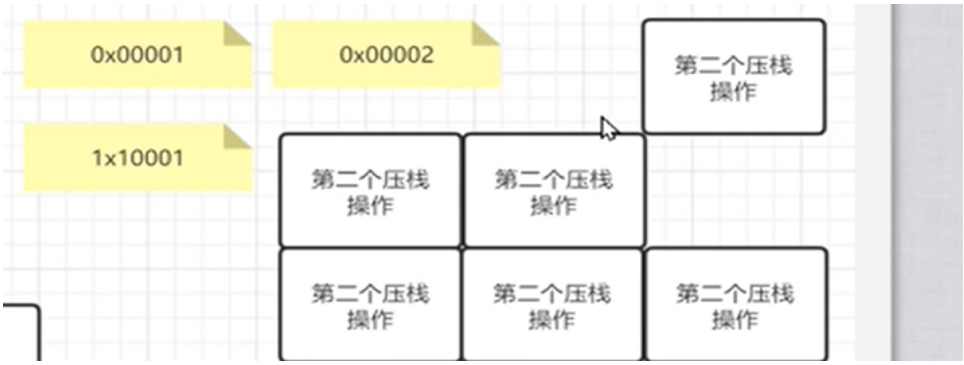
数据结构集

day01 内存是平面

> 问：内存对于数据来说，是平面的还是立体的？
> 答：平面的。
> 问：平面的，你的数据之间，可以有哪些方式来存放？
> 答：放一起，分开放。

放一起的是开辟连续内存空间的数组，分开放的是再内存中不连续的链表

且每个内存空间都有一个指向这块放值的空间的地址



day02 栈 接口

栈的特性

先进先出（FILO）

栈的数据结构

栈的数据结构为数组

由Java源码可以看出，Stack类继承自Vector，而Vector用的就是对象数组来存取数据的

```
public
class Stack<E> extends Vector<E> {
    /**
     * Creates an empty Stack.
     */
    public Stack() {
    }
```

```
public class Vector<E>
    extends AbstractList<E>
    implements List<E>, RandomAccess, Cloneable, java.io.Serializable
{
    /**
     * The array buffer into which the components of the vector are
     * stored. The capacity of the vector is the length of this array buf
     * and is at least large enough to contain all the vector's elements.
     *
     * <p>Any array elements following the last element in the Vector are
     *
     * @serial
     */
    protected Object[] elementData;
```

栈的执行流程

栈有一个专门指向当前栈顶部的数据的top指针，指向的值为数组的下标

这个指针初始指向的值为-1

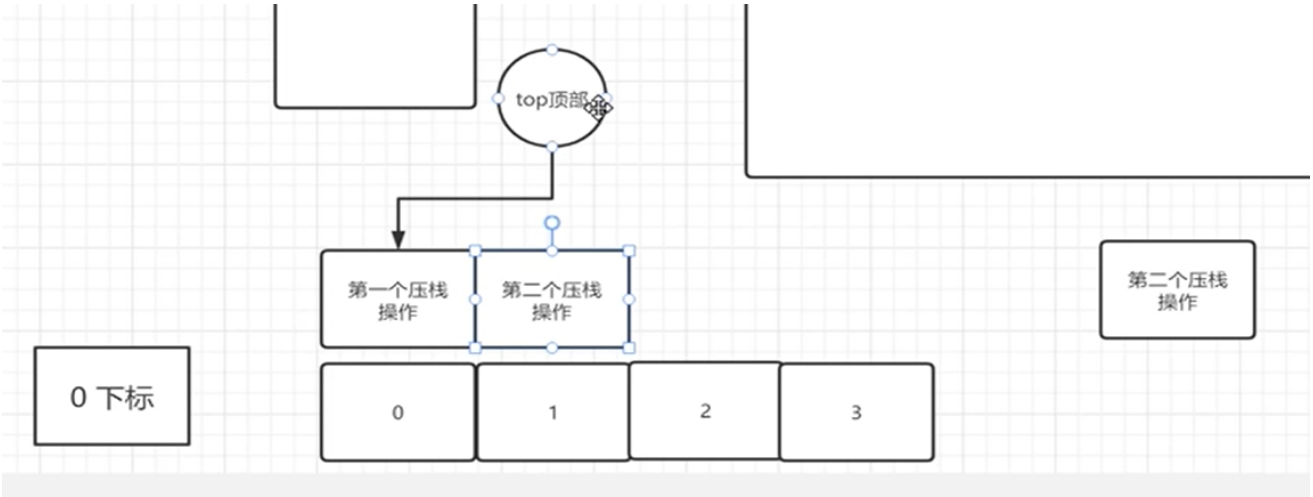
每当有push入栈（按顺序给栈数组的元素从零开始赋值），top指向的下标值都会+1

每当有pop弹栈都会让top的指向下标值-1

由上面可以知道弹栈不是真正的让数组的元素的值为空，只是让top的指向改变了，实际上被弹出的数据仍然再数组里

栈为空，那么top的指向值为-1

栈满，那么top的值为数组长度减1

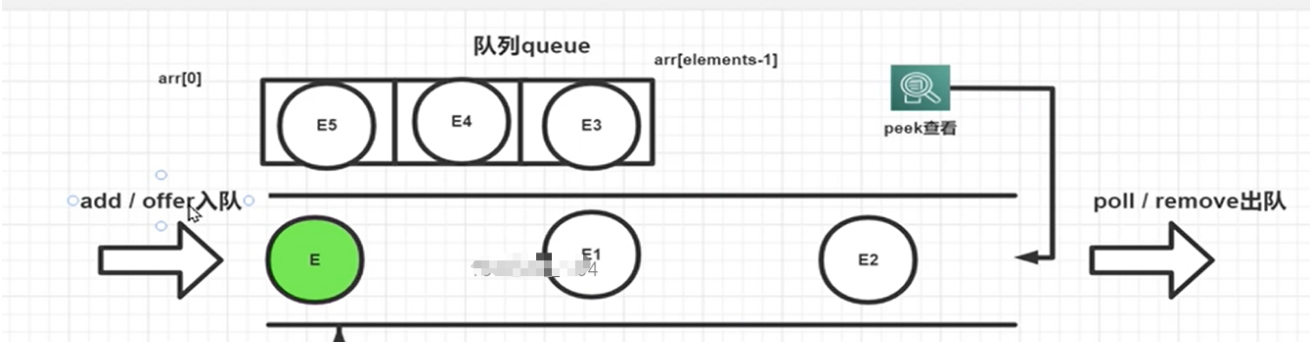



```java
public class MyStack<T> {
    private T [] arr; // 栈的底层是数组
    private int top = -1; // 栈顶的索引

    public MyStack() { arr = (T[]) new Object[10]; // 默认创建10个长度 }

    public MyStack(int capacity) { arr = (T[]) new Object[capacity]; // 自定义长度 }

    /**
     * 压栈
     */
    public void push(T value){
        if (isFull())
            throw new MyStackException("Stack is full");
        arr[++top] = value;
    }

    /**
     * 弹栈
     * @return 栈顶数据
     */
    public T pop(){
        if (isEmpty())
            throw new MyStackException("Stack is empty");
        return arr[top--];
    }
```

```java
public T pop(){
    if (isEmpty())
        throw new MyStackException("Stack is empty");
    return arr[top--];
}

/*
    查看栈顶元素
 */
public T peek(){
    if (isEmpty())
        throw new MyStackException("Stack is empty");
    return arr[top];
}

/**
 * 判断栈是否是空的
 * @return
 */
public boolean isEmpty() { return top == -1; }

/**
 * 判断栈是否满了
 * @return
```

day03 队列 接口

queue接口最有代表性的方法为 offer 和 poll，如果其他的类里有这两个方法那么他一定实现了队列接口，并且说明他的数据结构就是队列，那么他就拥有队列的特性



队列的特性

先进先出(FIFO)

Java中的队列是一个接口,所以他可以被是现成多种形式的队列

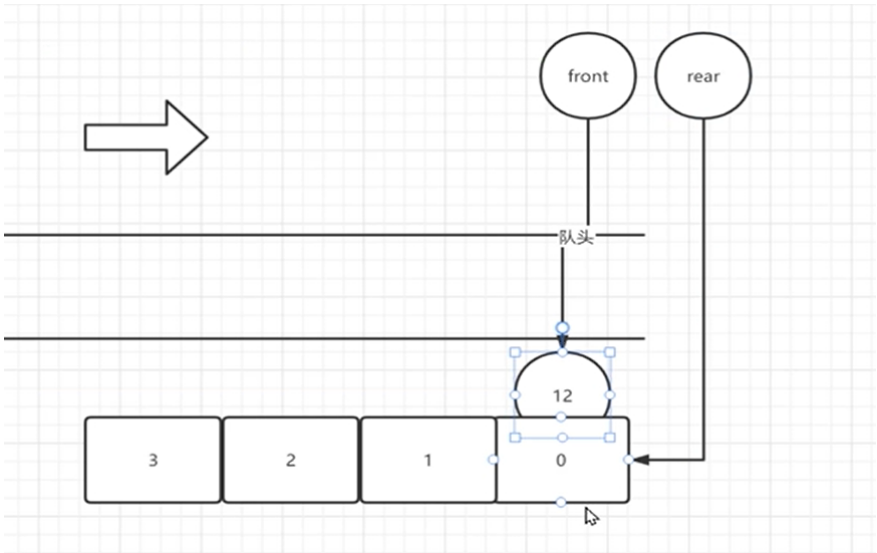
```
*/
public interface Queue<E> extends Collection<E> {
    /**
     * Inserts the specified element into this queue if
```

ArrayDeque和LinkedBlockoingDeque看开头就能知道这两个的数据结构分别使用的是数组和链表实现

```
ArrayDeque (java.util)
BlockingDeque (java.util.concurrent)
ConcurrentLinkedDeque (java.util.concurrent)
DualValueDeque (org.assertj.core.api.recursive.comparison)
IdentityLinkedList (sun.awt.util)
ImportStack in ConfigurationClassParser (org.springframework.context.a
JSONListAdapter (jdk.nashorn.internal.runtime)
KeepAliveStreamCleaner (sun.net.www.http)
LinkedBlockingDeque (java.util.concurrent)
LinkedList (java.util)
ListAdapter (jdk.nashorn.internal.runtime)
```

基本流程

最开始队头和队尾都指向数组的下标为零的位置



执行入队列方法，首先判断队列是否满了，没满就让队尾指针的值会加1，并且让队尾指针指向的前一个元素的值为当前入队的数据然后给专门记录队列所含元素个数的值+1

出队列的时候，会判断队列是否为空，不为空那么就让队头指针的指向的值+1，记录元素个数的element减1

自定义的queue

15828951064

```java
/**
 * 自定义队列
 */
public class Queue {
    private long [] arr; // 队列的底层是数组
    private int front = 0; // 队头索引
    private int rear = 0; // 队尾索引
    private int elements; // 队列中实际的元素个数

    public Queue() { arr = new long[10]; // 默认长度是10 }
    public Queue(int capacity) { arr = new long[capacity]; // 自定义长度 }

    /**
     * 进入队列 ： 从队尾进入
     * @param value
     */
    public void enqueue(long value){
        if (isFull())
            throw new IllegalArgumentException("Queue is full");
        arr[rear++] = value;
        elements++;
    }
```

```java
/**
 * 出队列 ： 从队头取出元素
 * @return
 */
public long dequeue(){
    if (isEmpty())
        throw new IllegalArgumentException("Queue is empty");
    elements--;
    return arr[front++];
}
```

```java
    /**
     * 查看队头的元素
     * @return
     */
    public long top(){
        if (isEmpty())
            throw new IllegalArgumentException("Queue is empty");
        return arr[front];
    }

    /**
     * 判断队列是否是空的
     * @return
     */
    public boolean isEmpty() { return elements == 0; }
```

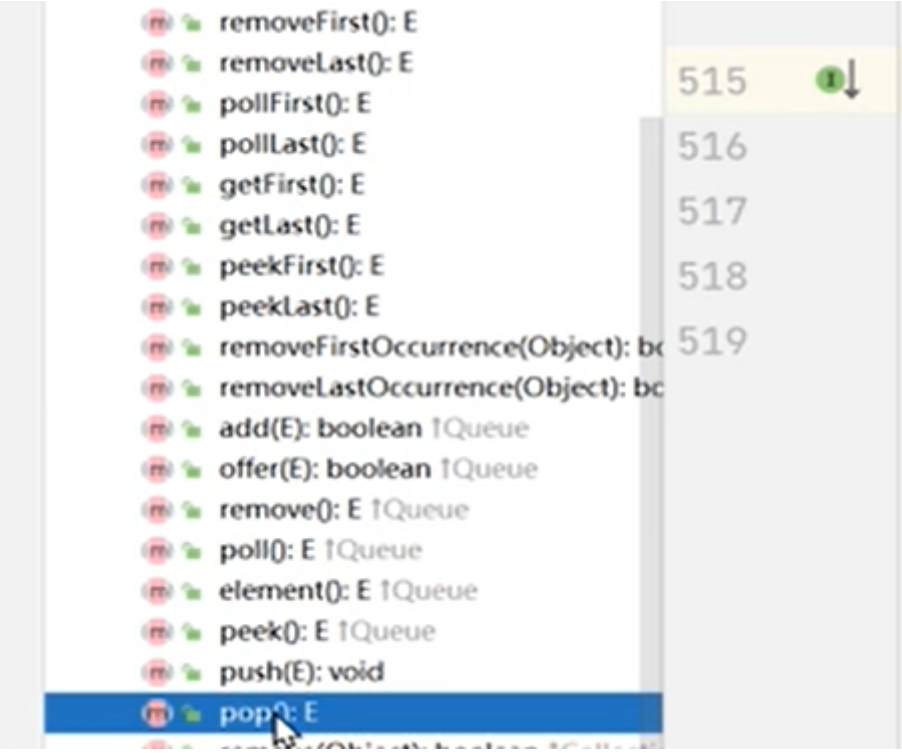
双端队列接口

也是一个接口

最神奇的地方在于他同时提供了队列的出入队列方法，还有提供栈的push和pop方法

说明他同时具有队列和栈的特性

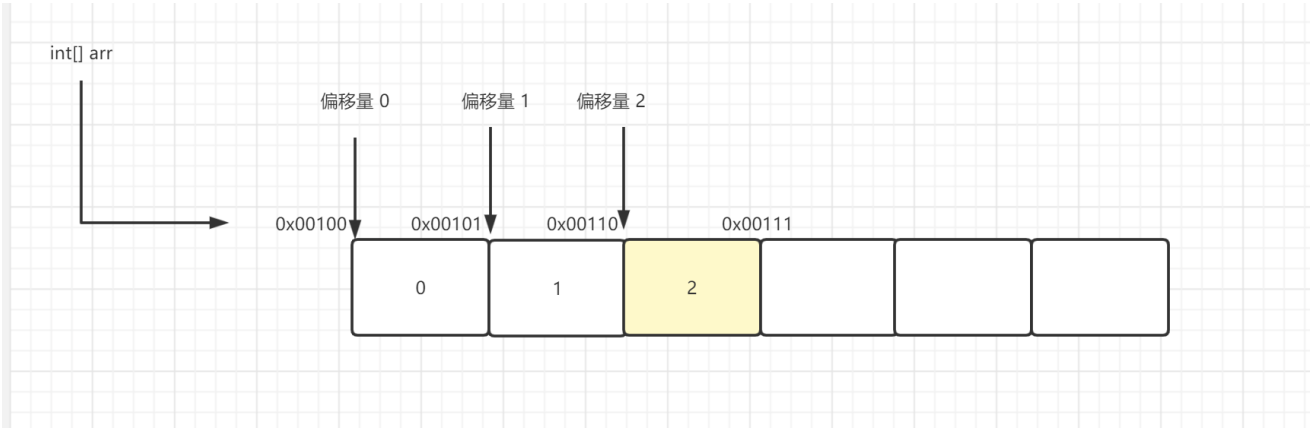
进去的是从一头进入的，出去可以从两端出



day04 数组

偏移量的概念：

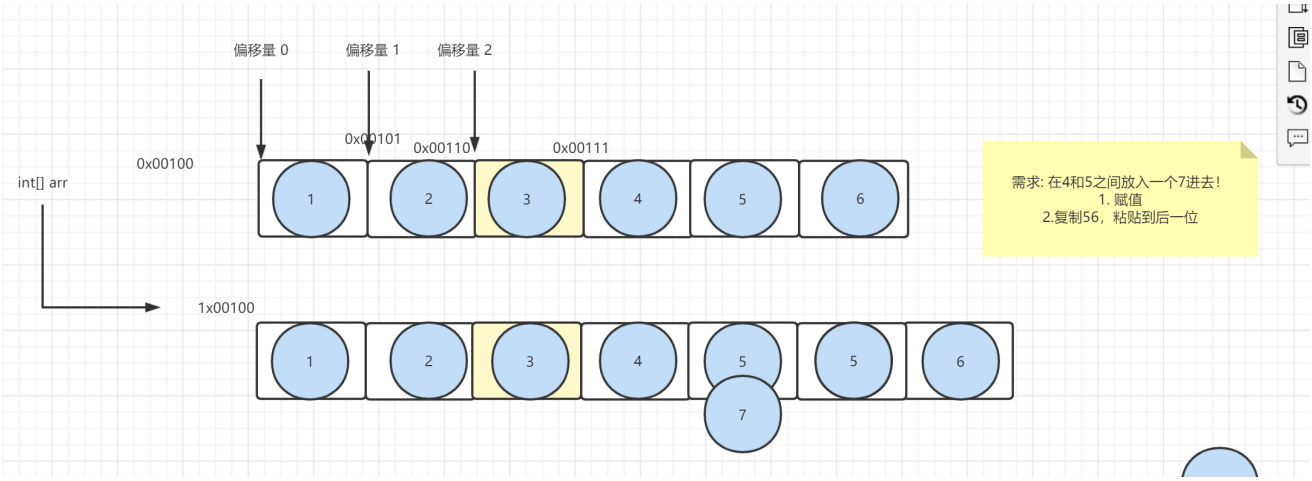
> 证明 查询快的原因



为什么增删慢

> 核心并非是对数组赋值慢，而是发生扩容的时候，比较慢！

数组一旦声明好，那么它的长度就是固定的！




```java
package com.woniuxy.day04Array;

import java.util.Arrays;

/**
 * @Author: 马宇航
 * @Todo: 演示数组的插入方法:arrayList底层源码，ArrayList的底层，扩容是 数组长度 + 数组长度>>1
 * @DateTime: 22/08/22/0022 10:03
 * @Component: 成都蜗牛学苑
 **/
public class TestArray {
    public static void main(String[] args) {
        int[] arr = {1,2,3,4,5,6};
        //需求是在4和5之间，插入一个7!
        //步骤1：扩容
        arr = Arrays.copyOf(arr, 7);
        Arrays.stream(arr).forEach(System.out::print);
        //步骤2：复制下标4和5位置的数据，到5和6的位置
        System.arraycopy(arr,4,arr,5,2);
        System.out.println();
        System.out.print("步骤2后的内容：");
        Arrays.stream(arr).forEach(System.out::print);
        //步骤3：对arr[4]进行赋值
        arr[4] = 7;
        System.out.println();
        System.out.print("步骤3后的内容：");
        Arrays.stream(arr).forEach(System.out::print);
    }
}
```

day05ArrayList源码解读

1. 如何读源码

1. 先看类上的继承关系
2. 看属性，特别是数据结构的内部类，比如Node，Entry，TreeNode等内部类，都是用来定义数据结构！内部定义的final 常量属性(记住大概内容)!
3. 再看构造方法：可以回答目录2的答案。
4. 新增方法：可以回答目录3的答案。

5. 修改方法
6. 删除方法
7. 查询方法

2. ArrayList无参构造创建出来后，默认容量是多少?

答案：0;

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

3. 当初始化一个ArrayList后，第三次扩容后，容量是多少?

答案：22； 初始化是0 第一次扩容 是 10，第二次 15 第三次 22。

根据源码推理答案：

首先，先看add方法：

```
public boolean add(E e) {
        // 确保是否需要扩容 size指的是当前放入的数据量  size = 9
        ensureCapacityInternal(size + 1);  // Increments modCount!!
        elementData[size++] = e;
        return true;
    }
```

```
grow(minCapacity); //扩容核心方法
```

```
private void grow(int minCapacity) { // minCapacity = 11
    // overflow-conscious code
    int oldCapacity = elementData.length;// 10
    //进行扩容数量的 10+ (10>>1) = 10 + 5 = 15  int类型非常重要
    // int 4byte 32bit    0000 1010 >> 1  0000 0101 = 5
    // 下一次扩容 15开始扩容 0000 1111 >> 1  0000 0111 = 7
    int newCapacity = oldCapacity + (oldCapacity >> 1);
    if (newCapacity - minCapacity < 0)//15 - 11 < 0 false
        newCapacity = minCapacity;
    if (newCapacity - MAX_ARRAY_SIZE > 0)//判断数组容量有没有超过上限
        newCapacity = hugeCapacity(minCapacity);
    // minCapacity is usually close to size, so this is a win:
    elementData = Arrays.copyOf(elementData, newCapacity);//扩容方法
}
```

day06ArrayList源码解读2

add第二个方法，和我们手写的数组中间插入的方法是一样的。

```
public void add(int index, E element) {
    rangeCheckForAdd(index);//检查下标是否越界

    ensureCapacityInternal(size + 1);  //检查是否需要扩容
    System.arraycopy(elementData, index, elementData, index + 1,
                     size - index);
    elementData[index] = element;
    size++;
}
```

删除方法：删掉的数据是直接从JVM中消失了么？

```
public E remove(int index) {
    rangeCheck(index);//是否越界

    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;
}
```

改变数据：

```
public E set(int index, E element) {
    rangeCheck(index);

    E oldValue = elementData(index);
    elementData[index] = element;
    return oldValue;
}
```

面试题：ArrayList是否是线程安全的？不是，那么怎么让它是线程安全的？

> 并发情况下，数据的计算会出问题！
>
> Collections.synchronizedList();
>
> 这种方式转换集合为线程安全的集合，可以，但是性能不好，不推荐使用！

```
List list = Collections.synchronizedList(ArrayListDemo.list);
```

正确的答案：

1. 先说Collections可以，但是性能不行；
2. CopyOnWriteArrayList 写时复制：底层原理是写入数据的时候，会复制一个新的数组给你写入，如果此时还有线程在读取，则，读取的是原本的那个数组。

> 数组有一个特点：一旦定义好，则容量不会发生改变。但是size在发生变化！

面试题：ArrayList通过fori循环，从0往后删除数据，会出现什么现象？怎么解决这种问题？

```
package com.woniuxy.day04Array;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

/**
 * @Author: 马宇航
 * @Todo: TODO
 * @DateTime: 22/08/24/0024 09:49
 * @Component: 成都蜗牛学苑
 **/
public class ArrayListDemo {
    static List list = new ArrayList();
    public static void main(String[] args) {
        for (int i = 0; i < 1000 ; i++) {
            ArrayListDemo.list.add(i);
        }
        for (int i = 0; i < 1000; i++) {
            list.remove(i);//list从前往后删，是跳着删，隔一个删一个
            System.out.println(list);
        }
    }
}
```

答案：现象1，最终会数组下标越界，这个越界可以拿出来面试的时候讲讲，源码如下：

```
//检查索引是否越界
private void rangeCheck(int index) {
    //索引 >= 数量(并非数组长度)
    if (index >= size)
        throw new IndexOutOfBoundsException(outOfBoundsMsg(index));
}
```

现象2，数据会跳着删，各一个删除一个。

答案：如何解决这个删除会出现跳着删和下标越界的问题？

倒着删，foreach行不？foreach不行，如下代码，会出现问题！！

真正删除，推荐使用Itr迭代删除。

```
package com.woniuxy.day04Array;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

/**
 * @Author: 马宇航
 * @Todo: TODO
 * @DateTime: 22/08/24/0024 09:49
 * @Component: 成都蜗牛学苑
 **/
public class ArrayListDemo {
    static List<Integer> list = new ArrayList();
    public static void main(String[] args) {
        for (int i = 0; i < 1000 ; i++) {
            ArrayListDemo.list.add(i);
        }
//        for (int i = 999; i >= 0; i--) {
//            list.remove(i);//list从前往后删，是跳着删，隔一个删一个
//            System.out.println(list.size());
//        }
        //遍历删除会出现并发修改异常，不能用，一定不能用!
        for (int i :list) {
            list.remove(i);
        }
    }
}
```

foreach会逐步next()方法找到下一个元素：ArrayList源码如下

```
public E next() {
    checkForComodification();//检查，来执行快速失败，抛出并发修改异常！
    int i = cursor;
    if (i >= size)
        throw new NoSuchElementException();
    Object[] elementData = ArrayList.this.elementData;
    if (i >= elementData.length)
        throw new ConcurrentModificationException();
    cursor = i + 1;
    return (E) elementData[lastRet = i];
}
```

ArrayList有一个**快速失败机制**！

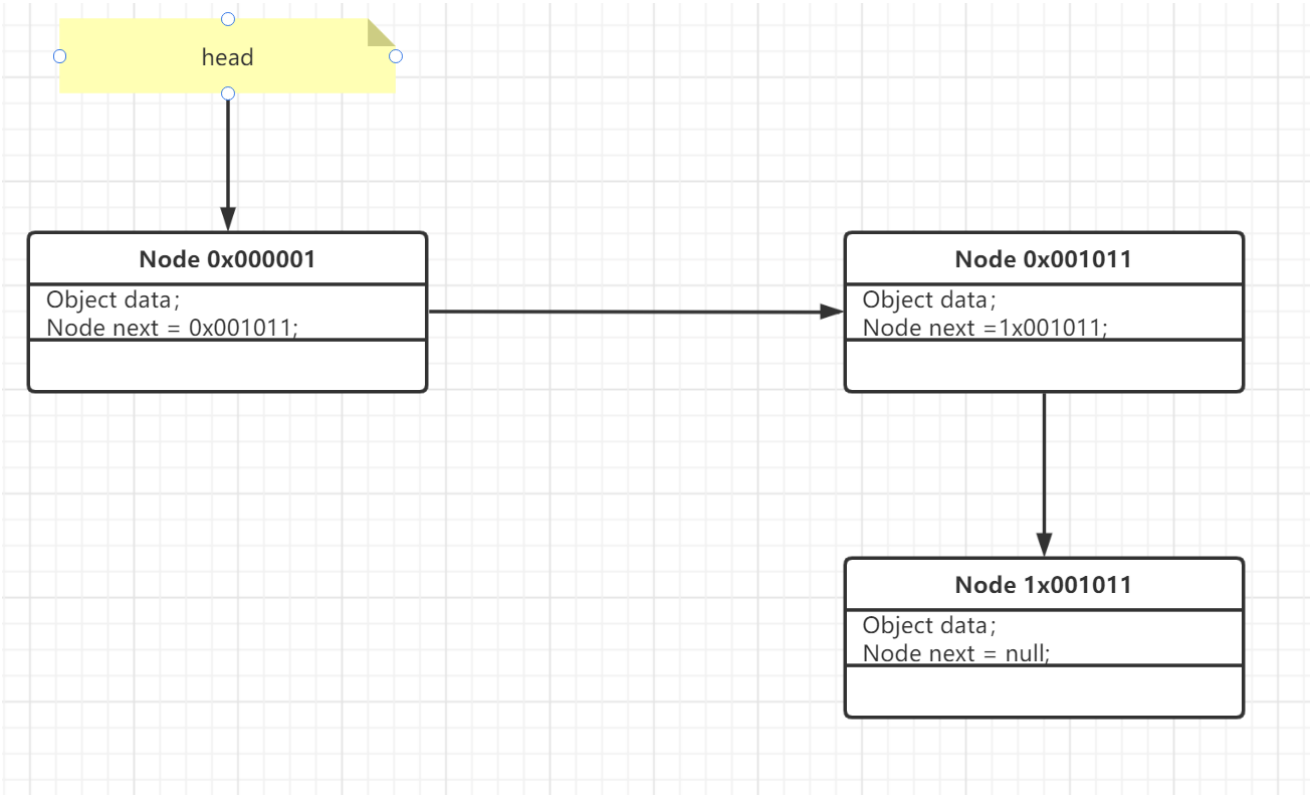
day07 链表

1. 面试：链表的特性？

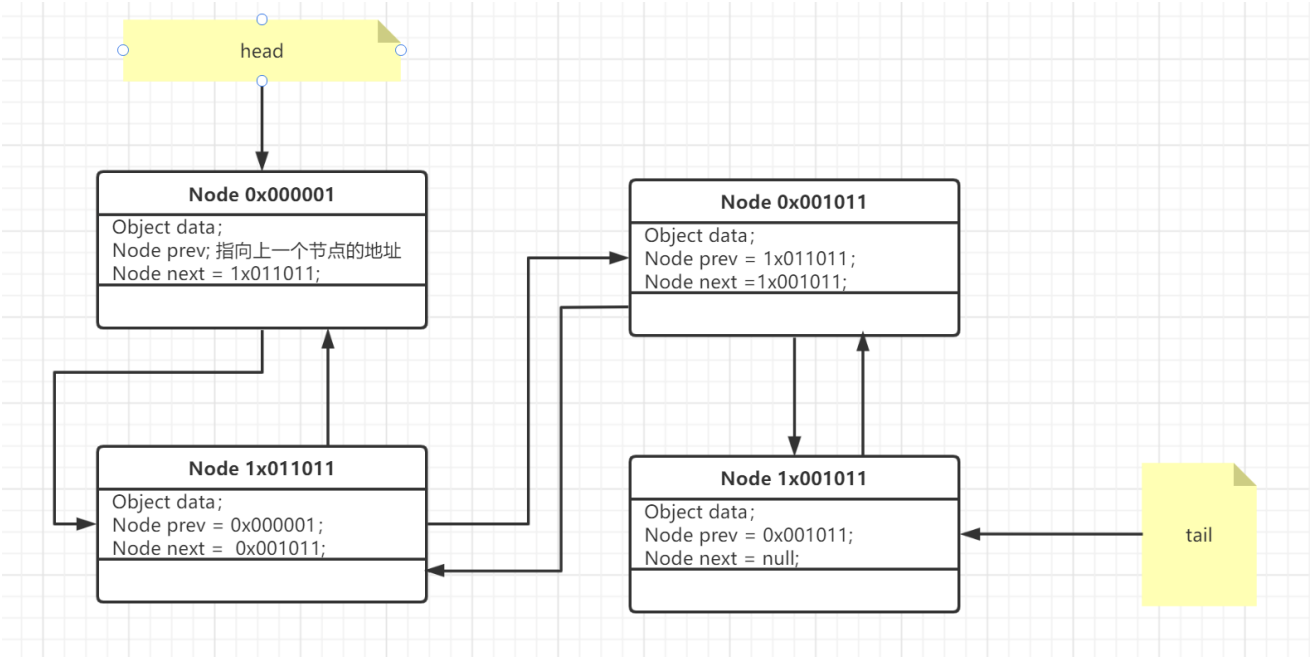
> 查询慢，增删快！
>
> 如下图所示，需要依次从头往后找，才能找到想要的数据。
>
> 增删快只需要修改属性Node next的值即可！

2. 面试：链表的几种接口？

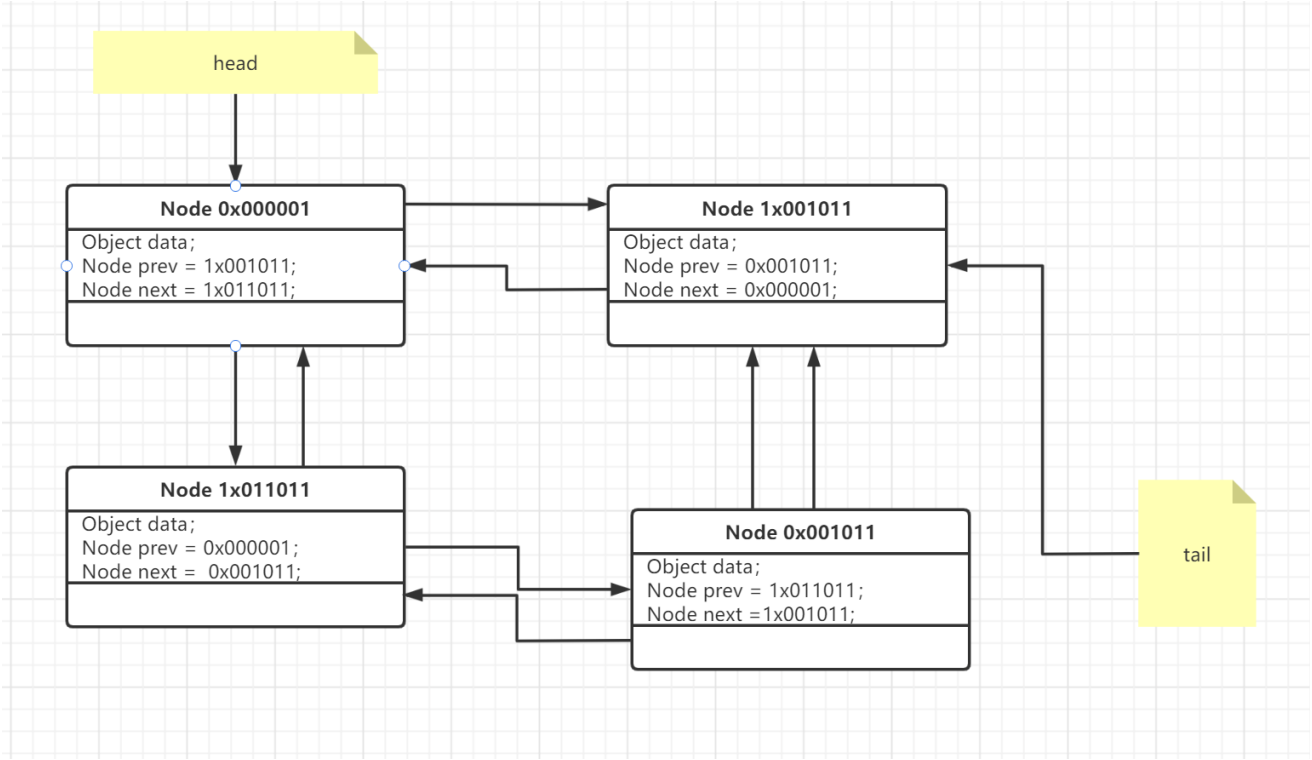
单向链表：


head
Node 0x000001
Object data;
Node next = 0x001011;
Node 0x001011
Object data;
Node next =1x001011;
Node 1x001011
Object data;
Node next = null;

双向链表


head
Node 0x000001
Object data;
Node prev; 指向上一个节点的地址
Node next = 1x011011;
Node 0x001011
Object data;
Node prev = 1x011011;
Node next =1x001011;
Node 1x011011
Object data;
Node prev = 0x000001;
Node next = 0x001011;
Node 1x001011
Object data;
Node prev = 0x001011;
Node next = null;
tail

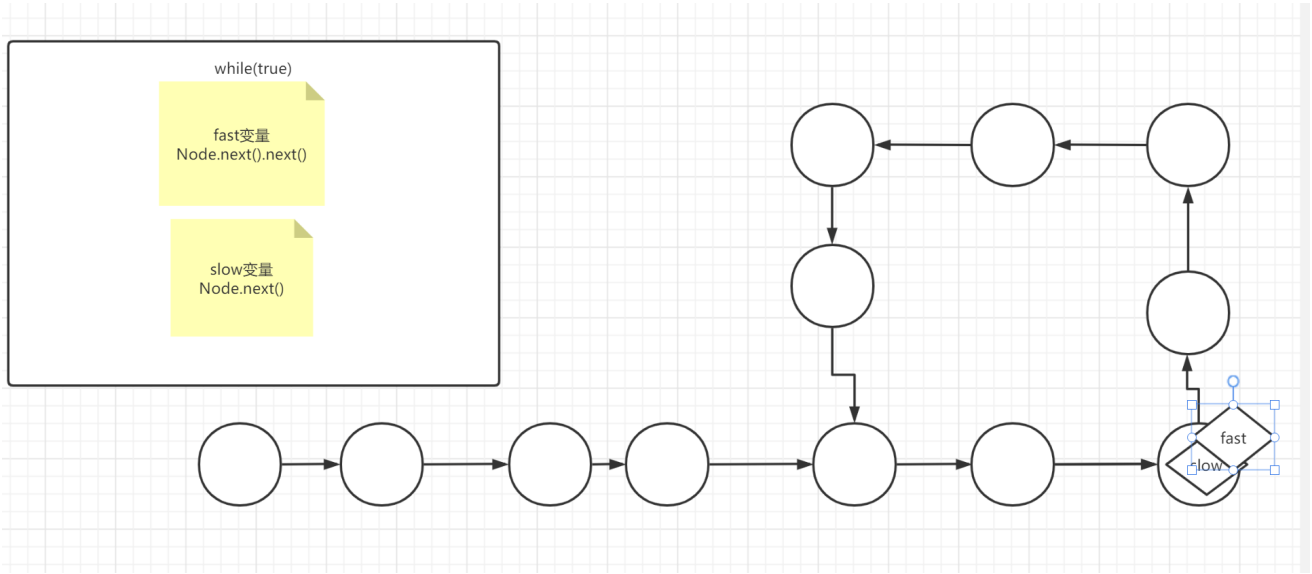
环形链表



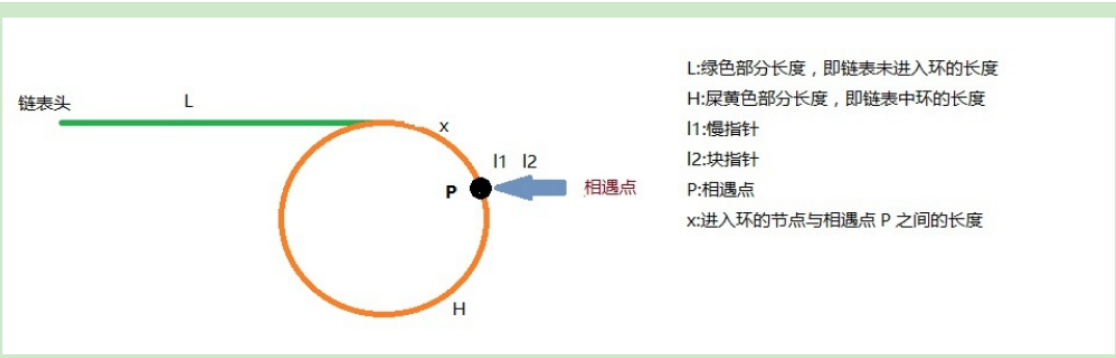
3. 面试：如何判断一个链表中有没有环？求环开始的点在哪里，和相遇的点在哪里？

> 痛苦的事情，从前往后遍历，永远无法结束！
>
> 结束的条件，Node.next == null; 最后一个节点的next属性必须是null才能结束！

答案：快慢指针！！！！一次循环，快指针前进2个节点，慢指针前进1个节点，如果他们相遇了，则是有环。如果说没有环，那么我们要提供一个方法，如果next()方法是null，则表示没有环，结束死循环。



算法公式的提示：



2(L+x) = L+x+n*H (n >= 1) // n 为快指针在闭环上的圈数

=> 2L+2x = L+x+n*H (n >= 1)

=> L = n*H-x (n >= 1)

=> L = n*(H-x)+(n-1)x (n >= 1)

请写出求环链表的代码!

4. 跳表相关问题

5. 延伸出来的树的问题

day08 Hash表

Hash表：哈希表，或者散列表。

> Hash表存在的目的：所有的Hash表都要追求一个极致的目的，希望，我放入的所有数据，都能够**均匀分布**在整个Hash表上，而不会出现冲突!
>
> 怎么实现这个目的：一个优秀的Hash算法！！！最简单的Hash算法可以是直接 取余!

https://www.360kuai.com/pc/96d461b33e52c79ed?cota=4&sign=360_57c3bbd1&refer_scene=so_1

面试题：如果出现hash碰撞(冲突)，怎么办?

1. 链表法(拉链法)：常见于HashMap和HashTable!
2. 开放寻址方式: 性能太差
    1. 线性探测法（di是等差1,2,3,4,5，.....）
    2. 二次探测法（平方再探测di=1^2,-1^2,2^2,-2^2.....）

       hash(key)+1^2或[hash(key)-1^2]，hash(key)+2^2或[hash(key)-2^2]
3. 双重散列(常用方案)

    通过2个Hash算法来计算下标：hi=(h(key)+ i*h1(key))%m (0≤i≤m-1) h表示第一个Hash算法，h1表示第二个Hash算法!

只要你看见一个类的名字，包含Hash字段，则这个类，使用的是Hash表！！！我们常见的hash表都是一维数组实现的。因为Hash算法，计算出来的内容，可以是下标!

面试题：HashMap底层，是如何计算的Hash值(如何计算的下标！)?

答：HashMap底层，使用的是hashCode来计算Hash值，然后，使用位运算的&来计算的下标。

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

> 上面这一段代码，说明了2个方面的问题：
>
> 第一个：key可以为null么?
>
> 第二个：只是用的Object.hashCode方法么?

前置知识：

位运算！！！！！

```
1.位于 运算  &  有0是0
  一步到位，讲替代 % 取余运算 :   16个长度的数组，我希望无论你传入的值是多少，最后计算出来的结果，都是0~15之间;   使用%    x % 16
  x     ;         0000 0000 0000 0000 0000 0000 xxxx xxxx
            &     0000 0000 0000 0000 0000 0000 0000 1111
-------------------------------------------------------------
                  0000 0000 0000 0000 0000 0000 0000 xxx0
                  所以，任意数字x，经过上述计算后，结果是 xxxx，则 xxxx的可能取值范围是:     0000
 1111  === 0 ~ 15
最终 x % 16 可以替换为    x & 15 = x & (16 - 1) = x & (数组长度 - 1)

常见面试题：为什么，HashMap的长度必须是2的指数倍！ 答案在上面和下面的位或！提示：数组长度换成奇数!
反证法：如果hashMap长度是奇数或者非2的指数倍，会出现什么现象！
```

2.位或 运算 | 有1是1

```
//在HashMap中有一段这个代码，用到了位或！这段代码决定了，你传入17，最后返回32!传入33最后返回64!
static final int tableSizeFor(int cap) {
        int n = cap - 1;//为什么要减1？如果传入的是32，避免计算成64。
        n |= n >>> 1;//n = n | n >>> 1; >>>叫做无符号右移
        n |= n >>> 2;
        n |= n >>> 4;
        n |= n >>> 8;
        n |= n >>> 16;
        return (n < 0) ? 1 : (n >= MAXIMUM_CAPACITY) ? MAXIMUM_CAPACITY : n + 1;
    }
y -= x;// y = y - x;
cap = 0010 0110 1010 1110 0100 0100 0000 0111
n = n          //0010 0110 1010 1110 0100 0100 0000 0110
     | n >>> 1 //0001 0011 0101 0111 0010 0010 0000 0011
    ————————————————————————————————————————————————
    n          //0011 0111 1111 1111 0110 0110 0000 0111
     | n >>> 2 //0000 1101 1111 1111 1101 1001 1000 0001
    ————————————————————————————————————————————————
    n          //0011 1111 1111 1111 1111 1111 1000 0111
     | n >>> 4 //0000 0011 1111 1111 1111 1111 1111 1000
    ————————————————————————————————————————————————
后面移动没有变化  //0011 1111 1111 1111 1111 1111 1111 1111
     | n >>> 8 //0000 0000 0011 1111 1111 1111 1111 1111
    ————————————————————————————————————————————————
    n最终值      //0011 1111 1111 1111 1111 1111 1111 1111
    n+1         //0100 0000 0000 0000 0000 0000 0000 0000
```

同学们来实验一个二进制，通过这种运算后的结果是什么？

第一个： 0001 0000 0000 0000 0000 0000 0000 0000

第二个： 0000 0000 0000 0000 0000 0000 0010 0000

最终结果，算出来二进制数中只会有一个1 说明一定是2的指数倍！

3.异或 运算 ^ 相同得0 不同得1