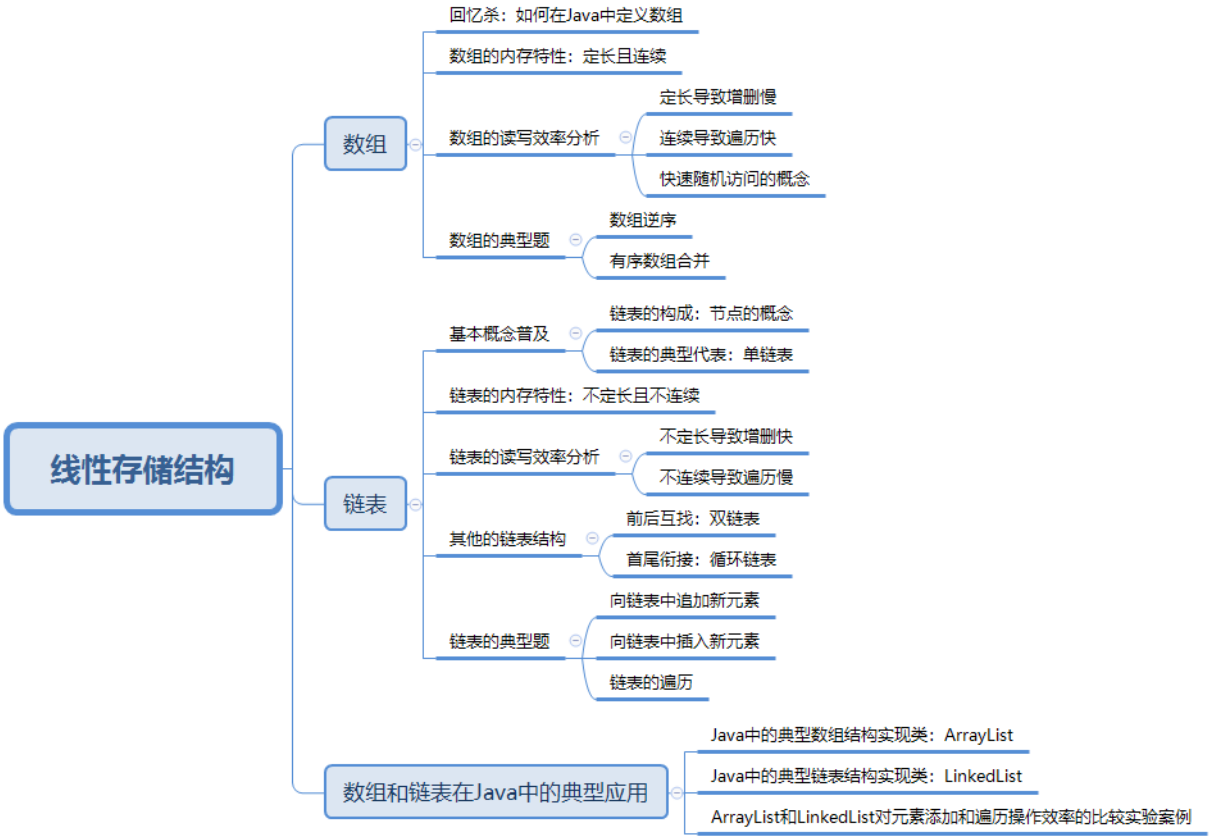


1.数组

①回忆杀：如何在Java中定义数组

在Java中定义数组的方式有两种：静态定义方式 + 动态定义方式

使用静态方式定义数组：在定义数组的时候指定数组的内容，但是不指定数组的长度，数组的长度由虚拟机自行根据元素的个数计算得到

```
int[] array = new int[] {0,1,2,3,4,5,6,7,8,9};  //数组的静态定义方式
```

使用动态方式定义数组：在定义数组的时候指定数组的长度，但是不指定数组的内容，虚拟机在为数组开辟内存空间的时候，使用默认值进行占位

```
int[] array = new int[10];  //数组的动态定义方式
```

注意：

1.在Java中定义数组的时候，数组的长度和内容只能够指定其中一个，不能即指定长度又指定内容，也不能不指定长度，也不指定内容

2.声明数组类型的时候，我们推荐将数组元素类型和[]放在一起，将类似于int[]整体看做一个独立的数据类型

3.在使用动态方式创建数组的时候，虚拟机在为数组开辟空间之后，这个数组中并不是“真空”的，而是使用元素默认值进行占位：

byte[]、short[]、int[]、long[]：默认值为0

float[]、double[]：默认值为0.0

boolean[]：默认值为false

char[]：默认值为Unicode值的0

引用数据类型数组，如String[]：默认值为null

4.不管数组中存储的元素类型是基本数据类型的元素还是引用数据类型的元素，数组类型本身是一种引用数据类型

②数组的内存特性：定长且连续

数组在内存中的特性，可以用一句话来概括：定长且连续

其中：

定长指的是在Java中，一个数组对象在内存中一旦被创建，其长度将不能被修改；如果想要修改一个数组的长度，那么只能重新new一个数组

连续指的是在Java中，存在于同一个数组中的所有元素，其内存地址之间是连续有规律的

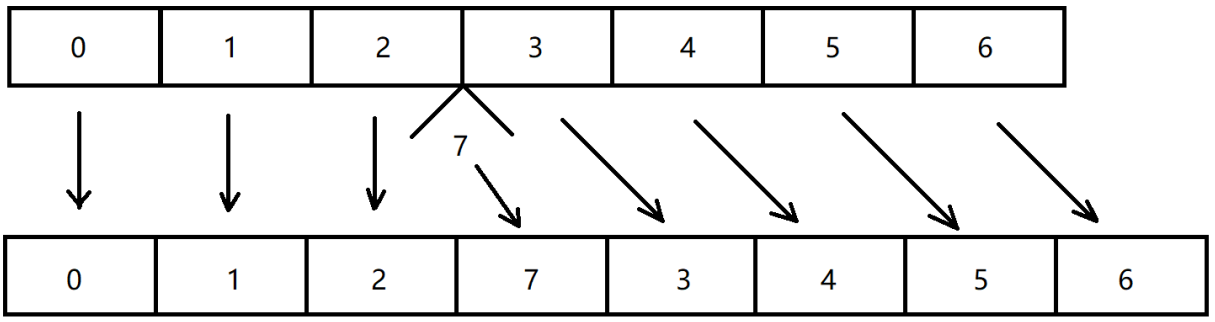
③数组的读写效率分析

1.定长导致增删慢

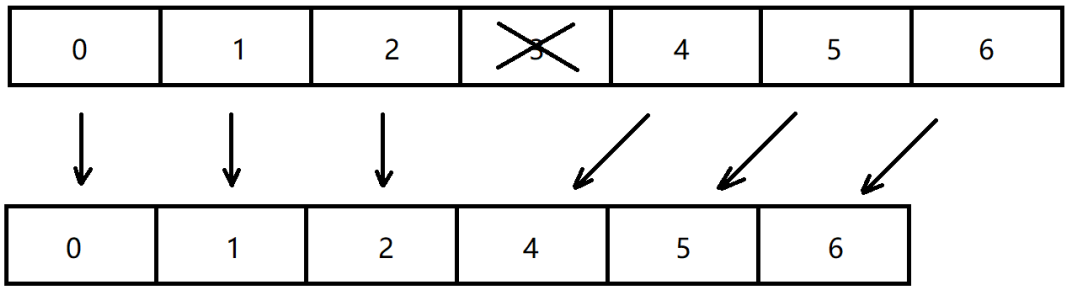
上文说到，Java中的数组在内存中一旦创建，其长度是不可变的，也就是说如果需要改变一个数组的长度，那么就需要重新创建一个数组

但是在Java中，创建对象属于十分消耗时间和内存的一种操作，所以如果在涉及到对数组元素的插入和删除，则必然涉及到对数组对象的重新创建和数组元素的拷贝

向数组中插入元素



删除数组中的元素



由此可见，在向数组中插入元素，或者删除数组元素的时候，都涉及到新数组的创建和原始元素的拷贝，所以这种操作是很慢的

2.连续导致遍历快

上文说到，在Java中，处于同一个数组中的元素之间的内存地址是有规律的，也就是说我们可以认为这些内存地址是连续存在的

只要是连续存在的内存地址，那么我们就可以直接通过某种方式计算得到某一位元素的内存地址，进而访问这个数组元素

也就是说，在通过下标访问数组中元素的时候，我们并不需要从数组的第一个元素开始，一个一个的向后查询这些元素，我们只要

根据这些规律计算得到目标元素的内存地址就可以了

假设这个数组中，下标为0的元素的内存地址是0x0010，并且内一个元素占用的内存空间是10，那么数组中元素的内存地址如下图所示：

0x0010	0x0020	0x0030	0x0040	0x0050	0x0060	0x0070
abc	bcd	cde	def	efg	fgh	ghi
下标：0	1	2	3	4	5	6

已知条件：数组中第一个元素的内存地址是0x0010（数组元素的首地址是0x0010）
数组中每一个元素占用的内存大小是10，
求：下标为5的元素的内存地址

下标为5的元素的内存地址 = 0x0010 + (10 * 5) = 0x0060

其中说明一下：数组中下标为0的元素的内存地址就是这个数组在内存中的起始地址，也称之为数组的首元素地址

由此可见，通过公式计算元素的内存地址，比逐个遍历数组元素进行查找要快的多

3.快速随机访问的概念

上文说到，在按照下标访问数组中元素的时候，我们并不需要逐个遍历数组中的元素进行查找

我们只需要按照数组元素首地址、单个数组元素大小和目标元素下标这三个参数直接套用公式，就能够计算得到目标元素的内存地址

而其中，数组的首地址和数组中单个元素的内存大小都是在创建数组的时候就已经确定的，所以我们只要告诉虚拟机，我们要访问的数组元素下标就可以了

下面，请各位同学记住这个公式：

数组目标元素内存地址 = 数组的首元素地址 + (数组元素占用内存大小 * 目标元素下标)

通过这种公式计算的方式得到数组元素内存地址的方式，称之为快速随机访问（Quick Random Access）

上面的公式，称之为快速随机访问公式

④数组的典型题

1.数组逆序

题目说明：

将一个数组中的所有元素以倒置的顺序重新存放在这个数组中

题目案例：

给定数组：[1,4,2,7,5,9,6]

倒序存储：[6,9,5,7,2,4,1]

思路解析：

使用两个变量i和j，分别指向数组的起点和数组的终点，i变量向后走，j变量向前走

在遍历数组过程中将array[i]和array[j]中的元素值使用一个临时空间进行互换

循环条件是i < j

2.有序数组合并

题目说明：

给定两个升序有序的数组，将这两个数组合并为一个升序有序的数组

题目案例：

给定两个升序有序的数组：

arr1 = [1,2,4,4,6,8,9]

arr2 = [0,1,3,6,7]

合并后结果为：[0,1,1,2,3,4,4,6,6,7,8,9]

思路解析：

使用两个变量i和j分别遍历数组arr1和数组arr2，

遍历过程中比较arr1[i]和arr2[j]之间的大小关系，并且将较小的一个元素落在结果数组中

哪一个数组中的元素落在结果数组中，哪一个数组的下标向前进1

直到有一个数组先遍历完成，将另一个数组中剩余的元素全部落在结果数组即可

2.链表

①基本概念普及

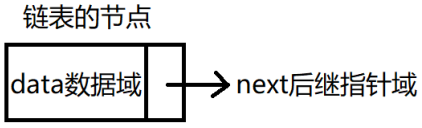
1.链表的构成：节点的概念

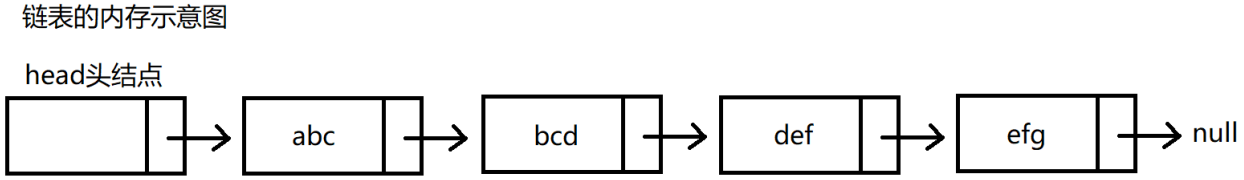
链表和数组不同，他的结构在内存中并不是连续的

大家可以想象一下现实生活中的锁链：一整条锁链是由一个一个的铁环构成的

一整条锁链，就可以看做是一个链表，锁链中的每一个铁环，就可以看做是链表的最基本单位：节点（Node）

链表的内存还意图如下：





从上图我们不难看出：

1.链表的节点一般分为两个部分：data数据域，用来存储要保存的数据，例如一个字符串、一个User对象等等；next后继指针域，用来保存下一个节点的内存地址，串起整个链表结构

2.在链表中，链表的第一个节点通常不存储任何数据，他仅仅用来引起整个链表，我们将这个特殊的节点称之为链表的头结点

3.在整条链表中，我们只要知道了链表头结点的内存地址，就可以顺着之后每一个节点的next后继指针域向下，逐个找到后续的所有节点

4.链表的最后一个节点的后继指针域取值为null，这一特性在遍历整个链表的时候，常用来判断是否还有后继节点

也可以打一个比方：整个链表就好比一列火车，头结点就是火车头，火车头是不搭载乘客的；后面的每一个节点就是普通的火车车厢，车厢中的乘客就是节点data数据域保存的数据

而节点的next后继指针域就好比车厢之间的钩子，沿着这个钩子就可以找到下一节车厢；如果一节车厢的钩子没有挂载其他车厢，说明这个车厢已经是最后一节车厢了

链表节点的Java代码，可以按照如下格式进行声明：

```
class Node {
    Object data;  //数据域的定义，为了能够保存任意数据类型的数据，故采用Object数据
    Node next;  //后继指针域，因为一个节点的下一位还是一个节点，所以通过一个Node数
}
```

2.链表的典型代表：单链表

上图演示的链表结构是一种非常常见，也非常典型的链表结构，称之为单链表。

之所以称之为单链表，是因为链表的每一个节点只有一个next后继指针域，所以在遍历这个链表的时候，只能够单向的从前向后进行遍历

不能够从后向前进行遍历，所以这种链表结构称之为单链表

除了单链表之外，在实际应用当中，根据应用场景的不同，我们还可能用到双链表、循环链表、十字链表等结构

这些特殊的链表结构，我们将在后续进行说明

②链表的内存特性：不定长且不连续

链表的内存特性，正好和数组是相反的，一句话概括就是：不定长且不连续

不定长指的是，在内存中，链表的节点数量是动态分配的，一个链表结构中存在多少个节点，取决于我们向这个链表中添加了多少元素

如果我们想要向这个链表中追加元素或者插入元素，那么我们只要新建一个节点保存这个元素，并且改变几个引用值，就可以完成操作

并不需要重建整个链表，更不需要拷贝原始链表中的任何元素

不连续指的是，在每次添加新元素到链表中的时候，链表的节点都是重新new出来的

正如大家知道的，每次new出来的对象，即使数据类型是一样的，但是他们之间的内存地址也是互相没有关系的

也就是说，即使是存储在同一个链表中的不同节点，他们之间的内存地址也是没有规律，不连续的

这样一来，如果我们想要遍历链表中所有的节点，或者按照下标找到链表中的特定节点，那么不得不每一次都重新从链表的头结点出发

一个一个的遍历节点，查找想要的元素

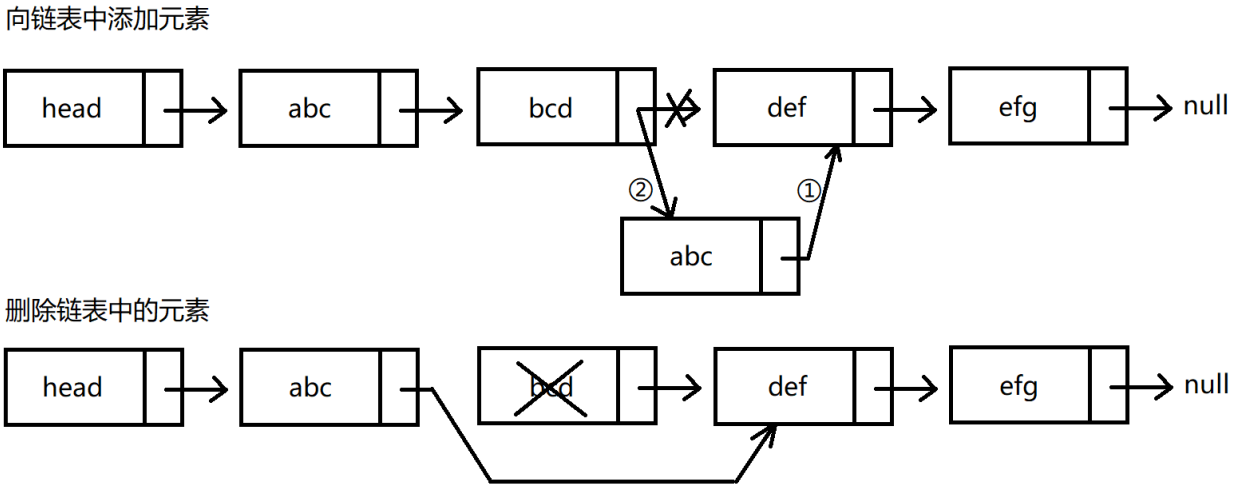
③链表的读写效率分析

1.不定长导致增删快

上文说到，在同一个链表中插入元素的时候，是不需要重新创建一整条链表的，我们只要创建一个新节点

并且改变原始链表中的一些后即指针的取值，就能够实现节点的添加操作；

同理，从链表中删除节点也是一样的操作，都不涉及到整个链表的重新创建



从上图我们不难看出，在向链表中添加节点和删除节点的时候，我们更多的是在操作节点的后继指针的取值，而并没有创建或者删除整个链表结构

这样一来，和数组相比，我们就能够节省下来大量用于创建对象和拷贝原始数据的时间

所以，链表的不定长特性导致链表增删元素比较快

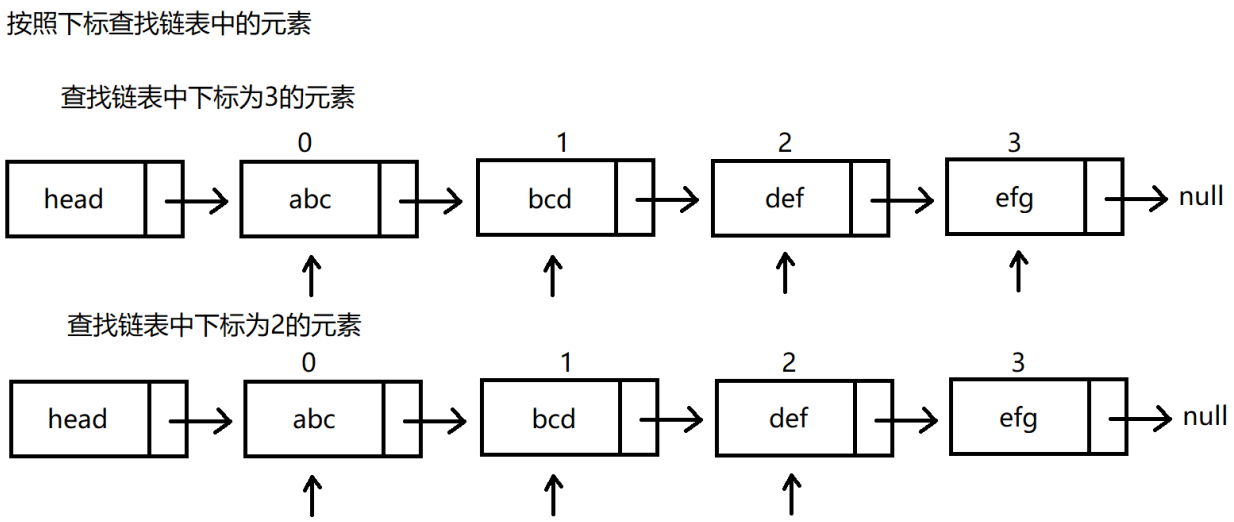
2.不连续导致遍历慢

上文说到，当一个链表需要存储一个新的元素的时候，都要重新new一个节点对象出来，而重新new出来的节点对象的内存地址

和其他节点的内存地址之间是没有任何关系的

所以从这一点看来，如果我们想要按照下标对链表中的元素进行访问，类似于数组中的快速随机访问的公式就是不可用的了

所以我们在按照下标访问链表元素的时候，就不得不每一次都从链表的头结点开始，每向后遍历一个节点就记一次数，知道到达目标下标的节点为止



所以从这个角度来件，从链表中查找元素的次数越多，花费的时间就越多

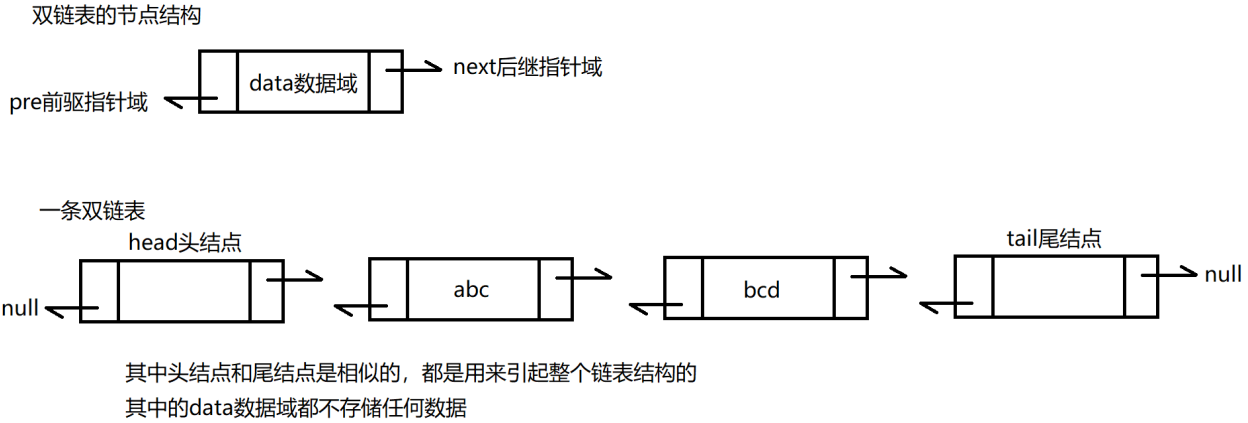
所以，链表节点之间内存地址不连续的特性，导致链表的遍历要慢于数组结构

④其他的链表结构

1.前后互找：双链表

如果我们在链表的节点中定义两个指针域，一个指向当前节点的下一个节点，一个指向当前节点的前一个节点

那么我们就可以从前后两个方向来遍历这个链表，由此也就产生了双链表结构

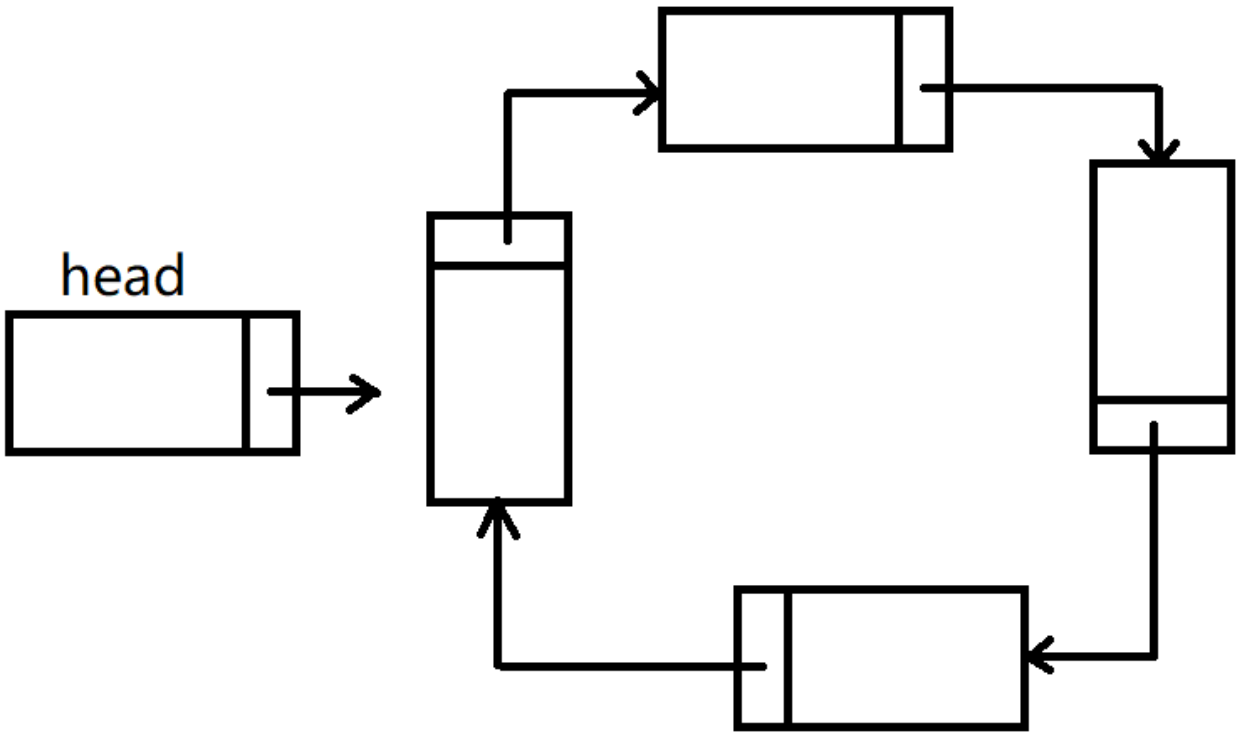


2.首尾衔接：循环链表

如果一个链表的最后一个节点的后继指针域并不是指向null，而是回过头来直接指向第一个存储数据的节点

那么这种结构就形成了环链表结构，也称之为循环链表

循环链表结构在诸如磁盘模拟算法、解决约瑟夫环问题等场景下，有着大量的应用



⑤链表的典型题

1.向链表中追加新元素

题目说明：

向一个单链表结构的最后面添加一个新元素

题目案例：

原始链表结构：abc -> bcd -> cde -> def

追加新元素：efg

得到链表结构：abc -> bcd -> cde -> def -> efg

思路解析：

在链表的封装类中定义一个Node类型的变量lastNode，用来始终指向最后一个节点

在向链表中追加元素的时候，直接使用lastNode来进行操作更加便捷

2.向链表中插入新元素

题目说明：

向链表的指定下标位插入新元素

题目案例：

原始链表结构：abc -> bcd -> cde -> def

向链表下标为2的位置插入一个新元素efg

得到链表结构：abc -> bcd -> efg -> cde -> def

思路解析：

首先遍历链表，找到下标为2的位置

新建一个节点保存数据，并通过修改后继指针域的方式，将新节点加入指定位置

3.链表的遍历

题目说明：

从链表中第一个存储数据的节点开始，向后遍历链表中所有的节点

并将节点数据域中保存的数据取得，进行打印输出

题目案例：

链表结构：abc -> bcd -> cde -> def

打印输出：[abc bcd cde def]

思路解析：

从链表头结点的下一个节点，也就是链表中真正存储元素的节点开始

向后逐个遍历节点，并取得节点数据域中的数据，进行打印输出

利用链表中最后一个节点的后继指针域的取值为null的特性，控制链表的遍历过程

3.数组和链表在Java中的典型应用

①数组和链表在Java中的典型应用类型

在Java中，如果我们想要在不同的应用场景下，使用数组或者链表的特性对数据进行存储和遍历

并不需要每一次都重新手动封装这些数据结构

因为在Java中已经替我们封装好了这些数据结构

1.ArrayList——数组的典型封装

ArrayList类型内部使用一个Object[]类型的数组对数据进行存储

从结构上来看，这是一个典型的数组结构的封装

并且为了提升添加元素的效率，ArrayList类型的对象在最初的时候，会给定一个长度为10的Object[]数组来存储元素

（值得一说的是，在JDK 1.8当中，如果使用ArrayList的空构造器创建一个对象，那么这个数组的默认长度是0，只有在第一次向ArrayList中添加元素的时候，才会扩展为10）

并且在之后每添加一个元素的时候，ArrayList都会判断当前的数组长度是否能够容纳下这个新元素

如果数组的长度不够了，那么ArrayList会对数组进行扩容，而扩容的方式也并不是简单的添加一个元素空间

而是将数组的长度扩展为原来的1.5倍，也就是说，ArrayList内部数组长度的默认扩展方式为：10 -> 15 -> 22 -> 33 -> 49 -> ...

2.LinkedList——链表的典型封装

LinkedList内部使用一个双链表来保存数据

之所以使用双链表，原因正如前面提到的，可以从链表头和链表尾两个方向来遍历链表

这样一来，不管是查找链表中的元素，还是向链表的指定位置添加新元素，都能够提升一定的效率，从而节省时间

②ArrayList和LinkedList对元素增删遍历的效率比较

其实如果想要比较数组和链表在增删元素以及遍历元素方面的效率差异，还是比较简单的

我们只要向ArrayList和LinkedList中添加100000个元素，然后再对这两个结构进行遍历，并记录这些操作的时间差

就能够很直观的看到其中的差异

案例代码如下：

```
public class TestSpeed {

    public static void main(String[] args) {

        //数组和链表的增删遍历效率比较

        long start = 0;
        long end = 0;

        ArrayList<Integer> l1 = new ArrayList<>();
        LinkedList<Integer> l2 = new LinkedList<>();

        System.out.println("-----添加效率测试-----");

        System.out.println("向ArrayList中插入100000个元素...");
        start = System.currentTimeMillis();
        for(int i = 0; i < 100000; i++) {
            l1.add(0, i);
        }
```

```
        end = System.currentTimeMillis();
        System.out.println("添加完成，用时" + (end - start) + "毫秒");

        System.out.println("向LinkedList中插入100000个元素...");
        start = System.currentTimeMillis();
        for(int i = 0; i < 100000; i++) {
            l2.add(0, i);
        }
        end = System.currentTimeMillis();
        System.out.println("添加完成，用时" + (end - start) + "毫秒");

        System.out.println("-----遍历效率测试-----");

        System.out.println("对ArrayList执行100000次元素查找...");
        start = System.currentTimeMillis();
        for(int i = 0; i < 100000; i++) {
            l1.get(i);
        }
        end = System.currentTimeMillis();
        System.out.println("遍历完成，用时" + (end - start) + "毫秒");

        System.out.println("对LinkedList执行100000次元素查找...");
        start = System.currentTimeMillis();
        for(int i = 0; i < 100000; i++) {
            l2.get(i);
        }
        end = System.currentTimeMillis();
        System.out.println("遍历完成，用时" + (end - start) + "毫秒");

    }

}
```

说明：

代码中的System.currentTimeMillis()方法，返回一个long型值，用来表示当前时间的时间戳取值

时间戳的起点是：格林尼治时间1970年1月1日的00:00:00 0毫秒； 北京时间的1970年1月1日的08:00:00 0毫秒

并且每过1毫秒，时间戳的取值就会+1

那么我们可以得知：距离时间戳越远的时间，System.currentTimeMillis()方法的返回值就越大

而计算两次System.currentTimeMillis()方法之间返回值之间的差值，就能够得到中间代码的运行用时的毫秒数

注意：1秒钟 = 1000毫秒