# 计算机体系结构

## 冯·诺依曼体系结构

1. 计算机处理的数据和指令一律用二进制数表示
2. 顺序执行程序

计算机运行过程中，把要执行的程序和处理的数据首先存入主存储器（内存），计算机执行程序时，将自动地并按顺序从主存储器中取出指令一条一条地执行，这一概念称作顺序执行程序。

1. 计算机硬件由运算器、控制器、存储器、输入设备和输出设备五大部分组成。

## 数据的机内表示

### 二进制表示

#### 机器数

由于计算机中符号和数字一样，都必须用二进制数串来表示，因此，**正负号也必须用0,1来表示**。

#### 原码

原码用第一位表示符号, 其余位表示值. 比如如果是8位二进制:

[+1]原 = 0000 0001

[-1]原 = 1000 0001

第一位是符号位. 因为第一位是符号位, 所以8位二进制数的取值范围就是:

[1111 1111 , 0111 1111]

即

[-127 , 127]

原码是人脑最容易理解和计算的表示方式

#### 反码

* 正数的反码是其本身
* 负数的反码是在其原码的基础上, 符号位不变，其余各个位取反.

[+1] = [00000001]原 = [00000001]反

[-1] = [10000001]原 = [11111110]反

可见如果一个反码表示的是负数，人脑无法直观的看出来它的数值， 通常要将其转换成原码再计算。

#### 补码

* 正数的补码就是其本身
* 负数的补码是在其原码的基础上, 符号位不变, 其余各位取反, 最后+1。 (即在反码的基础上+1)

[+1] = [00000001]原 = [00000001]反 = [00000001]补

[-1] = [10000001]原 = [11111110]反 = [11111111]补

1+（-1)= 00000001 + 11111111 = 00000000 = 0

对于负数, 补码表示方式也是人脑无法直观看出其数值的. 通常也需要转换成原码在计算其数值.

在计算机系统中，**数值一律用补码来表示和存储**。原因在于，使用补码，可以将符号位和数值域统一处理；同时，加法和减法也可以统一处理。此外，补码与原码相互转换，其运算过程是相同的，不需要额外的硬件电路。

#### 定点数与浮点数

**定点数是小数点固定的数**。在计算机中没有专门表示小数点的位，小数点的位置是约定默认的。一般固定在机器数的最低位之后，或是固定在符号位之后。前者称为定点纯整数，后者称为定点纯小数。

定点数表示法简单直观，但是 **数值表示的范围太小**，运算时容易产生溢出。

浮点数是小数点的位置可以变动的数。为增大数值表示范围，防止溢出，采用浮点数表示法。**浮点表示法类似于十进制中的科学计数法**。

在计算机上，通常使用2为基数的幂数来表式。一个浮点数a由两个数m和e来表示：a = m × b^e。在任意一个这样的系统中，我们选择一个基数b（记数系统的基）和精度p（即使用多少位来存储）。m （即尾数）是形如±d.ddd...ddd的p位数（每一位是一个介于0到b-1之间的整数，包括0和b-1）。**如果m的第一位是非0整数，m称作正规化的**。e是指数。

| 数符± | 阶码e | 尾数m |

数符表示尾数的符号位，阶码表示幂次，尾数表示规格化后的小数值。

* **32位单精度**：单精度二进制小数，使用32位存储。1 8 23 位长
* **64位双精度**：双精度二进制小数，使用64位存储。1 11 52 位长

### 位、字节、字

* 位(Bit)：电子计算机中最小的数据单位。每一位的状态只能是0或1。
* 字节(Byte)：**8个二进制位构成1个字节，它是存储空间的基本计量单位**。1个字节可以储存1个英文字母或者半个汉字，换句话说，1个汉字占据2个字节的存储空间。
* 字(Word)：**由若干个字节构成，字的位数叫做字长，不同档次的机器有不同的字长**。例如一台8位机，它的1个字就等于1个字节，字长为8位。如果是一台16位机，那么，它的1个字就由2个字节构成，字长为16位。**字是计算机进行数据处理和运算的单位**。

### 字节序

字节顺序是指占内存多于一个字节类型的数据在内存中的存放顺序，通常有小端、大端两种字节顺序。

* 小端字节序：低字节数据存放在内存低地址处，高字节数据存放在内存高地址处。
* 大端字节序：高字节数据存放在低地址处，低字节数据存放在高地址处。

基于X86平台的PC机是小端字节序的，而有的嵌入式平台则是大端字节序的。**所有网络协议也都是采用大端字节序的方式来传输数据的。所以有时我们也会把大端字节序方式称之为网络字节序**。

比如数字 0x12345678 在两种不同字节序CPU中的存储顺序如下所示：

 Big Endian
 低地址 高地址
 ---------------------------------------------------->
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 | 12 | 34 | 56 | 78 |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

 Little Endian
 低地址 高地址
 ---------------------------------------------------->
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 | 78 | 56 | 34 | 12 |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

从上面两图可以看出，采用Big Endian方式存储数据是符合我们人类的思维习惯的。

联合体union的存放顺序是所有成员都从低地址开始存放，利用该特性，就能判断CPU对内存采用Little-endian还是Big-endian模式读写。

### 字节对齐

现代计算机中内存空间都是按照字节划分的，从理论上讲似乎对任何类型的变量的访问可以从任何地址开始，但实际情况是在访问特定类型变量的时候经常在特定的内存地址访问，**这就需要各种类型数据按照一定的规则在空间上排列，而不是顺序的一个接一个的排放，这就是对齐**。

* 为什么要进行字节对齐？
1. 某些平台只能在特定的地址处访问特定类型的数据;
2. 最根本的原因是效率问题，字节对齐能提高存取数据的速度。

比如有的平台每次都是从偶地址处读取数据,对于一个int型的变量,若从偶地址单元处存放,则只需一个读取周期即可读取该变量，但是若从奇地址单元处存放，则需要2个读取周期读取该变量。

* 字节对齐的原则
1. 数据成员对齐规则：结构体或联合体的数据成员，第一个数据成员放在 offset 为0的地方，以后每个数据成员存储的起始位置要从该成员大小或者成员的子成员大小（只要该成员有子成员，比如说是数组，结构体等）的整数倍开始（比如int在32位机为4字节,则要从4的整数倍地址开始存储）。
2. 结构体作为成员：如果一个结构里有某些结构体成员，则结构体成员要从其内部最大元素大小的整数倍地址开始存储。(struct a里存有struct b,b里有char,int ,double等元素,那b应该从8的整数倍开始存储。)
3. 收尾工作：结构体的总大小，也就是sizeof的结果，必须是其内部最大成员的整数倍，不足的要补齐。