第3章 栈与队列

栈和队列是在软件设计中常用的两种数据结构，它们的逻辑结构和线性表相同，都是线性、有序的结构。二者的特点在于运算受到了限制：栈按照“后进先出”的规则进行操作，队列按“先进先出”的规则进行操作，故称栈和队列是运算受限制的线性表。

本章将要介绍栈和队列的逻辑结构定义、两种存储结构（顺序存储结构和链式存储结构）的实现，以及在两种存储结构上如何实现栈和队列的基本运算。

3.1 栈

3.1.1 栈的概念与运算

1. 栈的定义

栈（stack）是一种只允许在一端进行插入和删除的线性表，它是一种操作受限的线性表。在表中只允许进行插入和删除的一端称为**栈顶**（top），另一端称为**栈底**（bottom）。栈的插入操作通常称为入栈或进栈（push），而栈的删除操作则称为出栈或退栈（pop）。当栈中无数据元素时，称为**空栈**。

根据栈的定义可知，栈顶元素总是最后入栈的，因而是最先出栈；栈底元素总是最先入栈的，因而也是最后出栈。这种线性表是按照后进先出（last in first out，LIFO）的原则组织数据的，因此，栈也称为“后进先出”的线性表。

图 3.1 是一个栈的示意图，通常用指针 top 指示栈顶的位置，用指针 bottom 指向栈底。栈顶指针 top 动态反映栈的当前位置。元素是以 $a_1, a_2, \cdots, a_n$ 的顺序进栈，退栈的次序却是 $a_n, a_{n-1}, \cdots, a_1$。

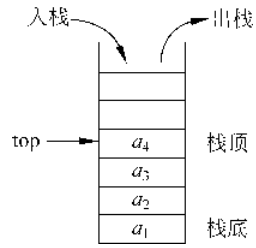

图 3.1 栈的存储与操作

2. 栈的操作

1）InitStack(S)

构造一个空栈 S。

2）StackEmpty(S)

栈的非空判断：若栈 S 不空，则返回 TRUE；否则，返回 FALSE。

3) StackFull(S)

判断栈满：若S为满栈，则返回TRUE，否则返回FALSE。注意：该运算只适用于栈的顺序存储结构。

4) Push(S,x)

进栈：若栈S不满，则将元素x压入S的栈顶，在顶部插入元素x。若栈满，则返回FALSE；否则，返回TRUE。

5) Pop(S)

出栈：若栈S不空，则返回栈顶元素，并从栈顶中删除该元素；否则，返回空元素NULL。

6) GetTop(S)

取栈顶元素值：若栈S不空，则返回栈顶元素的值；否则，返回空元素NULL。该操作与Pop(S)不同，它只是获得栈顶元素的值，该元素仍在栈顶不会改变。而Pop(S)是将栈顶元素的值读出，该元素同时从栈中移除。

栈的逻辑结构和先前学过的线性表相同，它是由多个数据元素形成的一种线性、有序的结构。如果它是非空的，则有且只有一个开始结点，有且只能有一个终端结点，其他结点前后所相邻的也只能是一个结点(直接前趋和直接后继)，但是栈的运算规则与线性表相比有更多的限制，栈(stack)是只能在一端进行插入和删除运算的线性表。

3.1.2　栈的存储方式

由于栈也是线性表，因此线性表的存储结构对栈也适用，通常栈有顺序栈和链栈两种存储结构，这两种存储结构的不同，则使得实现栈的基本运算的算法也有所不同。

需要了解的是，在顺序栈中有“上溢”和“下溢”的概念。顺序栈好比一个盒子，我们在里头放了一叠书，当要用书的话只能从第一本开始拿，那么当把书本放到这个栈中超过盒子的顶部时就放不下了，这时就是“上溢”，“上溢”也就是栈顶指针指出栈的外面，显然是出错了。反之，当栈中已没有书时，我们再去拿，看看没书，把盒子拎起来看看盒底，还是没有，这就是“下溢”。“下溢”本身可以表示栈为空栈，因此可以用它来作为控制转移的条件。

链栈则没有上溢的限制，它就像一条一头固定的链子，可以在活动的一头自由地增加链环(结点)而不会溢出，链栈不需要在头部附加头结点，因为栈都是在头部进行操作的，如果加了头结点，等于要在头结点之后的结点进行操作，反而使算法更复杂，所以只要有链表的头指针就可以了。

以上两种存储结构的栈的基本操作算法是不同的，这里主要介绍进栈和退栈的基本算法，以解决简单的应用问题。

1. 栈的顺序存储结构

与第2章讨论的一般的顺序存储结构的线性表一样，利用一组地址连续的存储单元依次存放自栈底到栈顶的数据元素，这种形式的栈也称为**顺序栈**。可以使用一维数组来作为栈的顺序存储空间。设指针top指向栈顶元素的当前位置，以数组小下标的一端作为栈底，通常以top=－1时为空栈，在元素进栈时指针top不断地加1，当top等于数组的最大下标值时则栈满。

图 3.2 展示了顺序栈中数据元素与栈顶指针的变化。

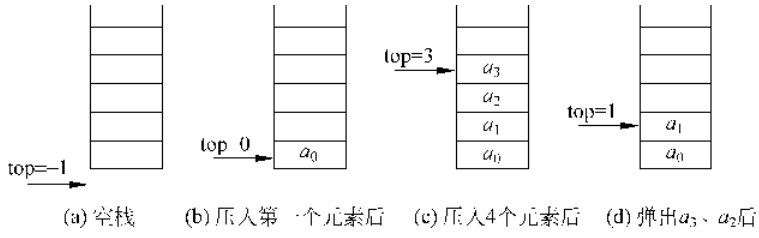

图 3.2 顺序栈的存储结构示意图

用 C 语言定义的顺序存储结构的栈如下：

```
#define TURE 1
#define FALSE 0
#define MAXNUM 100   /* 假定预分配的栈空间最多为 100 个元素 */
typedef struct
{ DataType stack[MAXNUM];
int top;
} seqstack;
```

注意：

(1) 顺序栈中元素用数组存放；

(2) 栈底位置是固定不变的，通常设置在数组中下标值小的一端；

(3) 栈顶位置是随着进栈和退栈操作而变化的，用一个整型量 top(通常称 top 为栈顶指针)来指示当前栈顶位置。

2. 顺序栈的基本操作算法

前提条件：设 s 是 seqstack 类型的指针变量。栈底位置在向量的低端，即 s－＞data[0]是栈底元素。

1) 栈的初始化

```
int initStack(seqstack * s)
{ /* 创建一个空栈由指针 s 指出 */
  s = (seqstack * )malloc(sizeof(seqstack));
  if (s == NULL) return FALSE;
  s－＞top = - 1;
  return TRUE;
}
```

2) 入栈操作

```
int push(seqstack * s, DataType x)
{ /* 将元素 x 插入到栈 s 中，作为 s 的新栈顶 */
  if(s－＞top＞ = MAXNUM - 1)  return FALSE;          /* 栈满 */
  s－＞top++ ;
  s－＞stack[s－＞top] = x;
  return  TRUE;
}
```

3）出栈操作

```
DataType pop(seqstack * s)
{ /* 若栈 s 不为空,则删除栈顶元素 */
  DataType x;
  if(s->top<0) return NULL;                    /* 栈空 */
  x = s->stack[s->top];
  s->top-- ;
  return x;
}
```

4）取栈顶元素操作

```
DataType gettop(seqstack * s)
{ /* 若栈 s 不为空,则返回栈顶元素 */
  if(s->top<0) return NULL;                    /* 栈空 */
  return (s->stack[s->top]);
}
```

取栈顶元素与出栈不同之处在于出栈操作改变栈顶指针 top 的位置,而取栈顶元素操作不改变栈的栈顶指针。

5）判栈空操作

```
int empty(seqstack * s)
{ /* 栈 s 为空时,返回为 TRUE; 非空时,返回为 FALSE */
  if(s->top<0) return TRUE;
  return FALSE;
}
```

6）置空操作

```
void setEmpty(seqstack * s)
{ /* 将栈 s 的栈顶指针 top,置为 -1 */
  s->top=-1;
}
```

3. 栈的链式存储结构

栈也可以采用链式存储结构表示,采用这种结构表示的栈简称为**链栈**。在一个链栈中,栈底就是链表的最后一个结点,而栈顶总是链表的第一个结点。因此,新入栈的元素即为链表新的第一个结点,只要系统还有存储空间,就不会有栈满的情况发生。一个链栈可由栈顶指针 top 唯一确定,当 top 为 NULL 时,是一个空栈。图 3.3 给出了链栈中数据元素与栈顶指针 top 的关系。

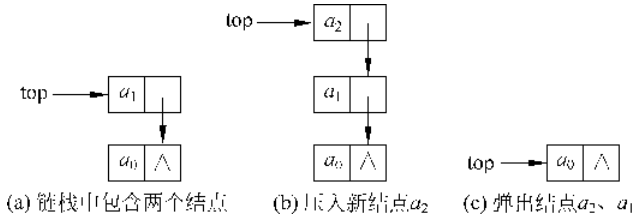

图 3.3　链栈的操作示意图

链栈的C语言定义为：

```
typedef struct Stacknode
{
  DataType data;
  Struct Stacknode * next;
}linkStack;
linkStack  * top
```

如果需要记录栈中元素个数，可将上述链栈的定义改为如下定义形式：

```
struct Stacknode
{ DataType data;
  Struct Stacknode * next;
}
struct linkStack
{ struct Stacknode * top;
  int len;
}
```

链栈创建时，将len的值初始化为0；在以后的操作中，每新增一个结点将len的值增1；每删除一个结点将len的值减1。通常，在实际应用中栈中元素的个数并不是需要关注的重点，所以多数情况下采用第一种定义形式。以下将要介绍的几个和链栈相关的操作就是建立在第一种结构定义的基础之上。

4. 链栈的基本操作算法

1）入栈操作

```
int push(linkStack * top,DataType x)
{ /* 将元素 x 压入链栈 top 中 */
  linkStack * p;
  p = (linkStack * )malloc(sizeof(linkStack)); /* 申请一个结点 */
  if(p == NULL) return FALSE;
  p -> data = x;
  p -> next = top;
  top = p;
  return TRUE;
}
```

2）出栈操作

```
DataType pop(linkStack * top)
{ /* 从链栈 top 中删除栈顶元素 */
  linkStack * p;
  DataType x;
  if (top == NULL) return NULL; /* 空栈 */
  p = top;
  top = top -> next;
  x = p -> data;
  free(p);
  return x;
}
```

3）取栈顶元素

```
DataType getTop(LinkStack * top)
{ if (top == NULL) return NULL;
  else return top->data;
}
```

在这里只介绍链栈的这 3 种操作，其他操作的算法与顺序栈并无什么差异，在此就不一一列出。在采用链式存储结构表示栈时，应该注意以下几个要点：

（1）在上一章线性表的链式存储结构中，为了操作的方便在第一个结点前附加一个头结点，头指针指向头结点。而在链栈中没有附加的头结点，栈顶指针 top 就是链栈的头指针。

（2）链栈中的结点是动态分配的，所以可以不用考虑“上溢”的问题；而在顺序栈中“上溢”的问题则是有可能发生的。

3.1.3　栈的应用举例

栈的应用非常广泛，只要问题满足后进先出的原则，均可使用栈作为其数据结构。通常在以下几个方面，栈的应用较为常见。

1. 算术表达式求值

通过设置两个栈，一个用于存放运算符，另一个用于存放运算数，实现较低优先级的运算符及其运算数的暂存。

2. 数制转换

通过设置一临时栈，存放每次整除后的余数，然后从栈顶依次弹出所有的余数，即可求得转换成某进制以后的数。

3. 括号匹配

在算法中设置一个栈，每读入一个括号，判断是左括号还是右括号。若是右括号，则或者可以使得置于栈顶的右括号得以消解，或者是不合法的情况；若是左括号，则压入栈中，期待着与之配对的右括号的出现。

4. 迷宫问题

在迷宫问题中，主要是利用栈保存那些“已经走过的路径”，以便于当向下无法再走时，从栈中弹出一步（返回一步），接着从上一步来继续探索下面可能的走法。

5. 递归问题

通常用于程序的嵌套调用中，保存每一级函数调用的返回地址。

下面举例介绍栈在数制转换及算术表达式运算中的使用。

1）数制转换

将一个非负的十进制整数 N 转换为另一个等值的基为 B 的 B 进制数，可以通过“除 B

取余法”来解决。

【例 3.1】 将十进制数 13 转化为二进制数。

解答：按除 2 取余法，得到的余数依次是 1、0、1、1，则十进制数转化为二进制数为 1101。

分析：由于最先得到的余数是转化结果的最低位，最后得到的余数是转化结果的最高位，因此很容易用栈来解决，在这里栈采用顺序存储结构实现。

转换算法如下：

```
typedef int DataType; //应将顺序栈的 DataType 定义改为整型
void conversion(int N,int B)
{//假设 N 是非负的十进制整数,输出等值的 B 进制数
   int i;
   seqstack S;
   initStack(&S);
   while(N)
   {//从右向左产生 B 进制的各位数字,并将其进栈
    push(&S,N % B); //将 bi 进栈 0<=i<=j
    N = N/B;
     }
   while(!Empty(&S))
    {//栈非空时退栈输出
      i = pop(&S);
      printf("%d",i);
     }
 }
```

2) 算术表达式求值

表达式求值是程序设计语言编译中的一个最基本问题，它的实现方法是栈的一个典型的应用实例。

在计算机中，任何一个表达式都是由操作数(operand)、运算符(operator)和界限符(delimiter)组成的。其中操作数可以是常数，也可以是变量或常量的标识符；运算符可以是算术运算符、关系运算符和逻辑符；界限符为左右括号和标识表达式开始、结束的起止符。在本节中，仅讨论简单算术表达式的求值问题。在这种表达式中只含加、减、乘、除四则运算，所有的运算对象均为单变量。表达式的起止符为“#”。

算术四则运算的规则为：

(1) 先乘除、后加减；

(2) 同级运算时先左后右；

(3) 先括号内，后括号外。

以上讨论的运算规则所针对的表达式一般都是运算符在两个操作数中间(除单目运算符外)，这也是平常在数学中表达式的形式，这种表达式称为**中缀表达式**。中缀表达式有时必须借助括号才能将运算顺序表达清楚，处理起来比较复杂。在编译系统中，对表达式的处理采用的是另外一种方法，即将中缀表达式转变为后缀表达式，然后对后缀表达式进行处理，后缀表达式也称为**逆波兰式**。

波兰表示法(也称为**前缀表达式**)是由波兰逻辑学家(Lukasiewicz)提出的，其特点是将

运算符置于运算对象的前面，如 $a+b$ 表示为＋ab。逆波兰式则是将运算符置于运算对象的后面，如 $a+b$ 表示为 ab＋。中缀表达式经过上述处理，得到后缀表达式后，运算时按从左到右的顺序进行，不需要括号。具体算法是：在计算表达式时，可以设置一个栈，从左到右扫描后缀表达式，每读到一个操作数就将其压入栈中；每到一个运算符时，则从栈顶取出两个操作数进行运算，并将结果压入栈中，一直到后缀表达式读完。最后的栈顶元素就是计算结果。

例如，中缀表达式 5＋(6－4/2)×3，转换为后缀表达式 5642/－3×＋，使用栈来计算该后缀表达式的值的过程如图 3.4 所示。

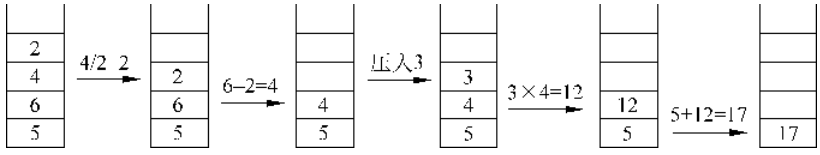

图 3.4　后缀表达式 5 6 4 2 / － 3 × ＋的计算过程

由上述计算过程可知，一旦得到某个表达式的后缀形式，那么计算其结果就是一件很容易的事。可是，计算机怎么将一个表达式的中缀形式转换为后缀形式呢？

以下介绍的运算符优先算法可以实现从中缀表达式到后缀表达式的转换并完成相应的计算，为了操作的需要表达式的两端加上起止符“＃”。整个算法过程如下：

计算机系统在处理表达式前，首先设置两个栈。

1）操作数栈(stackD)

存放处理表达式过程中的操作数。

2）运算符栈(stackR)

存放处理表达式过程中的运算符。开始时，在运算符栈中先在栈底压入一个表达式的结束符“＃”。

计算机系统在处理表达式时，从左到右依次读出表达式中的各个符号(操作数或运算符)，每读出一个符号后，根据运算规则进行如下的处理：

(1) 假如是操作数，则将其压入操作数栈，并依次读下一个符号；

(2) 假如是运算符，则

① 假如读出的运算符的优先级大于运算符栈栈顶运算符的优先级，则将其压入运算符栈，并依次读下一个符号；

② 假如读出的是表达式结束符“＃”，且运算符栈栈顶的运算符也为“＃”，则表达式处理结束，最后的表达式的计算结果在操作数栈的栈顶位置；

③ 假如读出的是“(”，则将其压入运算符栈；

④ 假如读出的是“)”，则

- 若运算符栈栈顶不是“(”，则从操作数栈连续退出两个操作数，从运算符栈中退出一个运算符，然后进行相应的运算，并将运算结果压入操作数栈；
- 若运算符栈栈顶为“(”，则从运算符栈退出“(”，依次读下一个符号。

⑤ 假如读出的运算符的优先级不大于运算符栈栈顶运算符的优先级，那么该运算符不能入栈，只能等待重新被读出。这种情况下执行的操作是：从操作数栈连续退出两个操作

数,从运算符栈中退出一个运算符,然后进行相应的运算,并将运算结果压入操作数栈。

上述过程涉及已存入栈中运算符与表达式中读出但还未入栈的运算符之间比较优先级的问题,表 3.1 给出了+、-、*、/、(、)和#的算术运算符间的优先级的关系,R1 表示已存入栈中运算符,R2 表示从表达式中读出的运算符。

表 3.1 运算符的优先级关系

R1 \ R2	+	-	*	/	(	)	#
+	>	>	<	<	<	>	>
-	>	>	<	<	<	>	>
*	>	>	>	>	<	>	>
/	>	>	>	>	<	>	>
(	<	<	<	<	<	=	error
)	>	>	>	>	error	>	>
#	<	<	<	<	<	error	=

看下面的例子。

【例 3.2】 对于一个中缀表达式 5+(6-4/2)×3,试用操作数栈和运算符栈这两个栈表示其运算过程。

分析:为了方便算法的执行,这里在表达式两端加上起止符,表达式成为#5+(6-4/2)×3#。使用操作数栈存放操作数和中间运算结果,使用运算符栈存放运算符。图 3.5 记录了表达式运算过程中栈内变化情况,整个过程详细描述如下:

(1) 初始状态下,两栈均为空,首先压入表达式开始标识符"#"。

(2) 依次从表达式中读出"5",压入数据栈;读出"+",由于"+"优先级高于操作符栈栈顶"#",所以压入操作符栈;读出"(",由于"("优先级高于操作符栈栈顶"+",所以压入栈内;……依次类推,直到将表达式中"/"压入操作符栈,"2"压入操作数栈。

(3) 继续读出符号")",由于")"优先级低于栈顶符号"/",所以")"不能入栈。接下来,从操作数栈弹出两个操作数,从操作符栈弹出一个操作符,执行 4/2=2 的运算,将所得结果压入操作数栈。

(4) 再次读出")",与操作符栈顶"-"比较,优先级仍然小于"-",还是不能入栈。接下来,从操作数栈弹出两个操作数,从操作符栈弹出一个操作符,执行 6-2=4 的运算,将所得结果压入操作数栈。

(5) 再次读出")",与操作符栈顶"("比较,根据算法思想,将二者消去。

(6) 从表达式中读出"*",与操作符栈顶"+"比较,优先级高于"+",压入栈内;继续读出"3",压入操作数栈。

(7) 从表达式中读出"#",与操作符栈顶"*"比较,"#"优先级低于"*",所以"#"不能入栈。接下来,从操作数栈弹出两个操作数,从操作符栈弹出一个操作符,执行 4×3=12 的运算,将"12"压入操作数栈。

(8) 再次读出刚刚不能入栈的"#",与栈顶"+"比较,"#"优先级低于"+",还是不能入栈。接下来从操作数栈中弹出"12"、"5",从操作符栈中弹出"+",执行 12+5=17 的操作,并将结果"17"压入操作数栈,此时栈内情况如图 3.5(8)所示。再次从表达式中读出

“＃”，与操作符栈顶“＃”比较，二者相等，到此为止，表达式处理结束。最后的表达式的计算结果在操作数栈的栈顶位置。

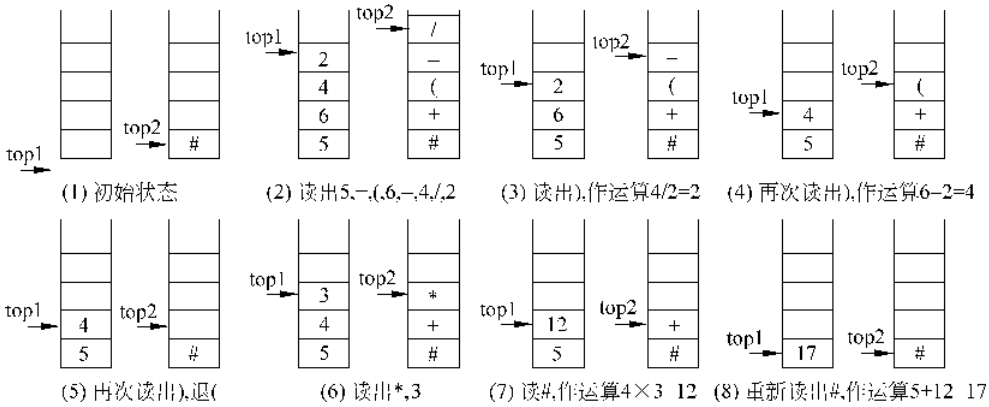

图 3.5　表达式运算过程中栈内变化情况

3.1.4　栈与递归的关系

在各种程序设计语言中都有子程序(或称函数、过程)调用功能。一个直接调用自己或通过一系列的调用语句间接调用自己的函数，称作**递归函数**。

递归是程序设计中一个强有力的工具。有很多数学函数是递归定义的，如大家熟悉的阶乘函数等；此外，有的数据结构，如二叉树、广义表等，由于结构本身固有的递归特性，则它们的操作可递归地描述；另外，还有一类问题，虽然问题本身没有明显的递归结构，但用递归求解比迭代求解更简单，如八皇后问题、Hanoi 塔问题等。

例如，对于第 1 章提到的计算 n 的阶乘问题，其递推公式为 $n!=n\times(n-1)!$。其递归算法的执行过程分递推与回归两个阶段。在递推阶段，把较复杂的问题(规模为 n)的求解推到比原问题简单一些的问题(规模小于 n)的求解。在回溯阶段，当获得最小规模问题的解之后，逐级返回，依次获得稍复杂的上一级问题的解。

【例 3.3】　试结合以下程序段具体分析一下计算机系统是如何完成阶乘运算中的递归调用的。

```
long f(int n)
 {if(n==0)
    return 1;
  else
     return n*f(n-1);
  }
main()
{
     ⋮
    …f(4);
    …;
  }
```

分析：当从主程序或其他函数非递归调用此阶乘函数时，首先把实参的值传送给形参n，同时把调用后的返回地址保存起来，以便调用结束之后返回之用。接着执行函数体，当n等于0时则返回函数值1，结束本次递归调用，并按返回地址（即返回到进行本次调用的调用函数的位置）继续向下执行；当n大于0时，则以实参n－1的值去调用本函数（递归调用），返回n的值与本次递归调用所求值的乘积。因为每进行一次递归调用，传送给形参n的值就减1，所以最终必然导致n的值为0，从而结束递归调用。接着不断地执行与递归调用相对应的返回操作，最后返回到进行非递归调用的调用函数（main函数）的位置向下执行。

图3.6给出了递归调用执行过程。从图3.6中可看到fact函数共被调用5次，即$f(4)$、$f(3)$、$f(2)$、$f(1)$、$f(0)$。其中，main()调用$f(4)$、$f(4)$调用$f(3)$、$f(3)$调用$f(2)$、$f(2)$调用$f(1)$、$f(1)$调用$f(0)$。每一次调用并未立即得到结果，而是进一步向深度递归调用，直到$n=0$时，函数fact才有结果为1，然后再一一返回计算，最终得到结果。

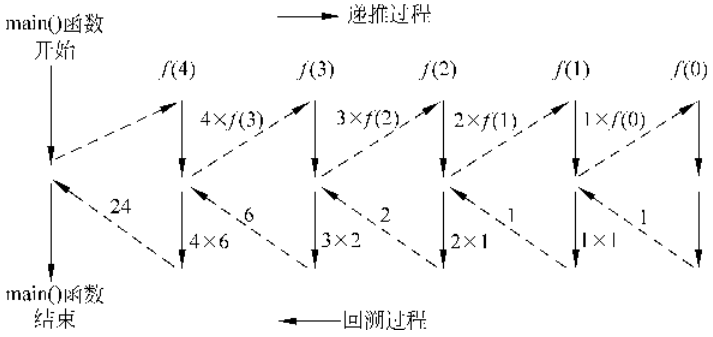

图3.6　阶乘运算中的递归调用执行过程及数值传递

通常，当在一个函数的运行期间调用另一个函数时，在运行被调用函数之前，系统需先完成3件事：

(1) 将所有的实在参数、返回地址等信息传递给被调用函数保存；

(2) 为被调用函数的局部变量分配存储区；

(3) 将控制转移到被调函数的入口。

而从被调用函数返回调用函数之前，系统也应完成3件工作：

(1) 保存被调函数的计算结果；

(2) 释放被调函数的数据区；

(3) 依照被调函数保存的返回地址将控制转移到调用函数。

当有多个函数构成嵌套调用时，按照“后调用先返回”的原则，上述函数之间的信息传递和控制转移必须通过“栈”来实现，即系统将整个程序运行时所需的数据空间安排在一个栈中，每当调用一个函数时，就为它在栈顶分配一个存储区，每当从一个函数退出时，就释放它的存储区，则当前正运行的函数的数据区必在栈顶。

一个递归函数的运行过程类似于多个函数的嵌套调用，只是调用函数和被调用函数是同一个函数，因此，和每次调用相关的一个重要的概念是递归函数运行的“层次”。假设调用该递归函数的主函数为第0层，则从主函数调用递归函数时进入第1层；从第i层递归调用