数据结构与算法

目录

数据结构与算法 1
数据结构 3
稀疏数组 3
思路 3
使用场景 3
稀疏数组结构 3
二维数组转稀疏数组 3
稀疏数组转二维数组 4
队列 6
思路 6
使用场景 6
队列结构 6
队列添加数据 6
队列取出数据 7
问题分析及优化 8
环形队列 8
单向链表 11
思路 11
使用场景 11
单向链表结构 11
添加（不考虑 id 顺序） 11
添加（根据 id 顺序） 12
插入（表头） 14
查找 15
有效个数 16
反转链表 16
逆序打印 17
查找倒数第 n 个节点 19
修改内容 19
删除 20
单向循环链表 22
双向链表 26
思路 26
使用场景 26
双向链表结构 26
查找 26
添加(链表头) 27
添加 28
插入（根据 id 大小排序） 28
修改 30
删除 30
栈 32

思路 32
结构 32
入栈 33
出栈 33
栈的使用（计算中缀表达式） 34
前缀（波兰式），中缀，后缀（逆波兰式）表达式 35
前缀表达式 35
中缀表达式 35
后缀表达式 35
例子 35
中缀转后缀 35
逆波兰计算器 38
算法 40
递归 40
思路 40
使用场景 40
打印 41
阶乘 41
迷宫问题 42

数据结构

稀疏数组

二维数组 twoArray

```
0   0   0   0   0
0   0   8   0   0
0   0   0   0   -16
0   0   0   0   0
0   0   0   0   0
```

稀疏数组 sparseArray

稀疏数组行标	二维数组行	二维数组列	值
0	5	5	2
1	1	2	8
2	2	4	-16

思路

当一个数组大部分元素为同一个值的时候，可以用稀疏数组来保存不同元素的位置和值以达到压缩的作用

使用场景

如棋盘，地图等

稀疏数组结构

第 0 列分别记录总行数，总列数，总个数
第 1~n 列分别记录元素坐在行，元素坐在列，元素数值

二维数组转稀疏数组

1. 遍历原始二维数组得到有效数据的个数 sum
2. 根据 sum 创建稀疏数组 sparseArray int[sum+1][3]
3. 将二维数组的有效数据存入到稀疏数组中

```java
// 二维数组转化稀疏数组
// 1.    遍历原始二维数组得到有效数据（非0）的个数sum
int sum = 0;
for (int i = 0; i < twoArray.length; i++) {
    for (int j = 0; j < twoArray[0].length; j++) {
        sum = twoArray[i][j] != 0 ? sum + 1 : sum;
    }
}
//2.    根据sum创建稀疏数组sparseArray int[sum+1][3]
int[][] sparseArray = new int[sum + 1][3];
//给稀疏数组赋值 00是二维数组的总行数，01是二维数组的总列数，02是有效数据
sparseArray[0][0] = twoArray.length;
sparseArray[0][1] = twoArray[0].length;
sparseArray[0][2] = sum;
// 3.    遍历二维数组并将二维数组的有效数据存入到稀疏数组中
// 计数器，记录是第几个有效数据
int count = 0;
for (int i = 0; i < twoArray.length; i++) {
    for (int j = 0; j < twoArray[0].length; j++) {
        if (twoArray[i][j] == 0) {
            continue;
        }
        count++;
        sparseArray[count][0] = i;
        sparseArray[count][1] = j;
        sparseArray[count][2] = twoArray[i][j];
    }
}
// 输出稀疏数组的形式
System.out.println();
System.out.println("2.稀疏数组");
for (int[] row : sparseArray) {
    System.out.printf("%d\t%d\t%d\t\n", row[0], row[1], row[2]);
}
```

稀疏数组转二维数组

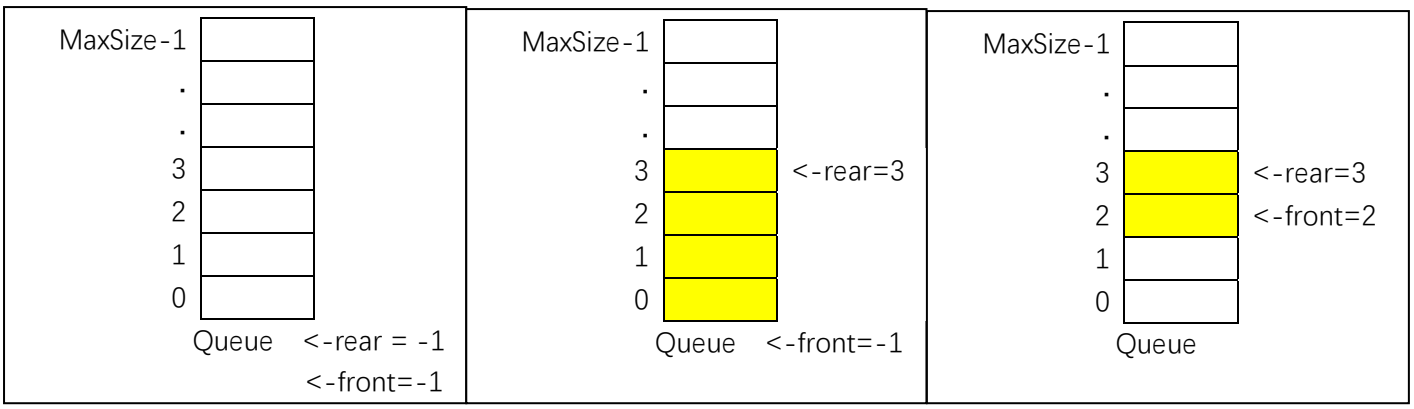
1. 先读取稀疏数组的第一行，根据第一行的数据创建二维数组如上图 twoArray=int[5][5]
2. 再读取稀疏数组后几行数据并赋值给二维数组

```
// 输出稀疏数组的形式
System.out.println();
System.out.println("2.稀疏数组");
for (int[] row : sparseArray) {
    System.out.printf("%d\t%d\t%d\t\n", row[0], row[1], row[2]);
}

//稀疏数组转二维数组

// 1.   先读取稀疏数组的第一行，根据第一行的数据创建二维数组如上图twoArray=int[5][5]
//读取稀疏数组的第0行第0个元素获得二维数组的行，第0行第1个元素获得二维数组的列
int[][] twoArray2 = new int[sparseArray[0][0]][sparseArray[0][1]];
System.out.println();
System.out.println("3.创建后未还原数据的二维数组");
for (int[] row : twoArray2) {
    for (int data : row) {
        System.out.printf("%d\t", data);
    }
    System.out.println();
}
// 2.   再读取稀疏数组后几行数据并赋值给二维数组
for (int i = 1; i < sparseArray.length; i++) {
    //sparseArray[i][0]是第几行，sparseArray[i][1]是第几列，sparseArray[i][2]是值
    twoArray2[sparseArray[i][0]][sparseArray[i][1]] = sparseArray[i][2];
}
//输出还原后的二维数组
System.out.println();
System.out.println("4.还原后的二维数组");
for (int[] row : twoArray2) {
    for (int data : row) {
        System.out.printf("%d\t", data);
    }
    System.out.println();
}
```

队列



思路

按队列是一个**有序列表**，可以用**数组**和**链表**实现。数组顺序存储，链表链式存储。
遵循**先入先出**的原则，即**先存入队列的数据，要先取出，后存入的要后取出**。

使用场景

如：叫号系统，秒杀系统，高并发情况下 API 限流等

队列结构

maxSize:队列最大容量
front:队列头的下标（相当于 C 语言链表的的头指针）随着数据输出而改变
rear:队列尾的下标（相当于 C 语言链表的的尾指针）随着数据输入而改变
queue:队列本身

队列添加数据

1. 先判断队列是否已满
2. rear 与 maxSize-1 对比，相等则队列已满 **注:**队列从 0 开始所以队列最后一个元素的下标为 maxSize-1
3. 未满则 rear 后移一位，并将数据存入 queue[rear]

```
/**
 * 判断队列是否满
 *
 * @return true: 队列满;
 * false:队列未满
 */
public boolean sizeFull() {
    return rear == maxSize - 1;
}
```

```java
/**
 * 添加数据到队列
 *
 * @param n 要添加的数据
 */
public void addQueue(int n) {
    //判断队列是否满
    if (sizeFull()) {
        System.out.println("队列已满，不能加入数据");
        return;
    }
    //让rear后移一位再存进数据到队列中
    array[++rear] = n;
}
```

队列取出数据

1. 先判断队列是否为空
2. rear 与 front 对比，相等则代表队列为空
3. 不空则 front 后移一位,并将数据从 queue[front]取出

```java
/**
 * 判断队列是否为空
 *
 * @return true: 队列为空;
 * false:队列不空
 */
public boolean sizeEmpty() {
    return rear == front;
}
```

```java
/**
 * 取出队列的数据，front后移
 *
 * @return 出队列的数据
 */
public int getQueue() {
    if (sizeEmpty()) {
        //抛出异常
        throw new RuntimeException("队列为空，不能取数据");
    }
    //front后移一位再队列中取出数据
    return array[++front];
}
```

问题分析及优化

1. 队列只能使用一次，没有达到复用的基本要求
2. 将这个数组使用算法，改进成环形队列通过求模的方式完成%

环形队列

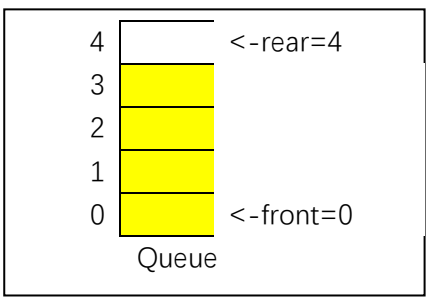
环形队列结构

maxSize:队列最大容量
front:队列头的下标指向队列第一个元素
rear:队列尾的下标指向队列的**最后一个元素的后一个位置，空出一个空间为约定**
queue:队列本身

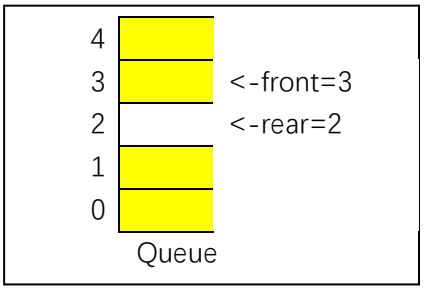
思路:

1. front:指向队列第一个元素，初始默认值为 0
2. rear:指向队列的**最后一个元素的后一个位置,**初始默认值为 0
3. 当队列满时，条件是(rear+1)%maxSize=front

举例子:
maxSize=5 时队列满即 rear 指向 4 即队列的最后一个元素，
rear+1=5　则 5%5=0，而 front 指向 0 则队列**[满]**



maxSize=5 时队列弹出元素后 front 指向元素 3 继续补充对象到满时则 rear 指向 2，rear+1=3　则 3%5=3，而 front 指向 3 则队列**[满]**
想象成环状则可能发生尾指针在头指针之前



4. 队列为空的条件 rear==front 则为[空]
5. 队列的有效数据 (rear+maxSize-front)%maxSize

队列添加数据

1. 先判断队列是否已满
2. (rear + 1) % maxSize == front 对比，相等则队列已满
   注:队列从 0 开始所以队列最后一个元素的下标为 maxSize-1
3. 未满则数据存入 queue[rear]
4. 将 rear 后移一位(求模防止数组越界) rear = (rear + 1) % maxSize

```
/**
 * 添加数据到队列
 *
 * @param n 要添加的数据
 */
public void addQueue(int n) {
    //判断队列是否满
    if (sizeFull()) {
        System.out.println("队列已满，不能加入数据");
        return;
    }
    //存进数据到队列中
    array[rear] = n;
    //rear后移，这里必须取模防止数组越界
    rear = (rear + 1) % maxSize;
}
```

队列取出数据

1. 先判断队列是否为空
2. rear 与 front 对比，相等则代表队列为空
3. 不空则先把 queue[front]保存的到临时变量
4. 将 front 后移一位,取模防止数组越界　(front + 1) % maxSize

5. 将临时保存的变量返回

```java
/**
 * 取出队列的数据，front后移
 *
 * @return 出队列的数据
 */
public int getQueue() {
    if (sizeEmpty()) {
        //抛出异常
        throw new RuntimeException("队列为空，不能取数据");
    }
    /*
        1.先把front指向的值保存的到临时变量
        2.将front后移一位,取模防止数组越界
        3.将临时保存的变量返回
    */
    int value = array[front];
    front = (front + 1) % maxSize;
    return value;
}
```

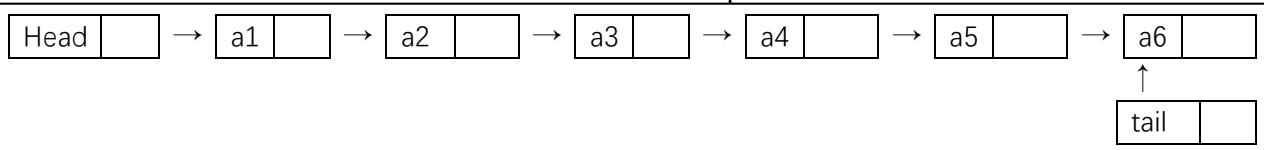
单向链表

内存结构

	头指针	地址	数据域(data)	指针域(next)
head	1200H	1000H	a2	1800H
	尾指针	1100H		
tail	1500H	1200H	a1	1000H
		1300H	a4	1700H
		1400H		
		1500H	a6	null
		1600H		
		1700H	a5	1500H
		1800H	a3	1300H

逻辑结构

地址	数据域（data）	指针域（next）
1200H	a1	1000H
1000H	a2	1800H
1800H	a3	1300H
1300H	a4	1700H
1700H	a5	1500H
1500H	a6	null



思路

1. 链表以节点的方式存储,像锁链一样一环扣一环，链式存储
2. 每个节点都有 data 域和 next 域，data 域负责存储数据，next 域指向下一个节点
3. 通过上图可知链表节点不一定连续存储
4. 链表有带头结点和没有头结点的
5. Head 节点不存放数据，作用是表示单向链表的头，tail 节点也不存放数据，作用是表示单向链表的尾
6. 单向链表的节点不能自我删除需要依靠辅助节点

使用场景

如： 对线性表的长度或者规模难以估计，频繁做插入删除操作

单向链表结构

head 头指针和 tail 尾指针，两个节点指针
节点结构：data：程序员自己设置变量，如 id，name 等
next：节点类 如 Node

添加（不考虑 id 顺序）

1. 创建 head 节点，表示链表头，tail 尾指针指向头节点
2. 通过尾指针获得链表最后一个元素并将下一节点的地址指向新节点
3. tail 尾指针向后移动。

```java
/**
 * 添加节点到单向链表（不考虑id顺序）
 * 1.找到当前链表的最后节点
 * 2.将最后这个节点的next指向新的节点
 *
 * @param newNode 新节点
 */
public void add(NodeOfSingle newNode) {
    // 拿到尾指针
    NodeOfSingle temp = tail;
    // 此时temp已经指向最后一个节点，把新节点的地址存入temp的next中完成添加节点操作
    temp.next = newNode;
    // 尾指针后移，确保尾指针指向的是最后一个节点
    tail = temp.next;
}
```

添加（根据 id 顺序）

1. 通过辅助变量 temp 找到新节点的插入位置，如果链表有当前编号则插入失败
2. 新的节点的 next 指向 temp.next，即 newNode.next=temp.next
3. 将 temp.next 指向新节点，即 temp.next=newNode

```java
public void addByOrder(NodeOfSingle newNode) {
    //获得指针的头，通过遍历找到插入的位置，即插入节点的前一个节点
    NodeOfSingle temp = head;
    //添加编号的是否存在，默认false
    boolean flag = false;
    while (true) {
        //判断是否到到链表最后也可以用temp==tail判断
        if (temp.next == null) {
            //如果到链表的最后，代表新节点的id最大所以尾结点指向新节点
            tail = newNode;
            break;
        }
        //找到位置，就在temp的后面插入节点
        if (temp.next.id > newNode.id) {
            break;
        } else if (temp.next.id == newNode.id) {
            //添加的节点已经存在
            flag = true;
            break;
        }
        //后移
        temp = temp.next;
    }
    //判断flag的值,不能添加说明编号存在
    if (flag) {
        System.out.printf("%d号节点已经存在,不能加入! \n", newNode.id);
    } else {
        /*
            插入链表中，把temp后面的节点赋值给新节点。
            假设链表中有1号节点和2号节点、4号节点、5号节点，需要插入的节点为3号节点
            插入流程：
            1. 3号节点的next存入4号节点的地址
            2. 2号节点的next存入3号节点的地址
            完成插入操作。
            注意：流程不能颠倒，否则4号节点地址丢失,后半段链表数据全部遗失
         */
        newNode.next = temp.next;
        temp.next = newNode;
    }
}
```

```java
/**
 * 插入新节点到指点节点的后面
 *
 * @param newNode 新节点
 * @param id      要插入的位置
 */
public void insert(NodeOfSingle newNode, int id) {
    //判断链表是否为空
    if (head.next == null) {
        System.out.println("链表为空");
        return;
    }
    //找到插入的节点，根据id找
    NodeOfSingle temp = head.next;
    //表示是否找到该节点
    boolean flag = false;
    while (temp != null) {
        //找到后
        if (temp.id == id) {
            flag = true;
            break;
        }
        temp = temp.next;
    }
    //根据flag是否找到要修改的节点
    if (flag) {
        tail = tail.id == id ? newNode : tail;
        //找到后把指定节点的nex赋值给新节点，然后指定节点指向新节点
        newNode.next = temp.next;
        temp.next = newNode;
    } else {
        System.out.printf("%d号节点不存在,不能插入! \n", id);
    }
}
```

插入（表头）

1. 新节点的 next 指向头指针的下一个节点
2. 头指针指向新节点
3. 判断是否是空链表插入第一个元素时，尾指针后移，如果不是则不移动

```java
/**
 * 插入新节点到链表头
 * @param newNode 新节点
 */
public void insertTop(NodeOfSingle newNode) {
    //获得链表
    NodeOfSingle temp = head;
    newNode.next = temp.next;
    temp.next = newNode;
    //如果是空链表插入第一个元素时，尾指针后移，如果不是则不移动
    tail = newNode.next==null?newNode:tail;
}
```

查找

1. 遍历对比 id 找到目标节点

```java
/**
 * 查找指定id的节点并打印数据
 * @param id 要查找的节点id
 */
public void select(int id) {
    NodeOfSingle temp = head;
    //标识是否找到待删除节点
    boolean flag = false;
    //遍历链表
    while (temp.next != null) {
        //找到temp下一个节点是否是目标节点
        if (temp.id == id) {
            flag = true;
            break;
        }
        temp = temp.next;
    }
    //找到
    if (flag) {
        //打印
        System.out.println(temp.name);
    } else {
        System.out.printf("%d号节点不存在! \n", id);
    }
}
```

有效个数.

1. 遍历链表
2. 定义一个全局遍历 length，增加节点，插入节点+1，删除节点-1

```
/**
 * 获得单向链表的节点个数
 *  实现的两种方法：
 *       1.遍历链表
 *       2.定义一个全局遍历length，增加节点，插入节点+1，删除节点-1
 * @param head 链表头节点
 * @return 返回有效节点的个数
 */
public int getLength(NodeOfSingle head){
    //判断链表是否为空
    if(head.next==null){
        return 0;
    }
    int length=0;
    NodeOfSingle temp =head.next;
    while (temp!=null){
        length++;
        temp=temp.next;
    }
    return length;
}
```

反转链表

1. 定义一个节点 reverseHead
2. 从头到尾遍历链表，每遍历一个节点，就将其取出，放到新链表的最前端
3. 原来的 head.next=reverseHead.next

```java
/**
 * 反转链表
 */
public static void reverseList(NodeOfSingle head) {
    //如果当前链表为空，或者只有一个节点，无需反转，直接返回
    if (head.next == null || head.next.next == null) {
        return;
    }
    //辅助节点,遍历链表
    NodeOfSingle temp = head.next;
    //指向当前节点的下一个节点
    NodeOfSingle next = null;
    //反转链表
    NodeOfSingle reverseHead = new NodeOfSingle( id: 0,  name: "");
    //遍历临时的链表，每遍历一个节点，就将其取出，并放到原来的Head的最前端
    while (temp != null) {
        //暂时保存下一个节点，否则链表丢失
        next = temp.next;
        //将reverseHead的下一个赋值给temp。逻辑参考insertTop方法
        temp.next = reverseHead.next;
        //将temp链接到新的链表上
        reverseHead.next =temp;
        //后移
        temp = next;
    }
    //完成后赋值回head
    head.next = reverseHead.next;
}
```

逆序打印

1. 方式 1：对各个节点进行压栈，利用栈的先进后出，实现逆序打印的效果
   a) 创建一个栈，将各个节点压入栈中
   b) 将链表节点压栈
   c) 弹栈
2. 方式 2：递归
   a) 创建临时节点
   b) 设置递归终止条件 temp == null
   c) 调用方法体本身

```java
/**
 * 逆序打印
 * 方式1：对各个节点进行压栈，利用栈的先进后出，实现逆序打印的效果
 * 方式2：递归
 */
public static void reverse1Print(NodeOfSingle head) {
    //判断是否为空
    if (head.next == null) {
        return;
    }
    //创建一个栈，将各个节点压入栈中
    Stack<NodeOfSingle> stack = new Stack<>();
    NodeOfSingle temp = head.next;
    //将链表节点压栈
    while (temp != null) {
        stack.push(temp);
        //后移
        temp = temp.next;
    }
    //弹栈
    while (stack.size() > 0) {
        //先进后出
        System.out.println(stack.pop());
    }
}
/**
 * 递归逆序打印
 * @param head 头结点
 */
public static void reverse2Print(NodeOfSingle head) {
    NodeOfSingle temp = head;
    //判断是否为空
    if (temp == null) {
        return;
    }
    reverse2Print(temp.next);
    if (temp.id != 0) {
        System.out.println(temp);
    }
}
```

查找倒数第 n 个节点

1. 接受 head 节点，以及 n
2. 获得链表有效个数 length
3. 从链表遍历到（length-n）个，得到倒数第 n 个节点

```java
/**
 * 查找倒数第n个节点
 * 1.接受head节点，以及n
 * 2.获得链表有效个数length
 * 3.从链表遍历到（length-n）个，得到倒数第n个节点
 */
public static NodeOfSingle findLastUnknownNode(NodeOfSingle head, int n) {
    //判断链表是否为空
    if (head.next == null) {
        return null;
    }
    //第一次遍历的到链表长度（节点个数）
    int length = getLength(head);
    //第二次遍历，获得倒数第N个
    if (n <= 0 || n > length) {
        return null;
    }
    //定义赋值变量
    NodeOfSingle temp=head.next;
    for (int i =0;i<length-n;i++){
        temp = temp.next;
    }
    return temp;
}
```

修改内容

1. 根据 newNode 的 id 找到要修改的节点，如果没有则警告
2. 把新节点的内容赋值到要修改的节点

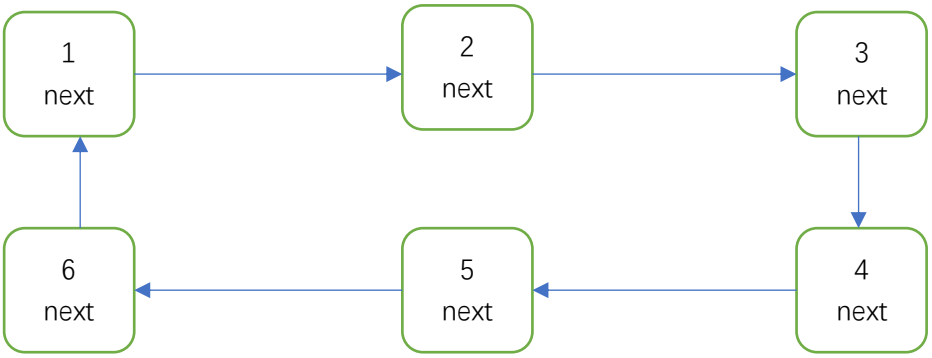
```java
/**
 * 修改节点信息，根据编号来修改即id不能改
 * 1.根据newNode的id来修改节点
 *
 * @param newNode 新节点
 */
public void update(NodeOfSingle newNode) {
    //判断链表是否为空
    if (head.next == null) {
        System.out.println("链表为空");
        return;
    }
    //找到需要修改的节点，根据id
    NodeOfSingle temp = head.next;
    //表示是否找到该节点
    boolean flag = false;
    while (temp != null) {
        //找到后
        if (temp.id == newNode.id) {
            flag = true;
            break;
        }
        temp = temp.next;
    }
    //根据flag是否找到要修改的节点
    if (flag) {
        //找到后修改内容
        temp.name = newNode.name;
    } else {
        System.out.printf("%d号节点不存在,不能修改! \n", newNode.id);
    }
}
```

删除

1.找到需要删除的节点的前一个节点，辅助变量 temo 指向删除的节点的前一个节点
2. temp 的 next 指向 temp 的下下个节点的地址，即 temp.next=temp.next.next
3.被删除的节点，将不会有其他引用指向，将会被 GC 回收

```java
/**
 * 删除节点
 * 1.temp获得head，遍历找到待删除节点的前一个节点
 * 2.比较时 temp.next.id和待删除节点的id比较
 * @param id 删除节点的id
 */
public void delete(int id) {
    NodeOfSingle temp = head;
    //标识是否找到待删除节点
    boolean flag = false;
    //遍历链表
    while (temp.next != null) {
        //找到temp下一个节点是否是目标节点
        if (temp.next.id == id) {
            //找到待删除节点的前一个节点
            flag = true;
            break;
        }
        temp = temp.next;
    }
    //找到
    if (flag) {
        //当删除的是最后一个节点的时候将tail指向temp
        tail = tail.id == id ? temp : tail;
        //可以删除
        temp.next = temp.next.next;
        //建议（不一定回收）jvm回收垃圾
        System.gc();
    } else {
        System.out.printf("%d号节点不存在，无法删除！\n", id);
    }
}
```

单向循环链表



应用

约瑟夫问题是个有名的问题：N 个人围成一圈，从第一个开始报数，第 M 个将出列，最后剩下一个，其余人都将出列。例如 N=6，M=5，被出列的顺序是：5，4，6，2，3，1

思路

1. 创建第一个节点，让 first 指向该节点，并形成环状
2. 每创建一个新节点，就把该节点加入到已有的环形链表

创建

1. 创建一个辅助指针 temp，创建 for 循环次数为节点个数
2. 根据循环次数 i 创建一个节点 newNode

判断循环次数 i 是否为 1

3. 次数为 1 时把 first 指向新节点 newNode,然后 first 的 next 指向自己，temp 指向 first
4. 次数为非 1 时把 temp 的 next 指向新节点 newNode，新节点 newNode 的 next 指向 first 形成一个环，temp 指向新节点即向前移动一位

```java
/**
 * 创建环形链表
 * @param numbers 节点个数
 */
public void addNode(int numbers) {
    //数据校验
    if (numbers < 1) {
        System.out.println("id的值不正确");
        return;
    }
    //辅助指针
    NodeOfSingleRing temp = null;
    for (int i = 1; i <= numbers; i++) {
        //根据编号创建节点
        NodeOfSingleRing newNode = new NodeOfSingleRing(i);
        if (i == 1) {
            first = newNode;
            //构建环
            first.setNext(first);
            temp=first;
        }else{
            temp.setNext(newNode);
            newNode.setNext(first);
            temp=newNode;
        }
    }
}
```

遍历

1. 让一个辅助指针（变量）temp，指向 first 节点
2. 然后通过一个 while 循环遍历

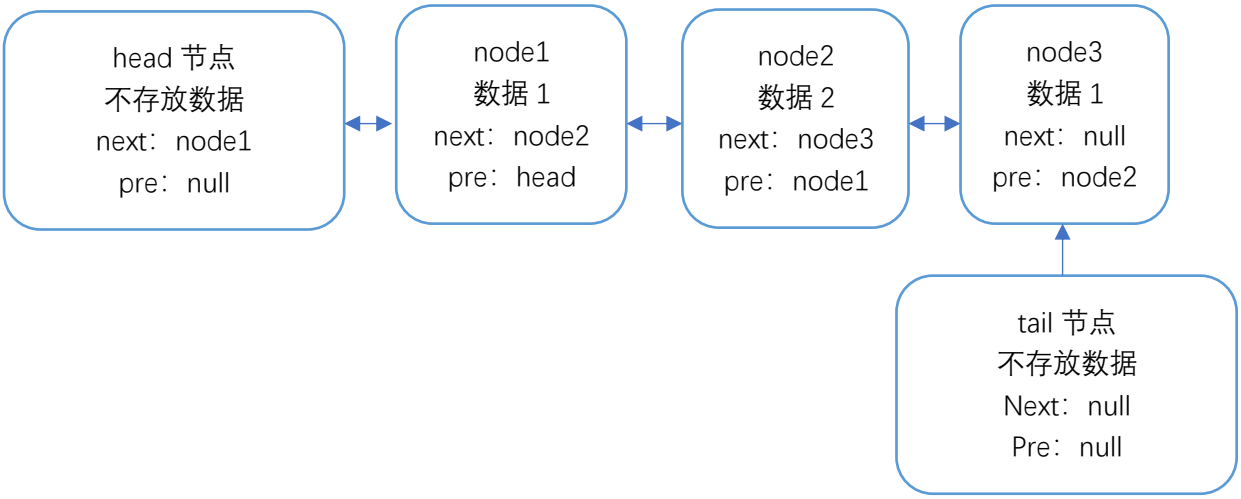
```java
/**
 * 遍历环形链表
 */
public void show() {
    if (first == null) {
        System.out.println("链表为空");
        return;
    }
    //first不能动，借助辅助指针
    NodeOfSingleRing temp = first;
    while (true) {
        System.out.printf("节点编号：%d\n", temp.getId());
        //遍历完成
        if (temp.getNext() == first) {
            break;
        }
        //节点后移
        temp = temp.getNext();
    }
}
```

出列

1. 创建一个辅助指针 helper，已经指向环形链表的最后节点。
2. 出列前 first 和 helper 移动 k-1 次
3. 当节点出列时，让 first 和 helper 指针同时移动 m-1 次
4. first 指向的节点出列
   first=first.getNext();
   helper.setNext(first);
5. 此时 helper 和 first 直接的节点完成出列

```java
/**
 * 根据输入，计算节点出列顺序
 * @param startNo  从第几个节点开始
 * @param countNum 计数
 * @param numbers  最初有多少节点
 */
public void countNodeSequence(int startNo, int countNum, int numbers) {
    //数据校验
    if (first == null || startNo < 1 || startNo > numbers) {
        System.out.println("参数有误，重新输入");
        return;
    }
    //1.创建辅助指针helper，获得循环链表最后节点
    NodeOfSingleRing helper = tail;
    //2.出列前，first和helper移动k-1次
    for (int j = 0; j < startNo - 1; j++) {
        first = first.getNext();
        helper = helper.getNext();
    }
    //只有一个节点时结束循环
    while (helper != first) {
        //3.让first和helper同时移动countNum-1
        for (int j = 0; j < countNum - 1; j++) {
            first = first.getNext();
            helper = helper.getNext();
        }
        //first指向的节点为出列节点
        System.out.printf("节点%d出列\n", first.getId());
        //4.first指向的节点出列
        first = first.getNext();
        helper.setNext(first);
    }
    System.out.printf("最后留着圈中的节点为%d\n", first.getId());
}
```

双向链表



思路

与单项链表相似，但是多了个后驱节点（指向上一个节点）

使用场景

如：　对线性表的长度或者规模难以估计，频繁做查找、插入、删除操作

双向链表结构

节点：data：数据区
　　next：指向下一个节点
　　pre：指向上一个节点

查找

　　与单链表相似，但是可以向前也可以向后查找

添加(链表头)

1. 获得头节点
2. 新节点的 pre 指向头节点 newNode.pre=temp;
3. 新节点指向 head 的下一个节点 newNode.next = temp.next;
4. 头节点的的 next 指向新节点 temp.next = newNode;

当新节点的下一个节点是否存在

5. 存在，新节点的下一个节点的 pre 指向新节点
6. 不存在，代表新节点为最后一个节点，尾指针指向新节点

```
/**
 * 插入新节点到链表头
 * 1.    获得头节点
 * 2.    新节点的pre指向头节点 newNode.pre=temp;
 * 3.    新节点指向head的下一个节点 newNode.next = temp.next;
 * 4.    头节点的的next指向新节点 temp.next = newNode;
 * 当新节点的下一个节点是否存在
 * 5.    存在，新节点的下一个节点的pre指向新节点
 * 6.    不存在，代表新节点为最后一个节点，尾指针指向新节点
 *
 * @param newNode 新节点
 */
public void insertTop(NodeOfDouble newNode) {
    // 获得链表
    NodeOfDouble temp = head;
    // 新节点的pre指向头节点
    newNode.pre = temp;
    // 新节点指向head的下一个节点
    newNode.next = temp.next;
    // 头节点的的next指向新节点
    temp.next = newNode;
    //当新节点的下一个节点是否存在
    if (newNode.next != null) {
        // 存在，新节点的下一个节点的pre指向新节点
        newNode.next.pre = newNode;
    } else {
        //不存在，代表新节点为最后一个节点，尾指针指向新节点
        tail = newNode;
    }
}
```

添加

1. 获得链表的尾指针 tail temp=tail
2. 将 temp 的 next 指向新节点 temp.next=newNode;
3. 新节点的 pre 指向 temp newNode.pre=temp;
4. 尾指针指向新节点 tail=newNode;

```
/**
 * 添加节点到双向链表的最后
 * 1.找到当前链表的最后节点
 * 2.将最后这个节点的next指向新的节点
 * 3.新节点的pre指向temp
 *
 * @param newNode 新节点
 */
public void add(NodeOfDouble newNode) {
    /* 拿到尾指针 */
    NodeOfDouble temp = tail;
    // 此时temp已经指向最后一个节点，把新节点的地址存入temp的next中完成添加节点操作
    temp.next = newNode;
    // 新节点的pre指向temp节点
    newNode.pre = temp;
    // 尾指针后移，确保尾指针指向的是最后一个节点
    tail = newNode;
}
```

插入（根据 id 大小排序）

1. 找到 id 比新节点 id 大的节点的前一个节点 temp，如果没有则新节点 id 最大
2. 新节点的 pre 指向 temp 节点 newNode.pre=temp;
3. 新节点指向 head 的下一个节点 newNode.next = temp.next;
4. temp 节点的 next 指向新节点 temp.next = newNode;

当新节点的下一个节点是否存在

5. 存在，新节点的下一个节点的 pre 指向新节点
6. 不存在，代表新节点为最后一个节点，尾指针指向新节点

```
    * 添加节点到单向链表（根据id顺序）
    * 1.    找到id比新节点id大的节点的前一个节点temp，如果没有则新节点id最大
    * 2.    新节点的pre指向temp节点 newNode.pre=temp;
    * 3.    新节点指向head的下一个节点 newNode.next = temp.next;
    * 4.    temp节点的next指向新节点 temp.next = newNode;
    * 当新节点的下一个节点是否存在
    * 5.    存在，新节点的下一个节点的pre指向新节点
    * 6.    不存在，代表新节点为最后一个节点，尾指针指向新节点
    *
    * @param newNode 新节点
    */
public void addByOrder(NodeOfDouble newNode) {
    //获得指针的头，通过遍历找到插入的位置，即插入节点的前一个节点
    NodeOfDouble temp = head;
    //添加编号的是否存在，默认false
    boolean flag = false;
    //最后节点的标识符，默认false
    boolean tailFlag = false;
    while (true) {
        //判断是否到到链表最后也可以用temp==tail判断
        if (temp.next == null) {
            //如果到链表的最后，代表新节点的id最大所以尾结点指向新节点
            tail = newNode;
            tailFlag = true;
            break;
        }
        //找到位置，就在temp的后面插入节点
        if (temp.next.id > newNode.id) {
            break;
        } else if (temp.next.id == newNode.id) {
            //添加的节点已经存在
            flag = true;
            break;
        }
        //后移
        temp = temp.next;
    }
    //判断flag的值,不能添加说明编号存在
    if (flag) {
        System.out.printf("%d号节点已经存在,不能加入! \n", newNode.id);
    } else {
        // 新节点的pre指向头节点
        newNode.pre = temp;
        // 新节点指向head的下一个节点
        newNode.next = temp.next;
        // 头节点的的next指向新节点
        temp.next = newNode;
        //当新节点的下一个节点是否存在
        if (!tailFlag) {
            // 存在，新节点的下一个节点的pre指向新节点
            newNode.next.pre = newNode;
        }
    }
}
```

修改

和单向链表一样

```
/**
 * 修改节点信息，根据编号来修改即id不能改，和单向链表一样
 * 1.根据newNode的id来修改节点
 *
 * @param newNode 新节点
 */
public void update(NodeOfDouble newNode) {
    // 判断链表是否为空
    if (head.next == null) {
        System.out.println("链表为空");
        return;
    }
    // 找到需要修改的节点，根据id
    NodeOfDouble temp = head.next;
    // 表示是否找到该节点
    boolean flag = false;
    while (temp != null) {
        // 找到后
        if (temp.id == newNode.id) {
            flag = true;
            break;
        }
        temp = temp.next;
    }
    // 根据flag是否找到要修改的节点
    if (flag) {
        // 找到后修改内容
        temp.name = newNode.name;
    } else {
        System.out.printf("%d号节点不存在,不能修改! \n", newNode.id);
    }
}
```

删除

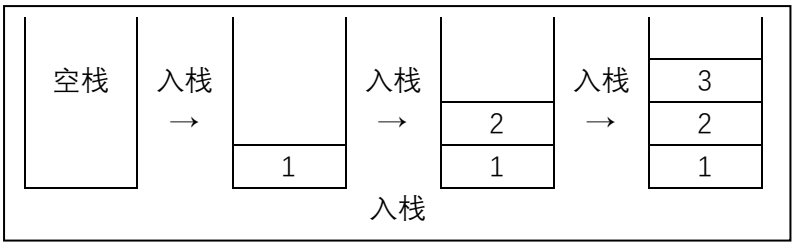
1. 遍历找到要查找的节点 temp
2. 如果不是，把 temp 的上一个节点的 next 指向 temp 的下一个节点，如果是最后一个则指向空
   temp.pre.next=temp.next;

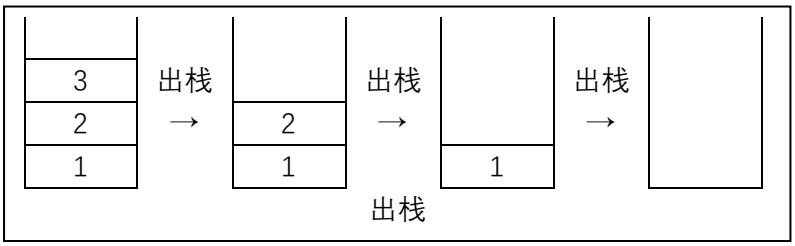
判断 temp 是否为最后一个节点

3. 如果是，tail 指向上一个节点 if(temp.next==null) tail=temp.pre;
4. 如果不是，把 temp 下一个节点的 pre 指向 temp 的上一个节点 temp.next.pre=temp.pre;
5. temp 的 pre 和 next 指向 null 完成后 temp 无指向为废弃节点 gc 会回收 temp.next = null;temp.pre = null;

```java
/**
 * 删除节点（双向链表可自我删除，无需辅助节点）
 * 1.   遍历找到要查找的节点temp
 * 2.   如果不是，把temp的上一个节点的next指向temp的下一个节点，如果是最后一个则指向空
 * 判断temp是否为最后一个节点
 * 3.   如果是，tail指向上一个节点 if(temp.next==null)
 * 4.   如果不是，把temp下一个节点的pre指向temp的上一个节点
 * 5.   temp的pre和next指向null 完成后temp无指向为废弃节点gc会回收
 *
 * @param id 删除节点的id
 */
public void delete(int id) {
    // 判断当前链表是否为空
    if (head.next == null) {
        System.out.println("链表为空，无法杀出");
        return;
    }
    NodeOfDouble temp = head.next;
    // 标识是否找到待删除节点
    boolean flag = false;
    // 遍历链表
    while (temp != null) {
        // 找到temp下一个节点是否是目标节点
        if (temp.id == id) {
            // 找到待删除节点的前一个节点
            flag = true;
            break;
        }
        temp = temp.next;
    }
    // 1.找到要查找的节点temp
    if (flag) {
        // 2.把temp的上一个节点的next指向temp的下一个节点，如果是最后一个则指向空
        temp.pre.next = temp.next;
        // 判断temp是否为最后一个节点，如果是tail指向上一个节点
        if (tail.id == id) {
            // 3.将尾结点指向temp的上一节点
            tail = temp.pre;
        } else {
            // 4.不是尾结点把temp下一个节点的pre指向temp的上一个节点
            temp.next.pre = temp.pre;
        }
        // 5.temp的pre和next指向null 完成后temp无指向为废弃节点gc会回收
        temp.next = null;
        temp.pre = null;
        // 建议（不一定回收）jvm回收垃圾
        System.gc();
    } else {
        System.out.printf("%d号节点不存在，无法删除！\n", id);
    }
}
```

栈





思路

栈是一个**先入后出**的有序列表
栈是限制线性表中元素的插入和删除只能在线性表的同一端进行的一种特殊线性表。允许插入和删除的一端，**变化的另一端，称为栈顶（top）**，另一端为固定的一端称为栈底（bottom）
最先放入栈中的元素在栈底，最后放入的元素在栈顶，删除元素则最先放入的最后删除，最后放入的最先删除

结构

使用数组模拟栈
定义一个 top 来表示栈顶，初始化为-1

```
/**
 * 栈的最大值
 */
private int maxSize;
/**
 * 数组模拟的栈，数据就存放在这
 */
private int[] stack;
/**
 * 栈顶，初始化为-1
 */
private int top = -1;
```

入栈

有数据加入栈时，top++;stack[top]=data;

```
/**
 * 判断栈是否已满
 *
 * @return 满true 不满false
 */
public boolean sizeFull() {
    return top == maxSize - 1;
}
```

```
/**
 * 入栈-push
 *
 * @param value 要入栈的数据
 */
public void push(int value) {
    //判断栈是否满
    if (sizeFull()) {
        System.out.println("栈满");
        return;
    }
    stack[++top] = value;
}
```

出栈

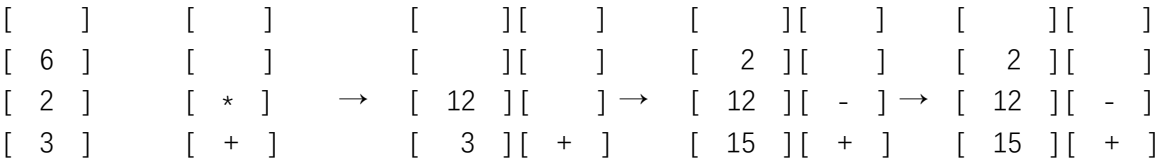
int value = stack[top];top--;return value;

```
/**
 * 判断栈是否为空
 *
 * @return 空true 不空false
 */
public boolean sizeEmpty() {
    return top == -1;
}
```

```java
/**
 * 出栈-pop
 *
 * @return 要出栈的数据
 */
public int pop() {
    //判断栈是否为空
    if (sizeEmpty()) {
        //抛出异常
        throw new RuntimeException("栈空");
    }
    return stack[top--];
}
```

栈的使用（计算中缀表达式）

计算 3+2*6-2=?



数栈 符号栈

numStack operStack

存放数 存放运算符

思路

1. 通过一个 index 值（索引）遍历表达式
2. 如果发现是一个数字，入数栈
3. 如果是符号分以下情况

   如果符号栈为空直接入栈

   如果有操作符，进行比较，如果优先级小于或者等于栈中的操作符，从数栈中 pop 出两个数，从符号栈中 pop 出一个符号，进行运算，将等到的结果入数栈，然后将当前操作符入符号栈

   如果优先级大于栈中的操作符就直接入符号栈
4. 当表达式扫描完毕，就顺序从数栈和符号栈中 pop 出相应的数和符号，并进行运算
5. 最后在数栈中只有一个数字，就是表达式的结果

前缀（波兰式），中缀，后缀（逆波兰式）表达式

前缀表达式

波兰式，从右到左扫描，遇到数字压入堆栈，遇到运算符时弹出栈顶的两个数做相应的运算，结果入栈，重复上述过程直到表达式最左端，最后的值为表达式的结果
例：-x+3456

中缀表达式

人类常用的运算表达式
例：(3+4)x5-6

后缀表达式

逆波兰表达式，与前缀表达式相似，运算符在操作数后
例：34+5x6-

例子

中缀表达式：(3+4)x5-6　　前缀表达式：-x+3456 后缀表达式：34+5x6-

中缀转后缀

1. 初始化一个栈和一个广义表：运算符栈 symbol 和存储中间结果的线性表 medium
2. 从左到右扫描中缀表达式
3. 遇到操作数时，将其插入 medium
4. 遇到运算符时，比较其余 symbol 栈运算符的优先级
    a) 如果 symbol 为空，或栈顶运算符为左括号，则直接将此运算符入栈
    b) 若优先级比栈顶运算符的高，也将运算符压入 symbol
    c) 否则，将 symbol 的栈顶的运算符弹出并插入到 medium 中，在此转到 4.a 与 symbol 中新的栈顶运算符相比较
5. 遇到括号时：
    a) 左括号：直接压入栈 symbol
    b) 右括号：依次弹出 symbol 栈顶的运算符，并插入 medium，知道遇到左括号为止，此时将这一对括号丢弃
6. 重复步骤 2 至 5，直到表达式的最右边
7. 将 symbol 中剩余的运算符一次弹出并插入 medium
8. 遍历 medium 即为后缀表达式

扫描到的元素	S2（栈底→栈顶）	S1（栈底→栈顶）	说明
1	1	Null	数字直接入栈
+	1	+	S1 为空，运算符直接入栈
(	1	+(	左括号，直接入栈
(	1	+((	同上
2	12	+((	数字
+	12	+((+	S1 栈顶为左括号，运算符直接入栈
3	123	+((+	数字
)	123+	+(	右括号，弹出运算符直至遇到左括号
*	123+	+(*	S1 栈顶为左括号，运算符直接入栈
4	123+4	+(*	数字
)	123+4*	+	右括号，弹出运算符直至遇到左括号
-	123+4*+	-	-与+优先级相同，因此弹出+，在压入-
5	123+4*+5	-	数字
到达最右端	123+4*+5-	空	S1 中剩余的运算符

```java
/**
 * 将中缀表达式转成对应的List
 *
 * @param s 表达式
 * @return 存放中缀的表达式的list
 */
public static List<String> toInfixExpressionList(String s) {
    // 定义一个List，存放中缀表达式对应的内容
    List<String> ls = new ArrayList<>();
    // 遍历s的指针
    int i = 0;
    // 对多位数的拼接
    StringBuilder str;
    // 遍历到一个字符就放进c
    char c;
    do {
        // 如果c是非数字，需要加入到ls asii:48=0,57=9
        if ((c = s.charAt(i)) < 48 || (c = s.charAt(i)) > 57) {
            ls.add("" + c);
            // i后移
            i++;
        } else {
            // 将str清空，考虑到多位数的问题
            str = new StringBuilder();
            while (i < s.length() && (c = s.charAt(i)) > 48 && (c = s.charAt(i)) < 57) {
                // 拼接
                str.append(c);
                i++;
            }
            ls.add(str.toString());
        }
    } while (i < s.length());
    return ls;
}
```

```java
/**
 * 中缀转后缀
 * @param ls 中缀list
 * @return 后缀list
 */
public static List<String> parseSuffixExpressionList(List<String> ls) {
    // 符号栈
    Stack<String> symbol = new Stack<>();
    // 中间结果
    List<String> medium = new ArrayList<>();
    String[] contrast = {"(", ")"};
    // 遍历ls
    for (String item : ls) {
        // 如果是一个数，就加入到medium
        if (item.matches( regex: "\\d+")) {
            medium.add(item);
        } else if (contrast[0].equals(item)) {
            // 如果是左括号
            symbol.push(item);
        } else if (contrast[1].equals(item)) {
            // 如果是右括号，则依次弹出symbol的运算符，并压入medium，直到遇到左括号为止，然后将这对括号丢弃
            while (!contrast[0].equals(symbol.peek())) {
                medium.add(symbol.pop());
            }
            // 将(弹出symbol，消除小括号
            symbol.pop();
        } else {
            // 当item的优先级小于等于symbol栈顶运算符，
            // 将symbol的栈顶运算符弹出并加入到medium中，再转到4.a与symbol中新的栈顶运算符进行比较
            while (symbol.size() != 0 && getValue(symbol.peek()) >= getValue(item)) {
                medium.add(symbol.pop());
            }
            //将item压入栈中
            symbol.push(item);
        }
    }
    //将s1中剩余的运算符依次弹出并加入s2
    while (symbol.size() != 0) {
        medium.add(symbol.pop());
    }
    //因为是存放到list中，按顺序输出就是对于的后缀表达式
    return medium;
}
```

```java
/**
 * 比较运算符优先级
 *
 * @param operation 运算符
 * @return 级别
 */
public static int getValue(String operation) {
    int result = 0;
    switch (operation) {
        case "+":
        case "-":
            result = 1;
            break;
        case "*":
        case "/":
            result = 2;
            break;
        default:
            if (!("(".equals(operation) || ")".equals(operation))) {
                System.out.println("不存在该运算符");
            }
            break;
    }
    return result;
}
```

逆波兰计算器

1.从左到右扫描，将 3 和 4 压入栈；
2.遇到+运算符，隐藏弹出 4 和 3（4 为栈顶元素，3 为次顶元素），计算 3+4 的值，得到 7，再将 7 入栈；
3.将 5 入栈；
4.接下来*运算符，因此弹出 5 和 7，极端出 7*5=35，将 35 入栈；
5.将 6 入栈；
6.最后-运算符，计算出 35-6 的值，得出最终结果 29

```java
/**
 * 将逆波兰表达式，依次将数据和运算符放入到ArrayList中
 *
 * @param suffixExpresion 后缀表达式
 * @return 拆分后的表达式
 */
public static List<String> getListString(String suffixExpresion) {
    // 分割suffixExpression
    String[] split = suffixExpresion.split( regex: " ");
    List<String> list = new ArrayList<>(Arrays.asList(split));
    return list;
}
```

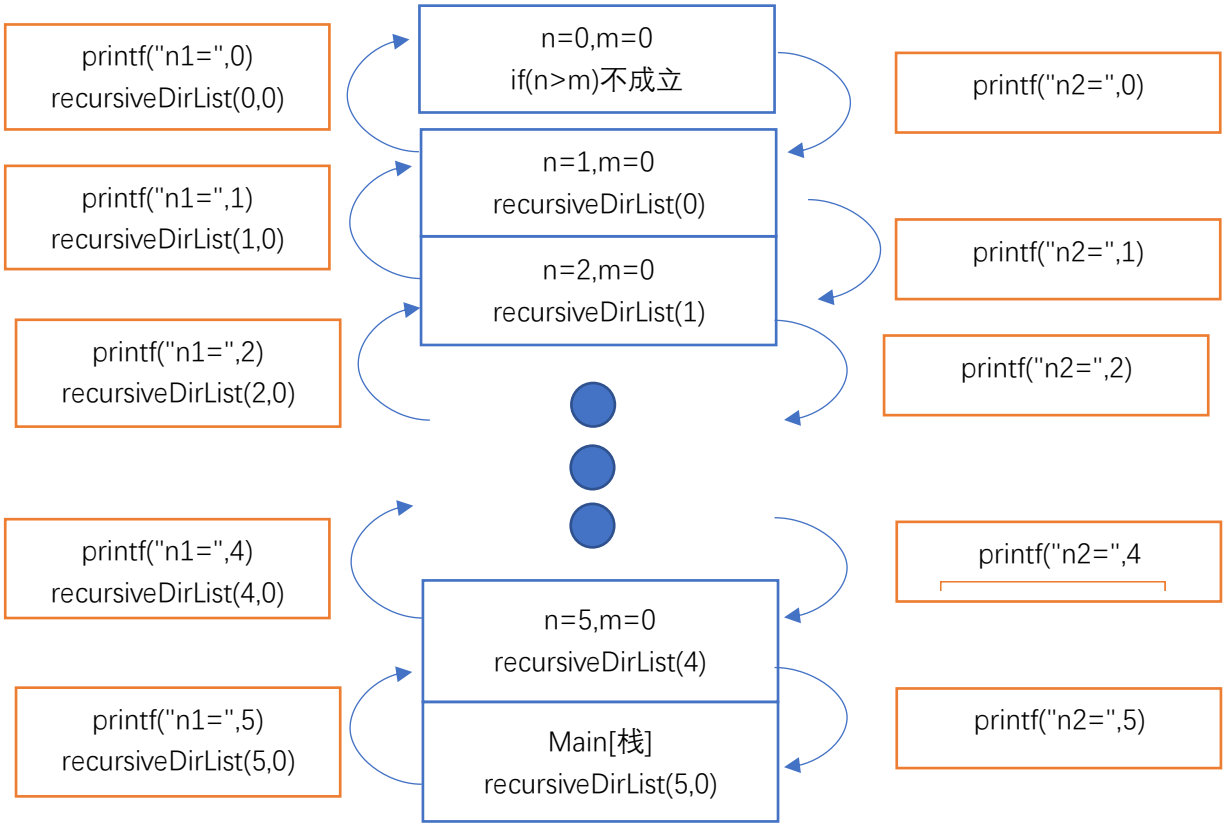
```java
/**
 * 完成对逆波兰表达式的运算
 * 1.从左到右扫描，将3和4压入栈；
 * 2.遇到+运算符，隐藏弹出4和3（4为栈顶元素，3为次顶元素），计算3+4的值，得到7，再将7入栈；
 * 3.将5入栈；
 * 4.接下来*运算符，因此弹出5和7，极端出7*5=35，将35入栈；
 * 5.将6入栈；
 * 6.最后-运算符，计算出35-6的值，得出最终结果29
 *
 * @param list 拆分后的表达式
 * @return 结果或者异常
 */
public static int calculate(List<String> list) {
    // 创建栈，只需要一个栈
    Stack<String> stack = new Stack<>();
    // 遍历list
    for (String item : list) {
        // 使用正则表达式取数,匹配多位数
        if (item.matches( regex: "\\d+")) {
            // 入栈
            stack.push(item);
        } else {
            // pop出两个数，并运算，再入栈
            int num2 = Integer.parseInt(stack.pop());
            int num1 = Integer.parseInt(stack.pop());
            int res = 0;
            switch (item) {
                case "+":
                    res = num1 + num2;
                    break;
                case "-":
                    res = num1 - num2;
                    break;
                case "*":
                    res = num1 * num2;
                    break;
                case "/":
                    res = num1 / num2;
                    break;
                default:
                    throw new RuntimeException("运算符有误");
            }
            // res入栈
            stack.push( item: res + "");
        }
    }
    // 最后留着stack中的数据是运算结果
    return Integer.parseInt(stack.pop());
}
```

算法

递归

思路

递归就是自己调用自己，每次传入不同的变量，直到完成结束条件后不断返回上一层
当程序执行到一个方法时，就会开辟一个独立的空间（栈）
每个空间的数据（局部变量）都是独立的
递归必须向着退出递归的条件逼近
执行完毕或者遇到 return 就返回，谁调用返回给谁



使用场景

数学问题：如 8 皇后问题，汉罗塔问题，阶乘问题，迷宫问题，球和篮子的问题
算法问题：快排，归并排序，二分查找，分治算法等
用栈解决的问题，递归代码简洁

打印

```java
/**
 * 打印问题
 *
 * @param n 操作数
 * @param m 出口数
 */
public static void recursiveDirList(int n, int m) {
    //结果n1=5  n1=4    n1=3    n1=2    n1=1    n1=0
    System.out.printf("n1=%d\t", n);
    //递归出口的条件
    if (n > m) {
        //本体调用
        recursiveDirList( n: n - 1, m);
    }
    //结果n2=0  n2=1    n2=2    n2=3    n2=4    n2=5
    System.out.printf("\tn2=%d", n);
}
```

阶乘

```java
/**
 * 阶乘问题
 * @param n 阶乘到多少
 * @return 阶乘答案
 */
public static int factorial(int n) {
    //递归结束条件
    if (n == 1) {
        return 1;
    } else {
        //不满足结束条件
        return factorial( n: n - 1) * n;
    }
}
```

迷宫问题

1	1	1	1	1	1	1
1	2	0	0	0	0	1
1	2	2	2	0	0	1
1	1	1	2	0	0	1
1	0	0	2	0	0	1
1	0	0	2	0	0	1
1	0	0	2	2	2	1
1	1	1	1	1	1	1

思路

1. 0 为没走过的路径，1 为墙，2 表示通路，3 表示该点已经走过但不通
2. 走迷宫策略为下->右->上->左，如果该点走不通，再回溯
   递归方法：
3. 将终点等于 2 设置为递归出口
4. 判断当前坐标是否为 0 即为没走过的路径，如果不是结束算法
5. 如果是，将当前坐标设置为 2，随后判断当前坐标的下方位置是否为 0，如果是进入下一层递归，如果不是返回 false，并判断右侧位置是否为 0，如果是进入下一次递归，如果不是返回 false，一同方法判断上分与左侧位置，直至终点为 2 或者所有可行走坐标变为 3 时完成算法

```java
/**
 * 迷宫回溯算法
 * @param maze    地图
 * @param x0rigin 起点x坐标
 * @param y0rigin 起点y坐标
 * @param xEnd    结束x坐标
 * @param yEnd    结束y坐标
 * @return 是否找到通路
 */
public static boolean readyGo(int[][] maze, int x0rigin, int y0rigin, int xEnd, int yEnd) {
    //递归出口，当终点被设置为2时代表找到通路
    if (maze[xEnd][yEnd] == 2) {
        return true;
    } else {
        if (maze[x0rigin][y0rigin] == 0) {
            //假设改坐标可以走通
            maze[x0rigin][y0rigin] = 2;
            //执行策略 下->右->上->左
            if (readyGo(maze, xOrigin: x0rigin + 1, y0rigin, xEnd, yEnd)) {
                //向下走
                return true;
            } else if (readyGo(maze, x0rigin, yOrigin: y0rigin + 1, xEnd, yEnd)) {
                //向右走
                return true;
            } else if (readyGo(maze, xOrigin: x0rigin - 1, y0rigin, xEnd, yEnd)) {
                //向上走
                return true;
            } else if (readyGo(maze, x0rigin, yOrigin: y0rigin - 1, xEnd, yEnd)) {
                //向左走
                return true;
            } else {
                //这个坐标走不通
                maze[x0rigin][y0rigin] = 3;
                return false;
            }
        } else {
            //如果maze[i][j]!=0,可能是1,2,3
            return false;
        }
    }
}
```