第 1 章　计算机系统知识

1.1　计算机系统基础知识

1.1.1　计算机系统硬件基本组成

计算机系统是由硬件和软件组成的，它们协同工作来运行程序。计算机的基本硬件系统由运算器、控制器、存储器、输入设备和输出设备 5 大部件组成。运算器、控制器等部件被集成在一起统称为中央处理单元（Central Processing Unit，CPU）。CPU 是硬件系统的核心，用于数据的加工处理，能完成各种算术、逻辑运算及控制功能。存储器是计算机系统中的记忆设备，分为内部存储器和外部存储器。前者速度高、容量小，一般用于临时存放程序、数据及中间结果。而后者容量大、速度慢，可以长期保存程序和数据。输入设备和输出设备合称为外部设备（简称外设），输入设备用于输入原始数据及各种命令，而输出设备则用于输出计算机运行的结果。

1.1.2　中央处理单元

中央处理单元（CPU）是计算机系统的核心部件，它负责获取程序指令、对指令进行译码并加以执行。

1．CPU 的功能

（1）程序控制。CPU 通过执行指令来控制程序的执行顺序，这是 CPU 的重要功能。

（2）操作控制。一条指令功能的实现需要若干操作信号配合来完成，CPU 产生每条指令的操作信号并将操作信号送往对应的部件，控制相应的部件按指令的功能要求进行操作。

（3）时间控制。CPU 对各种操作进行时间上的控制，即指令执行过程中操作信号的出现时间、持续时间及出现的时间顺序都需要进行严格控制。

（4）数据处理。CPU 通过对数据进行算术运算及逻辑运算等方式进行加工处理，数据加工处理的结果被人们所利用。所以，对数据的加工处理也是 CPU 最根本的任务。

此外，CPU 还需要对系统内部和外部的中断（异常）做出响应，进行相应的处理。

2．CPU 的组成

CPU 主要由运算器、控制器、寄存器组和内部总线等部件组成，如图 1-1 所示。

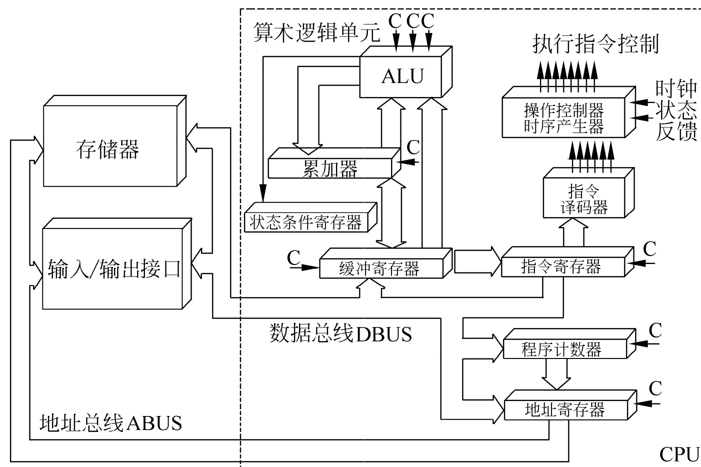


图 1-1　CPU 基本组成结构示意图

1）运算器

运算器由算术逻辑单元（Arithmetic and Logic Unit，ALU）、累加寄存器、数据缓冲寄存器和状态条件寄存器等组成，它是数据加工处理部件，用于完成计算机的各种算术和逻辑运算。相对控制器而言，运算器接受控制器的命令而进行动作，即运算器所进行的全部操作都是由控制器发出的控制信号来指挥的，所以它是执行部件。运算器有如下两个主要功能。

（1）执行所有的算术运算，例如加、减、乘、除等基本运算及附加运算。

（2）执行所有的逻辑运算并进行逻辑测试，例如与、或、非、零值测试或两个值的比较等。

下面简要介绍运算器中各组成部件的功能。

（1）算术逻辑单元（ALU）。ALU 是运算器的重要组成部件，负责处理数据，实现对数据的算术运算和逻辑运算。

（2）累加寄存器（AC）。AC 通常简称为累加器，它是一个通用寄存器，其功能是当运算器的算术逻辑单元执行算术或逻辑运算时，为 ALU 提供一个工作区。例如，在执行一个减法运算前，先将被减数取出暂存在 AC 中，再从内存储器中取出减数，然后同 AC 的内容相减，将所得的结果送回 AC 中。运算的结果是放在累加器中的，运算器中至少要有一个累加寄存器。

（3）数据缓冲寄存器（DR）。在对内存储器进行读/写操作时，用 DR 暂时存放由内存储器读/写的一条指令或一个数据字，将不同时间段内读/写的数据隔离开来。DR 的主要作用为：作为 CPU 和内存、外部设备之间数据传送的中转站；作为 CPU 和内存、外围设备之间在操作速度上的缓冲；在单累加器结构的运算器中，数据缓冲寄存器还可兼作为操作数寄存器。

（4）状态条件寄存器（PSW）。PSW 保存由算术指令和逻辑指令运行或测试的结果建立的各种条件码内容，主要分为状态标志和控制标志，例如运算结果进位标志（C）、运算结果溢出标志（V）、运算结果为 0 标志（Z）、运算结果为负标志（N）、中断标志（I）、方向标志（D）和单步标志等。这些标志通常分别由 1 位触发器保存，保存了当前指令执行完成之后的状态。通常，一个算术操作产生一个运算结果，而一个逻辑操作产生一个判决。

2）控制器

运算器只能完成运算，而控制器用于控制整个 CPU 的工作，它决定了计算机运行过程的自动化。它不仅要保证程序的正确执行，而且要能够处理异常事件。控制器一般包括指令控制逻辑、时序控制逻辑、总线控制逻辑和中断控制逻辑等几个部分。

指令控制逻辑要完成取指令、分析指令和执行指令的操作，其过程分为取指令、指令译码、按指令操作码执行、形成下一条指令地址等步骤。

（1）指令寄存器（IR）。当 CPU 执行一条指令时，先把它从内存储器取到缓冲寄存器中，再送入 IR 暂存，指令译码器根据 IR 的内容产生各种微操作指令，控制其他的组成部件工作，完成所需的功能。

（2）程序计数器（PC）。PC 具有寄存信息和计数两种功能，又称为指令计数器。程序的执行分两种情况，一是顺序执行，二是转移执行。在程序开始执行前，将程序的起始地址送入 PC，该地址在程序加载到内存时确定，因此 PC 的内容即是程序第一条指令的地址。执行指令时，CPU 自动修改 PC 的内容，以便使其保持的总是将要执行的下一条指令的地址。由于大多数指令都是按顺序来执行的，所以修改的过程通常只是简单地对 PC 加 1。当遇到转移指令时，后继指令的地址根据当前指令的地址加上一个向前或向后转移的位移量得到，或者根据转移指令给出的直接转移的地址得到。

（3）地址寄存器（AR）。AR 保存当前 CPU 所访问的内存单元的地址。由于内存和 CPU 存在着操作速度上的差异，所以需要使用 AR 保持地址信息，直到内存的读/写操作完成为止。

（4）指令译码器（ID）。指令包含操作码和地址码两部分，为了能执行任何给定的指令，必须对操作码进行分析，以便识别所完成的操作。指令译码器就是对指令中的操作码字段进行分析解释，识别该指令规定的操作，向操作控制器发出具体的控制信号，控制各部件工作，完成所需的功能。

时序控制逻辑要为每条指令按时间顺序提供应有的控制信号。总线逻辑是为多个功能部件

服务的信息通路的控制电路。中断控制逻辑用于控制各种中断请求，并根据优先级的高低对中断请求进行排队，逐个交给 CPU 处理。

3）寄存器组

寄存器组可分为专用寄存器和通用寄存器。运算器和控制器中的寄存器是专用寄存器，其作用是固定的。通用寄存器用途广泛并可由程序员规定其用途，其数目因处理器不同有所差异。

3．多核 CPU

核心又称为内核，是 CPU 最重要的组成部分。CPU 中心那块隆起的芯片就是核心，是由单晶硅以一定的生产工艺制造出来的，CPU 所有的计算、接收/存储命令、处理数据都由核心执行。各种 CPU 核心都具有固定的逻辑结构，一级缓存、二级缓存、执行单元、指令级单元和总线接口等逻辑单元都会有合理的布局。

多核即在一个单芯片上面集成两个甚至更多个处理器内核，其中，每个内核都有自己的逻辑单元、控制单元、中断处理器、运算单元，一级 Cache、二级 Cache 共享或独有，其部件的完整性和单核处理器内核相比完全一致。

CPU 的主要厂商 AMD 和 Intel 的双核技术在物理结构上有所不同。AMD 将两个内核做在一个 Die（晶元）上，通过直连架构连接起来，集成度更高。Intel 则是将放在不同核心上的两个内核封装在一起，因此将 Intel 的方案称为“双芯”，将 AMD 的方案称为“双核”。从用户端的角度来看，AMD 的方案能够使双核 CPU 的管脚、功耗等指标跟单核 CPU 保持一致，从单核升级到双核，不需要更换电源、芯片组、散热系统和主板，只需要刷新 BIOS 软件即可。

多核 CPU 系统最大的优点（也是开发的最主要目的）是可满足用户同时进行多任务处理的要求。

单核多线程 CPU 是交替地转换执行多个任务，只不过交替转换的时间很短，用户一般感觉不出来。如果同时执行的任务太多，就会感觉到“慢”或者“卡”。而多核在理论上则是在任何时间内每个核执行各自的任务，不存在交替问题。因此，单核多线程和多核（一般每核也是多线程的）虽然都可以执行多任务，但多核的速度更快。

虽然采用了 Intel 超线程技术的单核可以视为是双核，4 核可以视为是 8 核。然而，视为是 8 核一般比不上实际是 8 核的 CPU 性能。

要发挥 CPU 的多核性能，就需要操作系统能够及时、合理地给各个核分配任务和资源（如缓存、总线、内存等），也需要应用软件在运行时可以把并行的线程同时交付给多个核心分别处理。

1.1.3 数据表示

各种数值在计算机中表示的形式称为机器数，其特点是采用二进制计数制，数的符号用 0

和 1 表示，小数点则隐含，表示不占位置。机器数对应的实际数值称为数的真值。

机器数有无符号数和带符号数之分。无符号数表示正数，在机器数中没有符号位。对于无符号数，若约定小数点的位置在机器数的最低位之后，则是纯整数；若约定小数点的位置在机器数的最高位之前，则是纯小数。对于带符号数，机器数的最高位是表示正、负的符号位，其余位则表示数值。

为了便于运算，带符号的机器数可采用原码、反码和补码等不同的编码方法，机器数的这些编码方法称为码制。

1）原码、反码、补码和移码

（1）原码表示法。数值 X 的原码记为 $[X]_{原}$，如果机器字长为 n（即采用 n 个二进制位表示数据），则原码的定义如下：

若 X 是纯整数，则 $[X]_{原}=\begin{cases} X & 0 \leqslant X \leqslant 2^{n-1}-1 \\ 2^{n-1}+|X| & -(2^{n-1}-1) \leqslant X \leqslant 0 \end{cases}$

若 X 是纯小数，则 $[X]_{原}=\begin{cases} X & 0 \leqslant X < 1 \\ 2^{0}+|X| & -1 < X \leqslant 0 \end{cases}$

【例 1.1】 若机器字长 n 等于 8，分别给出+1，–1，+127，–127，+45，–45，+0.5，–0.5 的原码表示。

$[+1]_{原}$=0 0000001　　$[-1]_{原}$=1 0000001

$[+127]_{原}$=0 1111111　　$[-127]_{原}$=1 1111111

$[+45]_{原}$=0 0101101　　$[-45]_{原}$=1 0101101

$[+0.5]_{原}$=0 ◊1000000　　$[-0.5]_{原}$=1 ◊1000000（其中，◊ 是小数点的位置）

在原码表示法中，最高位是符号位，0 表示正号，1 表示负号，其余的 n–1 位表示数值的绝对值。数值 0 的原码表示有两种形式：$[+0]_{原}$=0 0000000，$[-0]_{原}$=1 0000000 。

（2）反码表示法。数值 X 的反码记作 $[X]_{反}$，如果机器字长为 n，则反码的定义如下：

若 X 是纯整数，则 $[X]_{反}=\begin{cases} X & 0 \leqslant X \leqslant 2^{n-1}-1 \\ 2^{n}-1+X & -(2^{n-1}-1) \leqslant X \leqslant 0 \end{cases}$

若 X 是纯小数，则 $[X]_{反}=\begin{cases} X & 0 \leqslant X < 1 \\ 2-2^{-(n-1)}+X & -1 < X \leqslant 0 \end{cases}$

【例 1.2】 若机器字长 n 等于 8，分别给出+1，–1，+127，–127，+45，–45，+0.5，–0.5 的反码表示。

$[+1]_{反}$=0 0000001　　$[-1]_{反}$=1 1111110

$[+127]_{反}$=0 1111111　　$[-127]_{反}$=1 0000000

$[+45]_{反}$=0 0101101　　$[-45]_{反}$=1 1010010

$[+0.5]_{反}$=0 ◊1000000　　$[-0.5]_{反}$=1◊ 0111111（其中，◊是小数点的位置）

在反码表示中，最高位是符号位，0 表示正号，1 表示负号，正数的反码与原码相同，负数的反码则是其绝对值按位求反。数值 0 的反码表示有两种形式：$[+0]_{反}$=0 0000000，$[-0]_{反}$=1 1111111。

（3）补码表示法。数值 X 的补码记作$[X]_{补}$，如果机器字长为 n，则补码的定义如下：

若 X 是纯整数，则 $[X]_{补}=\begin{cases} X & 0 \leqslant X \leqslant 2^{n-1}-1 \\ 2^n+X & -2^{n-1} \leqslant X \leqslant 0 \end{cases}$

若 X 是纯小数，则 $[X]_{补}=\begin{cases} X & 0 \leqslant X < 1 \\ 2+X & -1 \leqslant X < 0 \end{cases}$

【例 1.3】 若机器字长 n 等于 8，分别给出+1，–1，+127，–127，+45，–45，+0.5，–0.5 的补码表示。

$[+1]_{补}$=0 0000001　　$[-1]_{补}$=1 1111111

$[+127]_{补}$=0 1111111　　$[-127]_{补}$=1 0000001

$[+45]_{补}$=0 0101101　　$[-45]_{补}$=1 1010011

$[+0.5]_{补}$=0 ◊1000000　　$[-0.5]_{补}$=1◊ 1000000（其中，◊是小数点的位置）

在补码表示中，最高位为符号位，0 表示正号，1 表示负号，正数的补码与其原码和反码相同，负数的补码则等于其反码的末位加 1。在补码表示中，0 有唯一的编码：$[+0]_{补}$=0 0000000，$[-0]_{补}$=00000000。

（4）移码表示法。移码表示法是在数 X 上增加一个偏移量来定义的，常用于表示浮点数中的阶码。如果机器字长为 n，规定偏移量为 2^{n-1}，则移码的定义如下：

若 X 是纯整数，则 $[X]_{移}=2^{n-1}+X$ $(-2^{n-1} \leqslant X < 2^{n-1})$；若 X 是纯小数，则 $[X]_{补}$=1+X $(-1 \leqslant X < 1)$。

【例 1.4】 若机器字长 n 等于 8，分别给出+1，–1，+127，–127，+45，–45，+0，–0 的移码表示。

$[+1]_{移}$=1 0000001　　$[-1]_{移}$=0 1111111

$[+127]_{移}$=1 1111111　　$[-127]_{移}$=0 0000001

$[+45]_{移}$=1 0101101　　$[-45]_{移}$=0 1010011

$[+0]_{移}$=1 0000000　　$[-0]_{移}$=10000000

实际上，在偏移 2^{n-1} 的情况下，只要将补码的符号位取反便可获得相应的移码表示。

2）定点数和浮点数

（1）定点数。所谓定点数，就是小数点的位置固定不变的数。小数点的位置通常有两种约定方式：定点整数（纯整数，小数点在最低有效数值位之后）和定点小数（纯小数，小数点在

最高有效数值位之前）。

设机器字长为 n，各种码制下带符号数的范围如表 1-1 所示。

表 1-1　机器字长为 n 时各种码制表示的带符号数的范围

码　　制	定 点 整 数	定 点 小 数
原码	$-(2^{n-1}-1)\sim+(2^{n-1}-1)$	$-(1-2^{-(n-1)})\sim+(1-2^{-(n-1)})$
反码	$-(2^{n-1}-1)\sim+(2^{n-1}-1)$	$-(1-2^{-(n-1)})\sim+(1-2^{-(n-1)})$
补码	$-2^{n-1}\sim+(2^{n-1}-1)$	$-1\sim+(1-2^{-(n-1)})$
移码	$-2^{n-1}\sim+(2^{n-1}-1)$	$-1\sim+(1-2^{-(n-1)})$

（2）浮点数。当机器字长为 n 时，定点数的补码和移码可表示 2^n 个数，而其原码和反码只能表示 2^n-1 个数（0 的表示占用了两个编码），因此，定点数所能表示的数值范围比较小，在运算中很容易因结果超出范围而溢出。浮点数是小数点位置不固定的数，它能表示更大范围的数。

在十进制中，一个数可以写成多种表示形式。例如，83.125 可写成 $10^3\times0.083125$ 或 $10^4\times0.0083125$ 等。同样，一个二进制数也可以写成多种表示形式。例如，二进制数1011.10101 可以写成 $2^4\times0.101110101$、$2^5\times0.0101110101$ 或 $2^6\times0.00101110101$ 等。由此可知，一个二进制数 N 可以表示为更一般的形式 $N=2^E\times F$，其中 E 称为阶码，F 称为尾数。用阶码和尾数表示的数称为浮点数，这种表示数的方法称为浮点表示法。

在浮点表示法中，阶码为带符号的纯整数，尾数为带符号的纯小数。浮点数的表示格式如下：

阶符	阶码	数符	尾数

很明显，一个数的浮点表示不是唯一的。当小数点的位置改变时，阶码也随着相应改变，因此可以用多个浮点形式表示同一个数。

浮点数所能表示的数值范围主要由阶码决定，所表示数值的精度则由尾数决定。为了充分利用尾数来表示更多的有效数字，通常采用规格化浮点数。规格化就是将尾数的绝对值限定在区间[0.5, 1]。当尾数用补码表示时，需要注意如下问题。

① 若尾数 $M\geqslant0$，则其规格化的尾数形式为 $M=0.1\times\times\times\cdots\times$，其中，×可为 0，也可为 1，即将尾数限定在区间[0.5, 1]。

② 若尾数 $M<0$，则其规格化的尾数形式为 $M=1.0\times\times\times\cdots\times$，其中，×可为 0，也可为 1，即将尾数 M 的范围限定在区间[–1, –0.5]。

如果浮点数的阶码（包括 1 位阶符）用 R 位的移码表示，尾数（包括 1 位数符）用 M 位的补码表示，则这种浮点数所能表示的数值范围如下。

最大的正数：$+(1-2^{-M+1})\times 2^{(2^{R-1}-1)}$，最小的负数：$-1\times 2^{(2^{R-1}-1)}$

（3）工业标准 IEEE 754。IEEE 754 是由 IEEE 制定的有关浮点数的工业标准，被广泛采用。该标准的表示形式如下：

$$(-1)^S 2^E \left(b_0 b_1 b_2 b_3 \cdots b_{p-1}\right)$$

其中，$(-1)^S$ 为该浮点数的数符，当 S 为 0 时表示正数，S 为 1 时表示负数；E 为指数（阶码），用移码表示；$\left(b_0 b_1 b_2 b_3 \cdots b_{p-1}\right)$ 为尾数，其长度为 P 位，用原码表示。

目前，计算机中主要使用 3 种形式的 IEEE 754 浮点数，如表 1-2 所示。

表 1-2　3 种形式的 IEEE 754 浮点数格式

参　　数	单精度浮点数	双精度浮点数	扩充精度浮点数
浮点数字长	32	64	80
尾数长度 P	23	52	64
符号位 S	1	1	1
指数长度 E	8	11	15
最大指数	+127	+1023	+16 383
最小指数	–126	–1022	–16 382
指数偏移量	+127	+1023	+16 383
可表示的实数范围	$10^{-38}\sim10^{38}$	$10^{-308}\sim10^{308}$	$10^{-4932}\sim10^{4932}$

根据 IEEE 754 标准，被编码的值分为 3 种不同的情况：规格化的值、非规格化的值和特殊值，规格化的值为最普遍的情形。

① 规格化的值。

当阶码部分的二进制值不全为 0 也不全为 1 时，所表示的是规格化的值。例如，在单精度浮点格式下，阶码为 10110011 时，偏移量为+127（01111111），则其表示的真值为 10110011−01111111=00110100，转换为十进制后为 52。

对于尾数部分，由于约定小数点左边隐含有一位，通常这位数就是 1，因此单精度浮点数尾数的有效位数为 24 位，即尾数为 1.××…×。也就是说，不溢出的情况下尾数 M 的值在 1≤M<2 之中，这是一种获得一个额外精度位的表示技巧。

例如，单精度浮点数格式下，$b_0 b_1 \cdots b_{22} = 0100\,1001\,1000\,1000\,1001\,011$ 时，其对应的尾数真值为 $1+2^{-2}+2^{-5}+2^{-8}+2^{-9}+2^{-13}+2^{-17}+2^{-20}+2^{-22}+2^{-23}$ 即尾数的真值为 1.28724038600921630859375（在程序中以十进制方式输出时，由于精度的原因不能完全给出此值）。

【例 1.5】 利用 IEEE 754 标准将数 176.0625 表示为单精度浮点数。

解：首先将该十进制转换成二进制数。

$(176.0625)_{10}=(10110000.0001)_2$

其次对二进制数进行规格化处理：10110000.0001＝1◊01100000001×2^7。这就保证了使 b_0 为 1，而且小数点应当在◊位置上。将 b_0 去掉并扩展为单精度浮点数所规定的 23 位尾数 01100000001000000000000。

然后求阶码，上述表示中的指数为 7，而单精度浮点数规定指数的偏移量为 127（注意，不是前面移码描述中所提到的 128），即在指数 7 上加 127。那么，E=7+127=134，则指数的移码表示为 10000110。

最后，可得到$(176.0625)_{10}$的单精度浮点数表示形式：

0 10000110 01100000001000000000000

② 非规格化的值。

当阶码部分的二进制值全为 0 时，所表示的数是非规格化的。在这种情况下，指数的真值为 1−偏移量（对于单精度浮点数为−126，对于双精度浮点数为−1022），尾数的值就是二进制形式对应的小数，不包含隐含的 1。

非规格化数有两个用途：一是用来表示数值 0，二是表示那些非常接近于 0 的数。因为在规格化表示方式下，必须使尾数大于等于 1，因此不能表示出 0。实际上，+0.0 的浮点表示是符号、阶码和尾数的二进制表示都全为 0。需要注意的是，符号位为 1 而阶码和尾数部分全为 0 时表示−0.0。也就是说，+0.0 和−0.0 在浮点表示时有所不同。

③ 特殊值。

当阶码部分的二进制值全为 1 时，表示特殊的值。当尾数部分全部为 0 时表示无穷大，当符号位为 0 时表示+∞，当符号位为 1 时表示−∞。当浮点运算溢出时，用无穷来表示。当尾数部分不全为 0 时，称为“NaN”，即“不是一个数”。当运算结果不是实数或者无穷，就表示为 NaN。

（4）浮点数的运算。设有浮点数 $X=M\times 2^i$，$Y=N\times 2^j$，求 $X\pm Y$ 的运算过程要经过对阶、求尾数和（差）、结果规格化并判溢出、舍入处理和溢出判别等步骤。

① 对阶。使两个数的阶码相同。令 $K=|i-j|$，把阶码小的数的尾数右移 K 位，使其阶码加上 K。

② 求尾数和（差）。

③ 结果规格化并判溢出。若运算结果所得的尾数不是规格化的数，则需要进行规格化处理。当尾数溢出时，需要调整阶码。

④ 舍入处理。在对结果右规时，尾数的最低位将因移出而丢掉。另外，在对阶过程中也会将尾数右移使最低位丢掉。这就需要进行舍入处理，以求得最小的运算误差。

⑤ 溢出判别。以阶码为准，若阶码溢出，则运算结果溢出；若阶码下溢（小于最小值），

则结果为 0；否则结果正确，无溢出。

浮点数相乘，其积的阶码等于两乘数的阶码相加，积的尾数等于两乘数的尾数相乘。浮点数相除，其商的阶码等于被除数的阶码减去除数的阶码，商的尾数等于被除数的尾数除以除数的尾数。乘除运算的结果都需要进行规格化处理并判断阶码是否溢出。

1.1.4 校验码

计算机系统运行时，为了确保数据在传送过程中正确无误，一是提高硬件电路的可靠性，二是提高代码的校验能力，包括查错和纠错。通常使用校验码的方法来检测传送的数据是否出错。其基本思想是把数据可能出现的编码分为两类：合法编码和错误编码。合法编码用于传送数据，错误编码是不允许在数据中出现的编码。合理地设计错误编码以及编码规则，使得数据在传送中出现某种错误时会变成错误编码，这样就可以检测出接收到的数据是否有错。

所谓码距，是指一个编码系统中任意两个合法编码之间至少有多少个二进制位不同。例如，4 位 8421 码的码距为 1，在传输过程中，该代码的一位或多位发生错误，都将变成另外一个合法的编码，因此这种代码无检错能力。下面简要介绍常用的 3 种校验码：奇偶校验码、海明码和循环冗余校验码。

1．奇偶校验码

奇偶校验（Parity Codes）是一种简单有效的校验方法。这种方法通过在编码中增加一位校验位来使编码中 1 的个数为奇数（奇校验）或者为偶数（偶校验），从而使码距变为 2。对于奇校验，它可以检测代码中奇数位出错的编码，但不能发现偶数位出错的情况，即当合法编码中的奇数位发生了错误时，即编码中的 1 变成 0 或 0 变成 1，则该编码中 1 的个数的奇偶性就发生了变化，从而可以发现错误。

常用的奇偶校验码有 3 种：水平奇偶校验码、垂直奇偶校验码和水平垂直校验码。

2．海明码

海明码（Hamming Code）是由贝尔实验室的 Richard Hamming 设计的，是一种利用奇偶性来检错和纠错的校验方法。海明码的构成方法是在数据位之间的特定位置上插入 k 个校验位，通过扩大码距来实现检错和纠错。

设数据位是 n 位，校验位是 k 位，则 n 和 k 必须满足以下关系：

$$2^k - 1 \geqslant n + k$$

海明码的编码规则如下。

设 k 个校验位为 $P_k,P_{k-1},\cdots,P_1$，n 个数据位为 $D_{n-1},D_{n-2},\cdots,D_1,D_0$，对应的海明码为 $H_{n+k},H_{n+k-1},\cdots,H_1$，那么：

（1）P_i 在海明码的第 2^{i-1} 位置，即 $H_j=P_i$，且 $j=2^{i-1}$，数据位则依序从低到高占据海明码中剩下的位置。

（2）海明码中的任何一位都是由若干个校验位来校验的。其对应关系如下：被校验的海明位的下标等于所有参与校验该位的校验位的下标之和，而校验位由自身校验。

对于 8 位的数据位，进行海明校验需要 4 个校验位（$2^3-1=7, 2^4-1=15>8+4$）。令数据位为 $D_7,D_6,D_5,D_4,D_3,D_2,D_1,D_0$，校验位为 P_4,P_3,P_2,P_1，形成的海明码为 $H_{12}, H_{11},\cdots,H_3,H_2,H_1$，则编码过程如下。

（1）确定 D 与 P 在海明码中的位置，如下所示：

H_{12}	H_{11}	H_{10}	H_9	H_8	H_7	H_6	H_5	H_4	H_3	H_2	H_1
D_7	D_6	D_5	D_4	P_4	D_3	D_2	D_1	P_3	D_0	P_2	P_1

（2）确定校验关系，如表 1-3 所示。

表 1-3　海明码的校验关系表

海　明　码	海明码的下标	校 验 位 组	说明（偶校验）
$H_1 (P_1)$	1	P_1	P_1 校验：P_1、D_0、D_1、D_3、D_4、D_6 即 $P_1= D_0 \oplus D_1 \oplus D_3 \oplus D_4 \oplus D_6$
$H_2 (P_2)$	2	P_2	
$H_3 (D_0)$	3 = 1+2	P_1 , P_2	
$H_4 (P_3)$	4	P_3	P_2 校验：P_2、D_0、D_2、D_3、D_5、D_6 即 $P_2= D_0 \oplus D_2 \oplus D_3 \oplus D_5 \oplus D_6$
$H_5 (D_1)$	5 = 1+4	P_1 , P_3	
$H_6 (D_2)$	6 = 2+4	P_2 , P_3	
$H_7 (D_3)$	7 = 1+2+4	P_1 , P_2 , P_3	P_3 校验：P_3、D_1、D_2、D_3、D_7 即 $P_3= D_1 \oplus D_2 \oplus D_3 \oplus D_7$
$H_8 (P_4)$	8	P_4	
$H_9 (D_4)$	9 = 1+8	P_1 , P_4	
$H_{10} (D_5)$	10 = 2+8	P_2 , P_4	P_4 校验：P_4、D_4、D_5、D_6、D_7 即 $P_4= D_4 \oplus D_5 \oplus D_6 \oplus D_7$
$H_{11} (D_6)$	11 = 1+2+8	P_1 , P_2 , P_4	
$H_{12} (D_7)$	12 = 4+8	P_3 , P_4	

若采用奇校验，则将各校验位的偶校验值取反即可。

（3）检测错误。对使用海明编码的数据进行差错检测很简单，只需做以下计算：

$G_1 = P_1 \oplus D_0 \oplus D_1 \oplus D_3 \oplus D_4 \oplus D_6$

$G_2 = P_2 \oplus D_0 \oplus D_2 \oplus D_3 \oplus D_5 \oplus D_6$

$G_3 = P_3 \oplus D_1 \oplus D_2 \oplus D_3 \oplus D_7$

$G_4 = P_4 \oplus D_4 \oplus D_5 \oplus D_6 \oplus D_7$

若采用偶校验，则 $G_4G_3G_2G_1$ 全为 0 时表示接收到的数据无错误（奇校验应全为 1）。当

第3章 数据结构

数据结构是程序设计的重要基础，它所讨论的内容和技术对从事软件项目的开发有重要作用。学习数据结构要达到的目标是学会从问题出发，分析和研究计算机加工的数据的特性，以便为应用所涉及的数据选择适当的逻辑结构、存储结构及其相应的操作方法，为提高利用计算机解决问题的效率服务。

数据结构是指数据元素的集合及元素间的相互关系和构造方法。元素之间的相互关系是数据的逻辑结构，数据元素及元素之间关系的存储称为存储结构（或物理结构）。数据结构按照逻辑关系的不同分为线性结构和非线性结构两大类，其中，非线性结构又可分为树结构和图结构。

算法与数据结构密切相关，数据结构是算法设计的基础，设计合理的数据结构可使算法简单而高效。

3.1 线性结构

线性结构是一种基本的数据结构，主要用于对客观世界中具有单一前驱和后继的数据关系进行描述。线性结构的特点是数据元素之间呈现一种线性关系，即元素“一个接一个排列”。

3.1.1 线性表

线性表是最简单、最基本也是最常用的一种线性结构。常采用顺序存储和链式存储，主要的基本操作是插入、删除和查找等。

1．线性表的定义

一个线性表是 $n(n \geqslant 0)$个元素的有限序列，通常表示为（$a_1, a_2, \cdots, a_n$）。非空线性表的特点如下。

（1）存在唯一的一个称作“第一个”的元素。

（2）存在唯一的一个称作“最后一个”的元素。

（3）除第一个元素外，序列中的每个元素均只有一个直接前驱。

（4）除最后一个元素外，序列中的每个元素均只有一个直接后继。

2．线性表的存储结构

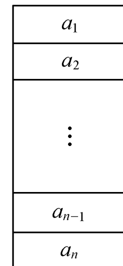


图 3-1　线性表的顺序存储

线性表的存储结构分为顺序存储和链式存储。

1）线性表的顺序存储

线性表的顺序存储是指用一组地址连续的存储单元依次存储线性表中的数据元素，从而使得逻辑上相邻的两个元素在物理位置上也相邻，如图 3-1 所示。在这种存储方式下，元素间的逻辑关系无须占用额外的空间来存储。

一般地，以 $\mathrm{LOC}(a_1)$表示线性表中第一个元素的存储位置，在顺序存储结构中，第 i 个元素 a_i 的存储位置为

$$\mathrm{LOC}(a_i)=\mathrm{LOC}(a_1)+(i-1)\times L$$

其中，L 是表中每个数据元素所占空间的字节数。根据该计算关系，可随机存取表中的任一个元素。

线性表采用顺序存储结构的优点是可以随机存取表中的元素，缺点是插入和删除操作需要移动元素。在插入前要移动元素以挪出空的存储单元，然后再插入元素；删除时同样需要移动元素，以填充被删除的元素空出来的存储单元。

在表长为 n 的线性表中插入新元素时，共有 n+1 个插入位置，在位置 1（元素 a_1 所在位置）插入新元素，表中原有的 n 个元素都需要移动，在位置 n+1（元素 a_n 所在位置之后）插入新元素时不需要移动任何元素，因此，等概率下（即新元素在 n+1 个位置插入的概率相同时）插入一个新元素需要移动的元素个数期望值 E_{insert} 为

$$E_{\text{insert}}=\sum_{i=1}^{n+1}P_i\times(n-i+1)=\frac{1}{n+1}\sum_{i=1}^{n+1}(n-i+1)=\frac{n}{2}$$

其中，P_{i} 表示在表中的位置 i 插入新元素的概率。

在表长为 n 的线性表中删除元素时，共有 n 个可删除的元素，删除元素 a_1 时需要移动 n–1 个元素，删除元素 a_n 时不需要移动元素，因此，在等概率下删除元素时需要移动的元素个数期望值 E_{delete} 为

$$E_{\text{delete}}=\sum_{i=1}^{n}q_i\times(n-i)=\frac{1}{n}\sum_{i=1}^{n}(n-i)=\frac{n-1}{2}$$

其中，q_i 表示删除第 i 个元素（即 a_i）的概率。

2）线性表的链式存储

线性表的链式存储是用通过指针链接起来的结点来存储数据元素，基本的结点结构如下所示：

数据域	指针域

其中，数据域用于存储数据元素的值，指针域则存储当前元素的直接前驱或直接后继的位置信息，指针域中的信息称为指针（或链）。

存储各数据元素的结点的地址并不要求是连续的，因此存储数据元素的同时必须存储元素之间的逻辑关系。另外，结点空间只有在需要的时候才申请，无须事先分配。

结点之间通过指针域构成一个链表，若结点中只有一个指针域，则称为线性链表（或单链表），如图 3-2 所示。

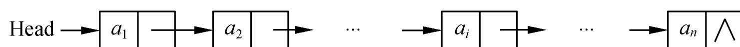


图 3-2　线性表的单链表存储

设线性表中的元素是整型，则单链表结点类型的定义为：

```
typedef struct node{
        int data;                   /*结点的数据域，此处假设为整型*/
        struct node *next;          /*结点的指针域*/
}NODE,*LinkList;
```

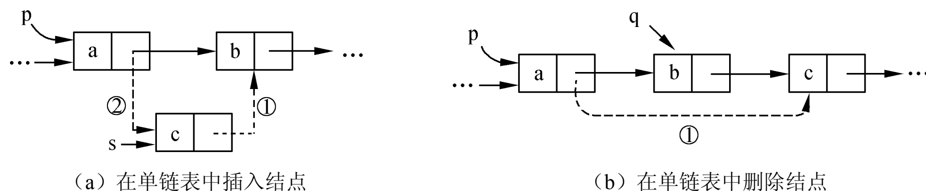
在链式存储结构中，只需要一个指针（称为头指针，如图 3-2 中的 head）指向第一个结点，就可以顺序地访问到表中的任意一个元素。

在链式存储结构下进行插入和删除，其实质都是对相关指针的修改。在单链表中，若在 p 所指结点后插入新元素结点（s 所指结点，已经生成），如图 3-3（a）所示，其基本步骤如下。

（1）s->next = p->next;

（2）p->next = s;

即先将 p 所指结点的后继结点指针赋给 s 所指结点的指针域，然后将 p 所指结点的指针域修改为指向 s 所指结点。



（a）在单链表中插入结点　　（b）在单链表中删除结点

图 3-3　在单链表中插入、删除结点时的指针变化示意图

同理，在单链表中删除 p 所指结点的后继结点时（如图 3-3（b）所示），步骤如下。

（1）q = p->next;

（2）p->next = p->next->next;

（3）free(q);

即先令临时指针 q 指向待删除的结点，然后修改 p 所指结点的指针域为指向 p 所指结点的后继的后继结点，从而将元素 b 所在的结点从链表中删除，最后释放 q 所指结点的空间。

在实际应用中，为了简化对链表状态的判定和处理，特别引入一个不存储数据元素的结点，称为头结点，将其作为链表的第一个结点并令头指针指向该结点。

下面给出单链表上查找、插入和删除运算的实现过程。

【函数】单链表的查找运算。

```
LinkList Find_List(LinkList L, int k) /*L 为带头结点单链表的头指针*/
/*在表中查找第 k 个元素，若找到，返回该元素结点的指针；否则，返回空指针 NULL*/
  {   LinkList p;    int i;
      i = 1; p = L->next;           /*初始时，令 p 指向第一个元素结点，i 为计数器*/
      while (p && i < k) {          /*顺指针链向后查找，直到 p 指向第 k 个元素结点或 p 为空指针*/
          p = p->next;    i++;
      }
      if (p && i == k) return p;    /*存在第 k 个元素且指针 p 指向该元素结点*/
      return NULL;                  /*第 k 个元素不存在，返回空指针*/
  } /*Find_List*/
```

【函数】单链表的插入运算。

```
int Insert_List (LinkList L, int k, int newElem) /*L 为带头结点单链表的头指针*/
  /*将元素 newElem 插入表中的第 k 个元素之前，若成功则返回 0，否则返回–1*/
  /*该插入操作等同于将元素 newElem 插入在第 k–1 个元素之后*/
{       LinkList p,s;                         /*p、s 为临时指针*/
        if (k == 1) p = L;                    /*元素 newElem 要插入到第 1 个元素之前*/
        else    p = Find_List(L,k-1);         /*查找表中的第 k–1 个元素并令 p 指向该元素结点*/
        if (!p) return –1;                    /*表中不存在第 k–1 个元素，不满足运算要求*/
        s = (NODE *)malloc(sizeof(NODE));     /*创建新元素的结点空间*/
```

```
        if (!s) return –1;
        s->data = newElem;
        s->next = p->next;    p->next = s;        /*将元素 newElem 插入第 k–1 个元素之后*/
        return 0;
} /* Insert_List */
```

【函数】单链表的删除运算。

```
int Delete_List (LinkList L, int k) /*L 为带头结点单链表的头指针*/
  /*删除表中的第 k 个元素结点，若成功则返回 0，否则返回–1*/
  /*删除第 k 个元素相当于令第 k–1 个元素结点的指针域指向第 k+1 个元素所在结点*/
{    LinkList p,q;                          /*p、q 为临时指针*/
     if (k == 1) p = L;                     /*删除的是第一个元素结点*/
     else    p = Find_List(L,k-1);          /*查找表中的第 k–1 个元素并令 p 指向该元素结点*/
     if (!p||!p->next) return –1;           /*表中不存在第 k 个元素*/
     q = p->next;                           /*令 q 指向第 k 个元素结点*/
     p->next = q->next;    free(q);         /*删除结点*/
     return 0;
} /* Delete_List */
```

当线性表采用链表作为存储结构时，不能对数据元素进行随机访问，但是具有插入和删除操作不需要移动元素的优点。

根据结点中指针域的设置方式，还有其他几种链表结构。

- 双向链表。每个结点包含两个指针，分别指出当前元素的直接前驱和直接后继。其特点是可以从表中任意的结点出发，从两个方向上遍历链表。
- 循环链表。在单向链表（或双向链表）的基础上令表尾结点的指针指向链表的第一个结点，构成循环链表。其特点是可以从表中任意结点开始遍历整个链表。
- 静态链表。借助数组来描述线性表的链式存储结构，用数组元素的下标表示元素所在结点的指针。

若双向链表中结点的 front 和 next 指针域分别指示当前结点的直接前驱和直接后继，则在双向链表中插入结点*s 时指针的变化情况如图 3-4（a）所示，其操作过程可表示为：

（1）s -> front = p -> front;

（2）p -> front -> next = s;　//或者表示为 s -> front -> next = s;

（3）s -> next = p;

（4）p -> front = s;

在双向链表中删除结点时指针的变化情况如图 3-4（b）所示，其操作过程可表示为：

（1）p -> front -> next = p -> next;

（2）p -> next -> front = p -> front; free(p);

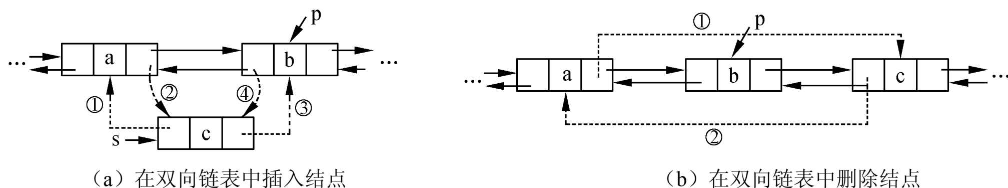


图 3-4　在双向链表插入和删除结点时的指针变化示意图

3.1.2　栈和队列

栈和队列是程序中常用的两种数据结构，它们的逻辑结构和线性表相同。其特点在于运算有所限制：栈按“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，故称为运算受限的线性表。

1．栈

1）栈的定义及基本运算

（1）栈的定义。

栈是只能通过访问它的一端来实现数据存储和检索的一种线性数据结构。换句话说，栈的修改是按先进后出的原则进行的。因此，栈又称为后进先出（Last In First Out，LIFO）的线性表。在栈中进行插入和删除操作的一端称为栈顶（Top），相应地，另一端称为栈底（Bottom）。不含数据元素的栈称为空栈。

（2）栈的基本运算。

① 初始化栈 InitStack(S)：创建一个空栈 S。

② 判栈空 isEmpty(S)：当栈 S 为空时返回“真”，否则返回“假”。

③ 入栈 Push(S,x)：将元素 x 加入栈顶，并更新栈顶指针。

④ 出栈 Pop(S)：将栈顶元素从栈中删除，并更新栈顶指针。若需要得到栈顶元素的值，可将 Pop(S)定义为一个返回栈顶元素值的函数。

⑤ 读栈顶元素 Top(S)：返回栈顶元素的值，但不修改栈顶指针。

在应用中常使用上述 5 种基本运算实现基于栈结构的问题求解。

2）栈的存储结构

（1）顺序存储。栈的顺序存储是指用一组地址连续的存储单元依次存储自栈顶到栈底的数据元素，同时附设指针 top 指示栈顶元素的位置。采用顺序存储结构的栈也称为顺序栈。在这种存储方式下，需要预先定义（或申请）栈的存储空间，也就是说，栈空间的容量是有限的。因此，在顺序栈中，当一个元素入栈时，需要判断是否栈满（栈空间中没有空闲单元），若栈满，则元素不能入栈。

（2）栈的链式存储。用链表作为存储结构的栈也称为链栈。由于栈中元素的插入和删除仅在栈顶一端进行，因此不必另外设置头指针，链表的头指针就是栈顶指针。链栈的表示如图 3-5 所示。

（3）栈的应用。栈的典型应用包括表达式求值、括号匹配等，在计算机语言的实现以及将递归过程转变为非递归过程的处理中，栈有重要的作用。

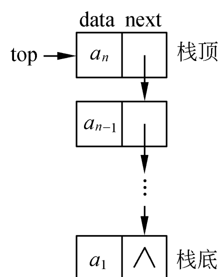


图 3-5　链栈示意图

2．队列

1）队列的定义及基本运算

（1）队列的定义。队列是一种先进先出（First In First Out，FIFO）的线性表，它只允许在表的一端插入元素，而在表的另一端删除元素。在队列中，允许插入元素的一端称为队尾（Rear），允许删除元素的一端称为队头（Front）。

（2）队列的基本运算。

① 初始化队列 InitQueue(Q)：创建一个空的队列 Q。

② 判队空 isEmpty(Q)：当队列为空时返回“真”，否则返回“假”。

③ 入队 EnQueue(Q,x)：将元素 x 加入到队列 Q 的队尾，并更新队尾指针。

④ 出队 DelQueue(Q)：将队头元素从队列 Q 中删除，并更新队头指针。

⑤ 读队头元素 FrontQue(Q)：返回队头元素的值，但不更新队头指针。

2）队列的存储结构

（1）队列的顺序存储。队列的顺序存储结构又称为顺序队列，它也是利用一组地址连续的存储单元存放队列中的元素。由于队列中元素的插入和删除限定在表的两端进行，因此设置队头指针和队尾指针，分别指出当前的队头和队尾。

下面设顺序队列 Q 的容量为 6，其队头指针为 front，队尾指针为 rear，头、尾指针和队列

中元素之间的关系如图 3-6 所示。

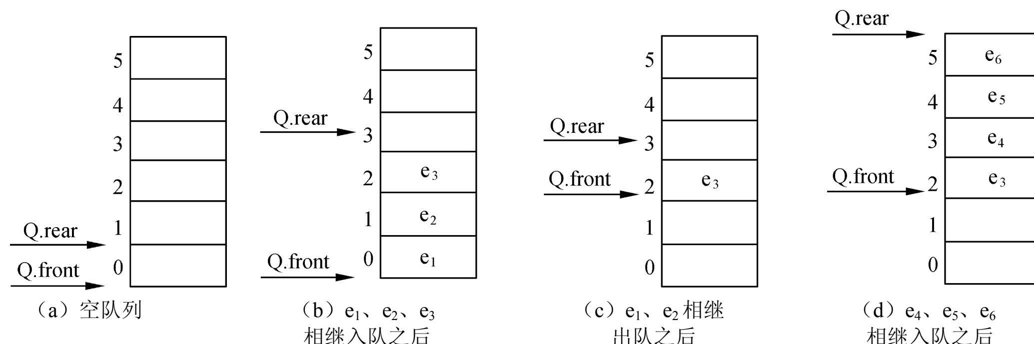


图 3-6　队列的头、尾指针与队列中元素之间的关系

在顺序队列中，为了降低运算的复杂度，元素入队时只修改队尾指针，元素出队时只修改队头指针。由于顺序队列的存储空间容量是提前设定的，所以队尾指针会有一个上限值，当队尾指针达到该上限时，就不能只通过修改队尾指针来实现新元素的入队操作了。若将顺序队列假想成一个环状结构（通过整除取余运算实现），则可维持入队、出队操作运算的简单性，如图 3-7 所示，称之为循环队列。

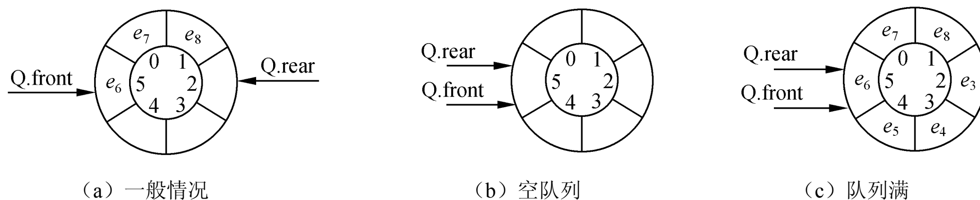


图 3-7　循环队列的头、尾指针示意图

设循环队列 Q 的容量为 MAXSIZE，初始时队列为空，且 Q.rear 和 Q.front 都等于 0，如图 3-8（a）所示。

元素入队时，修改队尾指针 Q.rear = (Q.rear+1)% MAXSIZE，如图 3-8（b）所示。

元素出队时，修改队头指针 Q.front = (Q.front+1)% MAXSIZE，如图 3-8（c）所示。

根据队列操作的定义，当出队操作导致队列变为空时，则有 Q.rear==Q.front，如图 3-8（d）所示；若入队操作导致队列满，则 Q.rear==Q.front，如图 3-8（e）所示。

在队列空和队列满的情况下，循环队列的队头、队尾指针指向的位置是相同的，此时仅仅

根据 Q.rear 和 Q.front 之间的关系无法断定队列的状态。为了区别队空和队满的情况，可采用以下两种处理方式：其一是设置一个标志，以区别头、尾指针的值相同时队列是空还是满；其二是牺牲一个存储单元，约定以“队列的尾指针所指位置的下一个位置是队头指针时”表示队列满，如图 3-8（f）所示，而头、尾指针的值相同时表示队列为空。

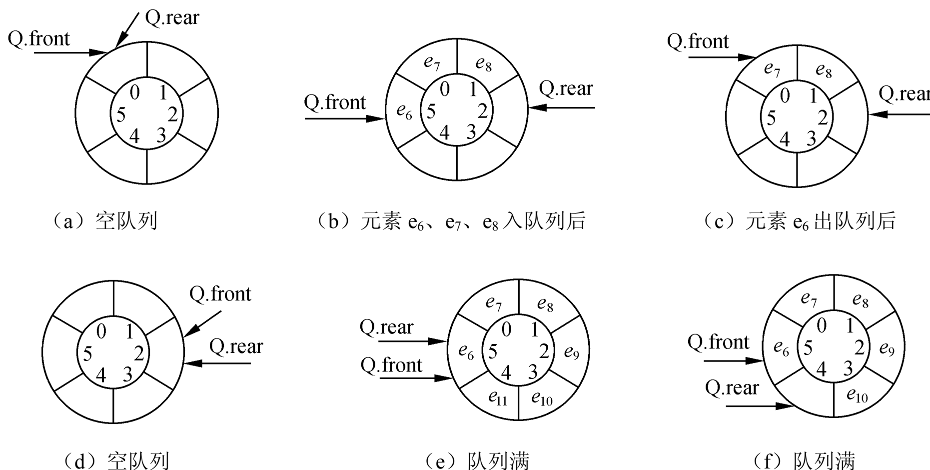


图 3-8　循环队列的头、尾指针示意图

设队列中的元素类型为整型，则循环队列的类型定义如下。

```
#define  MAXQSIZE  100
typedef  struct  {
    int  *base;          /*循环队列的存储空间，假设队列中存放的是整型数*/
    int  front;          /*指示队头，称为队头指针*/
    int  rear;           /*指示队尾，称为队尾指针*/
}SqQueue;
```

【函数】创建一个空的循环队列。

```
int InitQueue(SqQueue *Q)
/*创建容量为 MAXQSIZE 的空队列，若成功返回 0，否则返回–1*/
{   Q -> base = (int *)malloc(MAXQSIZE*sizeof(int));
    if (!Q->base) return –1;
    Q->front = 0; Q->rear = 0; return 0;
}/*InitQueue*/
```

【函数】元素入循环队列。

```
int EnQueue(SqQueue *Q, int e) /*元素 e 入队，若成功返回 0，否则返回–1*/
{    if ( (Q->rear+1)% MAXQSIZE == Q->front) return –1;
     Q->base[Q->rear] = e;
     Q->rear = (Q->rear + 1)% MAXQSIZE;
     return 0;
}/*EnQueue*/
```

【函数】元素出循环队列。

```
int DelQueue(SqQueue *Q, int *e)
/*若队列不空，则删除队头元素，由参数 e 带回其值并返回 0，否则返回–1*/
{    if ( Q->rear == Q->front) return –1;
     *e = Q->base[Q->front];
     Q->front = (Q->front + 1)% MAXQSIZE ;
     return 0;
}/*DelQueue*/
```

（2）队列的链式存储。队列的链式存储也称为链队列。这里为了便于操作，给链队列添加一个头结点，并令头指针指向头结点。因此，队列为空的判定条件是头指针和尾指针的值相同，且均指向头结点。队列的一种链式存储如图 3-9 所示。

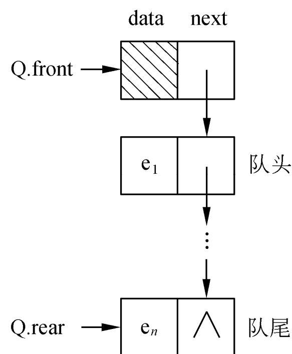


图 3-9 链队列示意图

3）队列的应用

队列结构常用于处理需要排队的场合，例如操作系统中处理打印任务的打印队列、离散事件的计算机模拟等。

3.1.3 串

串（字符串）是一种特殊的线性表，其数据元素为字符。计算机中非数值问题处理的对象经常是字符串数据，例如，在汇编和高级语言的编译程序中，源程序和目标程序都是字符串；在事务处理程序中，姓名、地址等一般也是作为字符串处理的。串具有自身的特性，运算时常常把一个串作为一个整体来处理。这里介绍串的定义、基本运算、存储结构及串的模式匹配算法。

1. 串的定义及基本运算

1）串的定义

串是仅由字符构成的有限序列，是一种线性表。一般记为 $S='a_1a_2\cdots a_n'$，其中，S 是串名，单引号括起来的字符序列是串值。

2）串的几个基本概念

- 空串：长度为零的串称为空串，空串不包含任何字符。
- 空格串：由一个或多个空格组成的串。虽然空格是一个空白字符，但它也是一个字符，在计算串长度时要将其计算在内。
- 子串：由串中任意长度的连续字符构成的序列称为子串。含有子串的串称为主串。子串在主串中的位置是指子串首次出现时，该子串的第一个字符在主串中的位置。空串是任意串的子串。
- 串相等：指两个串长度相等且对应序号的字符也相同。
- 串比较：两个串比较大小时以字符的 ASCII 码值（或其他字符编码集合）作为依据。实质上，比较操作从两个串的第一个字符开始进行，字符的码值大者所在的串为大；若其中一个串先结束，则以串长较大者为大。

3）串的基本操作

（1）赋值操作 StrAssign(s,t)：将串 s 的值赋给串 t。

（2）连接操作 Concat(s,t)：将串 t 接续在串 s 的尾部，形成一个新串。

（3）求串长 StrLength(s)：返回串 s 的长度。

（4）串比较 StrCompare(s,t)：比较两个串的大小。返回值–1、0 和 1 分别表示 s<t、s=t 和 s>t 三种情况。

（5）求子串 SubString(s,start,len)：返回串 s 中从 start 开始的、长度为 len 的字符序列。

以上 5 种最基本的串操作构成了串的最小操作子集，利用它们可以实现串的其他运算。

2．串的存储结构

串可以进行顺序存储或链式存储。

（1）串的顺序存储结构。串的顺序存储结构是指用一组地址连续的存储单元来存储串值的字符序列。由于串中的元素为字符，所以可通过程序语言提供的字符数组定义串的存储空间，也可以根据串长的需要动态申请字符串的空间。

（2）串的链式存储。当用链表存储串中的字符时，每个结点中可以存储一个字符，也可以存储多个字符，此时要考虑存储密度问题。在链式存储结构中，结点大小的选择会直接影响对串的处理效率。

3．串的模式匹配

子串的定位操作通常称为串的模式匹配，它是各种串处理系统中最重要的运算之一。子串也称为模式串。