# 第 3 章

# 栈 和 队 列

在第 2 章当中通过对数组和链表相关知识的学习可以发现：数组和链表属于数据结构当中最基本的物理结构，在具体业务场景中使用时，往往需要进一步地进行封装，简化其操作方式，提升其操作效率。在对数组和链表进行高级封装的过程中，如果从元素的操作顺序上加以约束，就能够得到两种逻辑结构：栈和队列。

在对栈和队列这两种逻辑结构进行具体实现的过程中，其底层既可以使用数组进行实现，也可以使用链表进行实现。例如在 Java 语言的原生 API 中，对于栈结构的实现有 Stack 类型，其内部与 ArrayList 相似，都是使用数组进行封装的；对于队列结构的实现有 LinkedList 类型，其内部使用双链表结构进行封装，但是从 JDK 1.6 版本开始，LinkedList 同样实现了栈结构的相关操作，可以作为栈结构的实现进行使用，所以从这一点看来，对于栈和队列的学习重点应该放在两种结构的元素操作逻辑及相应的使用场景上。

本章内容的思维导图如图 3-1 所示。

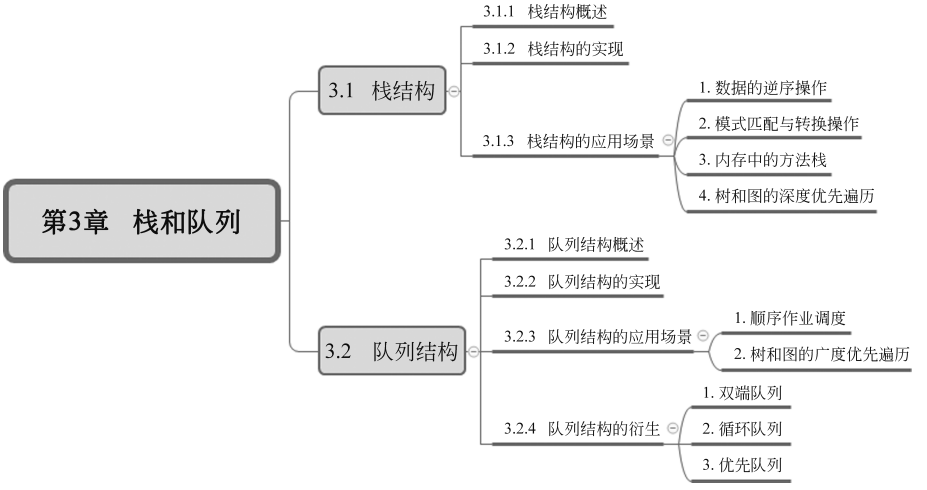


图 3-1　栈和队列章节思维导图

# 3.1　栈结构

## 3.1.1　栈结构概述

栈结构的逻辑特点是元素后进先出(Last In First Out,LIFO)。可以将栈结构理解为一个一端开口而另一端封闭的容器,在向容器中添加元素时,后加入的元素会“压住”先加入的元素,只有将后加入的元素逐个退出栈结构,才能将先加入其中的元素取出,而元素加入、退出栈结构的操作都是从同一端进行的。

在栈结构中,将元素加入栈结构的操作称为入栈(push),将元素从栈结构中退出的操作称为出栈(pop)。最先入栈元素所在的位置称为栈底(bottom),元素入栈、出栈所操作的一端称为栈顶(top)。栈底元素一定是栈结构中最先入栈的元素,而位于栈顶的元素一定是最后入栈的元素。在执行出栈操作时,栈顶元素最先出栈,栈底元素最后出栈。在 Java 中栈结构的实现类 Stack 类型下,还提供了看栈顶(peek)操作,即仅返回栈顶元素取值,但不将栈顶元素出栈操作的方法实现。

需要注意的是,当栈结构为空时对其继续执行出栈操作或看栈顶操作被认为是不正确的,此时通常以抛出异常的方式结束操作。

栈结构示意图如图 3-2 所示。

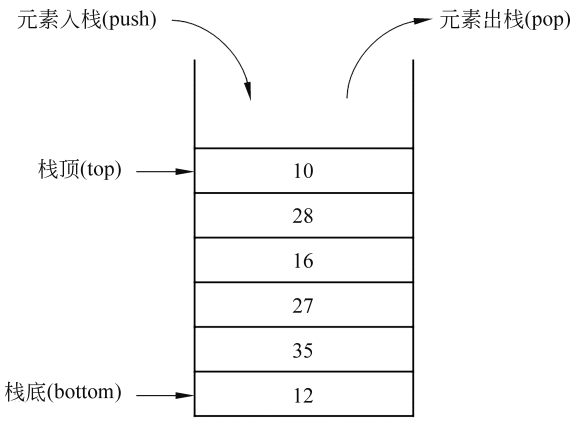


图 3-2　栈结构示意图

用一个形象恰当的例子解释栈结构:影视剧中所展示的枪支的弹夹与栈结构高度相似。在枪支的弹夹中,第一颗被压入的子弹将会被最后一个打出去,而最后一颗被压入的子弹将会被第 1 个打出去,所以很多编程人员也将元素入栈称为“元素压栈”,而将元素出栈也称为“元素弹栈”。

## 3.1.2　栈结构的实现

在通过数组实现栈结构时,通常使用一个 int top 变量保存数组中栈顶元素的下标取

值，初始化状态下 top＝－1。元素入栈的操作通过数组元素尾追加的方式实现，即以数组中有效元素的尾端为栈顶方向，此时 top＋＋，而元素出栈操作则相当于删除数组中 top 下标指向的元素，元素删除后 top－－。需要注意的是，在通过数组实现栈结构时，同样需要考虑因为元素入栈导致的数组扩容的相关需求。

通过数组实现的栈结构及相关操作如图 3-3 所示。

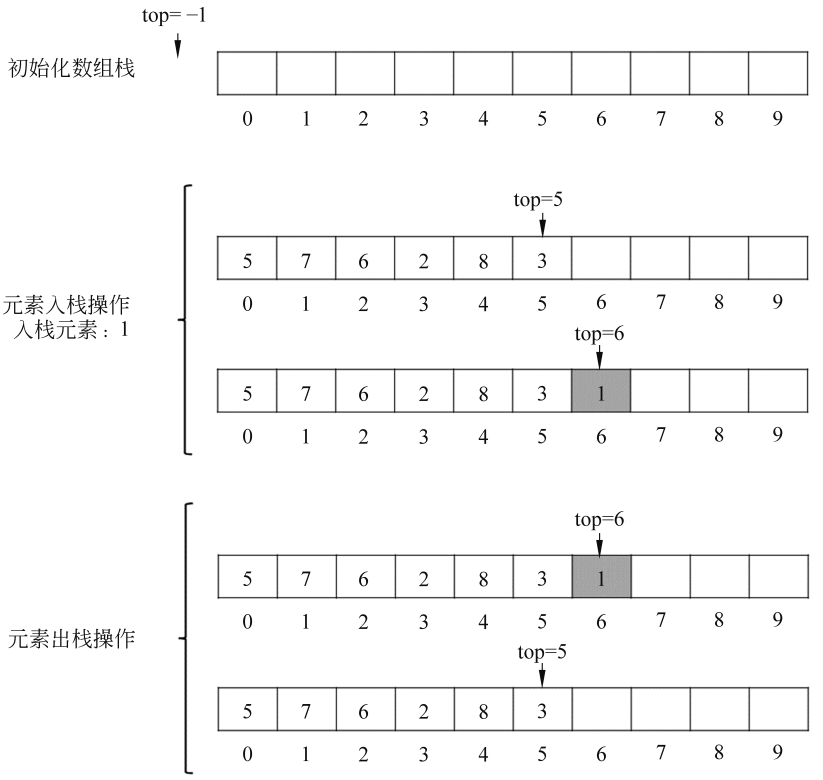


图 3-3　通过数组实现的栈结构及相关操作

在通过链表实现栈结构的时候，元素入栈通常采用头节点插入的方式进行实现，即以链表头部为栈顶方向。相对地，元素出栈操作则是使用头节点删除的方式进行实现。

通过链表实现的栈结构及相关操作如图 3-4 所示。

### 3.1.3　栈结构的应用场景

对于栈结构来讲容易理解的是，如果将一组数据按顺序逐个进行入栈，然后逐个出栈，即可得到一个数据顺序与原始序列完全相反的新序列，而栈结构也因为这种特性在各种开发场景中广泛用于“逆序”操作。

栈结构“逆序”的特点，大体上可以分为两种使用方式：①直观的数据逆序操作。例如将数据序列{1,2,3,4,5}中的数据依次入栈后再依次出栈，即可得到元素顺序为{5,4,3,2,1}的新序列。这种操作还可以通过灵活地指定每个元素入栈、出栈的时机，实现数据序列整体

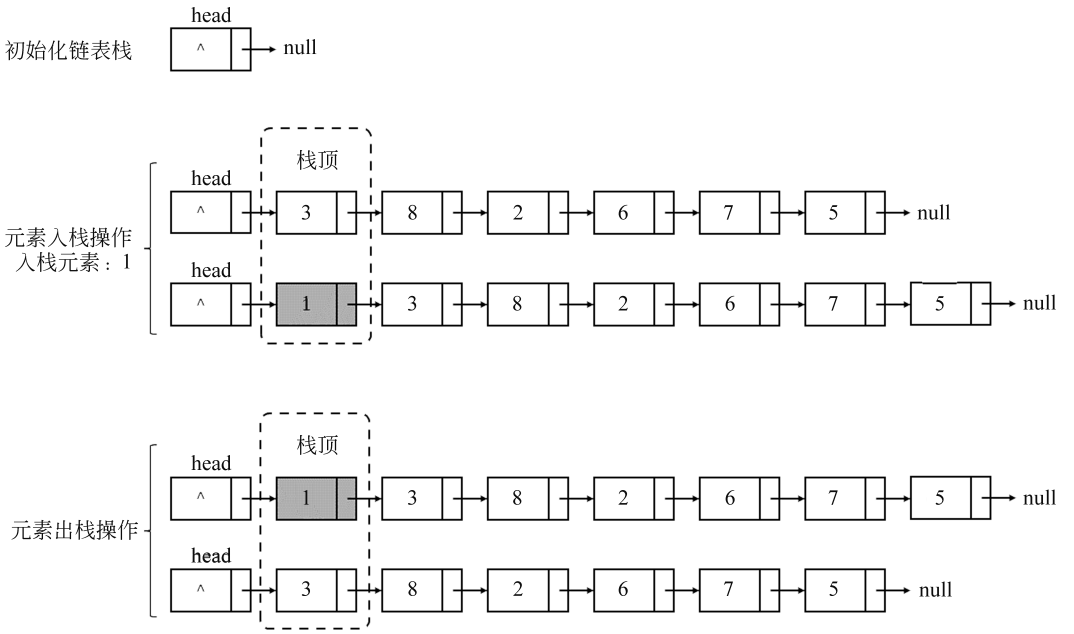


图 3-4　通过链表实现的栈结构及相关操作

或者部分的逆序操作；②实现操作的延后性。假设现有 A、B、C 共 3 个方法，它们之间相互调用，其中 A 方法内部调用 B 方法，B 方法内部调用 C 方法，并且程序整体以 A 方法的调用为起点。对于上述案例来讲，A 方法虽然最先被调用，B 方法次之，C 方法最后被调用，但是在内存中最先执行完毕的应该是 C 方法。因为只有 C 方法运行完成，并将运行结果返回 B 方法，B 方法才能继续向下运行，最终在运行结束时将结果返回给调用它的 A 方法，A 方法最终结束运行。如同在上述案例中这种“先调用、后结束”的操作场景，也可以通过栈结构进行实现。

本节给出了栈结构比较常见的几种具体应用场景，并且结合一些问题案例进行说明与实现。

**1. 数据的逆序操作**

对于使用栈结构实现最直观的数据逆序操作没有太大的难度，唯一需要注意的是，如果将原始序列中的数据一次性全部加入栈结构中，再一次性全部出栈，最终只会得到整体逆序的新序列，但是在某些应用场景下，原始序列中的元素可能通过部分入栈、部分出栈、再部分入栈、再部分出栈的不规则操作，得到部分逆序的新序列。下面通过两个案例来对上述情况进行演示。

**【例 3-1】** 存在一个空栈，现有 4 个元素，其入栈顺序为 a、b、c、d，则不可以得到的出栈顺序为(　　)。

A. a,b,c,d　　B. d,c,b,a　　C. c,a,b,d　　D. a,c,b,d

在例 3-1 中，给出了元素入栈的顺序，但是对元素出栈的顺序并未进行要求，也就是说

此时有可能出现部分元素入栈、部分元素出栈的不规则操作的情况，但是不论如何进行元素入栈、出栈操作都必须保证元素之间“后进先出”的顺序关系。下面对4个选项是否可能出现分别进行分析。

选项A：a入栈，a出栈，b入栈，b出栈，c入栈，c出栈，d入栈，d出栈，可行。

选项B：a入栈，b入栈，c入栈，d入栈，d出栈，c出栈，b出栈，a出栈，可行。

选项C：a入栈，b入栈，c入栈，c出栈，此时b压在a上，所以不可能直接出栈a，不可行。

选项D：a入栈，a出栈，b入栈，c入栈，c出栈，b出栈，d入栈，d出栈，可行。

**【例3-2】** 设有一栈结构S，元素s1、s2、s3、s4、s5、s6依次进栈，如果6个元素出栈的顺序是s2、s3、s4、s6、s5、s1，则栈的容量至少应该是（　　）。

A. 2　　B. 3　　C. 5　　D. 6

例3-2的分析思路与例3-1相似，元素的入栈顺序一定，为了得到目标的元素出栈顺序，就需要通过元素部分入栈、部分出栈的方式进行操作。在操作过程中，栈结构内同时存在元素数量的最大值，即为栈结构的最小容量。下面对分析过程进行说明。

为了得到例3-2中元素的出栈顺序，需按照如下顺序对入栈元素进行操作：入栈s1、入栈s2，此时栈内元素为{s1，s2}；出栈s2，栈内元素为{s1}；入栈s3，栈内元素为{s1，s3}；出栈s3，栈内元素为{s1}；入栈s4，栈内元素为{s1，s4}；出栈s4，栈内元素为{s1}；入栈s5、s6，栈内元素为{s1，s5，s6}；出栈s6，栈内元素为{s1，s5}；出栈s5，栈内元素为{s1}；出栈s1，栈空，操作完成。在上述过程中，栈内最多同时存在3个元素，所以栈结构最小容量为3。

**2. 模式匹配与转换操作**

模式匹配与转换操作可以认为是通过栈结构实现操作延后性的直观体现。

在一些应用场景当中，给出的某一部分数据可能无法直接进行运算操作。这一部分数据需要等到一个合适的操作指令的出现，才能逐个回退着向前进行计算。这样的运算方式在计算机的各种模式匹配与转换算法中经常会用到。下面通过两个具体案例对栈结构在模式匹配和转换操作中的应用进行说明。

**【例3-3】** 括号的匹配算法。在编译器对程序语法结构正确性进行检查的过程中，经常需要对语句中出现的括号是否匹配进行测试。现有一表达式，其中只包含(、)及运算数，例如(1+2)×3、3×((1+2)+(5+4))÷12等形式。表达式中，括号之间可以嵌套使用，如(())，也可以互相之间并列存在，如()()(()())。设计一个程序，验证以字符串形式输入的表达式中括号是否相互匹配。

在例3-3中，当遇见表达式中的左括号时，无法立即确认后续表达式中是否存在与之匹配的右括号，此时可以对遇见的左括号进行入栈操作，对后续表达式内容继续进行遍历，直到遇见一个右括号，位于栈顶的左括号出栈，与之进行匹配。因为表达式中的括号可以嵌套使用，所以在栈结构中可能同时存在多个左括号，而最先入栈的左括号，一定是表达式中最外层括号的组成部分，所以随着表达式的遍历，栈底的左括号一定是最后被匹配上的，这一点就表现出了括号匹配操作的延迟性。如果表达式遍历完成，栈结构中依然存在剩余的左

括号，则表示表达式中存在多余的左括号，表达式不合法；如果在表达式遍历过程中遇见右括号，而此时栈结构已经为空，则表示表达式中存在多余的右括号，表达式不合法；只有在表达式遍历到最后一个右括号的同时，栈结构恰好出栈最后一个左括号，才表示表达式匹配正确。

下面给出通过栈结构实现的表达式括号匹配算法的示例代码：

```
/**
 Chapter3_StackAndQueue
 com.ds.stack
 TestStack.java
 栈结构的应用测试类
 */
public class TestStack {

    //对表达式中的左右括号进行匹配，从而判断表达式是否合法
    public boolean testExpression(String exp) {

        //1.创建用于进行括号匹配的栈结构
        Stack<Character> stack = new Stack<>();

        //2.将表达式字符串转换成字符数组
        char[] chs = exp.toCharArray();

        //3.遍历表达式，进行括号匹配
        for(char ch : chs) {
            switch(ch) {
                //遇见左括号:入栈等待匹配
                case '(':
                    stack.push(ch);
                    break;
                //遇见右括号
                case ')':
                    //如果此时栈结构为空，则表示存在多余右括号，表达式不合法
                    if(stack.isEmpty()) {
                        return false;
                    }
                    //栈结构不为空，出栈栈顶左括号，表示匹配
                    stack.pop();
                    break;
                //对于其他符号直接忽略
                default:
                    continue;
            }
        }
```

```
            /*
             4.如果表达式遍历结束且栈结构不为空,则表示表达式中存在多余的左括号,表达式不
合法
             */
            if(!stack.isEmpty()) {
                return false;
            }

            //其余状况下,表达式合法
            return true;

        }

    }
```

**【例 3-4】** 逆波兰表达式的转换。逆波兰表达式是计算机当中用于存储运算表达式的一种表示方式。在将一个正常的运算式转换为逆波兰表达式的时候，首先取得运算符左右两侧运算数的优先表示，然后将运算符放在最后进行表示。例如运算式 $a+b$ 转换成逆波兰表达式为 $ab+$；运算式 $(a+b)\times c$ 转换成逆波兰表达式为 $ab+c\times$，其中 $ab+$ 可以看作 $(a+b)$ 部分的运算结果，与运算数 $c$ 同时表示×运算左右两侧的运算数。现给出一组由字符串表示的逆波兰表达式，如{"$a$", "$b$", "$c$", "$d$", "+", "$e$", "×", "+", "$f$", "+", "×"}，设计一个程序将逆波兰表达式转换为一般的运算式形式。例如将上述逆波兰表达式转换为一般运算式可以表示为 $(a\times((b+((c+d)\times e))+f))$。为了方便起见：①逆波兰表达式中仅包含+、−、×、/四则运算；②在转换过程中，得到任意部分的子运算式后都要在左右两端添加括号以表示子运算式的界限。

在进行逆波兰表达式转换过程中，首先扫描到的一定是运算数，但是在仅得到运算数的情况下是没有办法执行运算操作的，所以此时运算数之间的运算延迟可以通过对运算数进行入栈操作来完成。在继续扫描逆波兰表达式的过程中如果遇见运算符，则将位于栈顶的两个运算数出栈，作为运算符两侧的运算数，结合运算符构成运算式的子部分，并将这一子部分重新入栈，表示作为下一运算符的运算数使用。在逆波兰表达式数组遍历结束后，栈结构中剩余的唯一字符串，即为逆波兰表达式转换得到的对应运算式。需要注意的是，对于减法和除法这样不符合交换律的运算操作，栈顶最先出栈的运算数表示其右侧运算数，其次出栈的栈顶元素表示其左侧运算数，而对于加法、乘法等符合交换律的运算，运算数的左右位置并不重要。

下面给出通过栈结构实现的逆波兰表达式转换运算式算法的示例代码：

```
/**
 Chapter3_StackAndQueue
 com.ds.stack
 TestStack.java
```

```
 栈结构的应用测试类
 */
public class TestStack {

    //逆波兰表达式转换运算式的方法
    public String evalRPN(String[] rpn) {

        //1.创建用于运算式转换的栈结构
        Stack<String> stack = new Stack<>();

        //2.遍历逆波兰表达式数组,进行转化
        String left;
        String right;
        String tmp;
        for(String str : rpn) {

            switch(str) {
                //对于四则运算符的操作
                case "+":
                case "-":
                case "*":
                case "/":
                    //先出栈的是右侧运算数
                    right = stack.pop();
                    //后出栈的是左侧运算数
                    left = stack.pop();
                    //拼接子运算式并入栈
                    tmp = "(" + left + str + right + ")";
                    stack.push(tmp);
                    break;
                //对于一般运算数直接入栈
                default:
                    stack.push(str);
                    break;
            }

        }

        //转换结束,栈中唯一字符串为转换完毕的运算式
        return stack.pop();

    }

}
```

**注意**：在计算机中对运算式进行表示的方案可以分为前缀表达式、中缀表达式和后缀

表达式，其中前缀表达式亦称为波兰表达式；后缀表达式亦称为逆波兰表达式。中缀表达式中的运算符出现在运算数的中间，也就是常见的一般运算式。

**3. 内存中的方法栈**

在 Java 的内存模型当中，如果一个方法被调用，JVM 就会创建一个与被调用方法相对应的“栈帧”，并将这个“栈帧”加入方法栈当中。如果在这个方法内部，存在对其他方法的调用操作，则当执行到这一内部方法调用的时候，JVM 同样会为内部调用方法再创建新的栈帧，并入栈方法栈。在 Java 的内存结构当中，之所以使用栈结构描述方法之间的相互调用关系，是因为外部方法在执行过程中，总是依赖内部方法运行完毕后所产生的结果进一步向下执行，所以先调用的方法总是后结束，而在内部被后续调用的方法总是需要先结束，而这种“先调用后结束、后调用先结束”的延迟操作特性，正适合使用栈结构进行实现。

下面给出一组方法相互调用的示例，并以图示的方式演示方法栈中栈帧的变化过程，代码如下：

```
/**
 Chapter3_StackAndQueue
 com.ds.stack
 TestMethodStack.java
 方法栈演示代码
 */
public class TestMethodStack {

    public void methodA() {
        //方法 A 的执行代码
    }

    public void methodB() {
        methodA();
    }

    public void methodC() {
        methodB();
    }

    public void methodD() {
        //方法 D 的执行代码
    }

    public static void main(String[] args) {
        TestMethodStack test = new TestMethodStack();
        test.methodC();
        test.methodD();
    }

}
```

上述方法调用对应的方法栈的变化过程如图 3-5 所示。

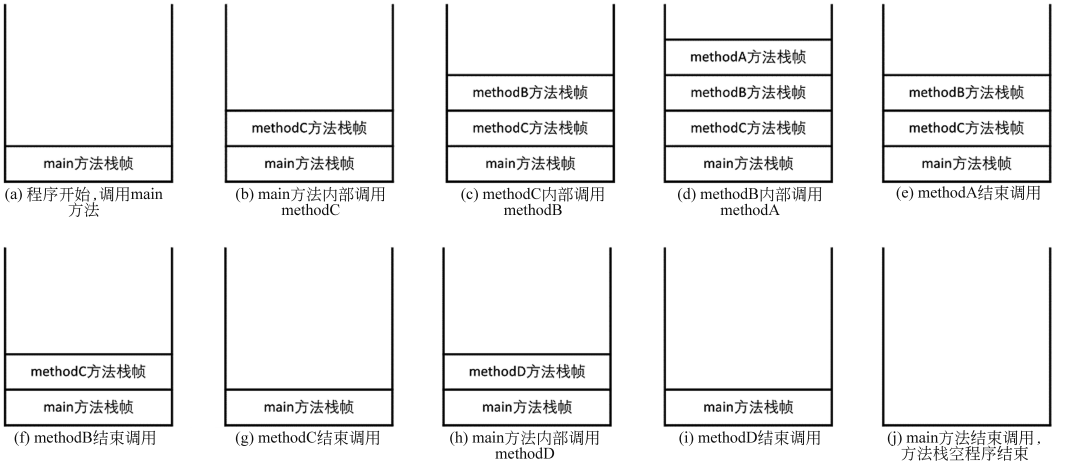


图 3-5　方法栈的变化过程

**4. 树和图的深度优先遍历**

在第 6 章与第 9 章的内容当中，将会继续对树和图这两种非常重要的数据结构进行说明。在针对树和图两种结构的基础操作当中都存在深度优先遍历(Depth First Search，DFS)操作，而在对树或者图进行深度优先遍历操作时，最常用的方式有两种：①通过栈结构实现；②通过递归结构实现。在第 4 章介绍递归结构的内容中还会详细地讲解递归结构在内存中的实现方式，而这一点仍然与前述讲解的方法栈息息相关。

鉴于目前尚未对树结构和图结构进行讲解，缺乏对于二者的基本认知，故而将此部分知识点放在第 6 章与第 9 章中详细说明，在此不过多进行阐述。

## 3.2　队列结构

### 3.2.1　队列结构概述

队列结构的逻辑特点是元素先进先出(First In First Out，FIFO)。可以将队列结构理解成为一个两端开口，一端入、一端出的结构，最先加入队列的元素也是最先退出队列的元素，最后加入队列的元素也是最后退出队列的元素。

在队列结构中，将元素从一端加入队列的操作称为入队列(offer)，将元素从另一端退出队列的操作称为出队列(poll)。元素入队列的一端称为队列尾(rear)，元素出队列的一端称为队列头(front)。最先入队列的元素位于队列头位置，最后入队列的元素位于队列尾位置。在执行出队列操作的时候，队列头元素最先出队列，队列尾元素最后出队列。

与栈结构相似的是，当队列为空的时候执行出队列操作也是被视为不正确的，同样可以通过抛出异常的方式结束操作。

队列结构示意图如图 3-6 所示。

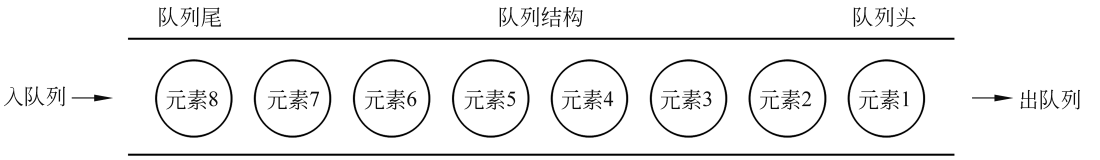


图 3-6　队列结构示意图

队列结构与现实生活中的排队非常相似，先来排队的人优先被服务，后来排队的人需要等前面的人都离开后才能被服务，而在排队过程中，先来排队的人位于队伍最前端，最后来的人位于队伍最末端。人员进入队伍和离开队伍的位置分别位于队伍两端，中间不允许插队。

### 3.2.2　队列结构的实现

队列结构同样可以使用数组或者链表实现。

在使用数组对队列结构进行实现时，通常设置两个 int 型变量 front 和 rear，其中变量 front 始终指向队列头元素所在下标，变量 rear 始终指向下一个入队列元素所要占用位置的下标。初始状态下，front 和 rear 的取值相等，均为 0。在元素入队列时，入队列元素加入数组中 rear 取值下标的位置并执行 rear＋＋；在元素出队列时，从数组中 front 取值下标的位置退出队列头元素并执行 front＋＋。在元素入队列、出队列过程中，如果 front ＝＝ rear 条件达成（注意此时 front 和 rear 取值可能并不是 0），则表示队列为空。

需要注意的是，通过数组结构实现队列仍然需要考虑因为元素入队列导致的数组扩容问题。同时因为在元素入队列、出队列操作过程中，front 和 rear 变量都是单向递增的，所以当元素出队列后，数组中下标取值小于 front 的部分将不会再被后续入队列元素占用，此时将会导致数组空间的浪费。这一情况在通过数组实现的循环队列结构中有所改善，详细内容将在 3.2.4 节中进行讲解。

通过数组实现的队列结构及相关操作如图 3-7 所示。

通过链表对队列结构进行实现相对简单。在实现过程中，同样需要设置两个节点类型的引用变量 front 和 rear，分别指向链表中队列头节点和队列尾节点（此处注意区分与数组实现队列结构过程中，rear 变量指向位置的不同）。初始状态下，front 和 rear 取值相同，都指向 null，并且反向判断 front 和 rear 同时指向 null 也可以作为当前队列为空的判别标准。元素入队列操作使用链表的尾追加方式实现。当初始队列为空时，保存入队列元素的新节点直接追加在链表的 head 头节点后面，并且 front 和 rear 变量同时指向新节点；如果初始队列非空，保存入队列元素的新节点追加在 rear 指向节点的后面，并且调整 rear＝rear.next。元素出队列操作使用链表的头删除方式实现。首先删除 front 变量指向的节点，并使用其中数据域取值作为返回值，然后调整 front＝head.next，即 front 变量始终指向链表头节点的直接后继节点。如果元素出队列后链表清空，则还需调整 rear＝null。

通过链表实现的队列结构及相关操作如图 3-8 所示。

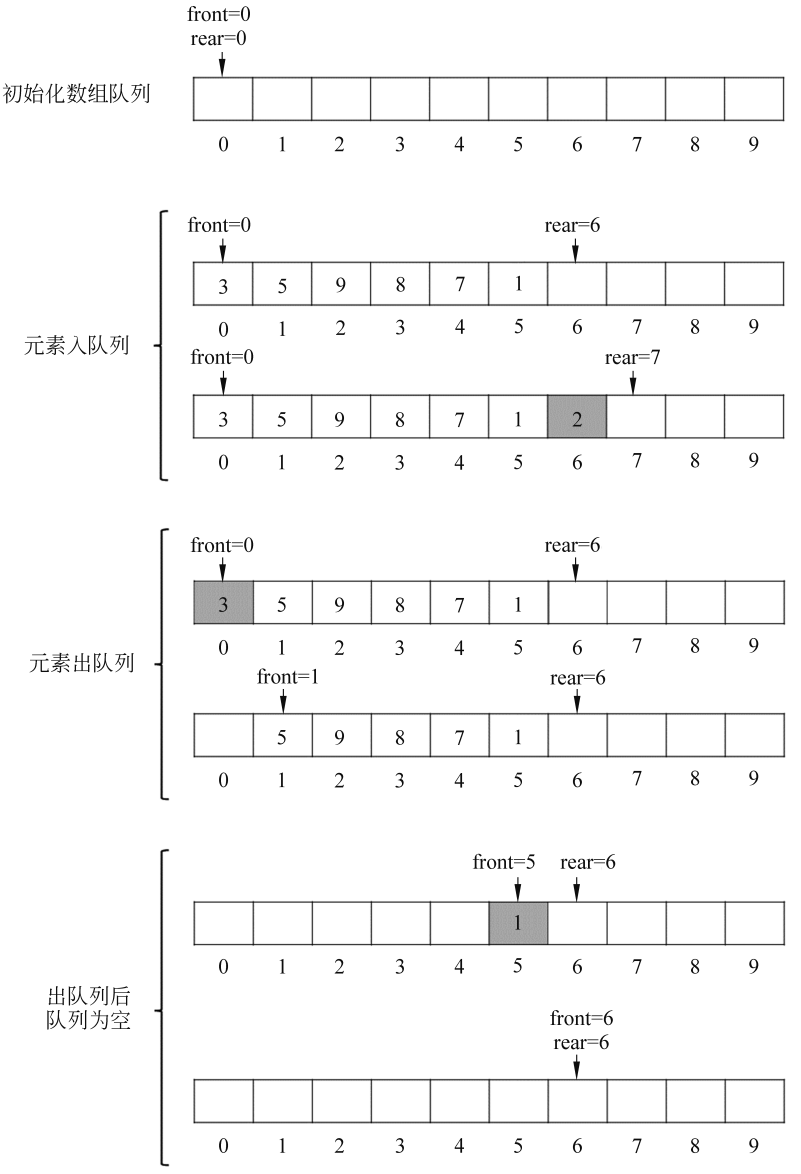


图 3-7　通过数组实现的队列结构及相关操作

## 3.2.3　队列结构的应用场景

队列结构的应用场景，总是与等待、排队相关。先来排队的元素先被处理，后来排队的元素后被处理。这种特点在很多算法与操作系统的底层应用中非常常见。下面通过两组具体的案例对队列结构的应用场景进行讲解。

**1. 顺序作业调度**

**【例 3-5】** 作业调度是操作系统底层为不同请求分配系统资源时进行的操作。如果多

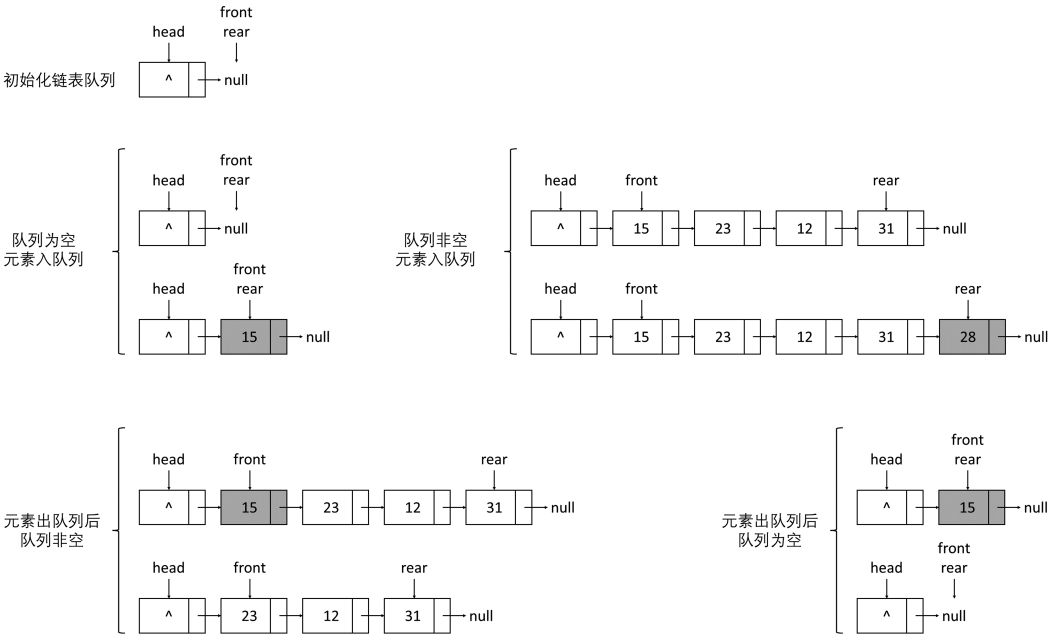


图 3-8　通过链表实现的队列结构及相关操作

个请求需要申请同一系统资源进行支持，而操作系统对多个请求按照资源申请的先后顺序进行处理，即为顺序作业调度。

现假设在一台具有单核 $k$ 线程 CPU 的计算机上同时产生了 $n$ 个请求，每个请求需要占用 CPU 的某一线程一段时间进行处理，同一 CPU 线程同一时间下只能处理一个请求，并且每个请求所占用的 CPU 线程都是由调度随机分配的。设计一个程序模拟上述场景，并在 $n$ 个请求全部处理完毕后，统计 CPU 每个线程被占用的时间。

上述需求是一个典型的顺序作业调度场景。在实现模拟的过程中，可以为每个 CPU 的线程设计一个任务队列，当某一请求发出后，根据随机值为其分配一个用来处理的 CPU 线程，并将这个任务加入对应线程编号的任务队列当中。当所有任务分配完毕后，对每个任务队列循环执行任务出队列操作，直到任务队列为空为止。在出队列过程中，将每个任务对线程的占用时间累加，即可得到每个 CPU 线程被占用的时间总和。

上述模拟场景的实现代码如下：

```
/**
 Chapter3_StackAndQueue
 com.ds.queue
 TestQueue.java
 队列结构的应用测试类
 * /
public class TestQueue {
```

```
    /**
     对顺序作业调度场景进行模拟的代码
     参数 int k 表示 CPU 的线程数量
     参数 long[] taskMills 表示请求对 CPU 线程的占用时间
     返回值表示在处理完所有请求后,每个 CPU 线程被占用的总时间
     */
    public long[] sequentialJobScheduling(int k, long[] taskMills) {

        /*
         1.为每个 CPU 线程创建一个任务队列
         为方便操作,将所有任务队列保存在一个长度为 k 的数组当中
         */
        Queue<Long>[] taskQueues = new Queue[k];
        for(int i = 0; i < k; i++) {
            taskQueues[i] = new LinkedList<>();
        }

        /*
         2.为每个请求分配一个 CPU 线程进行操作
         即将每个请求对应的操作时间根据随机值加入对应的任务队列
         CPU 线程编号的取值范围为[0, k-1]
         */
        int queueNum;
        for(int i = 0; i < taskMills.length; i++) {
            queueNum = (int)(Math.random() * k);
            taskQueues[queueNum].offer(taskMills[i]);
        }

        //3.请求分配结束后,对所有线程的占用时间进行统计
        long[] times = new long[k];
        Queue<Long> q;
        for(int i = 0; i < k; i++) {
            q = taskQueues[i];
            while(!q.isEmpty()) {
                times[i] += q.poll();
            }
        }

        return times;

    }

}
```

**2. 树和图的广度优先遍历**

在 3.1.3 节当中曾说明过,对于树和图两种结构可以使用栈结构或者递归实现其深度优先遍历操作。在深度优先遍历操作之外,树和图两种结构还存在广度优先遍历(Breadth First Search,DFS)操作,而树和图的广度优先遍历操作则是通过队列结构实现的。同样

地，这一部分内容将留在第 6 章与第 9 章进行详细讲解。

### 3.2.4 队列结构的衍生

**1. 双端队列**

双端队列就是两端均可以进行元素出入队列操作的队列结构。在双端队列中，元素入队列和出队列的操作又可以被详细地分为 offerFirst、offerLast 和 pollFirst、pollLast 两组。通过上述 4 种操作，可以根据具体的业务需求灵活地实现对队列结构中任意一端顶端数据的入队列、出队列操作。在双端队列的具体实现过程中，offerFirst 和 pollFirst 两种操作都是从结构的起始端进行元素的入队列和出队列操作；offerLast 和 pollLast 两种操作都是从结构的最末端进行元素入队列和出队列的操作。图 3-9 较为明确地演示了这 4 种操作与双端队列实现结构两端的关系。

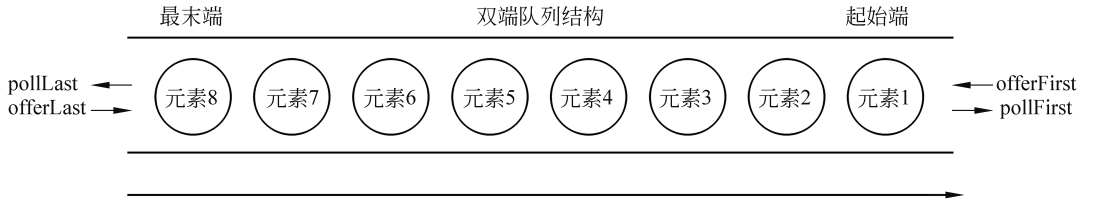


图 3-9 双端队列结构

从理论上来讲，双端队列既可以通过数组方式实现，也可以通过双链表方式实现，但是如 3.2.2 节讲解过的，如果通过数组实现一个不限制长度的队列结构，就有可能导致数组中出现空间浪费的情况，况且双端队列的元素出入队列操作是在数组两端均可实现的，所以不论选择数组结构的哪一端作为元素出入队列的起点都有更大的概率引起数组结构频繁地对元素进行移动操作，所以在此对数组实现双端队列结构的方案不做过多阐述。

在通过双链表结构实现双端队列结构时，因为在双链表中存在 head 和 tail 两个节点，分别从两端引起整个双链表结构，所以在定位链表结构的起始位置与结束位置的时候更加方便。在双端队列不为空的前提下，pollFirst 操作相当于删除 head 节点的直接后继节点；pollLast 操作相当于删除 tail 节点的直接前驱节点。不论双端队列是否为空，offerFirst 操作都相当于向 head 节点的直接后继位置进行新节点的插入；offerLast 操作都相当于向 tail 节点的直接前驱位置进行新节点的插入。

通过双链表实现的双端队列结构及相关操作如图 3-10 所示。

双端队列结构的一大经典应用是在“作业窃取”操作中的使用。

用一个相对通俗的案例解释“作业窃取”操作：假设某学校食堂有 4 个窗口可以打饭，在每个窗口前都有一条相互独立的队伍进行排队，每位前来就餐的学生可以随机选择四个队伍中的一个进行排队。如果在排队过程中，某一窗口前排队的学生都已经取餐完毕，则这个窗口就已经处于空闲状态。空闲状态下的窗口就可以呼唤其他学生队伍中处于队伍末端的同学前来打饭，这就是典型的“作业窃取”操作场景。

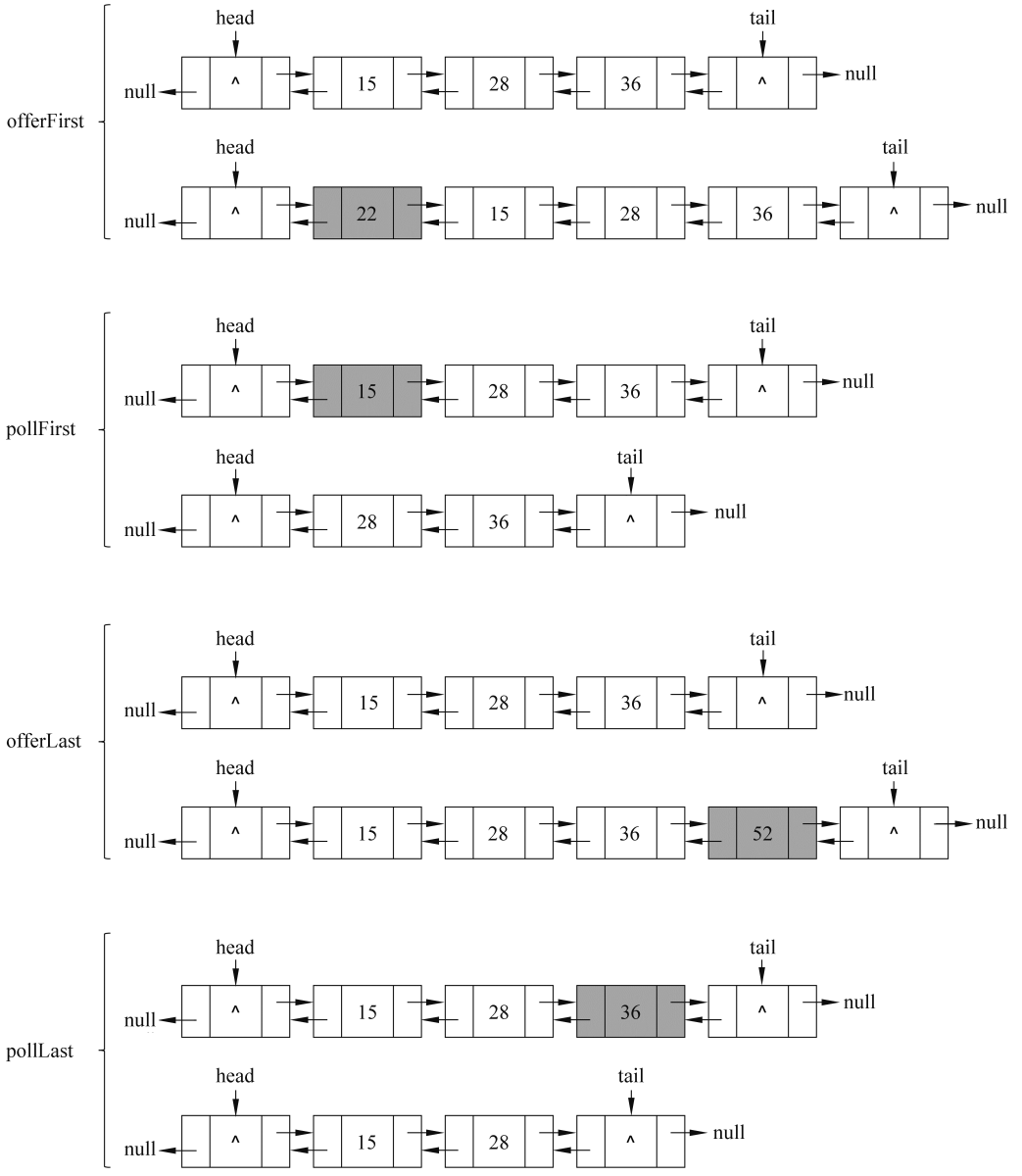


图 3-10　双链表实现的双端队列结构及相关操作

如果将 4 个食堂窗口看作 CPU 的 4 个独立线程，则在窗口前排队的学生队伍就是当前 CPU 线程的任务队列。每个 CPU 线程每次从自己任务队列的最前端出队列一个任务进行处理，而当有其他线程空闲时，这些空闲线程就可以从其他 CPU 线程任务队列的末尾“窃取(出队列)”一个任务进行处理。这样一来就可以最大限度地发挥多线程 CPU 的性能优势，不造成 CPU 线程资源的浪费，从而提升整体的任务处理效率。

在 Java 中提供的 ForkJoinPool 线程池及其相关 API 就是基于上述理论实现的，而其

中对于任务队列的实现正是使用了双端队列。

**2. 循环队列**

在 3.2.2 节中曾讲解过，如果使用数组实现队列，在不限定队列结构长度的情况下会在元素不断入队列、出队列的操作过程中产生大量数组空间的浪费，但是如果指定队列结构的长度，则会在入队列元素数量达到数组总长度时导致队列结构在逻辑上为满。此时即使出队列一些元素，队列头方向释放出来的数组空间也无法重复利用，依然会因为队列结构在逻辑上为满，导致新元素不能入队列。这种数组结构中存在空闲空间，但是队列结构在逻辑上为满、元素无法入队列的情况，称为队列结构的“假溢出”。如果将通过数组结构实现、限定长度的队列结构设计为循环队列，则不仅能有效地解决假溢出的问题，还能够不断地重复利用元素出队列后队列头方向产生的空闲数组空间。

在实现循环队列结构时，通常会在初始化阶段设置循环队列的长度。在通过设定的队列长度创建底层数组结构时，要求数组长度为循环队列长度＋1。在循环队列结构中，同样设置初值为 0 的 front 和 rear 两个变量，用来指定队列头元素所在下标和下一个入队列元素所占用的数组下标，即在循环队列非空的情况下，rear 变量同样指向循环队列结构中队列尾元素所在下标＋1 的位置。

初始状态下的循环队列结构如图 3-11 所示。

图 3-11　初始状态下的循环队列结构

假设当前循环队列内部数组的长度为 size，即循环队列为满时可容纳 size－1 个元素。

当元素入队列时，占用数组中下标为 rear 的位置，并将 rear 移动到(rear＋1) % size 的位置，即执行 rear＝(rear＋1) % size 的操作。之所在执行上述计算时使用 rear＋1 的取值对数组长度 size 进行取余运算，是因为在边界条件下如果入队列元素占用的是数组中最大下标位置，则可以通过上述操作将 rear 标记位直接移动回数组下标取值为 0 的位置，相当于重复利用数组结构中前序部分的空位，达到元素循环入队列的效果。例如在使用长度为 10 的数组实现的元素容纳数量为 9 的循环队列中，当前 rear 标记位已经指向数组中下标为 9 的位置，那么在新元素入队列并占用数组中下标为 9 的位置后，计算 rear＝(rear＋1) % size＝(9＋1) % 10＝0，即可将 rear 标记位重新指回数组的起始位置。

元素加入循环队列的两种情况如图 3-12 所示。

同样地，当元素出队列时，在取得数组中 front 下标位元素执行出队列操作后执行 front＝(front＋1) % size 操作，其目的同样是为了保证在边界条件下 front 标记位能够返回数组的起始下标位置。

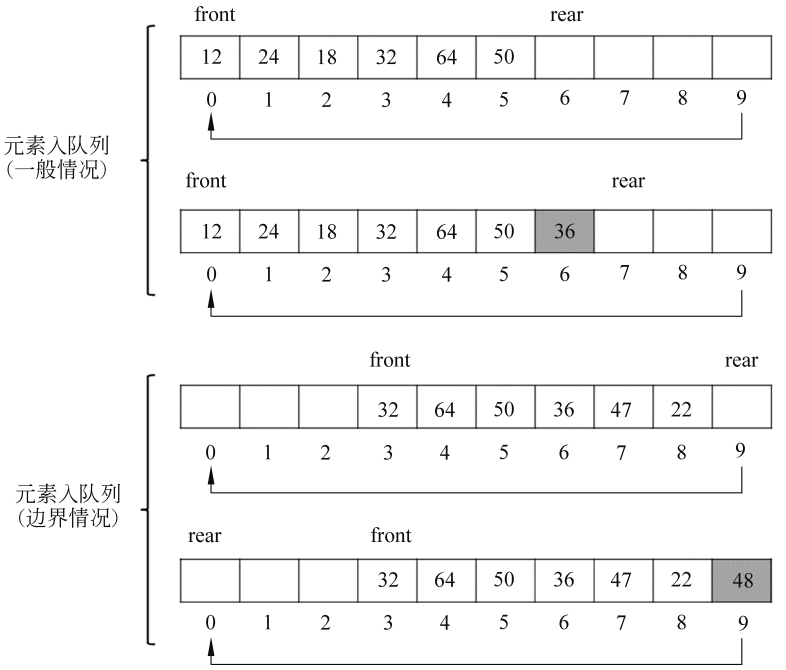


图 3-12　元素加入循环队列的两种情况

元素退出循环队列的两种情况如图 3-13 所示。

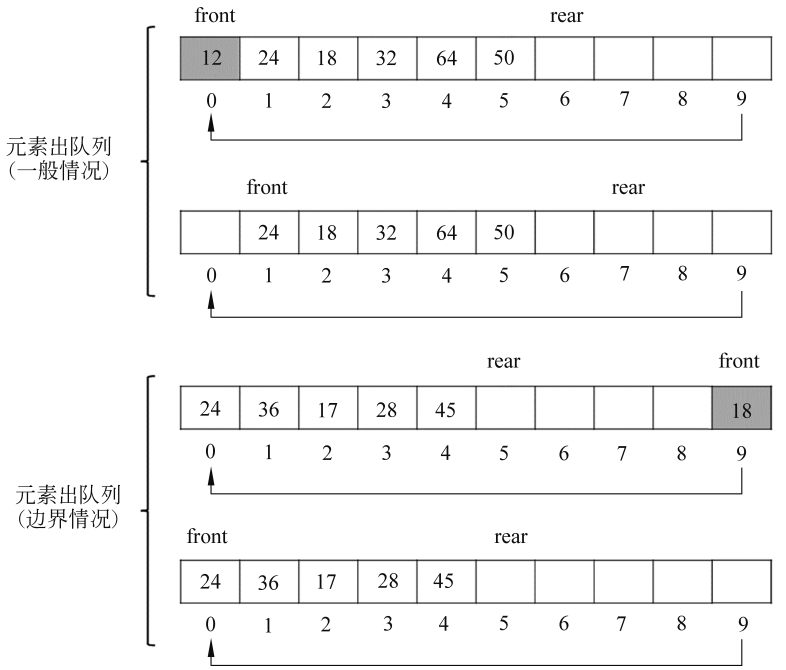


图 3-13　元素退出循环队列的两种情况

在判断循环队列是否为空时，同样可以使用 front == rear 的条件进行判断，但是如果要判断循环队列为满的情况，则需要进行如下讨论：如果将数组结构被填满作为循环队列结构为满的判断标准，则根据 rear 变量取值为队列尾元素下标+1 来计算，当数组被填满时同样会得到 front == rear 的结果，也就是说如果将数组被填满视作循环队列为满，则判断循环队列为满和循环队列为空的条件重叠。为了避免这种情况发生，则浪费掉数组中的 1 位空间，即当数组中只有 1 位空间空闲时，就认为循环队列为满。此时，front 和 rear 两变量之间的关系符合(rear+1) % size == front。这种做法也解释了循环队列底层的数组结构的长度等于循环队列长度+1 的原因，数组结构+1 的长度就是在判断循环队列为满时浪费掉的 1 位空间。

循环队列为空与为满的情况如图 3-14 所示。

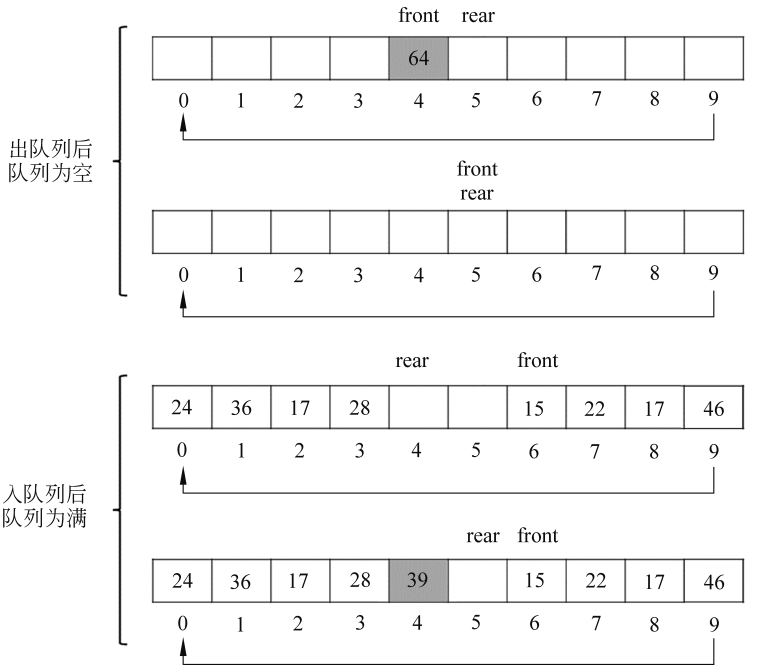


图 3-14 循环队列为空与为满的情况

### 3. 优先队列

在原生队列结构中，元素存储的顺序总是与元素入队列的顺序相关，但是在优先队列中元素存储的顺序则是与元素的权值相关。在优先队列中允许使用者通过自定义的方式，对加入队列的元素之间的权值进行比较，权值最大或者最小者处于队列头位置。也就是说优先队列每次出队列操作得到的取值，总是其中具有最高权值或者最低权值的元素，与元素加入队列的顺序无关。例如下列元素的取值，即代表元素的权值，元素取值为{3,5,1,0,9,4,2,8}。现将这些元素加入一个高权值优先的优先队列当中，则上述元素出队列的顺序为{9,8,5,4,3,2,1,0}。

Java 中同样提供了优先队列的实现类，即 PriorityQueue 类型，而 PriorityQueue 类型则是借助了堆结构实现其中元素的按照权值出队列功能。关于堆结构的知识将在第 7 章中进行详细讲解，在此不做过多阐述。

# 第 4 章

# 递归、查找与排序

第 2 章已经对数组和链表这两种物理结构进行了讲解，但是针对数组和链表结构中元素的操作，除了基本的增、删、改、查外还有很多其他的方式，这其中较为常见的就是元素查找与排序操作。对于数组和链表中元素的查找，除了基本的顺序查找方式外最为常用的就是二分查找。二分查找虽然对数据序列中元素的排列顺序有一定要求，但是其可以在对数级别的时间复杂度下完成对目标元素的查找，并且在二分查找的基础上，还衍生出了插值查找和斐波那契查找等其他查找算法，而元素排序操作根据其实现思路的不同，又可以详细分为多种不同的排序算法，不同的排序算法在时间复杂度、空间复杂度及稳定性方面也具有不同的表现，但是不论查找算法还是排序算法，在实现的过程中又都有用到递归结构的场景，所以本章将对递归结构、多种查找及排序算法进行说明。为了方便讲解，本章中所有的代码示例均以数组结构作为数据序列载体。

本章内容的思维导图如图 4-1 所示。

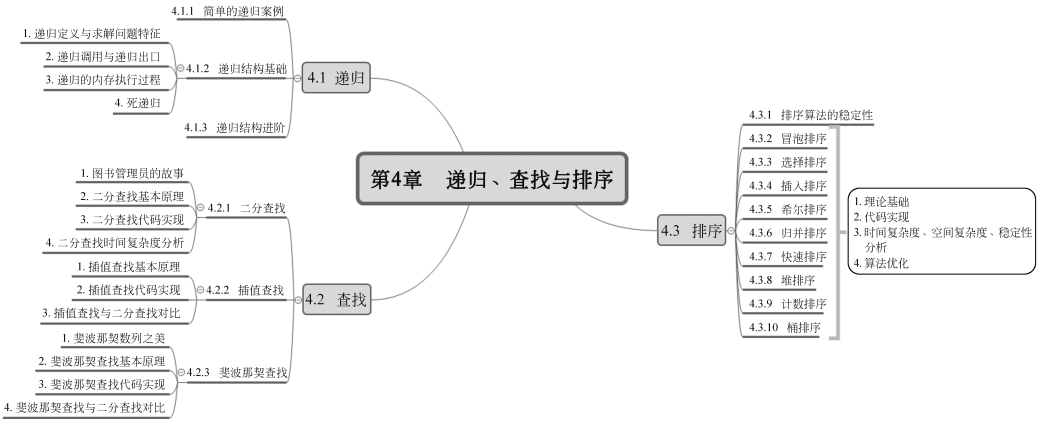


图 4-1　递归、查找与排序章节思维导图

## 4.1　递归

递归结构一直以来都是数据结构与算法学习过程中的重点和难点，甚至夸张一些来讲，是否熟练掌握递归结构及能否通过递归思想解决问题，可以作为衡量一个程序员水平的标杆。