第4章

栈与队列

本章将定制并实现更加基本，且更为常用的两类数据结构——栈与队列。与此前介绍的向量和列表一样，它们也属于线性序列结构，故其中存放的数据对象之间也具有线性次序。相对于一般的序列结构，栈与队列的数据操作范围仅限于逻辑上的特定某端。然而，得益于其简洁性与规范性，它们既成为构建更复杂、更高级数据结构的基础，同时也是算法设计的基本出发点，甚至常常作为标准配置的基本数据结构以硬件形式直接实现。因此无论就工程或理论而言，其基础性地位都是其它结构无法比拟的。

在信息处理领域，栈与队列的身影随处可见。许多程序语言本身就是建立于栈结构之上，无论PostScript或者Java，其实时运行环境都是基于栈结构的虚拟机。再如，网络浏览器多会将用户最近访问过的地址组织为一个栈。这样，用户每访问一个新页面，其地址就会被存放至栈顶；而用户每按下一次“后退”按钮，即可沿相反的次序返回此前刚访问过的页面。类似地，主流的文本编辑器也大都支持编辑操作的历史记录功能，用户的编辑操作被依次记录在一个栈中。一旦出现误操作，用户只需按下“撤销”按钮，即可取消最近一次操作并回到此前的编辑状态。

在需要公平且经济地对各种自然或社会资源做管理或分配的场合，无论是调度银行和医院的服务窗口，还是管理轮耕的田地和轮伐的森林，队列都可大显身手。甚至计算机及其网络自身内部的各种计算资源，无论是多进程共享的CPU时间，还是多用户共享的打印机，也都需要借助队列结构实现合理和优化的分配。

相对于向量和列表，栈与队列的外部接口更为简化和紧凑，故亦可视作向量与列表的特例，因此C++的继承与封装机制在此可以大显身手。得益于此，本章的重点将不再拘泥于对数据结构内部实现机制的展示，并转而更多地从其外部特性出发，结合若干典型的实际问题介绍栈与队列的具体应用。

在栈的应用方面，本章将在1.4节的基础上，结合函数调用栈的机制介绍一般函数调用的实现方式与过程，并将其推广至递归调用。然后以降低空间复杂度的目标为线索，介绍通过显式地维护栈结构解决应用问题的典型方法和基本技巧。此外，还将着重介绍如何利用栈结构，实现基于试探回溯策略的高效搜索算法。在队列的应用方面，本章将介绍如何实现基于轮值策略的通用循环分配器，并以银行窗口服务为例实现基本的调度算法。

§4.1 栈

4.1.1 ADT接口

■ 入栈与出栈

栈（stack）是存放数据对象的一种特殊容器，其中的数据元素按线性的逻辑次序排列，故也可定义首、末元素。不过，尽管栈结构也支持对象的插入和删除操作，但其操作的范围仅限于栈的某一特定端。也就是说，若约定新的元素只能从某一端插入其中，则反过来也只能从这一端删除已有的元素。禁止操作的另一端，称作盲端。

图4.1 一摞椅子即是一个栈

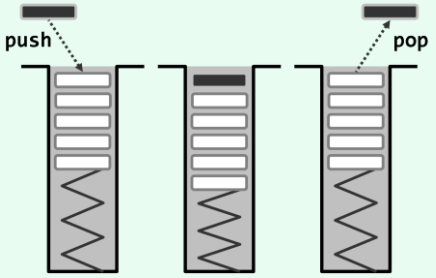

图4.2 栈操作

如图4.1所示，数把椅子叠成一摞即可视作一个栈。为维持这一放置形式，对该栈可行的操作只能在其顶部实施：新的椅子只能叠放到最顶端；反过来，只有最顶端的椅子才能被取走。因此比照这类实例，栈中可操作的一端更多地称作栈顶（stack top），而另一无法直接操作的盲端则更多地称作栈底（stack bottom）。

作为抽象数据类型，栈所支持的操作接口可归纳为表4.1。其中除了引用栈顶的top()等操作外，如图4.2所示，最常用的插入与删除操作分别称作入栈（push）和出栈（pop）。

表4.1 栈ADT支持的操作接口

操作接口	功 能
size()	报告栈的规模
empty()	判断栈是否为空
push(e)	将e插至栈顶
pop()	删除栈顶对象
top()	引用栈顶对象

■ 后进先出

由以上关于栈操作位置的约定和限制不难看出，栈中元素接受操作的次序必然始终遵循所谓“后进先出”（last-in-first-out，LIFO）的规律：从栈结构的整个生命期来看，更晚（早）出栈的元素，应为更早（晚）入栈者；反之，更晚（早）入栈者应更早（晚）出栈。

4.1.2 操作实例

表4.2给出了一个存放整数的栈从被创建开始，按以上接口实施一系列操作的过程。

表4.2 栈操作实例

操 作	输 出	栈（左侧为栈顶）
Stack()		
empty()	true	
push(5)		5
push(3)		3 5
pop()	3	5
push(7)		7 5
push(3)		3 7 5
top()	3	3 7 5
empty()	false	3 7 5

操 作	输 出	栈（左侧为栈顶）
push(11)		11 3 7 5
size()	4	11 3 7 5
push(6)		6 11 3 7 5
empty()	false	6 11 7 5
push(7)		7 6 11 3 7 5
pop()	7	6 11 3 7 5
pop()	6	11 3 7 5
top()	11	11 3 7 5
size()	4	11 3 7 5

4.1.3 Stack模板类

既然栈可视作序列的特例，故只要将栈作为向量的派生类，即可利用C++的继承机制，基于2.2.3节定义的向量模板类实现栈结构。当然，这里需要按照栈的习惯，对各接口重新命名。

按照表4.1所列的ADT接口，可描述并实现Stack模板类如代码4.1所示。

```
1 #include "../Vector/Vector.h" //以向量为基类，派生出栈模板类
2 template <typename T> class Stack: public Vector<T> { //将向量的首/末端作为栈底/顶
3 public: //size()、empty()以及其它开放接口，均可直接沿用
4    void push ( T const& e ) { insert ( size(), e ); } //入栈：等效于将新元素作为向量的末元素插入
5    T pop() { return remove ( size() - 1 ); } //出栈：等效于删除向量的末元素
6    T& top() { return ( *this ) [size() - 1]; } //取顶：直接返回向量的末元素
7 };
```

代码4.1 Stack模板类

既然栈操作都限制于向量的末端，参与操作的元素没有任何后继，故由2.5.5节和2.5.6节的分析结论可知，以上栈接口的时间复杂度均为常数。

套用以上思路，也可直接基于3.2.2节的List模板类派生出Stack类（习题[4-1]）。

§4.2 栈与递归

习题[1-17]指出，递归算法所需的空间量，主要决定于最大递归深度。在达到这一深度的时刻，同时活跃的递归实例达到最多。那么，操作系统具体是如何实现函数（递归）调用的？如何记录调用与被调用函数（递归）实例之间的关系？如何实现函数（递归）调用的返回？又是如何维护同时活跃的所有函数（递归）实例的？所有这些问题的答案，都可归结于栈。

4.2.1 函数调用栈

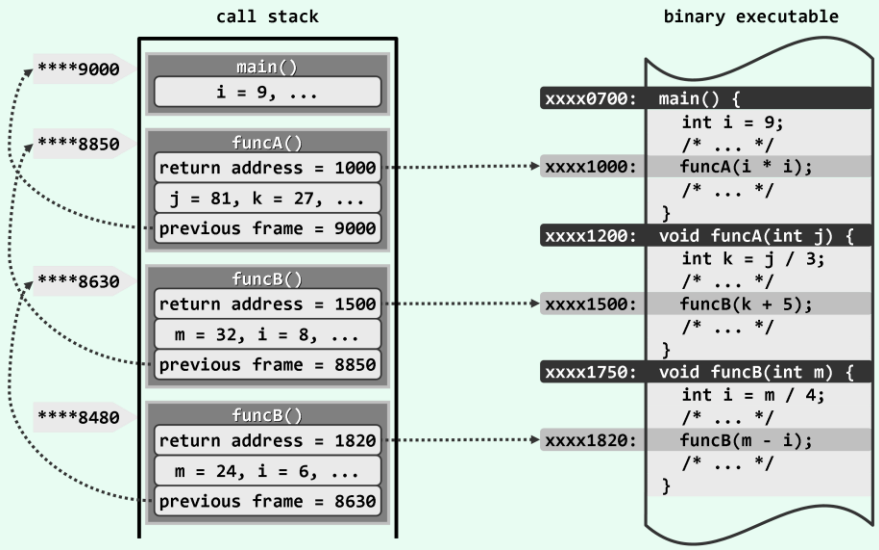

图4.3 函数调用栈实例：主函数main()调用funcA()，funcA()调用funcB()，funcB()再自我调用

在Windows等大部分操作系统中，每个运行中的二进制程序都配有一个调用栈（call stack）或执行栈（execution stack）。借助调用栈可以跟踪属于同一程序的所有函数，记录它们之间的相互调用关系，并保证在每一调用实例执行完毕之后，可以准确地返回。

■ 函数调用

如图4.3所示，调用栈的基本单位是帧（frame）。每次函数调用时，都会相应地创建一帧，记录该函数实例在二进制程序中的返回地址（return address），以及局部变量、传入参数等，并将该帧压入调用栈。若在该函数返回之前又发生新的调用，则同样地要将与新函数对应的一帧压入栈中，成为新的栈顶。函数一旦运行完毕，对应的帧随即弹出，运行控制权将被交还给该函数的上层调用函数，并按照该帧中记录的返回地址确定在二进制程序中继续执行的位置。

在任一时刻，调用栈中的各帧，依次对应于那些尚未返回的调用实例，亦即当时的活跃函数实例（active function instance）。特别地，位于栈底的那帧必然对应于入口主函数main()，若它从调用栈中弹出，则意味着整个程序的运行结束，此后控制权将交还给操作系统。

仿照递归跟踪法，程序执行过程出现过的函数实例及其调用关系，也可构成一棵树，称作该程序的运行树。任一时刻的所有活跃函数实例，在调用栈中自底到顶，对应于运行树中从根节点到最新活跃函数实例的一条调用路径。

此外，调用栈中各帧还需存放其它内容。比如，因各帧规模不一，它们还需记录前一帧的起始地址，以保证其出栈之后前一帧能正确地恢复。

■ 递归

作为函数调用的特殊形式，递归也可借助上述调用栈得以实现。比如在图4.3中，对应于funcB()的自我调用，也会新压入一帧。可见，同一函数可能同时拥有多个实例，并在调用栈中各自占有一帧。这些帧的结构完全相同，但其中同名的参数或变量，都是独立的副本。比如在funcB()的两个实例中，入口参数m和内部变量i各有一个副本。

4.2.2 避免递归

今天，包括C++在内的各种高级程序设计语言几乎都允许函数直接或间接地自我调用，通过递归来提高代码的简洁度和可读性。而Cobol和Fortran等早期的程序语言虽然一开始并未采用栈来实现过程调用，但在其最新的版本中也陆续引入了栈结构来支持过程调用。

尽管如此，系统在后台隐式地维护调用栈的过程中，难以区分哪些参数和变量是对计算过程有实质作用的，更无法以通用的方式对它们进行优化，因此不得不将描述调用现场的所有参数和变量悉数入栈。再加上每一帧都必须保存的执行返回地址以及前一帧起始位置，往往导致程序的空间效率不高甚至极低；同时，隐式的入栈和出栈操作也会令实际的运行时间增加不少。

因此在追求更高效率的场合，应尽可能地避免递归，尤其是过度的递归。实际上，我们此前已经介绍过相应的方法和技巧。例如，在1.4.4节中将尾递归转换为等效的迭代形式；在1.4.5节中采用动态规划策略，将Fibonacci数算法中的二分递归改为线性递归，直至完全消除递归。

既然递归本身就是操作系统隐式地维护一个调用栈而实现的，我们自然也可以通过显式地模拟调用栈的运转过程，实现等效的算法功能。采用这一方式，程序员可以精细地裁剪栈中各帧的内容，从而尽可能降低空间复杂度的常系数。尽管算法原递归版本的高度概括性和简洁性将大打折扣，但毕竟在空间效率方面可以获得足够的补偿。

§4.3 栈的典型应用

4.3.1 逆序输出

在栈所擅长解决的典型问题中，有一类具有以下共同特征：首先，虽有明确的算法，但其解答却以线性序列的形式给出；其次，无论是递归还是迭代实现，该序列都是依逆序计算输出的；最后，输入和输出规模不确定，难以事先确定盛放输出数据的容器大小。因其特有的“后进先出”特性及其在容量方面的自适应性，使用栈来解决此类问题可谓恰到好处。

■ 进制转换

考查如下问题：任给十进制整数n，将其转换为λ进制的表示形式。比如λ = 8时有

$12345_{(10)} = 30071_{(8)}$

一般地，设 $n = (d_m \ldots d_2\ d_1\ d_0)_{(\lambda)} = d_m\times\lambda^m + \ldots + d_2\times\lambda^2 + d_1\times\lambda^1 + d_0\times\lambda^0$

若记 $n_i = (d_m \ldots d_{i+1}\ d_i)_{(\lambda)}$

则有 $d_i = n_i \ \%\ \lambda$ 和 $n_{i+1} = n_i \ /\ \lambda$

这一递推关系对应的计算流程如下。可见，其输出的确为长度不定的逆序线性序列。

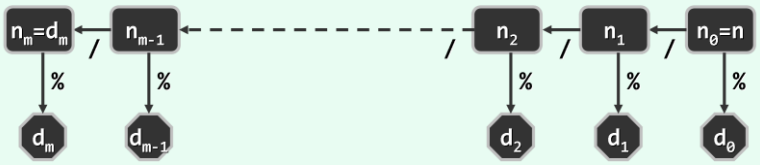

图4.4 进制转换算法流程

■ 递归实现

根据如图4.4所示的计算流程，可得到如代码4.2所示递归式算法。

```
1 void convert ( Stack<char>& S, __int64 n, int base ) { //十进制数n到base进制的转换（递归版）
2    static char digit[] //0 < n, 1 < base <= 16，新进制下的数位符号，可视base取值范围适当扩充
3    = { '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'A', 'B', 'C', 'D', 'E', 'F' };
4    if ( 0 < n ) { //在尚有余数之前，不断
5       convert ( S, n / base, base ); //通过递归得到所有更高位
6       S.push ( digit[n % base] ); //输出低位
7    }
8 } //新进制下由高到低的各数位，自顶而下保存于栈S中
```

代码4.2 进制转换算法（递归版）

尽管新进制下的各数位须按由低到高次序逐位算出，但只要引入一个栈并将算得的数位依次入栈，则在计算结束后只需通过反复的出栈操作即可由高到低地将其顺序输出。

■ 迭代实现

这里的静态数位符号表在全局只需保留一份，但与一般的递归函数一样，该函数在递归调用栈中的每一帧都仍需记录参数S、n和base。将它们改为全局变量固然可以节省这部分空间，但依然不能彻底地避免因调用栈操作而导致的空间和时间消耗。为此，不妨考虑改写为如代码4.3所示的迭代版本，既能充分发挥栈处理此类问题的特长，又可将空间消耗降至$\mathcal{O}(1)$。

```
1 void convert ( Stack<char>& S, __int64 n, int base ) { //十进制数n到base进制的转换（迭代版）
2    static char digit[] //0 < n, 1 < base <= 16，新进制下的数位符号，可视base取值范围适当扩充
3    = { '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'A', 'B', 'C', 'D', 'E', 'F' };
4    while ( n > 0 ) { //由低到高，逐一计算出新进制下的各数位
5       int remainder = ( int ) ( n % base ); S.push ( digit[remainder] ); //余数（当前位）入栈
6       n /= base; //n更新为其对base的除商
7    }
8 } //新进制下由高到低的各数位，自顶而下保存于栈S中
```

代码4.3 进制转换算法（迭代版）

4.3.2 递归嵌套

具有自相似性的问题多可嵌套地递归描述，但因分支位置和嵌套深度并不固定，其递归算法的复杂度不易控制。栈结构及其操作天然地具有递归嵌套性，故可用以高效地解决这类问题。以下先从混洗的角度介绍栈的递归嵌套性，然后再讲解其在具体问题中的应用。

■ **栈混洗**

考查三个栈A、B和S。其中，B和S初始为空；A含有n个元素，自顶而下构成输入序列：

$$A = < a_1, a_2, \ldots, a_n]$$

这里，分别用尖括号、方括号示意栈顶、栈底，这也是本小节将统一采用的约定。

以下，若只允许通过S.push(A.pop())弹出栈A的顶元素并随即压入栈S中，或通过B.push(S.pop())弹出S的顶元素并随即压入栈B中，则在经过这两类操作各n次之后，栈A和S有可能均为空，原A中的元素均已转入栈B。此时，若将B中元素自底而上构成的序列记作：

$$B = [a_{k1}, a_{k2}, \ldots, a_{kn} >$$

则该序列称作原输入序列的一个栈混洗（stack permutation）。

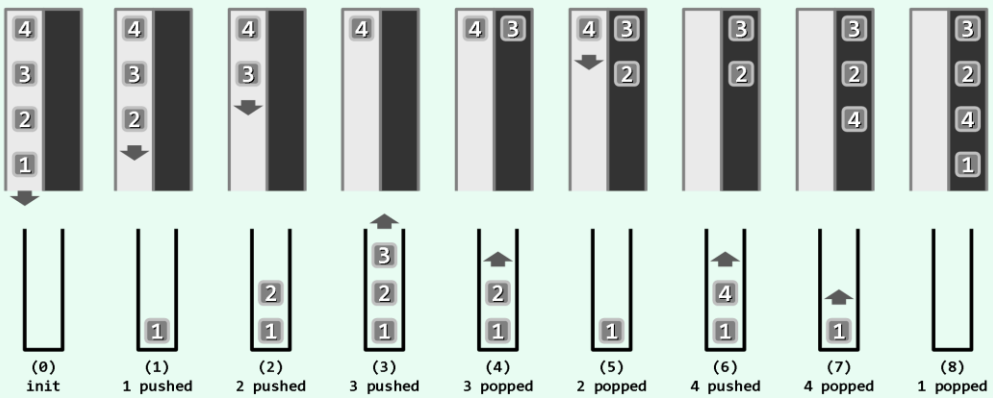

图4.5 栈混洗实例：从< 1, 2, 3, 4]到[3, 2, 4, 1 >（上方左侧为栈A，右侧为栈B；下方为栈S）

如图4.5所示，设最初栈A = < 1, 2, 3, 4]，栈S和B均为空；经过“随机的”8次操作，A中元素全部转入栈B中。此时，栈B中元素所对应的序列[3, 2, 4, 1 >，即是原序列的一个栈混洗。除了“实施出栈操作时栈不得为空”，以上过程并无更多限制，故栈混洗并不唯一。就此例而言，[1, 2, 3, 4 >、[4, 3, 2, 1 >以及[3, 2, 1, 4 >等也是栈混洗。

从图4.5也可看出，一般地对于长度为n的输入序列，每一栈混洗都对应于由栈S的n次push

和n次pop构成的某一合法操作序列，比如[3, 2, 4, 1]即对应于操作序列：

{ push, push, push, pop, pop, push, pop, pop }

反之，由n次push和n次pop构成的任何操作序列，只要满足“任一前缀中的push不少于pop”这一限制，则该序列也必然对应于某个栈混洗（习题[4-4]）。

■ 括号匹配

对源程序的语法检查是代码编译过程中重要而基本的一个步骤，而对表达式括号匹配的检查则又是语法检查中必需的一个环节。其任务是，对任一程序块，判断其中的括号是否在嵌套的意义下完全匹配（简称匹配）。比如在以下两个表达式中，前者匹配，而后者不匹配。

a / (b [i - 1] [j + 1] + c [i + 1] [j - 1]) * 2

a / (b [i - 1] [j + 1]) + c [i + 1] [j - 1]) * 2

■ 递归实现

不妨先只考虑圆括号。用'+'表示表达式的串接。

不难理解，一般地，若表达式S可分解为如下形式：

$S = S_0 + "(" + S_1 + ")" + S_2 + S_3$

其中S_0和S_3不含括号，且S_1中左、右括号数目相等，则S匹配当且仅当S_1和S_2均匹配。

按照这一理解，可采用分治策略设计算法如下：将表达式划分为子表达式S_0、S_1和S_2，分别递归地判断S_1和S_2是否匹配。这一构思可具体实现如代码4.4所示。

```
1  void trim ( const char exp[], int& lo, int& hi ) { //删除exp[lo, hi]不含括号的最长前缀、后缀
2     while ( ( lo <= hi ) && ( exp[lo] != '(' ) && ( exp[lo] != ')' ) ) lo++; //查找第一个和
3     while ( ( lo <= hi ) && ( exp[hi] != '(' ) && ( exp[hi] != ')' ) ) hi--; //最后一个括号
4  }
5
6  int divide ( const char exp[], int lo, int hi ) { //切分exp[lo, hi]，使exp匹配仅当子表达式匹配
7     int mi = lo; int crc = 1; //crc为[lo, mi]范围内左、右括号数目之差
8     while ( ( 0 < crc ) && ( ++mi < hi ) ) //逐个检查各字符，直到左、右括号数目相等，或者越界
9        {  if ( exp[mi] == ')' )  crc--; if ( exp[mi] == '(' )  crc++;  } //左、右括号分别计数
10    return mi; //若mi <= hi，则为合法切分点；否则，意味着局部不可能匹配
11 }
12
13 bool paren ( const char exp[], int lo, int hi ) { //检查表达式exp[lo, hi]是否括号匹配（递归版）
14    trim ( exp, lo, hi ); if ( lo > hi ) return true; //清除不含括号的前缀、后缀
15    if ( exp[lo] != '(' ) return false; //首字符非左括号，则必不匹配
16    if ( exp[hi] != ')' ) return false; //末字符非右括号，则必不匹配
17    int mi = divide ( exp, lo, hi ); //确定适当的切分点
18    if ( mi > hi ) return false; //切分点不合法，意味着局部以至整体不匹配
19    return paren ( exp, lo + 1, mi - 1 ) && paren ( exp, mi + 1, hi ); //分别检查左、右子表达式
20 }
```

代码4.4 括号匹配算法（递归版）

其中，trim()函数用于截除表达式中不含括号的头部和尾部，即前缀S_0和后缀S_3。divide()

函数对表达式做线性扫描，并动态地记录已经扫描的左、右括号数目之差。如此，当已扫过同样多的左、右括号时，即确定了一个合适的切分点mi，并得到子表达式S_1 = exp(lo, mi)和S_2 = exp(mi, hi]。以下，经递归地检查S_1和S_2，即可判断原表达式是否匹配。

在最坏情况下divide()需要线性时间，且递归深度为$O(n)$，故以上算法共需$O(n^2)$时间。此外，该方法也难以处理含有多种括号的表达式（习题[4-5]和[4-15]），故有必要进一步优化。

■ 迭代实现

实际上，只要将push、pop操作分别与左、右括号相对应，则长度为n的栈混洗，必然与由n对括号组成的合法表达式彼此对应（习题[4-4]）。比如，栈混洗[3, 2, 4, 1 >对应于表达式"((( )) ( ))"。按照这一理解，借助栈结构，只需扫描一趟表达式，即可在线性时间内，判定其中的括号是否匹配。这一新的算法，可简明地实现如代码4.5所示。

```
1  bool paren ( const char exp[], int lo, int hi ) { //表达式括号匹配检查，可兼顾三种括号
2     Stack<char> S; //使用栈记录已发现但尚未匹配的左括号
3     for ( int i = lo; i <= hi; i++ ) /* 逐一检查当前字符 */
4        switch ( exp[i] ) { //左括号直接进栈；右括号若与栈顶失配，则表达式必不匹配
5           case '(': case '[': case '{': S.push ( exp[i] ); break;
6           case ')': if ( ( S.empty() ) || ( '(' != S.pop() ) ) return false; break;
7           case ']': if ( ( S.empty() ) || ( '[' != S.pop() ) ) return false; break;
8           case '}': if ( ( S.empty() ) || ( '{' != S.pop() ) ) return false; break;
9           default: break; //非括号字符一律忽略
10       }
11    return S.empty(); //整个表达式扫描过后，栈中若仍残留（左）括号，则不匹配；否则（栈空）匹配
12 }
```

代码4.5 括号匹配算法（迭代版）

新算法的流程控制简单，而且便于推广至多类括号并存的场合。它自左向右逐个考查各字符，忽略所有非括号字符。凡遇到左括号，无论属于哪类均统一压入栈S中。若遇右括号，则弹出栈顶的左括号并与之比对。若二者属于同类，则继续检查下一字符；否则，即可断定表达式不匹配。当然，栈S提前变空或者表达式扫描过后栈S非空，也意味着不匹配。

图4.6给出了一次完整的计算过程。表达式扫描完毕时，栈S恰好为空，故知表达式匹配。

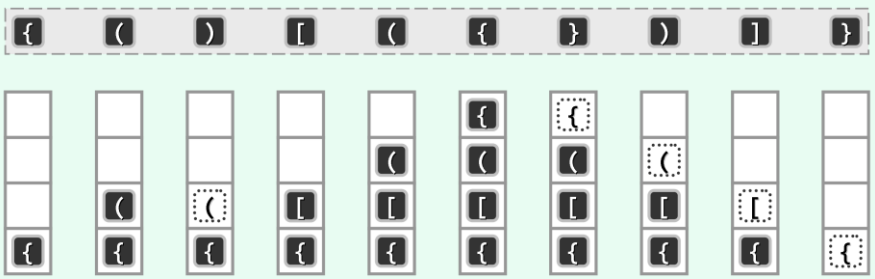

图4.6 迭代式括号匹配算法实例

（上方为输入表达式；下方为辅助栈的演变过程；虚框表示在（右）括号与栈顶（左）括号匹配时对应的出栈操作）

4.3.3 延迟缓冲

在一些应用问题中，输入可分解为多个单元并通过迭代依次扫描处理，但过程中的各步计算往往滞后于扫描的进度，需要待到必要的信息已完整到一定程度之后，才能作出判断并实施计算。在这类场合，栈结构则可以扮演数据缓冲区的角色。

■ 表达式求值

在编译C++程序的预处理阶段，源程序中的所有常量表达式都需首先计算并替换为对应的具体数值。而在解释型语言中，算术表达式的求值也需随着脚本执行过程中反复进行。

比如，在UNIX Shell、DOS Shell和PostScript交互窗口中分别输入：

```
$       echo $(( 0 + ( 1 + 23 ) / 4 * 5 * 67 - 8 + 9 ))
\>      set /a (( 0 + ( 1 + 23 ) / 4 * 5 * 67 - 8 + 9 ))
GS>     0 1 23 add 4 div 5 mul 67 mul add 8 sub 9 add =
```

都将返回“2011”。

可见，不能简单地按照“先左后右”的次序执行表达式中的运算符。关于运算符执行次序的规则（即运算优先级），一部分决定于事先约定的惯例（比如乘除优先于加减），另一部分则决定于括号。也就是说，仅根据表达式的某一前缀，并不能完全确定其中各运算符可否执行以及执行的次序；只有在已获得足够多后续信息之后，才能确定其中哪些运算符可以执行。

■ 优先级表

我们首先如代码4.6所示，将不同运算符之间的运算优先级关系，描述为一张二维表格。

```
1  #define N_OPTR 9 //运算符总数
2  typedef enum { ADD, SUB, MUL, DIV, POW, FAC, L_P, R_P, EOE } Operator; //运算符集合
3  //加、减、乘、除、乘方、阶乘、左括号、右括号、起始符与终止符
4
5  const char pri[N_OPTR][N_OPTR] = { //运算符优先等级 [栈顶] [当前]
6     /*              |-------------------- 当 前 运 算 符 --------------------| */
7     /*              +      -      *      /      ^      !      (      )      \0 */
8     /* --  + */    '>',   '>',   '<',   '<',   '<',   '<',   '<',   '>',   '>',
9     /* |   - */    '>',   '>',   '<',   '<',   '<',   '<',   '<',   '>',   '>',
10    /* 栈  * */    '>',   '>',   '>',   '>',   '<',   '<',   '<',   '>',   '>',
11    /* 顶  / */    '>',   '>',   '>',   '>',   '<',   '<',   '<',   '>',   '>',
12    /* 运  ^ */    '>',   '>',   '>',   '>',   '>',   '<',   '<',   '>',   '>',
13    /* 算  ! */    '>',   '>',   '>',   '>',   '>',   '>',   ' ',   '>',   '>',
14    /* 符  ( */    '<',   '<',   '<',   '<',   '<',   '<',   '<',   '=',   ' ',
15    /* |   ) */    ' ',   ' ',   ' ',   ' ',   ' ',   ' ',   ' ',   ' ',   ' ',
16    /* -- \0 */    '<',   '<',   '<',   '<',   '<',   '<',   '<',   ' ',   '='
17 };
```

代码4.6 运算符优先级关系的定义

在常规的四则运算之外，这里还引入了乘方和阶乘运算。其中阶乘属于一元运算，且优先级最高。为统一算法的处理流程，将左、右括号以及标识表达式尾部的字符'\0'，也视作运算符。

■ 求值算法

基于运算符优先级如上的定义和判定规则，可实现表达式求值算法如代码4.7所示。

```
1  float evaluate ( char* S, char*& RPN ) { //对（已剔除白空格的）表达式S求值，并转换为逆波兰式RPN
2     Stack<float> opnd; Stack<char> optr; //运算数栈、运算符栈
3     optr.push ( '\0' ); //尾哨兵'\0'也作为头哨兵首先入栈
4     while ( !optr.empty() ) { //在运算符栈非空之前，逐个处理表达式中各字符
5        if ( isdigit ( *S ) ) { //若当前字符为操作数，则
6           readNumber ( S, opnd ); append ( RPN, opnd.top() ); //读入操作数，并将其接至RPN末尾
7        } else //若当前字符为运算符，则
8           switch ( orderBetween ( optr.top(), *S ) ) { //视其与栈顶运算符之间优先级高低分别处理
9              case '<': //栈顶运算符优先级更低时
10                optr.push ( *S ); S++; //计算推迟，当前运算符进栈
11                break;
12             case '=': //优先级相等（当前运算符为右括号或者尾部哨兵'\0'）时
13                optr.pop(); S++; //脱括号并接收下一个字符
14                break;
15             case '>': { //栈顶运算符优先级更高时，可实施相应的计算，并将结果重新入栈
16                char op = optr.pop(); append ( RPN, op ); //栈顶运算符出栈并续接至RPN末尾
17                if ( '!' == op ) { //若属于一元运算符
18                   float pOpnd = opnd.pop(); //只需取出一个操作数，并
19                   opnd.push ( calcu ( op, pOpnd ) ); //实施一元计算，结果入栈
20                } else { //对于其它（二元）运算符
21                   float pOpnd2 = opnd.pop(), pOpnd1 = opnd.pop(); //取出后、前操作数
22                   opnd.push ( calcu ( pOpnd1, op, pOpnd2 ) ); //实施二元计算，结果入栈
23                }
24                break;
25             }
26             default : exit ( -1 ); //逢语法错误，不做处理直接退出
27          }//switch
28    }//while
29    return opnd.pop(); //弹出并返回最后的计算结果
30 }
```

代码4.7 表达式的求值及RPN转换

该算法自左向右扫描表达式，并对其中字符逐一做相应的处理。那些已经扫描过但（因信息不足）尚不能处理的操作数与运算符，将分别缓冲至栈opnd和栈optr。一旦判定已缓存的子表达式优先级足够高，便弹出相关的操作数和运算符，随即执行运算，并将结果压入栈opnd。

请留意这里区分操作数和运算符的技巧。一旦当前字符由非数字转为数字，则意味着开始进入一个对应于操作数的子串范围。由于这里允许操作数含有多个数位，甚至可能是小数，故可调用readNumber()函数（习题[4-6]），根据当前字符及其后续的若干字符，利用另一个栈解析出当前的操作数。解析完毕，当前字符将再次聚焦于一个非数字字符。

■ 不同优先级的处置

按照代码4.7，若当前字符为运算符，则在调用orderBetween()函数（习题[4-7]），将其与栈optr的栈顶操作符做一比较之后，即可视二者的优先级高低，分三种情况相应地处置。

1）若当前运算符的优先级更高，则optr中的栈顶运算符尚不能执行

以表达式" 1 + 2 * 3 ... "为例，在扫描到运算符'*'时，optr栈顶运算符为此前的'+'，由于pri['+']['*'] = '<'，当前运算符'*'优先级更高，故栈顶运算符'+'的执行必须推迟。

请注意，由代码4.6定义的优先级表，无论栈顶元素如何，当前操作符为'('的所有情况均统一归入这一处理方式；另外，无论当前操作符如何，栈顶操作符为'('的所有情况也统一按此处理。也就是说，所有左括号及其后紧随的一个操作符都会相继地被直接压入optr栈中，而此前的运算符则一律押后执行——这与左括号应有的功能完全吻合。

2）反之，一旦栈顶运算符的优先级更高，则可以立即弹出并执行对应的运算

以表达式" 1 + 2 * 3 - 4 ... "为例，在扫描到运算符'-'时，optr栈顶运算符为'*'，由于pri['*']['-'] = '>'，意味着当前运算符的优先级更低，故栈顶运算符'*'可立即执行。

类似地，根据代码4.6定义的优先级表，无论栈顶元素如何，当前操作符为')'的情况也几乎全部归入这一处理方式。也就是说，一旦抵达右括号，此前在optr栈缓冲的运算符大都可以逐一弹出并执行——这与右括号应有的功能也完全吻合。

3）当前运算符与栈顶运算符的优先级“相等”

对右括号的上述处理方式，将在optr栈顶出现操作符'('时终止——由代码4.6可知，pri['(')[')'] = '='。此时，将弹出栈顶的'('，然后继续处理')'之后的字符。不难看出，这对左、右括号在表达式中必然相互匹配，其作用在于约束介乎二者之间的那段子表达式的优先级关系，故在其“历史使命”完成之后，算法做如上处置理所应当。

除左、右括号外，还有一种优先级相等的合法情况，即pri['\0']['\0'] = '='。由于在算法启动之初已经首先将字符'\0'压入optr栈，故在整个表达式已被正确解析并抵达表达式结束标识符'\0'时，即出现这一情况。对于合法的表达式，这种情况只在算法终止前出现一次。既然同是需要弹出栈顶，算法不妨将这种情况按照优先级相等的方式处置。

■ 语法检查及鲁棒性

为简洁起见，以上算法假设输入表达式的语法完全正确；否则，有可能会导致荒诞的结果。读者可在此基础上，尝试扩充语法检查以及对各种非法情况的处理功能（习题[4-12]）。

4.3.4 逆波兰表达式

■ RPN

逆波兰表达式（reverse Polish notation，RPN）是数学表达式的一种，其语法规则可概括为：操作符紧邻于对应的（最后一个）操作数之后。比如“1 2 +”即通常习惯的“1 + 2”。

按此规则，可递归地得到更复杂的表达式，比如RPN表达式

```
1 2 + 3 4 ^ *
```

即对应于常规的表达式

```
( 1 + 2 ) * 3 ^ 4
```

RPN表达式亦称作后缀表达式（postfix），原表达式则称作中缀表达式（infix）。尽管RPN表达式不够直观易读，但其对运算符优先级的表述能力，却毫不逊色于常规的中缀表达式；而其在计算效率方面的优势，更是常规表达式无法比拟的。RPN表达式中运算符的执行次序，可更为简捷地确定，既不必在事先做任何约定，更无需借助括号强制改变优先级。具体而言，各运算符被执行的次序，与其在RPN表达式中出现的次序完全吻合。以上面的" 1 2 + 3 4 ^ * "为例，三次运算的次序{ +, ^, * }，与三个运算符的出现次序完全一致。

■ 求值算法

根据以上分析，采用算法4.1即可高效地实现对RPN表达式的求值。

```
rpnEvaluation(expr)
输入：RPN表达式expr（假定语法正确）
输出：表达式数值
{
   引入栈S，用以存放操作数;
   while (expr尚未扫描完毕) {
      从expr中读入下一元素x;
      if (x是操作数) 将x压入S;
      else { //x是运算符
         从栈S中弹出运算符x所需数目的操作数;
         对弹出的操作数实施x运算，并将运算结果重新压入S;
      } //else
   } //while
   返回栈顶; //也是栈底
}
```

算法4.1 RPN表达式求值

可见，除了一个辅助栈外，该算法不需要借助任何更多的数据结构。此外，算法的控制流程也十分简明，只需对RPN表达式做单向的顺序扫描，既无需更多判断，也不含任何分支或回溯。

算法4.1的一次完整运行过程，如表4.3所示。

表4.3 RPN表达式求值算法实例（当前字符以方框注明，操作数栈的底部靠左）

操作数栈	表 达 式	注 解
	0 ! 1 + 2 3 ! 4 + ^ * 5 ! 67 - 8 9 + - -	初始化，引入操作数栈
0	[0] ! 1 + 2 3 ! 4 + ^ * 5 ! 67 - 8 9 + - -	操作数0入栈
1	0 [!] 1 + 2 3 ! 4 + ^ * 5 ! 67 - 8 9 + - -	0出栈，运算'!'结果入栈
1 1	0 ! [1] + 2 3 ! 4 + ^ * 5 ! 67 - 8 9 + - -	操作数1入栈
2	0 ! 1 [+] 2 3 ! 4 + ^ * 5 ! 67 - 8 9 + - -	1和1出栈，运算'+'结果入栈
2 2	0 ! 1 + [2] 3 ! 4 + ^ * 5 ! 67 - 8 9 + - -	操作数2入栈
2 2 3	0 ! 1 + 2 [3] ! 4 + ^ * 5 ! 67 - 8 9 + - -	操作数3入栈
2 2 6	0 ! 1 + 2 3 [!] 4 + ^ * 5 ! 67 - 8 9 + - -	3出栈，运算'!'结果入栈
2 2 6 4	0 ! 1 + 2 3 ! [4] + ^ * 5 ! 67 - 8 9 + - -	操作数4入栈

操作数栈	表 达 式	注　解
2 2 10	0 ! 1 + 2 3 ! 4 [+] ^ * 5 ! 67 - 8 9 + - -	6和4出栈，运算'+'结果入栈
2 1024	0 ! 1 + 2 3 ! 4 + [^] * 5 ! 67 - 8 9 + - -	2和10出栈，运算'^'结果入栈
2048	0 ! 1 + 2 3 ! 4 + ^ [*] 5 ! 67 - 8 9 + - -	2和1024出栈，运算'*'结果入栈
2048 5	0 ! 1 + 2 3 ! 4 + ^ * [5] ! 67 - 8 9 + - -	操作数5入栈
2048 120	0 ! 1 + 2 3 ! 4 + ^ * 5 [!] 67 - 8 9 + - -	5出栈，运算'!'结果入栈
2048 120 67	0 ! 1 + 2 3 ! 4 + ^ * 5 ! [67] - 8 9 + - -	操作数67入栈
2048 53	0 ! 1 + 2 3 ! 4 + ^ * 5 ! 67 [-] 8 9 + - -	120和67出栈，运算'-'结果入栈
2048 53 8	0 ! 1 + 2 3 ! 4 + ^ * 5 ! 67 - [8] 9 + - -	操作数8入栈
2048 53 8 9	0 ! 1 + 2 3 ! 4 + ^ * 5 ! 67 - 8 [9] + - -	操作数9入栈
2048 53 17	0 ! 1 + 2 3 ! 4 + ^ * 5 ! 67 - 8 9 [+] - -	8和9出栈，运算'+'结果入栈
2048 36	0 ! 1 + 2 3 ! 4 + ^ * 5 ! 67 - 8 9 + [-] -	53和17出栈，运算'-'结果入栈
2012	0 ! 1 + 2 3 ! 4 + ^ * 5 ! 67 - 8 9 + - [-]	2048和36出栈，运算'-'结果入栈

可见，只有操作数可能需要借助栈S做缓存，运算符则均可直接执行而不必保留。

另外，只要RPN表达式合法，在整个求值计算的过程中，当前运算符所需的操作数无论多少，都必然恰好按次序存放在当前栈的顶部。当上例处理到运算符'^'时，对应的操作数2和10恰为次栈顶和栈顶；当处理到运算符'*'时，对应的操作数2和1024也恰为次栈顶和栈顶。

■ 手工转换

按照以下步骤，即可完成从中缀表达式到RPN的转换。以如下中缀表达式为例（习题[4-9]）：

(0 ! + 1) * 2 ^ (3 ! + 4) - (5 ! - 67 - (8 + 9))

首先，假设在事先并未就运算符之间的优先级做过任何约定。于是，我们不得不通过增添足够多的括号，以如下方式，显式地指定该表达式的运算次序：

((((0) ! + 1) * (2 ^ ((3) ! + 4))) - (((5) ! - 67) - (8 + 9)))

然后，将各运算符后移，使之紧邻于其对应的右括号的右侧：

((((0) ! 1) + (2 ((3) ! 4) +) ^) * (((5) ! 67) - (8 9) +) -) -

最后抹去所有括号：

0 ! 1 + 2 3 ! 4 + ^ * 5 ! 67 - 8 9 + - -

稍事整理，即得到对应的RPN表达式：

0 ! 1 + 2 3 ! 4 + ^ * 5 ! 67 - 8 9 + - -

可见，操作数之间的相对次序，在转换前后保持不变；而运算符在RPN中所处的位置，恰好就是其对应的操作数均已就绪且该运算可以执行的位置。

■ 自动转换

实际上，95页代码4.7中evaluate()算法在对表达式求值的同时，也顺便完成了从常规表达式到RPN表达式的转换。在求值过程中，该算法借助append()函数（习题[4-8]）将各操作数和运算符适时地追加至串rpn的末尾，直至得到完整的RPN表达式（习题[4-9]）。

这里采用的规则十分简明：凡遇到操作数，即追加至rpn；而运算符只有在从栈中弹出并执行时，才被追加。这一过程，与上述手工转换的方法完全等效，其正确性也因此得以确立。

将RPN自动转换过程与RPN求值过程做一对照，即不难看出，后者只不过是前者的忠实再现。

§4.4 *试探回溯法

4.4.1 试探与回溯

■ 忒修斯的法宝

古希腊神话中半人半牛的怪物弥诺陶洛斯（Minotaur），藏身于一个精心设计、结构极其复杂的迷宫之中。因此，找到并消灭它绝非易事，而此后如何顺利返回而不致困死更是一个难题。不过，在公主阿里阿德涅（Ariadne）的帮助下，英雄忒修斯（Theseus）还是想出了个好办法，他最终消灭了怪物，并带着公主轻松地走出迷宫。

实际上，忒修斯所使用的法宝，只不过是一团普通的线绳。他将线绳的一端系在迷宫的入口处，而在此后不断检查各个角落的过程中，线团始终握在他的手中。线团或收或放，跟随着忒修斯穿梭于蜿蜒曲折的迷宫之中，确保他不致迷路。

忒修斯的高招，与现代计算机中求解很多问题的算法异曲同工。事实上，很多应用问题的解，在形式上都可看作若干元素按特定次序构成的一个序列。以经典的旅行商问题（traveling salesman problem, TSP）为例，其目标是计算出由给定的n个城市构成的一个序列，使得按此序列对这些城市的环游成本（比如机票价格）最低。尽管此类问题本身的描述并不复杂，但遗憾的是，由于所涉及元素（比如城市）的每一排列都是一个候选解，它们往往构成一个极大的搜索空间。通常，其搜索空间的规模与全排列总数大体相当，为$n! = \mathcal{O}(n^n)$。因此若采用蛮力策略，逐一生成可能的候选解并检查其是否合理，则必然无法将运行时间控制在多项式的范围以内。

■ 剪枝

为此，必须基于对应用问题的深刻理解，利用问题本身具有的某些规律尽可能多、尽可能早地排除搜索空间中的候选解。其中一种重要的技巧就是，根据候选解的某些局部特征，以候选解子集为单位批量地排除。通常如图4.7所示，搜索空间多呈树状结构，而被排除的候选解往往隶属于同一分支，故这一技巧也可以形象地称作剪枝（pruning）。

与之对应的算法多呈现为如下模式。从零开始，尝试逐步增加候选解的长度。更准确地，这一过程是在成批地考查具有特定前缀的所有候选解。这种从长度上逐步向目标解靠近的尝试，称作试探（probing）。作为解的局部特征，特征前缀在试探的过程中一旦被发现与目标解不合，则收缩到此前一步的长度，然后继续试探下一可能的组合。特征前缀长度缩减的这类操作，称作回溯（backtracking），其效果等同于剪枝。如此，只要目标解的确存在就迟早会被发现，而且只要剪枝所依据的特征设计得当，计算的效率就会大大提高。

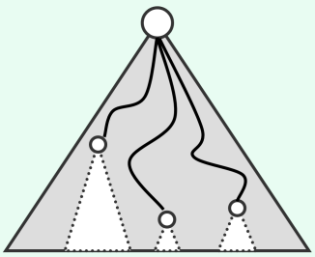

图4.7 通过剪枝排除候选解子集

■ 线绳与粉笔

回到开头的传说故事。不难看出，忒修斯藉以探索迷宫的正是试探回溯法。当然，这一方法的真正兑现还依赖于有形的物质基础——忒修斯的线绳。忒修斯之所以能够在迷宫中有条不紊地进行搜索，首先是得益于这团收放自如的线绳。这一点不难理解，所有算法的实现都必须建立在特定的数据结构之上。

以下两个实例，将介绍如何借助适当的数据结构以高效地实现试探回溯策略。我们将看到，栈结构在此过程中所扮演的正是忒修斯手中线绳的角色。当然，这里还需解决故事中隐含的另一技术难点：如何保证搜索过的部分不被重复搜索。办法之一就是，在剪枝的位置留下某种标记。同样地，这类标记也需兑现为具体的数据结构。倘若建议忒修斯在回溯时不妨用粉笔就地做个记号，那么我们的算法也应配有以数据结构形式实现的“粉笔”。

4.4.2 八皇后

■ 问题描述

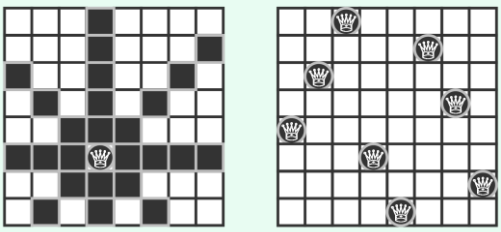

图4.8 (a)皇后的控制范围；(b)8皇后问题的一个解

如图4.8(a)，国际象棋中皇后的势力范围覆盖其所在的水平线、垂直线以及两条对角线。现考查如下问题：在n×n的棋盘上放置n个皇后，如何使得她们彼此互不攻击——此时称她们构成一个可行的棋局。对于任何整数$n \geq 4$，这就是n皇后问题。

由鸽巢原理可知，在n行n列的棋盘上至多只能放置n个皇后。反之，n个皇后在n×n棋盘上的可行棋局通常也存在，比如图4.8(b)即为在8×8棋盘上，由8个皇后构成的一个可行棋局。

■ 皇后

皇后是组成棋局和最终解的基本单元，故可如代码4.8所示实现对应的Queen类。

```
1  struct Queen { //皇后类
2     int x, y; //皇后在棋盘上的位置坐标
3     Queen ( int xx = 0, int yy = 0 ) : x ( xx ), y ( yy ) {};
4     bool operator== ( Queen const& q ) const { //重载判等操作符，以检测不同皇后之间可能的冲突
5        return    ( x == q.x ) //行冲突（这一情况其实并不会发生，可省略）
6                  || ( y == q.y ) //列冲突
7                  || ( x + y == q.x + q.y ) //沿正对角线冲突
8                  || ( x - y == q.x - q.y ); //沿反对角线冲突
9     }
10    bool operator!= ( Queen const& q ) const { return ! ( *this == q ); } //重载不等操作符
11 };
```

代码4.8 皇后类

可见，每个皇后对象均由其在棋盘上的位置坐标确定。此外，这里还通过重载判等操作符，实现了对皇后位置是否相互冲突的便捷判断。具体地，这里按照以上棋规，将同行、同列或同对角线的任意两个皇后视作“相等”，于是两个皇后相互冲突当且仅当二者被判作“相等”。

■ 算法实现

基于试探回溯策略，可如代码4.9所示，实现通用的N皇后算法。

既然每行能且仅能放置一个皇后，故不妨首先将各皇后分配至每一行。然后，从空棋盘开始，逐个尝试着将她们放置到无冲突的某列。每放置好一个皇后，才继续试探下一个。若当前皇后在任何列都会造成冲突，则后续皇后的试探都必将是徒劳的，故此时应该回溯到上一皇后。

```
1  void placeQueens ( int N ) { //N皇后算法（迭代版）：采用试探/回溯的策略，借助栈记录查找的结果
2     Stack<Queen> solu; //存放（部分）解的栈
3     Queen q ( 0, 0 ); //从原点位置出发
4     do { //反复试探、回溯
5        if ( N <= solu.size() || N <= q.y ) { //若已出界，则
6           q = solu.pop(); q.y++; //回溯一行，并继续试探下一列
7        } else { //否则，试探下一行
8           while ( ( q.y < N ) && ( 0 <= solu.find ( q ) ) ) //通过与已有皇后的比对
9              { q.y++; nCheck++; } //尝试找到可摆放下一皇后的列
10          if ( N > q.y ) { //若存在可摆放的列，则
11             solu.push ( q ); //摆上当前皇后，并
12             if ( N <= solu.size() ) nSolu++; //若部分解已成为全局解，则通过全局变量nSolu计数
13             q.x++; q.y = 0; //转入下一行，从第0列开始，试探下一皇后
14          }
15       }
16    } while ( ( 0 < q.x ) || ( q.y < N ) ); //所有分支均已或穷举或剪枝之后，算法结束
17 }
```

代码4.9 N皇后算法

这里借助栈solu来动态地记录各皇后的列号。当该栈的规模增至N时，即得到全局解。该栈即可依次给出各皇后在可行棋局中所处的位置。

■ 实例

图4.9给出了利用以上算法，得到四皇后问题第一个解的完整过程。

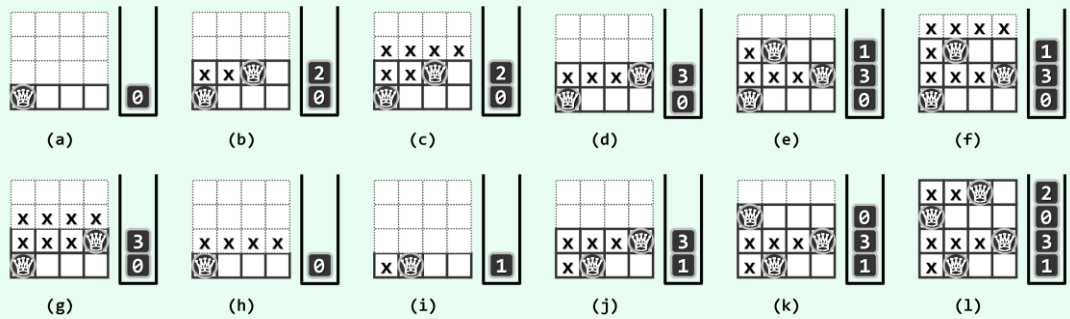

图4.9 四皇后问题求解过程（棋盘右侧为记录解的栈solu）

首先试探第一行皇后，如图(a)所示将其暂置于第0列，同时列号入栈。接下来试探再第二行皇后，如图(b)所示在排除前两列后，将其暂置于第2列，同时列号入栈。然而此后试探第三行皇后时，如图(c)所示发现所有列均有冲突。于是回溯到第二行，并如图(d)所示将第二行皇后调整到第3列，同时更新栈顶列号。后续各步原理相同，直至图(l)栈满时得到一个全局解。

如此不断地试探和回溯，即可得到所有可行棋局。可见，通过剪枝我们对原规模为4！= 24的搜索空间实现了有效的筛选。随着问题规模的增加，这一技巧的优化效果将更为明显。

4.4.3 迷宫寻径

■ **问题描述**

路径规划是人工智能的基本问题之一，要求依照约定的行进规则，在具有特定几何结构的空间区域内，找到从起点到终点的一条通路。以下考查该问题的一个简化版本：空间区域限定为由n × n个方格组成的迷宫，除了四周的围墙，还有分布其间的若干障碍物；只能水平或垂直移动。我们的任务是，在任意指定的起始格点与目标格点之间，找出一条通路（如果的确存在）。

■ **迷宫格点**

格点是迷宫的基本组成单位，故首先需要实现Cell类如代码4.10所示。

```
typedef enum { AVAILABLE, ROUTE, BACKTRACKED, WALL } Status; //迷宫单元状态
//原始可用的、在当前路径上的、所有方向均尝试失败后回溯过的、不可使用的（墙）

typedef enum { UNKNOWN, EAST, SOUTH, WEST, NORTH, NO_WAY } ESWN; //单元的相对邻接方向
//未定、东、南、西、北、无路可通

inline ESWN nextESWN ( ESWN eswn ) { return ESWN ( eswn + 1 ); } //依次转至下一邻接方向

struct Cell { //迷宫格点
   int x, y; Status status; //x坐标、y坐标、类型
   ESWN incoming, outgoing; //进入、走出方向
};

#define LABY_MAX 24 //最大迷宫尺寸
Cell laby[LABY_MAX][LABY_MAX]; //迷宫
```

代码4.10 迷宫格点类

可见，除了记录其位置坐标外，格点还需记录其所处的状态。共有四种可能的状态：原始可用的（AVAILABLE）、在当前路径上的（ROUTE）、所有方向均尝试失败后回溯过的（BACKTRACKED）、不可穿越的（WALL）。属于当前路径的格点，还需记录其前驱和后继格点的方向。既然只有上、下、左、右四个连通方向，故以EAST、SOUTH、WEST和NORTH区分。特别地，因尚未搜索到而仍处于初始AVAILABLE状态的格点，邻格的方向都是未知的（UNKNOWN）；经过回溯后处于BACKTRACKED状态的格点，与邻格之间的连通关系均已关闭，故标记为NO_WAY。

■ **邻格查询**

在路径试探过程中需反复确定当前位置的相邻格点，可如代码4.11所示实现查询功能。

```
inline Cell* neighbor ( Cell* cell ) { //查询当前位置的相邻格点
   switch ( cell->outgoing ) {
      case EAST  : return cell + LABY_MAX; //向东
      case SOUTH : return cell + 1;        //向南
      case WEST  : return cell - LABY_MAX; //向西
      case NORTH : return cell - 1;        //向北
```

```
7       default    : exit ( -1 );
8    }
9 }
```

代码4.11 查询相邻格点

■ 邻格转入

在确认某一相邻格点可用之后，算法将朝对应的方向向前试探一步，同时路径延长一个单元。为此，需如代码4.12所示实现相应的格点转入功能。

```
1 inline Cell* advance ( Cell* cell ) { //从当前位置转入相邻格点
2    Cell* next;
3    switch ( cell->outgoing ) {
4       case EAST:  next = cell + LABY_MAX; next->incoming = WEST;  break; //向东
5       case SOUTH: next = cell + 1;        next->incoming = NORTH; break; //向南
6       case WEST:  next = cell - LABY_MAX; next->incoming = EAST;  break; //向西
7       case NORTH: next = cell - 1;        next->incoming = SOUTH; break; //向北
8       default : exit ( -1 );
9    }
10   return next;
11 }
```

代码4.12 转入相邻格点

■ 算法实现

在以上功能的基础上，可基于试探回溯策略实现寻径算法如代码4.13所示。

```
1 // 迷宫寻径算法：在格单元s至t之间规划一条通路（如果的确存在）
2 bool labyrinth ( Cell Laby[LABY_MAX][LABY_MAX], Cell* s, Cell* t ) {
3    if ( ( AVAILABLE != s->status ) || ( AVAILABLE != t->status ) ) return false; //退化情况
4    Stack<Cell*> path; //用栈记录通路（Theseus的线绳）
5    s->incoming = UNKNOWN; s->status = ROUTE; path.push ( s ); //起点
6    do { //从起点出发不断试探、回溯，直到抵达终点，或者穷尽所有可能
7       Cell* c = path.top(); //检查当前位置（栈顶）
8       if ( c == t ) return true; //若已抵达终点，则找到了一条通路；否则，沿尚未试探的方向继续试探
9       while ( NO_WAY > ( c->outgoing = nextESWN ( c->outgoing ) ) ) //逐一检查所有方向
10         if ( AVAILABLE == neighbor ( c )->status ) break; //试图找到尚未试探的方向
11      if ( NO_WAY <= c->outgoing ) //若所有方向都已尝试过
12         { c->status = BACKTRACKED; c = path.pop(); }//则向后回溯一步
13      else //否则，向前试探一步
14         { path.push ( c = advance ( c ) ); c->outgoing = UNKNOWN; c->status = ROUTE; }
15   } while ( !path.empty() );
16   return false;
17 }
```

代码4.13 迷宫寻径

该问题的搜索过程中，局部解是一条源自起始格点的路径，它随着试探、回溯相应地伸长、缩短。因此，这里借助栈path按次序记录组成当前路径的所有格点，并动态地随着试探、回溯做入栈、出栈操作。路径的起始格点、当前的末端格点分别对应于path的栈底和栈顶，当后者抵达目标格点时搜索成功，此时path所对应的路径即可作为全局解返回。

■ 实例

图4.10给出了以上迷宫寻径算法的一次运行实例。

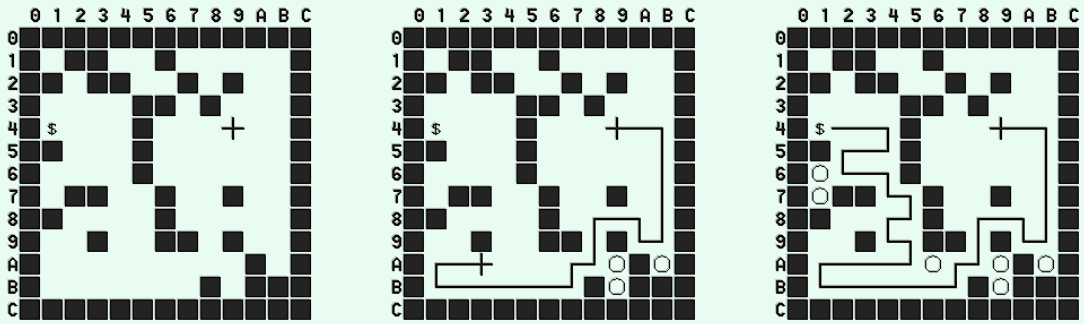

图4.10 迷宫寻径算法实例

左侧为随机生成的13×13迷宫。算法启动时，其中格点分为可用（AVAILABLE，白色）与障碍（WALL，黑色）两种状态。在前一类中，随机指定了起始格点（+）和目标格点（$)。中图为算法执行过程的某一时刻，可见原先为可用状态的一些格点已经转换为新的状态：转入ROUTE状态的格点，依次联接构成一条（尚未完成的）通路；曾参与构成通路但后因所有前进方向均已尝试完毕而回溯的格点，则进而从ROUTE转入TRIED状态（以圆圈注明）。

■ 正确性

该算法会尝试当前格点的所有相邻格点，因此通过数学归纳可知，若在找到全局解后依然继续查找，则该算法可以抵达与起始格点连通的所有格点。因此，只要目标格点与起始格点的确相互连通，则这一算法必将如右图所示找出一条联接于二者之间的通路。

从算法的中间过程及最终结果都可清晰地看出，这里用以记录通路的栈结构的确相当于忒修斯手中的线绳，它确保了算法可沿着正确地方向回溯。另外，这里给所有回溯格点所做的状态标记则等效于用粉笔做的记号，正是这些标记确保了格点不致被重复搜索，从而有效地避免了沿环路的死循环现象。

■ 复杂度

算法的每一步迭代仅需常数时间，故总体时间复杂度线性正比于试探、回溯操作的总数。由于每个格点至多参与试探和回溯各一次，故亦可度量为所有被访问过的格点总数——在图4.10中，也就是最终路径的总长度再加上圆圈标记的数目。

§4.5 队列

4.5.1 概述

■ 入队与出队

与栈一样，队列（queue）也是存放数据对象的一种容器，其中的数据对象也按线性的逻辑

次序排列。队列结构同样支持对象的插入和删除，但两种操作的范围分别被限制于队列的两端——若约定新对象只能从某一端插入其中，则只能从另一端删除已有的元素。允许取出元素的一端称作队头（front），而允许插入元素的另一端称作队尾（rear）。

以如图4.11所示顺序盛放羽毛球的球桶为例。通常，我们总是从球托所指的一端将球取出，而从另一端把球纳入桶中。因此如果将球托所指的一端理解为队头，另一端理解为队尾，则桶中的羽毛球即构成一个队列，其中每只球都属于该队列的一个元素。

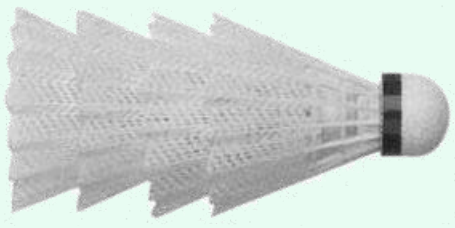

图4.11 在球桶中顺序排列的一组羽毛球可视作一个队列

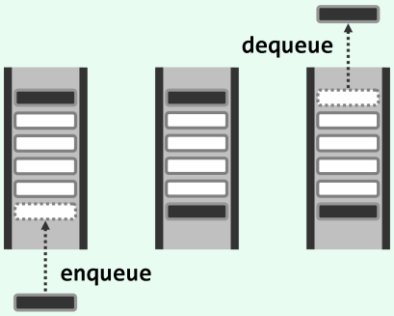

图4.12 队列操作

一般地如图4.12所示，元素的插入与删除也是修改队列结构的两种主要方式，站在被操作对象的角度，分别称作入队（enqueue）和出队（dequeue）操作。

■ 先进先出

由以上的约定和限制不难看出，与栈结构恰好相反，队列中各对象的操作次序遵循所谓先进先出（first-in-first-out，FIFO）的规律：更早（晚）出队的元素应为更早（晚）入队者，反之，更早（晚）入队者应更早（晚）出队。

4.5.2 ADT接口

作为一种抽象数据类型，队列结构必须支持以下操作接口。

表4.4 队列ADT支持的操作接口

操 作	功 能
size()	报告队列的规模（元素总数）
empty()	判断队列是否为空
enqueue(e)	将e插入队尾
dequeue()	删除队首对象
front()	引用队首对象

4.5.3 操作实例

按照表4.4定义的ADT接口，表4.5给出了一个队列从被创建开始，经过一系列操作的过程。

表4.5 队列操作实例（元素均为整型）

操作	输出	队列（右侧为队头）	操作	输出	队列（右侧为队头）

操作	输出	队列（右侧为队头）
Queue()		
empty()	true	
enqueue(5)		5
enqueue(3)		3 5
dequeue()	5	3
enqueue(7)		7 3
enqueue(3)		3 7 3
front()	3	3 7 3
empty()	false	3 7 3

操作	输出	队列（右侧为队头）
enqueue(11)		11 3 7 3
size()	4	11 3 7 3
enqueue(6)		6 11 7 3
empty()	false	6 11 3 7 3
enqueue(7)		6 11 3 7 3
dequeue()	3	7 6 11 3 7
dequeue()	7	7 6 11 3
front()	3	7 6 11 3
size()	4	7 6 11 3

4.5.4 Queue模板类

既然队列也可视作序列的特例，故只要将队列作为列表的派生类，即可利用C++的继承机制，基于3.2.2节已实现的列表模板类，实现队列结构。同样地，也需要按照队列的习惯对各相关的接口重新命名。按照表4.4所列的ADT接口，可描述并实现Queue模板类如下。

```
1 #include "../List/List.h" //以List为基类
2 template <typename T> class Queue: public List<T> { //队列模板类（继承List原有接口）
3 public: //size()、empty()以及其它开放接口均可直接沿用
4    void enqueue ( T const& e ) { insertAsLast ( e ); } //入队：尾部插入
5    T dequeue() { return remove ( first() ); } //出队：首部删除
6    T& front() { return first()->data; } //队首
7 };
```

代码4.14 Queue模板类

由代码4.14可见，队列的enqueue()操作等效于将新元素作为列表的末元素插入，dequeue()操作则等效于删除列表的首元素，front()操作可直接返回对列表首元素的引用。而size()及empty()等接口，均可直接沿用基类的同名接口。

这里插入和删除操作的位置分别限制于列表的末端和首端，故由3.3.5节的分析结论可知，队列结构以上接口的时间复杂度均为常数。

套用以上思路，也可直接基于2.2.3节的Vector模板类派生出Stack类（习题[4-2]）。

§4.6 队列应用

4.6.1 循环分配器

为在客户（client）群体中共享的某一资源（比如多个应用程序共享同一CPU），一套公平且高效的分配规则必不可少，而队列结构则非常适于定义和实现这样的一套分配规则[①]。

具体地，可以借助队列Q实现一个资源循环分配器，其总体流程大致如算法4.2所示。

```
RoundRobin { //循环分配器
   Queue Q(clients); //参与资源分配的所有客户组成队列Q
   while (!ServiceClosed()) { //在服务关闭之前，反复地
      e = Q.dequeue(); //队首的客户出队，并
      serve(e); //接受服务，然后
      Q.enqueue(e); //重新入队
   }
}
```

算法4.2 利用队列结构实现的循环分配器

在以上所谓轮值（round robin）算法中，首先令所有参与资源分配的客户组成一个队列Q。接下来是一个反复轮回式的调度过程：取出当前位于队头的客户，将资源交予该客户使用；在经过固定的时间之后，回收资源，并令该客户重新入队。得益于队列“先进先出”的特性，如此既可在所有客户之间达成一种均衡的公平，也可使得资源得以充分利用。

这里，每位客户持续占用资源的时间，对该算法的成败至关重要。一方面，为保证响应速度，这一时间值通常都不能过大。另一方面，因占有权的切换也需要耗费一定的时间，故若该时间值取得过小，切换过于频繁，又会造成整体效率的下降。因此，往往需要通过实测确定最佳值。

反过来，在单一客户使用多个资源的场合，队列也可用以保证资源的均衡使用，提高整体使用效率。针式打印机配置的色带，即是这样的一个实例，环形[②]色带收纳于两端开口的色带盒内。在打印过程中，从一端不断卷出的色带，在经过打印头之后，又从另一端重新卷入盒中，并如此往复。可见，此处色带盒的功能等效于一个队列，色带的各部分按照“先进先出”的原则被均衡地使用，整体使用寿命因而得以延长。

4.6.2 银行服务模拟

以下以银行这一典型场景为例，介绍如何利用队列结构实现顾客服务的调度与优化。

通常，银行都设有多个窗口，顾客按到达的次序分别在各窗口排队等待办理业务。为此，可首先定义顾客类Customer如下，以记录顾客所属的队列及其所办业务的服务时长。

```
1 struct Customer { int window; unsigned int time; }; //顾客类：所属窗口（队列）、服务时长
```

代码4.15 顾客对象

① 更复杂条件和需求下的调度分配算法，可参考排队论（queuing theory）的相关资料

② 严格地说，色带是个Möbius环，如此可进一步保证其“两”面都能被均衡使用

顾客在银行中接受服务的整个过程，可由如代码4.16所示的simulate()函数模拟。

```
void simulate ( int nWin, int servTime ) { //按指定窗口数、服务总时间模拟银行业务
   Queue<Customer>* windows = new Queue<Customer>[nWin]; //为每一窗口创建一个队列
   for ( int now = 0; now < servTime; now++ ) { //在下班之前，每隔一个单位时间
      if ( rand() % ( 1 + nWin ) ) { //新顾客以nWin/(nWin + 1)的概率到达
         Customer c ; c.time = 1 + rand() % 98; //新顾客到达，服务时长随机确定
         c.window = bestWindow ( windows, nWin ); //找出最佳（最短）的服务窗口
         windows[c.window].enqueue ( c ); //新顾客加入对应的队列
      }
      for ( int i = 0; i < nWin; i++ ) //分别检查
         if ( !windows[i].empty() ) //各非空队列
            if ( -- windows[i].front().time <= 0 ) //队首顾客的服务时长减少一个单位
               windows[i].dequeue(); //服务完毕的顾客出列，由后继顾客接替
   } //for
   delete [] windows; //释放所有队列（此前，~List()会自动清空队列）
}
```

代码4.16 银行服务模拟

这里，首先根据银行所设窗口的数量相应地建立多个队列。以下以单位时间为间隔反复迭代，直至下班。每一时刻都有一位顾客按一定的概率抵达，随机确定所办业务服务时长之后，归入某一“最优”队列。每经单位时间，各队列最靠前顾客（如果有的话）的待服务时长均相应减少一个单位。若时长归零，则意味着该顾客的业务已办理完毕，故应退出队列并由后一位顾客（如果有的话）接替。可见，顾客归入队列和退出队列的事件可分别由enqueue()和dequeue()操作模拟，查询并修改队首顾客时长的事件则可由front()操作模拟。

```
int bestWindow ( Queue<Customer> windows[], int nWin ) { //为新到顾客确定最佳队列
   int minSize = windows[0].size(), optiWin = 0; //最优队列（窗口）
   for ( int i = 1; i < nWin; i++ ) //在所有窗口中
      if ( minSize > windows[i].size() ) //挑选出
         { minSize = windows[i].size(); optiWin = i; } //队列最短者
   return optiWin; //返回
}
```

代码4.17 查找最短队列

为更好地为新到顾客确定一个队列，这里采用了“最短优先”的原则。如代码4.17所示，为此只需遍历所有的队列并通过size()接口比较其规模，即可找到其中的最短者。