第三章：字符串和矩阵

1. 字符串

1.1 字符串的简介

➢ **字符串的定义**

字符串（string，简称串）是由 n（n≥0）个字符组成的有限序列，串中所包含的字符个数称之为串的长度。串通常记作：S = "$s_1s_2s_3...s_n$"。其中，S 是串名，双引号是定界符（避免字符串与变量名或数的常量混淆），双引号包裹的内容就是串值。字符串和字符之间的关系，就好比于羊肉串和羊肉之间的关系。

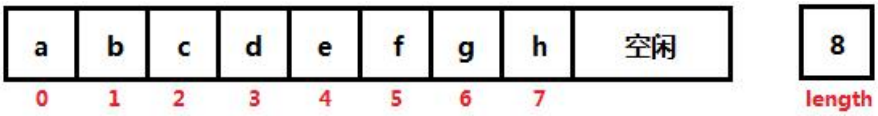
长度为 0 的串称之为**空串**，记作""，空串中不包含任何字符。由一个或多个空格组成的串成之为**空格串**，其长度就是串中包含的空格数。显然，在统计字符串的长度时需包含其中的空格，例如"hello world"的长度就是 11。

字符串中任意个连续的字符组成的子序列称之为该串的**子串**（substring），相应的包含子串的串称之为**主串**（primary string）。子串的第一个字符在主串中的序号，我们称之为子串在主串中的位置，例如子串"sxt"在主串"whsxt"中的索引位置就是 2。

➢ **字符串的存储结构**

字符串是数据元素为单个字符的线性表，一般采用顺序存储结构（因为链式存储结构的空间利用率低），即使用数组来存储串中的字符序列。例如，在 C、C++、Java 等语言中，字符串都是采用顺序存储结构，一般有三种方法来表示字符串长度：

(1) 用一个变量来表示串的长度，如下图所示：



(2) 用数组索引为 0 的位置来存放串的长度，串值从索引为 1 的位置开始存放，如下图所示：

(3) 在串末尾存储一个不会在串中出现的特殊字符作为字符串的终结符，例如在 C、C++和 Java 语言中就用'\0'来表示串的结束。这种方法不能直接得到串的长度，而是通过判断当前字符是否为'\0'来确定串是否结束，从而求得串的长度，如下图所示：

1.2 字符串的模式匹配

给定两个字符串 S = "$s_1s_2s_3...s_n$" 和 T = "$t_1t_2t_3...t_m$"，在主串 S 中寻找子串 T 的过程就称之为字符串的**模式匹配**（pattern matching），T 称为**模式**（pattern）。如果匹配成功，则返回 T 在 S 中的位置；如果匹配失败，则返回-1。

字符串的模式匹配是字符串的最基本操作，它广泛应用于文本处理、邮件过滤、杀毒软件、操作系统、数据库系统以及搜索引擎中。因此，字符串的模式匹配是字符串最为频繁的操作，如何简化其复杂性一直是算法研究中的经典问题。

1.2.1 BF 算法

- **BF 算法实现思路**

BF（Brute-Force）算法的基本思想就是暴力匹配（蛮力匹配），即从主串 S 的第一个字符开始和模式 T 的第一个字符进行比较。若相等，则继续比较两者后续的字符串；否则，从主串 S 的第二个字符开始和模式 T 的第一个字符进行比较。重复上述过程，直至 S 或 T 中的所有字符比较完毕。若 T 中的字符全部比较完毕，则证明匹配成功，返回本趟匹配的开始位置；否则就证明匹配失败，返回-1 即可。

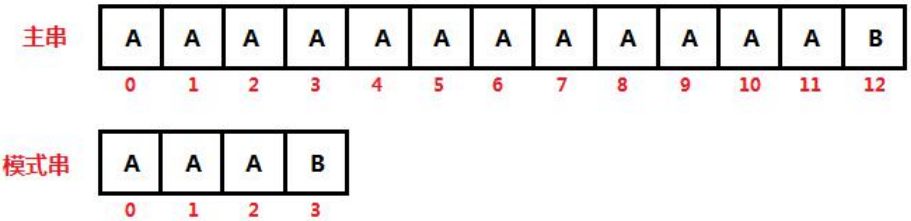
- **BF 算法的代码实现**

```
package com.bjsxt.p1.string;
/**
 * 字符串模式匹配之 BF 算法
 */
public class Test02 {
    public static void main(String[] args) {
        // 主串
        String destStr = "WHNBWHSXTNB";
        // 模式串
        String subStr = "WHSXT";
        // 测试 BF 算法
        int index = bfSearch(destStr, subStr);
        System.out.println(index);
    }
    /**
     * 实现 BF 算法
     * @param destStr 主串
     * @param subStr 模式串
     * @return 返回模式串的第一个字符在主串中的索引位置。
```

```
     *          如果主串中不存在该模式串，则返回-1
     */
    public static int bfSearch(String destStr, String subStr) {
        // 1.定义一个指针，用于指向主串中的第一个字符
        int x = 0;
        // 2.定义一个指针，用于指向模式串中的第一个字符
        int y = 0;
        // 3.定义一个循环，用于实现字符串的匹配操作
        while (x < destStr.length() && y < subStr.length()) {
            // 4.如果相等，则继续逐个比较后续字符
            if (destStr.charAt(x) == subStr.charAt(y)) {
                x++;
                y++;
            }
            // 5.如果不相等，则从主串的下一个字符起，从新与模式串的第一个字符进行比较
            else {
                x = x - y + 1;
                y = 0;
            }
        }
        // 6.如果模式串遍历完毕，则证明匹配成功，否则就证明匹配失败
        if (y == subStr.length()) {
            return x - y;
        }
        else {
            return -1;
        }
    }
}
```

- **BF 算法的时间复杂度**

假设主串 S 的长度为 n，模式串 T 的长度为 m，在匹配成功的情况下，我们来考虑最坏情况下的时间复杂度。在最坏情况下，每趟不成功的匹配都发生在模式串 T 的最后一个字符。



例如：S="AAAAAAAAAAAAB"，T="AAAB"，那么前 n-m 趟不成功的匹配比较了(n-m)*m 次，最后一趟匹配成功比较了 m 次，则总共匹配了(n-m)*m+m 次。若 m<<n，则 BF 算法时间复杂度为：O(n*m)。

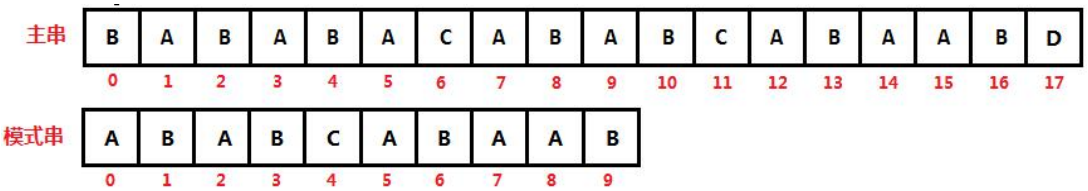
因此，虽然 BF 算法简单容易理解，但是它的执行效率却是非常低。接下来，我们就来学习 KMP 算法，在 KMP 算法中的时间复杂度为 O(n + m)。

1.2.2 KMP 算法

- **KMP 算法实现思路**

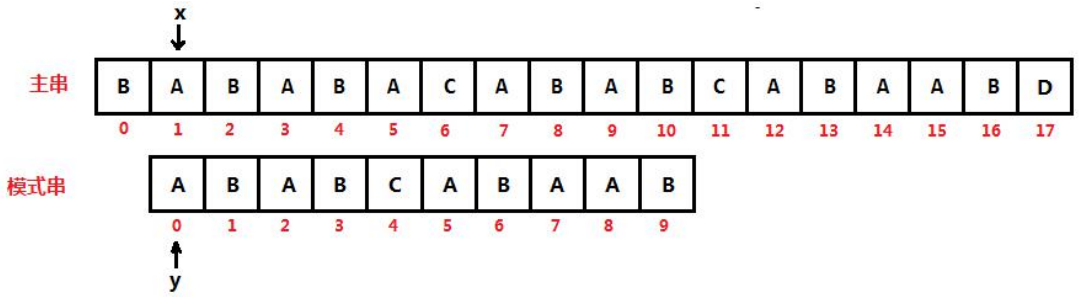
KMP 算法是一种改进的字符串匹配算法，由 D.E.Knuth，J.H.Morris 和 V.R.Pratt 提出并实现的，因此人们称它为 KMP 算法。KMP 算法的核心是利用匹配失败后的信息，尽量减少模式串与主串的匹配次数，从而达到快速匹配的目的。具体实现方案为，主串的 x 指针不回溯，通过修改模式串的 y 指针的指向，从而让模式串尽量的移动到有效的位置，这样的话就提高了字符串的匹配效率。

KMP 算法的整体思路是什么样子呢？让我们来看一组例子：

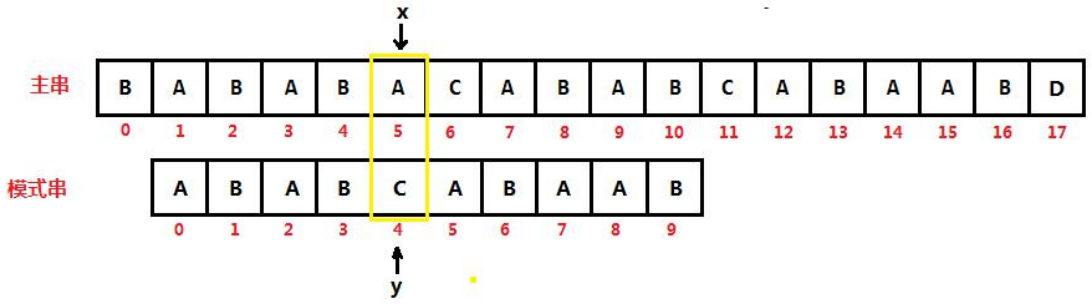


KMP 算法和 BF 算法的“开局”是一样的，即从主串 x=0 的字符开始和模式串 y=0 的字符进行比较。

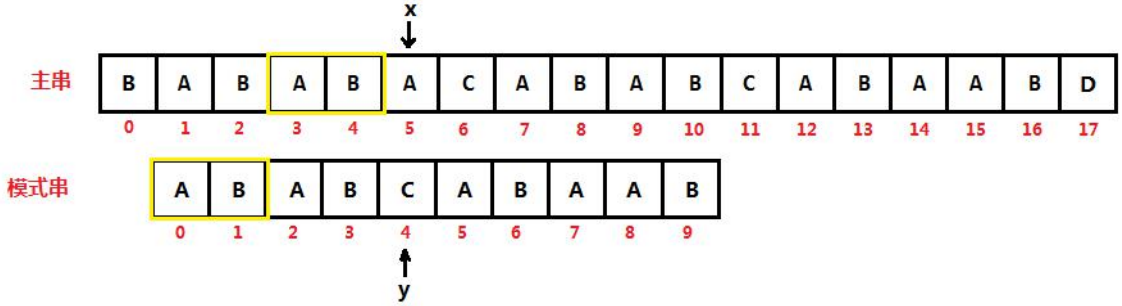
第一轮，主串和模式串的第一个字符不匹配，则让主串的 x=1 字符和模式串的 y=0 的字符进行匹配，如下图所示：



第二轮，主串索引 x 为[1, 4]的字符和模式串索引 y 为[0,3]的字符匹配，此时主串索引 x=5 的字符和模式串索引 y=4 的字符不匹配。

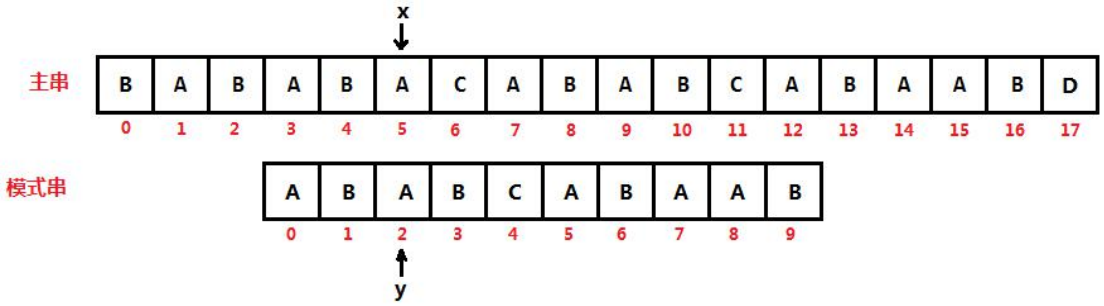


这时候，如何有效利用已匹配的字符串“ABAB”呢？我们可以发现，在字符串“ABAB”当中，后两个字符“AB”和前两位字符“AB”是相同的：



因此，此时主串 x 的指针无需回溯，只需修改 y 指向 2 即可，也就是让主串 x=5 的字符和模式

串 y=2 的字符进行比较，这样就减少了比较的次数。



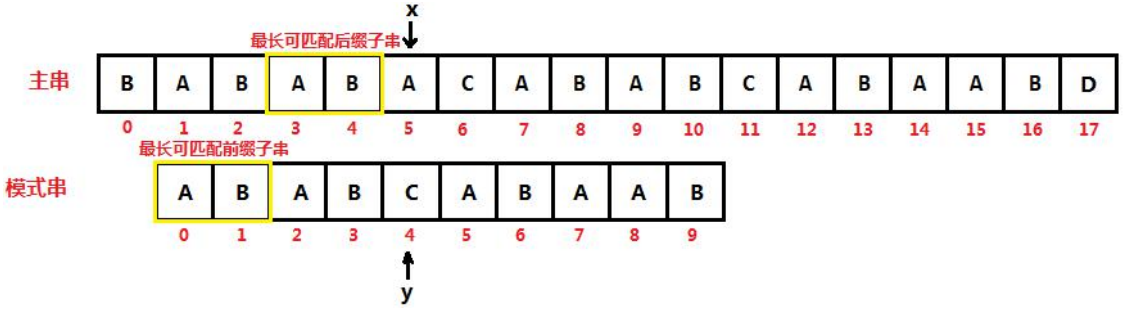
后面的操作步骤以此类推，此处我就不去做演示了。接下来，我们来讲解如何找到一个字符串前缀的“最长可匹配后缀子串”和“最长可匹配前缀子串”，也就是讲解 next 数组的计算。**next 数组有什么作用呢？next[y]的值表示，当“主串[x] != 模式串[y]”时，y 指针的下一步移动位置。**

- **next 数组的推导过程**

首先，我们先了解两个概念：“前缀”和“后缀”。“前缀”指的就是除了最后一个字符之外，剩余字符组成的字符串就称之为前缀。例如，字符串“ABABC”的前缀就是“ABAB”。“后缀”指的就是除了第一个字符之外，剩余字符组成的字符串就称之为后缀。例如，字符串“ABABC”的后缀就是“BABC”。

然后，我们再来认识“前缀组合”和“后缀组合”。“前缀组合”指的就是除了最后一个字符之外，一个字符串的全部头部组合就称之为“前缀组合”。例如，字符串"ABAB"的前缀组合有："A"、"AB"、"ABA"。“后缀组合”指除了第一个字符之外，一个字符串的全部尾部组合就称之为“后缀组合”。例如，字符串"ABAB"的后缀组合有："BAB"、"AB"、"B"。

next 数组保存就是已匹配过模式串前缀的“最长可匹配前缀子串”和“最长可匹配后缀子串”的最长的共有元素的长度。例如，主串索引 x=5 对应的字符和模式串索引 y=4 对应的字符不等，此时已匹模式串的字符串就是"ABABC"，字符串"ABABC"的前缀就为“ABAB”，则对应的“最长可匹配前缀子串”和“最长可匹配后缀子串”就是“AB”，也就是“ABAB”的最长共有元素长度就为 2，这也就意味着 next[4]存放的元素值就为 2，对应的我们需要将 y 指向模式串索引为 2 的位置。



就下来，我们以模式串"ABABCABAAB"为例，来讲解 next 数组的推导过程，如下图所示：

索引为 0 时，该字符串没有前缀，因此最长共有元素长度计为-1（方便 KMP 算法使用）。

索引为 1 时，该字符串的前缀为"A"，此时没有共有元素，因此最长共有元素长度为 0。

索引为 2 时，该字符串的前缀为"AB"，此时没有共有元素，因此最长共有元素长度为 0。

索引为 3 时，该字符串的前缀为"ABA"，此时共有元素为"A"，因此最长共有元素长度为 1。
索引为 4 时，该字符串的前缀为"ABAB"，此时共有元素为"AB"，因此最长共有元素长度为 2。
索引为 5 时，该字符串的前缀为"ABABC"，此时没有共有元素，因此最长共有元素长度为 0。
索引为 6 时，该字符串的前缀为"ABABCA"，此时共有元素为"A"，因此最长共有元素长度为 1。
索引为 7 时，该字符串的前缀为"ABABCAB"，此时共有元素为"AB"，因此最长共有元素长度为 2。
索引为 8 时，该字符串的前缀为"ABABCABA"，此时共有元素为"ABA"，因此最长共有元素长度为 3。
索引为 9 时，该字符串的前缀为"ABABCABAA"，此时共有元素为"A"，因此最长共有元素长度为 1。

因此，模式串"ABABCABAAB"对应的 next 数组就是{-1, 0, 0, 1, 2, 0, 1, 2, 3, 1}，如下图所示：

模式串	A	B	A	B	C	A	B	A	A	B
next数组	-1	0	0	1	2	0	1	2	3	1
	0	1	2	3	4	5	6	7	8	9

也就是说，主串索引 x 的字符和模式串索引 y 的字符不匹配，模式串 y 指向的位置就是 next[y]所对应的位置。例如，当 x=5，y=4 所对应的字符不匹配是，模式串 y 指向的位置就是 next[4]所对应的位置，也就是让 y 指向索引为 2 的位置。

- **KMP 算法的代码实现**

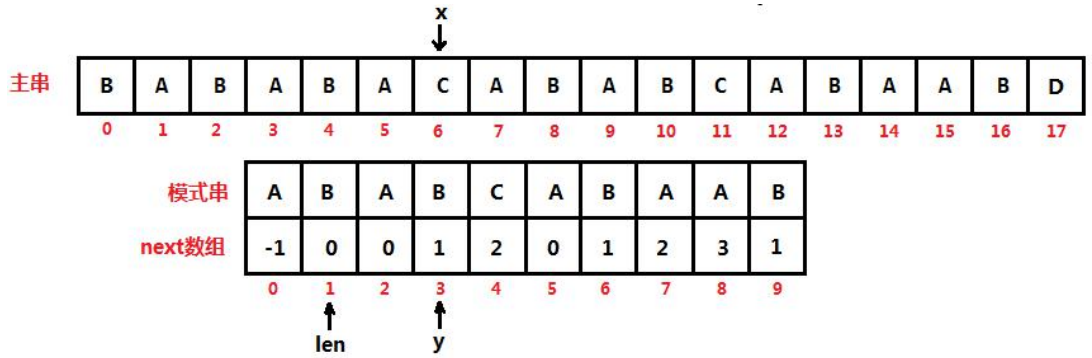
```
package com.bjsxt.p1.string;

import java.util.Arrays;

/**
 * 字符串模式匹配之 KMP 算法
 */
public class Test03 {
    public static void main(String[] args) {
        // 主串
        String destStr = "BABABACABABCABAABD";
        // 模式串
        String subStr = "ABABCABAAB";
        // 测试 next 数组
        int[] next = getNext(subStr);
        System.out.println(Arrays.toString(next));
        // 测试 KMP 算法
        int index = kmpSearch(destStr, subStr);
        System.out.println(index);
    }

    /**
     * 实现 KMP 算法
     * @param destStr 主串
     * @param subStr 模式串
     * @return 返回模式串的第一个字符在主串中的索引位置。若主串中不存在该模式串，则返回-1
     */
    public static int kmpSearch(String destStr, String subStr) {
```

```
        // 0.计算 next 数组
        int[] next = getNext(subStr);
        // 1.定义一个指针，用于指向主串中的第一个字符
        int x = 0;
        // 2.定义一个指针，用于指向模式串中的第一个字符
        int y = 0;
        // 3.定义一个循环，用于实现字符串的匹配操作
        while (x < destStr.length() && y < subStr.length()) {
            // 4.如果相等，则继续逐个比较后续字符
            if (y == -1 || destStr.charAt(x) == subStr.charAt(y)) {
                x++;
                y++;
            }
            // 5.如果不相等，则从主串的下一个字符起，从新与模式串的第一个字符进行比较
            else {
                // x = x - y + 1;
                y = next[y];
            }
        }
        // 6.如果模式串遍历完毕，则证明匹配成功，否则就证明匹配失败
        if (y == subStr.length()) {
            return x - y;
        }
        else {
            return -1;
        }
    }

    /**
     * 计算 next 数组
     * @param subStr 模式串
     * @return 返回 next 数组
     */
    public static int[] getNext(String subStr) {
        // 1.定义一个 next 数组
        int[] next = new int[subStr.length()];
        // 2.设置 next 数组的第一个元素值为-1
        next[0] = -1;
        // 3.定义两个变量
        int y = 0, len = -1;
        // 4.定义一个循环，用于计算 next 数组
        while (y < subStr.length() - 1) {
            // 5.处理 len 和 y 指向模式串中的字符相等的情况
            if (len == -1 || subStr.charAt(len) == subStr.charAt(y)) {
                y++;
                len++;
                next[y] = len;
            }
            // 6.处理 len 和 y 指向模式串中的字符不相等的情况
            else {
```

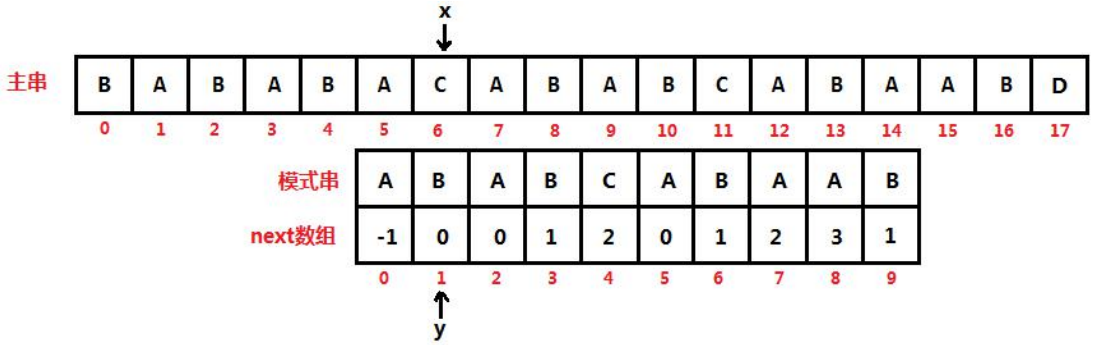
```
                len = next[len];
            }
        }
        // 7.返回计算出来的 next 数组
        return next;
    }
}
```

- **KMP 算法的优化**

接下来，我们来看一下上边的算法存在的缺陷，如下所示：



当 x 指针和 y 指针指向的字符不相等时，则我们应该是把 y 指针指向模式串索引为 1 的字符，然后继续和 x 指针指向的字符做比较。



不难发现，这一步是完全没有意义的，因为后面的字符 B 已经不匹配了，那前面的字符 B 也一定是不匹配的。

解决这个问题也很简单，我们只需在 getNext()方法中添加一个判断即可，代码如下所示：

```
/**
 * 计算 next 数组
 * @param subStr 模式串
 * @return 返回 next 数组
 */
public static int[] getNext(String subStr) {
    // 1.定义一个 next 数组
    int[] next = new int[subStr.length()];
    // 2.设置 next 数组的第一个元素值为-1
    next[0] = -1;
    // 3.定义两个变量
    int y = 0, len = -1;
    // 4.定义一个循环，用于计算 next 数组
```

```
    while (y < subStr.length() - 1) {
        // 5.处理 len 和 y 指向模式串中的字符相等的情况
        if (len == -1 || subStr.charAt(len) == subStr.charAt(y)) {
            // 如果 len 和 y 指向的字符相等，那么就执行 next[y] = next[len]
            if (subStr.charAt(++len) == subStr.charAt(++y)) {
                next[y] = next[len];
            }
            else {
                next[y] = len;
            }
        }
        // 6.处理 len 和 y 指向模式串中的字符不相等的情况
        else {
            len = next[len];
        }
    }
    // 7.返回计算出来的 next 数组
    return next;
}
```

2. 矩阵

在实际应用中，经常出现一些阶数很高的矩阵，常见的有特殊矩阵和稀疏矩阵。**特殊矩阵**（special matrix）指的是矩阵中有很多值相同的元素并且它们的分布有一定的规律。**稀疏矩阵**（sparse matrix）指的是矩阵中有很多零元素。我们对这些矩阵进行压缩存储，从而就可以节约存储空间，并且还能使矩阵的各种运算能有效的进行。

矩阵压缩存储的基本实现就是：（一）为多个值相同的元素只分配一个存储空间；（二）对零元素不分配存储空间。

2.1 特殊矩阵

2.1.1 对称矩阵的压缩存储

- **对称矩阵的介绍**

对称矩阵（Symmetric Matrices）是指以主对角线为对称轴，各元素对应相等的矩阵。如下图所示，就是一个 5 阶对称矩阵。

	0	1	2	3	4
0	3	6	4	7	8
1	6	2	8	4	2
2	4	8	1	6	9
3	7	4	6	0	5
4	8	2	9	5	7

假设对称矩阵是 n 阶矩阵，则必有 arr[i][j] = arr[j][i](0≤i≤n-1，0≤j≤n-1)的特点。

- **对称矩阵的压缩存储原理**

对称矩阵关于主对角线对称，因此我们只需要存储下三角形部分（包含对角线）即可。对于一个 n 阶对称矩阵，原来需要 n*n 个存储单元，现在只需要 n*(n+1)/2 个存储单元，节约了大约一半的存储空间，当 n 较大时，这是客观的一部分存储资源。

如何只存储下三角部分的元素呢？由于下三角形共有 n*(n+1)/2 个元素，因此可将这些元素按行存储到一个一维数组中即可，如下图所示：

这样，下三角中的元素 arr[i][j]（i≥j）就存储到一维数组中了。当 i≥j 时，此时一维数组中下标 k 和 i、j 的关系为：k = i*(i + 1)/2 + j。例如，对称矩阵中元素 arr[4][3]对应一维数组中的索引就为 13。当 i<j 时，此时一维数组中下标 k 和 i、j 的关系为：k = j*(j + 1)/2 + i。例如，对称矩阵中元素 arr[3][4]对应一维数组中的索引就为 13。

- **对称矩阵压缩存储的代码实现**

```
package com.bjsxt.p2.specialmatrix;

import java.util.Arrays;

/**
 * 对称矩阵的压缩存储
 */
public class Test01 {
    public static void main(String[] args) {
        // 定义一个 5 阶的对称矩阵
        int[][] matrix = {{3, 6, 4, 7, 8},
                          {6, 2, 8, 4, 2},
                          {4, 8, 1, 6, 9},
                          {7, 4, 6, 0, 5},
                          {8, 2, 9, 5, 7}};
        // 测试对称矩阵的压缩存储操作
        int[] arr = compress(5, matrix);
        System.out.println(Arrays.toString(arr));
```

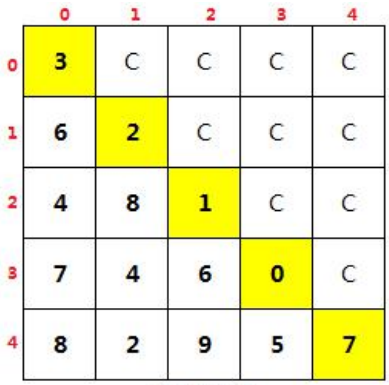
```
        // 测试对称矩阵的解压缩操作
        int[][] matrix2 = decompression(5, arr);
        for (int[] a : matrix2) {
            System.out.println(Arrays.toString(a));
        }
    }

    /**
     * 用于实现对称矩阵的解压缩操作
     * @param order 对称矩阵的阶数
     * @param arr 压缩存储之后的一维数组
     * @return 解压缩之后的对称矩阵
     */
    public static int[][] decompression(int order, int[] arr) {
        // 1.定义一个矩阵，用于保存解压缩之后的结果
        int[][] matrix = new int[order][order];
        // 2.定义一个嵌套循环，用于给 matrix 实现赋值的操作
        for (int i = 0; i < order; i++) {
            for (int j = 0; j < order; j++) {
                // 3.实现对称矩阵的解压缩操作
                if (i >= j) { // k = i*(i + 1)/2 + j
                    matrix[i][j] = arr[i*(i + 1)/2 + j];
                }
                else { // k = j*(j + 1)/2 + i
                    matrix[i][j] = arr[j*(j + 1)/2 + i];
                }
            }
        }
        // 4.返回解压缩之后的对称矩阵
        return matrix;
    }

    /**
     * 用于实现对称矩阵的压缩存储操作
     * @param order 对称矩阵的阶数
     * @param matrix 需要压缩存储的对称矩阵
     * @return 返回压缩存储之后的一维数组
     */
    public static int[] compress(int order, int[][] matrix) {
        // 1.定义一个一维数组，用于保存压缩存储之后的结果
        int[] arr = new int[order*(order+1)/2];
        // 2.定义一个嵌套循环，用于获得对称矩阵中下三角形及对角线中的元素
        for (int i = 0; i < order; i++) {
            for (int j = 0; j <= i; j++) {
                // 3.实现对称矩阵的压缩存储操作
                arr[i*(i + 1)/2 + j] = matrix[i][j];
            }
        }
        // 4.返回压缩存储之后的一维数组
        return arr;
```

```
    }
}
```

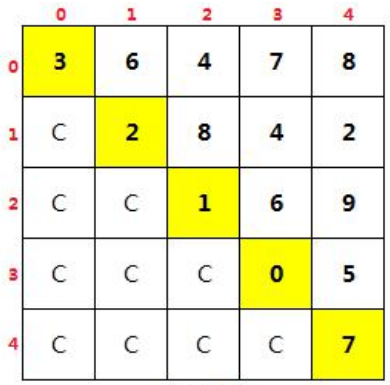
2.1.2 三角矩阵的压缩存储

- **三角矩阵的介绍**

以主对角线划分三角矩阵有**下三角矩阵**和**上三角矩阵**。下三角矩阵：主对角线以上部分均为同一个常数，如下图所示。上三角矩阵：主对角线以下部分均为同一个常数，如下图所示。



下三角矩阵



上三角矩阵

- **三角矩阵的压缩存储原理**

下三角矩阵的压缩存储与对称矩阵类似，不同之处仅在于除了要存储下三角形以及主对角线中的元素以外，还要存储主对角线上方的常数。因为是同一个常数，所以只存储一个即可。对于一个 n 阶下三角矩阵，则一共存储了 n*(n+1)/2+1 个元素。然后，将下三角形以及主对角线中的元素按行存储入一维数组中，最后再存储主对角线上方的常数，如下图所示：

上三角矩阵的压缩存储思想与下三角矩阵类似，依旧是按行存储上三角形以及主对角线中的元素，最后存储主对角线下方的常数。

- **上三角矩阵压缩存储的代码实现**

```
package com.bjsxt.p2.specialmatrix;

import java.util.Arrays;

/**
 * 下三角矩阵的压缩存储
 */
public class Test02 {
    public static void main(String[] args) {
        // 定义一个 5 阶的下三角矩阵
        int[][] matrix = {{3, 8, 8, 8, 8},
                          {6, 2, 8, 8, 8},
```

```
                        {4, 8, 1, 8, 8},
                        {7, 4, 6, 0, 8},
                        {8, 2, 9, 5, 7}};
        // 测试下三角矩阵的压缩存储操作
        int[] arr = compress(5, matrix);
        System.out.println(Arrays.toString(arr));
        // 测试下三角矩阵的解压缩操作
        int[][] matrix2 = decompression(5, arr);
        for (int[] a : matrix2) {
            System.out.println(Arrays.toString(a));
        }
    }

    /**
     * 用于实现下三角矩阵的解压缩操作
     * @param order 下三角矩阵的阶数
     * @param arr 压缩存储之后的一维数组
     * @return 解压缩之后的下三角矩阵
     */
    public static int[][] decompression(int order, int[] arr) {
        // 1.定义一个矩阵，用于保存解压缩之后的结果
        int[][] matrix = new int[order][order];
        // 2.定义一个嵌套循环，用于给 matrix 实现赋值的操作
        for (int i = 0; i < order; i++) {
            for (int j = 0; j < order; j++) {
                // 3.实现下三角矩阵的解压缩操作
                if (i >= j) { // k = i*(i + 1)/2 + j
                    matrix[i][j] = arr[i*(i + 1)/2 + j];
                }
                else {
                    matrix[i][j] = arr[order*(order+1)/2];
                }
            }
        }
        // 4.返回解压缩之后的下三角矩阵
        return matrix;
    }

    /**
     * 用于实现下三角矩阵的压缩存储操作
     * @param order 下三角矩阵的阶数
     * @param matrix 需要压缩存储的下三角矩阵
     * @return 返回压缩存储之后的一维数组
     */
    public static int[] compress(int order, int[][] matrix) {
        // 1.定义一个一维数组，用于保存压缩存储之后的结果
        int[] arr = new int[order*(order+1)/2 + 1];
        // 2.定义一个嵌套循环，用于获得下三角矩阵中下三角形及对角线中的元素
```

```
        for (int i = 0; i < order; i++) {
            for (int j = 0; j <= i; j++) {
                // 3.实现下三角矩阵的压缩存储操作
                arr[i*(i + 1)/2 + j] = matrix[i][j];
            }
        }
        // 4.保存主对角线上面的常数
        arr[order*(order+1)/2] = matrix[0][1];
        // 5.返回压缩存储之后的一维数组
        return arr;
    }
}
```

2.1.3 对角矩阵的压缩存储

- **对角矩阵的介绍**

在对角矩阵中，所有非零元素都集中在以主对角线为中心的带状区域，除了主对角线和若干条次对角线的元素之外，其余位置的元素都为零，如下图所示：

	0	1	2	3	4
0	3	6	0	0	0
1	9	2	5	0	0
2	0	7	1	6	0
3	0	0	4	0	2
4	0	0	0	8	7

假设对角矩阵是 n 阶矩阵，变量 i（0≤i≤n-1）和 j（0≤j≤n-1）代表就是矩阵中的索引，则满足|i - j|≤1 的元素就是带状区域的元素。

- **对角矩阵的压缩存储原理**

对角矩阵的压缩存储就是将带状区域的元素逐行存储到一维数组中，也就是把|i - j|≤1 位置的元素存入到一维数组中。假设在一个 n 阶对角矩阵中，每一行带状区域的元素最多为 3 个，则带状区域一共就有(3-1)*2+(n-2)*3 个元素，然后将其按行存入一维数组中，如下图所示：

这样，对角矩阵中的带状区域元素就存储到一维数组中了，在一维数组中下标 k 和 i、j 的关系为：k = 2*i+j。例如，对称矩阵中元素 arr[4][3]对应一维数组中的索引就为 11。

- **对角矩阵压缩存储的代码实现**

```
package com.bjsxt.p2.specialmatrix;
```

```java
import java.util.Arrays;

/**
 * 对角矩阵的压缩存储
 */
public class Test03 {
    public static void main(String[] args) {
        // 定义一个 5 阶的对角矩阵
        int[][] matrix = {{3, 6, 0, 0, 0},
                          {9, 2, 5, 0, 0},
                          {0, 7, 1, 6, 0},
                          {0, 0, 4, 0, 2},
                          {0, 0, 0, 8, 7}};
        // 测试对角矩阵的压缩存储操作
        int[] arr = compress(5, matrix);
        System.out.println(Arrays.toString(arr));
        // 测试对角矩阵的解压缩操作
        int[][] matrix2 = decompression(5, arr);
        for (int[] a : matrix2) {
            System.out.println(Arrays.toString(a));
        }
    }

    /**
     * 用于实现对角矩阵的解压缩操作
     * @param order 对角矩阵的阶数
     * @param arr 压缩存储之后的一维数组
     * @return 解压缩之后的对角矩阵
     */
    public static int[][] decompression(int order, int[] arr) {
        // 1.定义一个矩阵，用于保存解压缩之后的结果
        int[][] matrix = new int[order][order];
        // 2.定义一个嵌套循环，用于给 matrix 实现赋值的操作
        for (int i = 0; i < order; i++) {
            for (int j = 0; j < order; j++) {
                // 3.实现对角矩阵的解压缩操作
                if (Math.abs(i - j) <= 1) {
                    matrix[i][j] = arr[2*i+j];
                }
            }
        }
        // 4.返回解压缩之后的对角矩阵
        return matrix;
    }

    /**
     * 用于实现对角矩阵的压缩存储操作
```

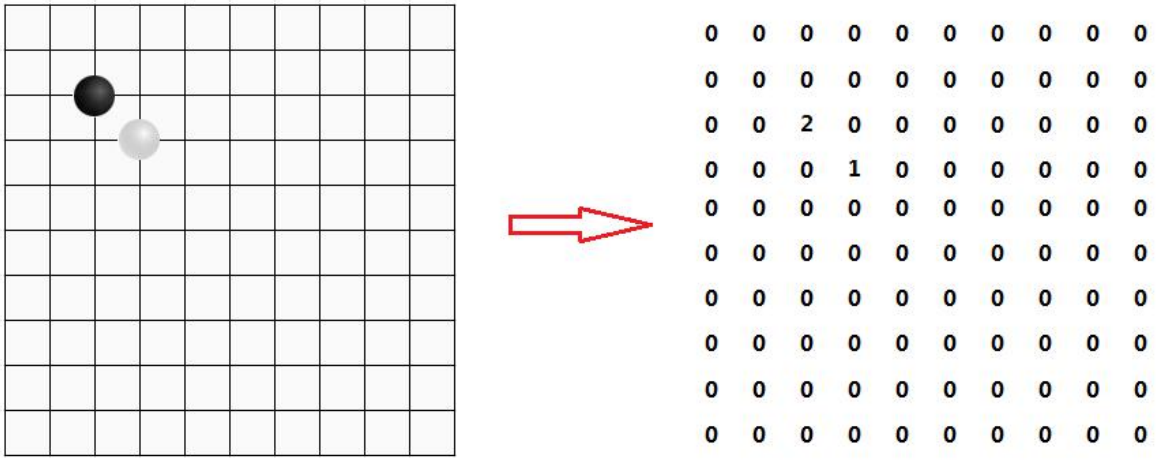
```
     * @param order 对角矩阵的阶数
     * @param matrix 需要压缩存储的对角矩阵
     * @return 返回压缩存储之后的一维数组
     */
    public static int[] compress(int order, int[][] matrix) {
        // 1.定义一个一维数组，用于保存压缩存储之后的结果
        int[] arr = new int[(3-1)*2+(order-2)*3];
        // 2.定义一个嵌套循环，用于获得对角矩阵中的所有元素
        for (int i = 0; i < order; i++) {
            for (int j = 0; j < order; j++) {
                // 3.实现对角矩阵的压缩存储操作
                if (Math.abs(i - j) <= 1) {
                    arr[2*i+j] = matrix[i][j];
                }
            }
        }
        // 4.返回压缩存储之后的一维数组
        return arr;
    }
}
```

2.2 稀疏矩阵

2.2.1 稀疏矩阵的简介

需求：编写一个五子棋程序，要求保存棋盘中每个棋子的位置，我们该如何实现呢？

解决：把棋盘当成一个矩阵（二维数组），然后通过矩阵来保存棋盘中每个棋子的位置，在矩阵中使用 1 代表白子，使用 2 代表黑子，其余位置则使用 0 来表示。



通过分析保存棋盘记录的矩阵，我们发现数值为 0 的元素数目远远多于非 0 元素的数目，并且非 0 元素分布没有任何规律，因此这种矩阵我们称之为**稀疏矩阵**（sparse matrix）。与之相反，若非 0 元素数目占大多数时，则称该矩阵为**稠密矩阵**（dense matrix）。

稀疏因子是用于描述稀疏矩阵非零元素的比例情况。假设一个 n*m 的矩阵中拥有 t 个非零元素，则稀疏因子δ的计算公式如下：δ=t/(n*m)。当计算出来稀疏因子的值小于等于 0.05 时，那么这个矩阵我们就可以称之为稀疏矩阵。

2.2.2 稀疏矩阵的压缩存储

存储矩阵的一般方法是采用二维数组，其优点是可以随机地访问每一个元素，因而能够较容易的实现矩阵的各种运算。但对于稀疏矩阵而言，若用二维数组来表示，会重复存储了很多个零元素，从而浪费了存储空间。因此，我们需要对稀疏矩阵进行压缩存储，以避免资源的不必要的浪费。

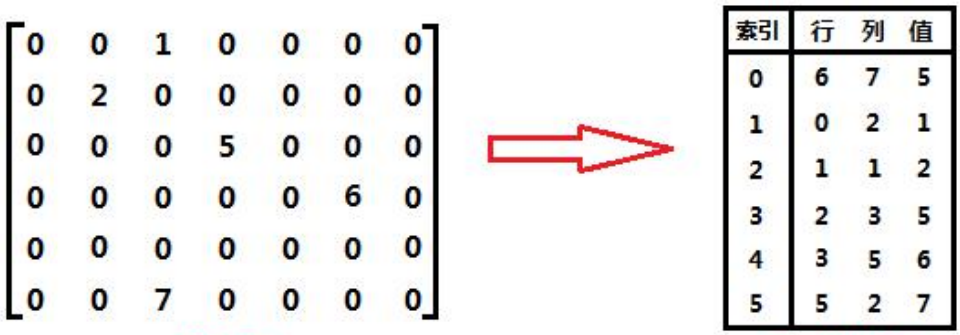
稀疏矩阵最常见的压缩存储方式有：三元组顺序表和十字链表法。

2.2.2.1 三元组顺序表

- **三元组顺序表压缩存储原理**

三元组顺序表存储数据的策略是只存储非零元素，但是稀疏矩阵中非零元素的分布是没有任何规律的。因此，在这种情况下，它的存储方案就是：分别存储每个非零元素的行索引（row）、列索引（col）和非零元素值（value）。

为了更可靠的描述（也为了方便实现解压缩的操作），我们需要再加一个“总体”信息概述，也就是在三元组顺序表的第一行存储：总行数、总列数和非零元素个数。如下所示：



如上图所示，我们把稀疏矩阵转换为三元组顺序表来存储后，原来需要 6*7 的存储空间，而压缩之后就只需要 7*3 的存储空间了，从而大大的节省了存储空间，避免资源的不必要的浪费。

- **三元组顺序表的代码实现**

掌握了三元组顺序表压缩存储稀疏矩阵的原理后，接下来我们来实现稀疏矩阵的压缩存储操作和三元组顺序表的解压缩存储操作，代码实现如下：

```
package com.bjsxt.p3.sparsematrix;

import java.util.Arrays;

/**
 * 稀疏矩阵压缩存储之三元组顺序表
 */
```

```java
public class Test01 {
    public static void main(String[] args) {
        // 定义一个 6 行 7 列的稀疏矩阵（一共有 5 个非零元素）
        int[][] matrix = {/*0  1  2  3  4  5  6*/
                    /*0*/ {0, 0, 1, 0, 0, 0, 0},
                    /*1*/ {0, 2, 0, 0, 0, 0, 0},
                    /*2*/ {0, 0, 0, 5, 0, 0, 0},
                    /*3*/ {0, 0, 0, 0, 0, 6, 0},
                    /*4*/ {0, 0, 0, 0, 0, 0, 0},
                    /*5*/ {0, 0, 7, 0, 0, 0, 0}};
        // 稀疏矩阵转化为三元组顺序表
        int[][] result = compress(matrix);
        for (int[] a : result) {
            System.out.println(Arrays.toString(a));
        }
        // 三元组顺序表转化为稀疏矩阵
        int[][] matrix2 = decompression(result);
        for (int[] a : matrix2) {
            System.out.println(Arrays.toString(a));
        }
    }

    /**
     * 把三元组顺序表转化为稀疏矩阵
     * @param result 三元组顺序表
     * @return 返回解压缩之后的稀疏矩阵
     */
    public static int[][] decompression(int[][] result) {
        // 1.获得稀疏矩阵中的总行数和总列数
        int rows = result[0][0], cols = result[0][1];
        // 2.定义一个矩阵，用于保存解压缩之后的结果
        int[][] matrix = new int[rows][cols];
        // 3.获得三元组顺序表中的非零元素信息，然后将非零元素存入 matrix 矩阵中
        for (int i = 1; i < rows; i++) {
            matrix[result[i][0]][result[i][1]] = result[i][2];
        }
        // 4.返回解压缩之后的稀疏矩阵
        return matrix;
    }

    /**
     * 把稀疏矩阵转化为三元组顺序表
     * @param matrix 需要压缩存储的稀疏矩阵
     * @return 返回压缩存储后的三元组顺序表
     */
    public static int[][] compress(int[][] matrix) {
        // 1.获得稀疏矩阵中的总行数和总列数
        int rows = matrix.length, cols = matrix[0].length;
```

```
        // 2.获得稀疏矩阵中的非零元素个数
        int sum = 0;
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                if (matrix[i][j] != 0) {
                    sum++;
                }
            }
        }
        // 3.定义一个三元组顺序表
        int[][] result = new int[sum + 1][3];
        // 4.设置三元组顺序表的总体概述信息
        result[0][0] = rows;
        result[0][1] = cols;
        result[0][2] = sum;
        // 5.获得稀疏矩阵中非零元素的信息，然后将其存入三元组顺序表中
        int idx = 1;
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                if (matrix[i][j] != 0) {
                    // 把非零元素的信息存入到三元组顺序表中
                    result[idx][0] = i;
                    result[idx][1] = j;
                    result[idx][2] = matrix[i][j];
                    // 更新 idx 变量的值
                    idx++;
                }
            }
        }
        // 6.返回三元组顺序表
        return result;
    }
}
```

2.2.2.2 十字链表法

- **十字链表法的压缩存储原理**

使用三元组顺序表来表示的稀疏矩阵，比起使用原始的二维数组来存储，节约了存储空间。但是，归根结底是依旧是使用数组存储稀疏矩阵，因为数组不利于执行“插入和删除”操作。因此，使用三元组顺序表不适合解决向矩阵中“添加或删除非 0 元素”的操作。

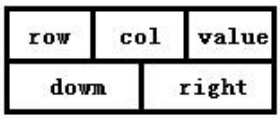
例如，A 和 B 分别为两个矩阵，我们执行将矩阵 B 加到矩阵 A 的操作，则矩阵 A 中的元素会发生很大的变化，之前的非 0 元素可能变为 0，而 0 元素也可能变为非 0 元素。此时，若还用三元组顺序表来存储，势必会为了保持三元组顺序表“以行序为主序”的特点，从而大量移动元素。

十字链表采用的是“数组+链表”的结构来存储稀疏矩阵，它能够灵活的插入因运算而产生的新的非 0 元素、删除因运算而产生的新的 0 元素，从而高效的实现矩阵的各种运算。

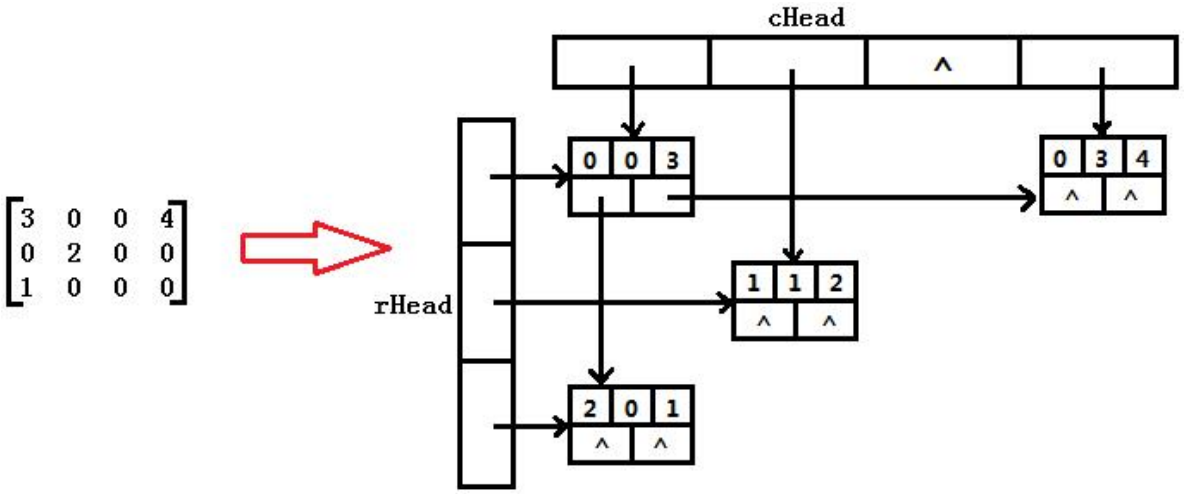
在十字链表中，矩阵的每一个非零元素用一个结点表示，该结点除了（row，col，value）以外，还存在着以下两个指针：

- right：用于链接同一行中的下一个非零元素。
- down：用以链接同一列中的下一个非零元素。

整个结点的结构如下图所示：



在十字链表中，我们定义一个存放所有行链表首节点的的一维数组（rHead），和一个存放所有列链表首节点的的一维数组(cHead)。同一行的非零元素通过 right 指针链接成一个单链表，同一列的非零元素通过 down 指针链接成一个单链表。这样，矩阵中任意一个非零元素 arr[i][j]所对应的结点既处在第 i 行的行链表上，又处在第 j 列的列链表上，这好像是处在一个十字交叉路口上，所以称其为十字链表。



➢ **十字链表法的代码实现**

十字链表类的代码实现

```
package com.bjsxt.p2.specialmatrix;

/**
 * 十字链表法
 */
public class CrossList {
    /**
     * 保存稀疏矩阵的总行数
     */
    private int rows;
    /**
     * 保存稀疏矩阵的总列数
     */
    private int cols;
```

```
    /**
     * 保存稀疏矩阵中的非零元素的个数
     */
    private int size;
    /**
     * 保存行链表首节点的一维数组
     */
    private Node[] rHead;
    /**
     * 保存列链表首节点的一维数组
     */
    private Node[] cHead;

    /**
     * 返回非零元素的个数
     * @return
     */
    public int size() {
        return this.size;
    }

    /**
     * 构造方法
     * @param matrix 需要压缩的稀疏矩阵
     */
    public CrossList(int[][] matrix) {
        // 初始化操作
        this.rows = matrix.length;
        this.cols = matrix[0].length;
        this.rHead = new Node[rows];
        this.cHead = new Node[cols];
        // 遍历稀疏矩阵中的所有元素
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                // 把非零元素存入到十字链表中
                if (matrix[i][j] != 0) {
                    insert(i, j, matrix[i][j]);
                }
            }
        }
    }

    /**
```

```
     * 将十字链表转化为稀疏矩阵
     * @return
     */
    public int[][] toSparseArray() {
        // 1.创建一个稀疏矩阵
        int[][] matrix = new int[rows][cols];
        // 2.遍历行链表数组，并实现对稀疏矩阵的赋值操作
        for (int i = 0; i < rHead.length; i++) {
            // 获得第 i 行链表的首节点
            Node head = rHead[i];
            // 遍历以 head 为首节点的单链表
            while (head != null) {
                // 实现稀疏矩阵的赋值操作
                matrix[head.row][head.col] = head.value;
                // 更新 head 节点
                head = head.right;
            }
        }
        // 3.返回稀疏矩阵
        return matrix;
    }

    /**
     * 向十字链表中插入一个节点
     * @param row 非零元素的行索引
     * @param col 非零元素的列索引
     * @param value 非零元素值
     */
    public void insert(int row, int col, int value) {
        // 1.创建一个节点对象
        Node node = new Node(row, col, value);
        // 2.将 node 节点插入到 row 行单链表的末尾
        // 2.1 处理 row 行单链表为空表的情况
        if (rHead[row] == null) {
            rHead[row] = node;
        }
        // 2.2 处理 row 行单链表不是空表的情况
        else {
            // 获得 row 行单链表的尾节点
            Node tempNode = rHead[row];
            while (tempNode.right != null) {
                tempNode = tempNode.right;
            }
```

```
            // 设置 tempNode 的 right 值为 node
            tempNode.right = node;
        }
        // 3.将 node 节点插入到 col 列单链表的末尾
        // 3.1 处理 col 列单链表为空表的情况
        if (cHead[col] == null) {
            cHead[col] = node;
        }
        // 3.2 处理 col 列单链表不是空表的情况
        else {
            // 获得 col 列单链表的尾节点
            Node tempNode = cHead[col];
            while (tempNode.down != null) {
                tempNode = tempNode.down;
            }
            // 设置 tempNode 的 down 值为 node
            tempNode.down = node;
        }
        // 4.更新 size 的值
        size++;
    }

    /**
     * 十字链表节点类
     */
    public static class Node {
        /**
         * 非零元素行索引
         */
        private int row;
        /**
         * 非零元素列索引
         */
        private int col;
        /**
         * 非零元素值
         */
        private int value;
        /**
         * 指向同一行的下一个非零元素节点
         */
        private Node right;
        /**
```

```
         * 指向同一列的下一个非零元素节点
         */
        private Node down;

        /**
         * 构造方法
         * @param row
         * @param col
         * @param value
         */
        public Node(int row, int col, int value) {
            this.row = row;
            this.col = col;
            this.value = value;
        }
    }
}
```

测试类的代码实现

```
package com.bjsxt.p2.specialmatrix;
import java.util.Arrays;

public class Test01 {
    public static void main(String[] args) {
        // 定义一个稀疏矩阵
        int[][] matrix = {/*0  1  2  3*/
                    /*0*/{3, 0, 0, 4},
                    /*1*/{0, 2, 0, 0},
                    /*2*/{1, 0, 0, 0}};
        // 创建一个十字链表对象，并实现稀疏矩阵的压缩操作
        CrossList list = new CrossList(matrix);
        // 实现稀疏矩阵的解压缩操作
        int[][] sparseArray = list.toSparseArray();
        for (int[] arr : sparseArray) {
            System.out.println(Arrays.toString(arr));
        }
    }
}
```