)
- 新增和删除
  - ArrayList尾部插入和删除，时间复杂度是O(1)；其他部分增删需要挪动数组，时间复杂度是O(n)
  - LinkedList头尾节点增删时间复杂度是O(1)，其他都需要遍历链表，时间复杂度是O(n)

## ArrayList 和 LinkedList 的区别是什么?

### 3. 内存空间占用




- ArrayList底层是数组，内存连续，节省内存
- LinkedList 是双向链表需要存储数据，和两个指针，更占用内存

### 4. 线程安全

- ArrayList和LinkedList都不是线程安全的
- 如果需要保证线程安全，有两种方案：
  - 在方法内使用，局部变量则是线程安全的
  - 使用线程安全的ArrayList和LinkedList

```
List<Object> syncArrayList = Collections.synchronizedList(new ArrayList<>());
List<Object> syncLinkedList = Collections.synchronizedList(new LinkedList<>());
```