第 5 章　数组和广义表

前几章讨论的线性结构中的数据元素都是非结构的原子类型，元素的值是不再分解的。本章讨论的两种数据结构——数组和广义表，可以看成是线性表在下述含义上的扩展：表中的数据元素本身也是一个数据结构。

数组是读者已经很熟悉的一种数据类型，几乎所有的程序设计语言都把数组类型设定为固有类型。本章以抽象数据类型的形式讨论数组的定义和实现，使读者加深对数组类型的理解。

5.1　数组的定义

类似于线性表，抽象数据类型**数组**可形式地定义为：

```
ADT Array {
    数据对象：D = { a_{j1 j2 … jn} | n(>0)称为数组的维数，b_i 是数组第 i 维的长度，
                                  j_i = 0, …, b_i - 1, i = 1, 2, …, n,
                                  j_i 是数组元素的第 i 维下标，a_{j1 j2 … jn} ∈ ElemSet }
    数据关系：R = { R_1, R_2, …, R_n }
              R_i = { < a_{j1 … ji … jn}, a_{j1 … ji+1 … jn} > |
                                  0 ≤ j_k ≤ b_k - 1, 1 ≤ k ≤ n 且 k ≠ i,
                                  0 ≤ j_i ≤ b_i - 2,
                                  a_{j1 … ji … jn}, a_{j1 … ji+1 … jn} ∈ D, i = 2, …, n }
    基本操作：
      NewArray(n, bound1, …, boundn);
        操作结果：若维数 n 和各维长度合法，则构建并返回相应的数组。
      FreeArray(A);
        操作结果：回收数组 A，须重新构建才能作为数组使用，返回 NULL。
      PutElem(A, e, index1, …, indexn);
        初始条件：A 是 n 维数组，e 为元素，随后是 n 个下标值。
        操作结果：若下标不超界，则将 e 的值赋给所指定的 A 的元素，并返回 OK。
      GetElem(A, index1, …, indexn);
        初始条件：A 是 n 维数组，随后是 n 个下标值。
        操作结果：若各下标不超界，则返回所指定的 A 的元素值。
} ADT Array
```

这是一个 C 语言风格的多维数组定义。从上述定义可见，n 维数组中含 $\prod_{i=1}^{n} b_i$ 个数据元素，每个元素都受 n 个关系的约束。在每个关系中，元素 $a_{j_1 j_2 \cdots j_n}$ $(0 \leqslant j_i \leqslant b_i - 2)$ 都有一个直接后继元素。因此，就其单个关系而言，这 n 个关系仍是线性关系。和线性表一样，所有的数据元素都必须属于同一数据类型。数组中的每个数据元素都对应于一组下标 $(j_1, j_2, \cdots, j_n)$，每个下标的取值范围是 $0 \leqslant j_i \leqslant b_i - 1$，$b_i$ 称为第 i 维的长度 $(i = 1, 2, \cdots, n)$。显

然，当 $n=1$ 时，n 维数组就退化为定长的线性表。反之，n 维数组也可以看成是线性表的推广。由此，也可以从另一个角度来定义 n 维数组。

可以把二维数组看成是这样一个定长线性表：它的每个数据元素也是一个定长线性表。例如，图 5.1(a)是一个二维数组，以 m 行 n 列的矩阵形式表示，它可以看成是一个线性表

$$A=(a_0\ a_1\ \cdots\ a_p)\quad p=m-1\ 或\ n-1$$

其中，每个数据元素 a_j 是一个如图 5.1(b)所示的列向量形式的线性表

$$a_j=(a_{0j},a_{1j},\cdots,a_{m-1,j})\quad 0\leqslant j\leqslant n-1$$

或者 a_i 是一个如图 5.1(c)所示的行向量形式的线性表

$$a_i=(a_{i0},a_{i1},\cdots,a_{i,n-1})\quad 0\leqslant i\leqslant m-1$$

$$\boldsymbol{A}_{m\times n}=\begin{bmatrix} a_{00} & a_{01} & a_{02} & \cdots & a_{0,n-1} \\ a_{10} & a_{11} & a_{12} & \cdots & a_{1,n-1} \\ \vdots & \vdots & \vdots & & \vdots \\ a_{m-1,0} & a_{m-1,1} & a_{m-1,2} & \cdots & a_{m-1,n-1} \end{bmatrix}$$

(a) 矩阵形式表示

$$\boldsymbol{A}_{m\times n}=\begin{bmatrix}\begin{bmatrix} a_{00} \\ a_{10} \\ \vdots \\ a_{m-1,0} \end{bmatrix}\begin{bmatrix} a_{01} \\ a_{11} \\ \vdots \\ a_{m-1,1} \end{bmatrix}\cdots\begin{bmatrix} a_{0,n-1} \\ a_{1,n-1} \\ \vdots \\ a_{m-1,n-1} \end{bmatrix}\end{bmatrix}$$

(b) 列向量的一维数组

$$\boldsymbol{A}_{m\times n}=((a_{00}a_{01}\cdots a_{0,n-1}),(a_{10}a_{11}\cdots a_{1,n-1}),\cdots,(a_{m-1,0}a_{m-1,1}\cdots a_{m-1,n-1}))$$

(c) 行向量的一维数组

图 5.1　二维数组示例

在 C 语言中，一个二维数组类型可以定义为其分量类型为一维数组类型的一维数组类型，也就是说

```
typedef  ElemType  Array2[m][n];
```

等价于

```
typedef  ElemType  Array1[n];
typedef  Array1  Array2[m];
```

同理，一个 n 维数组类型可以定义为其数据元素为 $n-1$ 维数组类型的一维数组类型。

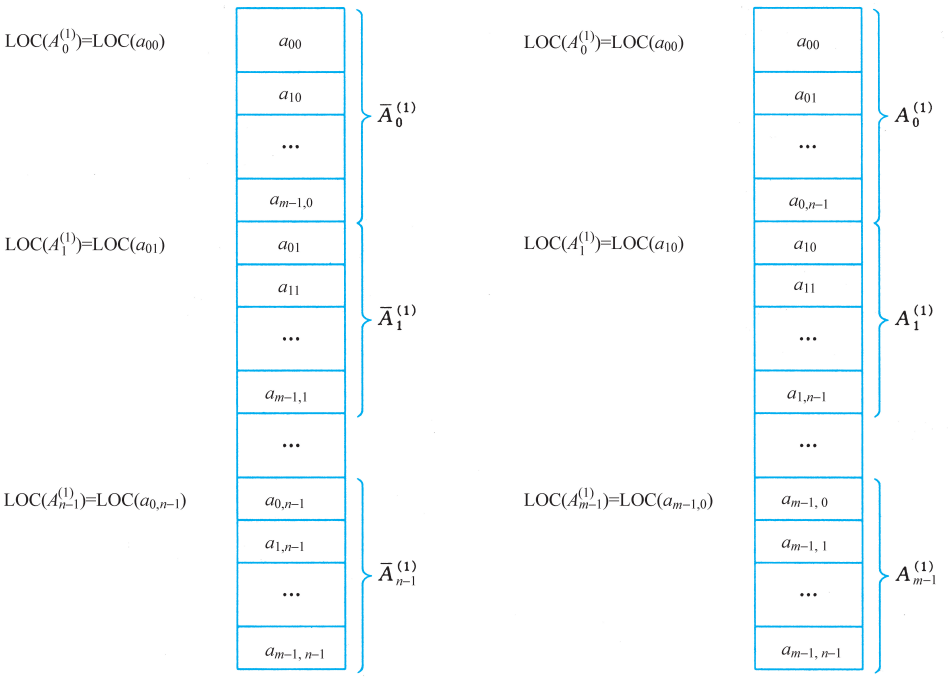
数组一旦被定义，它的维数和维界就不再改变。因此，除了结构的初建和回收之外，数组只有存取元素和修改元素值的操作。

5.2　数组的顺序表示和实现

数组一般不做插入或删除操作，一旦建立了数组，结构中的数据元素个数和元素之间的关系就不再发生变动。数组适合采用顺序存储结构表示。

内存的存储单元组织是一维结构，而数组是多维结构。用一组连续存储单元存放数组的数据元素就有个次序约定问题。例如图 5.1(a)的二维数组可以看成如图 5.1(c)的一维数组，也可看成如图 5.1(b)的一维数组。对应地，对二维数组可有两种存储方式：一种是以列序为主序(column major order)的存储方式，如图 5.2(a)所示；另一种是以行序为主序(row major order)的存储方式，如图 5.2(b)所示。在扩展 BASIC、PL/1、COBOL、Pascal、C/C++

和 Java 等绝大多数语言中，多维数组用的都是以行序为主序的存储结构，而在 FORTRAN 语言中，用的则是以列序为主序的存储结构。



(a) 以列序为主序　　(b) 以行序为主序

图 5.2　二维数组的两种存储方式

一旦规定了数组的维数和各维的长度，便可为它分配存储空间。只要给出一组下标便可求得相应数组元素的存储位置。下面仅用以行序为主序的存储结构为例予以说明。

假设每个数据元素占 L 个存储单元，则二维数组 $\boldsymbol{A}$ 中任一元素 a_{ij} 的存储位置可由下式确定

$$\mathrm{LOC}(i,j)=\mathrm{LOC}(0,0)+(b_2\times i+j)L \tag{5-1}$$

式中，LOC(i，j)是 a_{ij} 的存储位置；LOC(0,0)是 a_{00} 的存储位置，即二维数组 $\boldsymbol{A}$ 的起始存储位置，也称**基地址**或**基址**。

将式(5-1)推广到一般情况，可得到 n 维数组的数据元素存储位置的计算公式：

$$\begin{aligned}\mathrm{LOC}(j_1,j_2,\cdots,j_n)&=\mathrm{LOC}(0,0,\cdots,0)+(b_2\times\cdots\times b_n\times j_1+b_3\times\cdots\times b_n\times j_2\\&\quad+\cdots+b_n\times j_{n-1}+j_n)L\\&=\mathrm{LOC}(0,0,\cdots,0)+\Big(\sum_{i=1}^{n-1}j_i\prod_{k=i+1}^{n}b_k+j_n\Big)L\end{aligned}$$

可缩写成

$$\mathrm{LOC}(j_1,j_2,\cdots,j_n)=\mathrm{LOC}(0,0,\cdots,0)+\sum_{i=1}^{n}c_ij_i \tag{5-2}$$

其中，$c_n = L$，$c_{i-1} = b_i \times c_i$，$1 < i \leqslant n$。

式(5-2)称为 n 维数组的映像函数。容易看出，数组元素的存储位置是其下标的线性函数，一旦确定了数组的各维的长度，c_i 就是常数。由于计算各个元素存储位置的时间相等，所以存取数组中任一元素的时间也相等。我们称具有这一特点的存储结构为随机存储结构。

下面是数组的顺序存储表示和基本操作的函数原型。

```
//------数组的顺序存储表示 ------
#include <stdarg.h>                  // C 标准头文件，提供宏 va_start、va_arg 和 va_end
                                     // 用于存取变长参数表
#define MAX_ARRAY_DIM 8              // 假设数组维数的最大值为 8
typedef struct {
    ElemType   *base;                // 数组元素基址，由 ArrayNew 分配
    int        dim;                  // 数组维数
    int        *bounds;              // 数组维界基址，由 ArrayNew 分配
    int        *constants;           // 数组映像函数常量基址，由 ArrayNew 分配
} *Array;                            // 数组指针类型
//------基本操作的函数原型 ------
Array NewArray(int dim, …);          // 若维数 dim 和随后的各维长度合法，则构造并返回数组
Array FreeArray(Array A);            // 回收数组 A，返回 NULL
Status PutElem(Array A, ElemType e, …);
                                     // A 为 n 维数组，e 为元素，随后是 n 个下标值
                                     // 若下标不超界，则 e 值赋给 A 的指定元素
ElemType GetElem(Array A, …);        // A 为 n 维数组，随后是 n 个下标值
                                     // 若各下标不超界，则返回 A 的指定元素值
```

例 5-1 对于 C 语言的一个三维数组的声明：

```
int a[4][5][6];
```

可用定义的多维数组 Array 类型来声明变量 A 并调用构造函数得到对应的三维数组：

```
Array A =NewArray(3, 4, 5, 6);
```

图 5.3 给出了 A 的存储结构示例。

从函数原型可见，4 个中有 3 个参数表含 3 个点号。这是 C 语言的变长参数表，给函数接口提供了灵活性。调用函数时，对应的实参可以是 1 到多个。调用这 3 个函数时，对应数组维数，依次给出各维相应的维界(也称维长)或者下标。<stdarg.h>给出了可变参数表类型 va_list，以及获取参数相关的 3 个宏 va_start，va_arg 和 va_end。在对相关函数讲解时，可看到它们怎样运用。

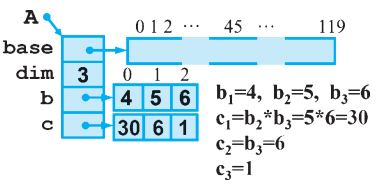


图 5.3 一个三维数组的存储结构

先来分析数组构造函数要做的工作。从图 5.3 可见，数组 A 的存储结构需要分配 4 块空间，首先是 A 指向的结构记录，余下 3 块分别由结构记录中的 3 个指针域 base、b 和 c 指向。base 指向的是元素存储空间，参数没有给出其容量，需要各维长度相乘才能得到。b 和 c 指向的空间分别存储维长和下标计算公式的常数，容量就是维数。常数需要按式(5-2)求

得。算法 5.1 实现了数组构造函数 NewArray 和回收函数 FreeArray。回收函数就是要释放数组已分配的 4 块空间,并置数组为 NULL(不再存在)。

```
Array NewArray(int dim, …) {   // 若维数 dim 和各维长度合法,则构造并返回相应的数组 A
    if (dim<1 || dim>MAX_ARRAY_DIM) return NULL;   // 若参数不合法,则返回 NULL 报错
    Array A;  int elemtotal=1, i;                  // elemtotal 用于求元素总值
    va_list ap;                                    // ap 指向…对应的首个实参
    if (!(A =(Array)malloc(sizeof( * A)))) exit(OVERFLOW);  // 分配数组的结构记录
    A->dim=dim;                                                // 维数
    if (!(A->bounds =(int * )malloc(dim * sizeof(int)))) exit(OVERFLOW);     // 维长向量
    va_start(ap, dim);          // ap 为 va_list 类型,是存放变长参数表信息的数组
    for (i=0; i<dim; ++i) {     // 依次取变长参数中的各维长度
        A->bounds[i]=va_arg(ap,int);
        if (A->bounds[i]<0) return NULL;     // 若维长是负数,则返回 NULL 报错
        elemtotal * =A->bounds[i];           // 累乘维长,求元素总数
    }                                      // 循环结束,求得元素总数,用于分配元素空间
    va_end(ap);                            // 取变长参数结束
    if (!(A->base =(ElemType * )calloc(elemtotal, sizeof(ElemType))))
        exit(OVERFLOW);
    if (!(A->constants =(int * )malloc(dim * sizeof(int))))      // 常数向量
          exit(OVERFLOW);
    // 以下求映像函数的常数 c,并存入 A->constants[i-1],i=1,2,…,dim
    A->constants[dim-1]=1;                                       // 常数 c1
    for (i=dim-2; i>=0; --i)
        A->constants[i]=A->bounds[i+1] * A->constants[i+1];  // 依次求 ci
    return A;                                  // 返回新建的数组
}
Array FreeArray(Array A) {  // 回收数组 A,返回 NULL
    if (!A) return NULL;
    free(A->base);  free(A->bounds);  free(A->constants);
                                               // 释放元素、维长和常数空间
    free(A);  return NULL;                     // 释放结构记录,返回 NULL
}
```

算法 5.1

如果说构造和回收是一对互逆的操作,那么对下标变量的赋值和取值也是一对互逆的操作。虽说互逆,但对下标变量赋值和取值的过程却是“大同小异”。它们从变长参数表提取下标并计算下标变量的相对地址是完全相同的,只是最后赋值和取值是“相逆”的。算法 5.2 是对函数的实现,它们都调用了辅助函数 CalcOff 求相对地址。

```
int CalcOff(Array A, va_list ap) {    // 辅助函数: PutElem 和 GetElem 调用此函数
    // 若 ap 指示的各下标值合法,则求出并返回该元素在 A 中的相对地址 off
    int i, idx, off=0;                 // off 用于累计相对地址
    for (i=0; i<A->dim; i++) {
        idx=va_arg(ap, int);           // 取第 i 个下标 ji
        if (idx<0 || idx>=A->bounds[i]) return -1;      // 若下标超界,返回-1 报错
        off+=A->constants[i] * idx;    // (ci * ji)加到 off
```

```
    }
    return off;                                  // 返回下标变量相对地址 off
}
Status PutElem(Array A, ElemType e, …) {
    // … 表示变长参数,依次为各维的下标值
    // 若各下标合法,则将 e 的值赋给 A 的指定元素
    if (!A) return ERROR;                        // 若 A 不存在,则返回 ERROR 报错
    va_list ap; va_start(ap,e);                  // ap 指示…对应的首个实参
    int off;
    if ((off=CalcOff(A, ap)) < 0)                // 若求得 off 值不正常,则返回 ERROR 报错
        return ERROR;
    * (A->base+off)=e;                           // 对指定的下标变量赋值
    return OK;
}
ElemType GetElem(Array A, …) {          // …依次为各维的下标值
                                        // 若各下标合法,则返回 A 的相应的元素值
    if (!A) return errV;                // 若 A 不存在,则返回 errV 报错
    va_list ap; va_start(ap, A);        // ap 指示…对应的首个实参
    int off;                            // 用于求相对地址
    if ((off=CalcOff(A, ap))<0) return errV;
                                        // 若求得 off 值不正常,则返回 errV 报错
    return * (A->base+off);             // 基址+相对地址,返回元素值
}
```

算法 5.2

5.3 矩阵的压缩存储

矩阵是很多科学与工程计算问题中研究和运用的数学对象。在此感兴趣的不是矩阵本身,而是如何存储矩阵元,使得矩阵的各种运算能有效地进行。

用高级程序设计语言编制程序时,几乎都用二维数组存储矩阵元。有的程序设计语言中提供了各种矩阵运算,使用方便。

然而,在数值分析中经常有一些高阶矩阵含许多值相同的元素或者是零元素。为了节省存储空间,可以对这类矩阵进行压缩存储,也就是为多个值相同的元只分配一个存储空间,对零元不分配空间。

假若值相同的元素或者零元素在矩阵中的分布有一定规律,则称此类矩阵为**特殊矩阵**;反之,称为稀疏矩阵。下面分别讨论它们的压缩存储。

5.3.1 特殊矩阵

若 n 阶矩阵 $\boldsymbol{A}$ 中的元满足下述性质

$$a_{ij}=a_{ji} \quad 1<i,j\leqslant n$$

则称为 n 阶**对称矩阵**。

对于对称矩阵,可以为每一对**对称元**分配一个存储空间,则可将 n^2 个元压缩存储到 $n(n+1)/2$ 个元的空间中。不失一般性,可以行序为主序存储其下三角(包括对角线)中

的元。

假设以一维数组 $sa[n(n+1)/2]$作为 n 阶对称矩阵 **A** 的存储结构，则 $sa[k]$和矩阵元 a_{ij}之间存在着一一对应的关系：

$$k=\begin{cases}\dfrac{i(i-1)}{2}+j-1 & i\geqslant j\\ \dfrac{j(j-1)}{2}+i-1 & i\leqslant j\end{cases} \tag{5-3}$$

对于任意给定一组下标(i,j)，均可在 sa 中找到矩阵元 a_{ij}，反之，对所有的 $k=0,1,2,\cdots,n(n+1)/2-1$，都能确定 $sa[k]$中的元在矩阵中的位置(i,j)。由此，称 $sa[n(n+1)/2]$为 n 阶对称矩阵 **A** 的压缩存储(见图 5.4)。

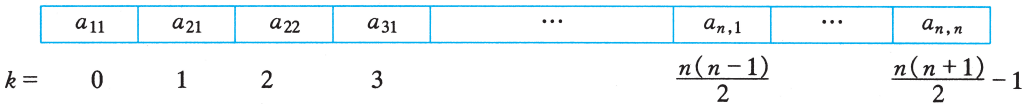
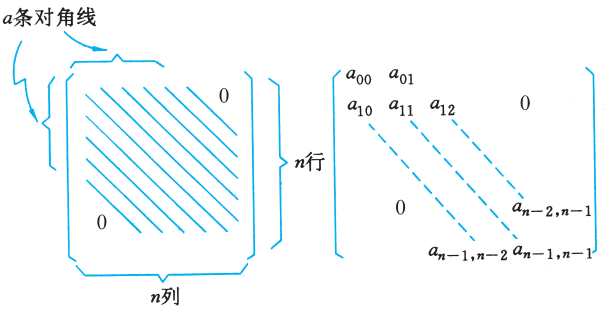


图 5.4　对称矩阵的压缩存储

这种压缩存储的方法同样也适用于三角矩阵。所谓**下(上)三角矩阵**是指矩阵的上(下)三角(不包括对角线)中的元均为常数 c 或零的 n 阶矩阵。则除了和对称矩阵一样，只存储其下(上)三角中的元之外，再加一个存储常数 c 的存储空间即可。

在数值分析中经常出现的还有另一类特殊矩阵是**对角矩阵**。在这种矩阵中，所有的非零元都集中在以主对角线为中心的带状区域中。即除了主对角线上和直接在对角线上、下方若干条对角线上的元之外，所有其他的元皆为零，如图 5.5 所示。对这种矩阵，也可按某个原则(或以行为主，或以对角线的顺序)将其压缩存储到一维数组上。



(a) 一般情形　　(b) 三对角矩阵

图 5.5　对角矩阵

在统称为特殊矩阵的这些矩阵中，非零元的分布都有明显规律，都可将其压缩存储到一维数组中，并找到每个非零元在一维数组中的对应关系。

然而，在实际应用中还经常会遇到另一类矩阵，其非零元较零元少很多，且分布没有规律，称为**稀疏矩阵**。这类矩阵的压缩存储就要比特殊矩阵复杂。这就是 5.3.2 节要讨论的问题。

5.3.2　稀疏矩阵

什么是**稀疏矩阵**？没有严格的定义，只是一个大致的表述。假设在 $m\times n$ 矩阵中，有 t

个元素不为零。令 $\delta=\frac{t}{m\times n}$，称 δ 为矩阵的**稀疏因子**。通常 $\delta\leqslant 0.05$ 时称该矩阵为稀疏矩阵。矩阵运算的种类很多，在稀疏矩阵的下列抽象数据类型定义中，只列举了几种常见的运算。

```
ADT SparseMat {
    数据对象: D = { a[i,j] | i=1,2,…,m; j=1,2,…,n;
                    a[i,j] ∈ ElemSet, m 和 n 分别为矩阵的行数和列数 }
    数据关系: R = { Row, Col }
              Row = { <a[i,j], a[i,j+1]> | 1≤i≤m, 1≤j≤n-1 }
              Col = { <a[i,j], a[i+1,j]> | 1≤i≤m-1, 1≤j≤n }
    基本操作:
        NewSMat();
            操作结果: 创建并返回稀疏矩阵。
        FreeSMat(M);
            初始条件: 稀疏矩阵 M 存在。
            操作结果: 回收稀疏矩阵 M。
        PrintSMat(M);
            初始条件: 稀疏矩阵 M 存在。
            操作结果: 输出稀疏矩阵 M。
        CopySMat(M);
            初始条件: 稀疏矩阵 M 存在。
            操作结果: 返回由 M 复制得到稀疏矩阵。
        AddSMat(M, N);
            初始条件: 稀疏矩阵 M 与 N 的行数和列数对应相等。
            操作结果: 求并返回稀疏矩阵的和 M+N。
        SubSMat(M, N);
            初始条件: 稀疏矩阵 M 与 N 的行数和列数对应相等。
            操作结果: 求并返回稀疏矩阵的差 M-N。
        MultSMat(M, N);
            初始条件: 稀疏矩阵 M 的列数等于 N 的行数。
            操作结果: 求并返回稀疏矩阵乘积 M×N。
        TransposeSMat(M);
            初始条件: 稀疏矩阵 M 存在。
            操作结果: 求并返回稀疏矩阵 M 的转置矩阵。
    } ADT SparseMat
```

如何对稀疏矩阵压缩存储呢？

按照压缩存储的概念，只存储稀疏矩阵的非零元。因此，除了存储非零元的值之外，还必须同时记下它所在行和列的位置(i,j)。反之，一个三元组(i,j,a_{ij})唯一确定了矩阵 $\boldsymbol{A}$ 的一个非零元。由此，稀疏矩阵可由表示非零元的三元组及其行列数唯一确定。例如，下列三元组表

$((1,2,12),(1,3,9),(3,1,-3),(3,6,14),(4,3,24),(5,2,18),(6,1,15),(6,4,-7))$

加上 6 和 7 这一对矩阵的行数、列数，便可作为图 5.6 中矩阵 $\boldsymbol{M}$ 的另一种描述。而由上述三元组表的不同存储表示可得到稀疏矩阵不同的压缩存储方法。

1. 三元组顺序表

以三元组作为顺序表的元素类型，就得到稀疏矩阵的一种压缩存储方式——**三元组顺**

$$M=\begin{bmatrix}0&12&9&0&0&0&0\\0&0&0&0&0&0&0\\-3&0&0&0&0&14&0\\0&0&24&0&0&0&0\\0&18&0&0&0&0&0\\15&0&0&-7&0&0&0\end{bmatrix}\qquad T=\begin{bmatrix}0&0&-3&0&0&15\\12&0&0&0&18&0\\9&0&0&24&0&0\\0&0&0&0&0&-7\\0&0&0&0&0&0\\0&0&14&0&0&0\\0&0&0&0&0&0\end{bmatrix}$$

图 5.6　稀疏矩阵 M 和 T

序表，以下是其类型定义。存储分配辅助操作 allocTSMat 为一个含 t 个非零元的 $m\times n$ 矩阵分配存储空间，但未存入非零元三元组。由一个二维数组存储的稀疏矩阵构造其三元组顺序表或者在矩阵操作中构造三元组顺序表，都可以调用 allocTSMat 分配存储空间。

```
//-----稀疏矩阵的三元组顺序表存储表示 -----
#define MAX_ROW 100              // 矩阵最大行数
#define MAX_COL 100              // 矩阵最大列数
typedef struct {
    int        i, j;             // 该非零元的行下标和列下标
    ElemType   e;                // 非零元
} Triple;                        // 三元组类型
typedef struct {
    Triple     *nzElem;          // 动态分配的非零元三元组表,nzElem[0]未用
    int         m, n, t;         // 矩阵的行数、列数和非零元个数
    int         size;            // 三元组表容量
} *TSMat;                        // 三元组顺序表指针类型
TSMat allocTSMat(int m, int n, int t) {  // 存储分配辅助操作(未存储非零元)
    if (m<1 || m>MAX_ROW || n<1 || n>MAX_COL) return NULL; // 参数不合法则返回 NULL
    TSMat M;
    if (!(M = (TSMat)malloc(sizeof(*M))))              // 分配结构记录和三元组表空间
        exit(OVERFLOW);
    if (!(M->nzElem = (Triple*)malloc((t+1) * sizeof(Triple)))) exit(OVERFLOW);
    M->m=m;  M->n=n;  M->t=M->size=t;
                                 // 置行、列、三元组数和容量
    return M;
}
```

nzElem 中表示非零元的三元组是以行序为主序顺序排列的，在下面的讨论可看出这样做有利于进行某些矩阵运算。现在讨论在这种压缩存储结构下如何实现矩阵的转置运算。

转置运算是一种简单常见的矩阵运算。对于一个 $m\times n$ 矩阵 $\boldsymbol{M}$，它的转置矩阵 $\boldsymbol{T}$ 是一个 $n\times m$ 矩阵，且 $\boldsymbol{T}(i,j)=\boldsymbol{M}(j,i)$，$1\leqslant i\leqslant n$，$1\leqslant j\leqslant m$。图 5.6 中的矩阵 $\boldsymbol{M}$ 和 $\boldsymbol{T}$ 互为转置矩阵。

显然，一个稀疏矩阵的转置矩阵仍然是稀疏矩阵。假设 a 和 b 是 TSMat 型的变量，分别表示矩阵 $\boldsymbol{M}$ 和 $\boldsymbol{T}$。那么，如何由 a 得到 b 呢？

从分析 a 和 b 之间的差异可见，只要做到以下 3 点。

(1) 将矩阵的行列值相互交换。

(2) 将每个三元组中的 i 和 j 相互调换。

(3) 重排三元组之间的次序便可实现矩阵的转置。

前二点是容易做到的,关键是如何实现第三点。即如何使 b->nzElem 中的三元组是以 ***T*** 的行(***M*** 的列)为主序依次排列的。

a->nzElem

下标	*i*	*j*	*e*
1	1	2	12
2	1	3	9
3	3	1	−3
4	3	6	14
5	4	3	24
6	5	2	18
7	6	1	15
8	6	4	−7

b->nzElem

下标	*i*	*j*	*e*
1	1	3	−3
2	1	6	15
3	2	1	12
4	2	5	18
5	3	1	9
6	3	4	24
7	4	6	−7
8	6	3	14

可以有以下两种处理方法。

(1) 按照 b->nzElem 中三元组的次序依次在 a->nzElem 中找到相应的三元组进行转置。换句话说,按照矩阵 ***M*** 的列序来进行转置。为了找到 ***M*** 的每一列中所有的非零元素,需要对其三元组表 a->nzElem 从第一行起整个扫描一遍,由于 a->nzElem 是以 ***M*** 的行序为主序来存放每个非零元的,由此得到的恰是 b->nzElem 应有的顺序。其具体算法描述如算法 5.3 所示。

```
TSMat TransposeTSMat(TSMat M) { // 采用三元组顺序表存储,求并返回稀疏矩阵 M 的转置矩阵
    if (!M) return NULL;                                    // 若 M 不存在,则返回 NULL 报错
    int p, q, col;
    TSMat T=allocTSMat(M->n, M->m, M->t);
    if (T->t) {
        q=1;
        for (col=1; col<=M->n; ++col)                       // 按列号从小到依次复制非零元
            for (p=1; p<=M->t; ++p)                         // 依次扫描三元组
                if (M->nzElem[p].j==col) {                  // 若是当前列号的非零元
                    T->nzElem[q].i=M->nzElem[p].j;          // 复制三元组到转置矩阵
                    T->nzElem[q].j=M->nzElem[p].i;
                    T->nzElem[q].e=M->nzElem[p].e;
                    q++;                                    // q 指示转置矩阵的三元组表末尾加入位置
                }
    }
    return T;                                               // 返回转置矩阵
}
```

算法 5.3

分析这个算法,主要的工作是在 p 和 col 的两重循环中完成的,故算法的时间复杂度为 $O(n\times t)$,即和 ***M*** 的列数及非零元的个数的乘积成正比。一般矩阵的转置算法为

```
for (col=1; col<=nu; ++col)
    for (row=1; row<=mu; ++row)
        T[col][row]=M[row][col];
```

其时间复杂度为 $O(m\times n)$。当非零元的个数 t 和 $m\times n$ 同数量级时,算法 5.3 的时间复杂