

# 第18讲：数据在内存中的存储

## 目录

1. 整数在内存中的存储
2. 大小端字节序和字节序判断
3. 浮点数在内存中的存储

正文开始

## 1. 整数在内存中的存储

在讲解操作符的时候，我们就讲过了下面的内容：

整数的2进制表示方法有三种，即 **原码、反码和补码**

有符号的整数，三种表示方法均有**符号位**和**数值位**两部分，符号位都是用0表示“正”，用1表示“负”，最高位的一位是被当做符号位，剩余的都是数值位。

**正整数的原、反、补码都相同。**

**负整数的三种表示方法各不相同。**

**原码：**直接将数值按照正负数的形式翻译成二进制得到的就是原码。

**反码：**将原码的符号位不变，其他位依次按位取反就可以得到反码。

**补码：**反码+1就得到补码。

**对于整形来说：数据存放内存中其实存放的是补码。**

为什么呢？

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；

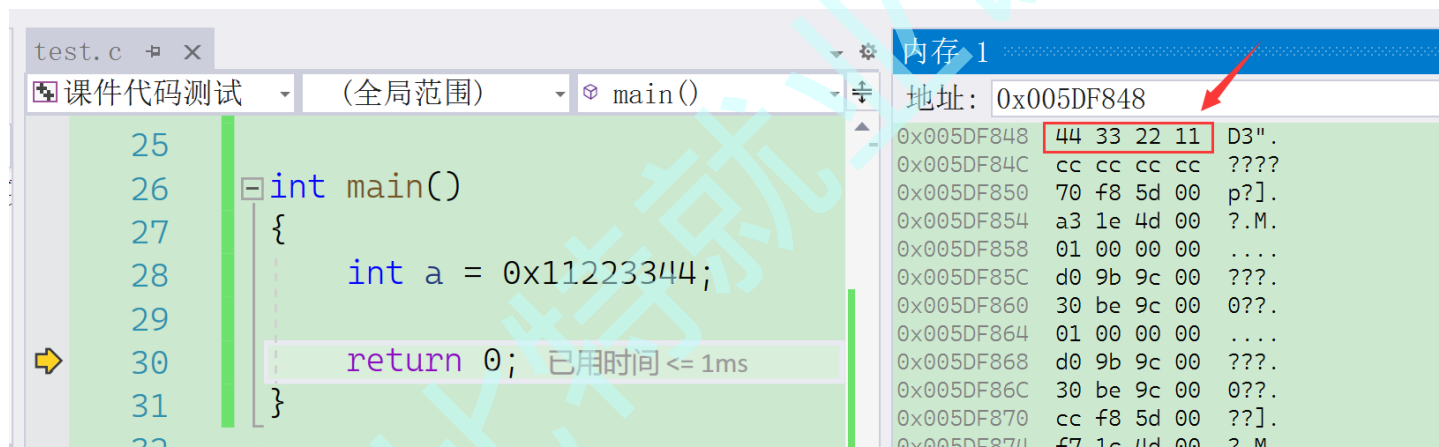
同时，加法和减法也可以统一处理（**CPU只有加法器**）此外，补码与原码相互转换，其运算过程是相同的，不需要额外的硬件电路。

## 2. 大小端字节序和字节序判断

当我们了解了整数在内存中存储后，我们调试看一个细节：

```
1 #include <stdio.h>
2
3 int main()
4 {
5     int a = 0x11223344;
6
7     return 0;
8 }
```

调试的时候，我们可以看到在a中的 `0x11223344` 这个数字是按照字节为单位，倒着存储的。这是为什么呢？



### 2.1 什么是大小端？

其实超过一个字节的数据在内存中存储的时候，就有存储顺序的问题，按照不同的存储顺序，我们分为大端字节序存储和小端字节序存储，下面是具体的概念：

大端（存储）模式：

是指数据的低位字节内容保存在内存的高地址处，而数据的高位字节内容，保存在内存的低地址处。

小端（存储）模式：

是指数据的低位字节内容保存在内存的低地址处，而数据的高位字节内容，保存在内存的高地址处。

上述概念需要记住，方便分辨大小端。

## 2.2 为什么有大小端？

为什么会有大小端模式之分呢？

这是因为在计算机系统中，我们是以字节为单位的，每个地址单元都对应着一个字节，一个字节为8 bit 位，但是在C语言中除了8 bit 的 `char` 之外，还有16 bit 的 `short` 型，32 bit 的 `long` 型（要看具体的编译器），另外，对于位数大于8位的处理器，例如16位或者32位的处理器，由于寄存器宽度大于一个字节，那么必然存在着一个如何将多个字节安排的问题。因此就导致了大端存储模式和小端存储模式。

例如：一个 16bit 的 `short` 型 `x`，在内存中的地址为 `0x0010`，`x` 的值为 `0x1122`，那么 `0x11` 为高字节，`0x22` 为低字节。对于大端模式，就将 `0x11` 放在低地址中，即 `0x0010` 中，`0x22` 放在高地址中，即 `0x0011` 中。小端模式，刚好相反。我们常用的 X86 结构是小端模式，而 KEIL C51 则为大端模式。很多的ARM，DSP都为小端模式。有些ARM处理器还可以由硬件来选择是大端模式还是小端模式。

## 2.3 练习

### 2.3.1 练习1

请简述大端字节序和小端字节序的概念，设计一个小程序来判断当前机器的字节序。（10分）-百度笔试题

```
1 //代码1
2 #include <stdio.h>
3 int check_sys()
4 {
5     int i = 1;
6     return (*(char *)&i);
7 }
8
9 int main()
10 {
11     int ret = check_sys();
12     if(ret == 1)
13     {
14         printf("小端\n");
15     }
16     else
17     {
18         printf("大端\n");
19     }
20     return 0;
21 }
```

```

22
23 //代码2
24 int check_sys()
25 {
26     union
27     {
28         int i;
29         char c;
30     }un;
31     un.i = 1;
32     return un.c;
33 }

```

### 2.3.2 练习2

```

1 #include <stdio.h>
2 int main()
3 {
4     char a = -1;
5     signed char b = -1;
6     unsigned char c = -1;
7     printf("a=%d,b=%d,c=%d",a,b,c);
8     return 0;
9 }

```

### 2.3.3 练习3

```

1 #include <stdio.h>
2 int main()
3 {
4     char a = -128;
5     printf("%u\n",a);
6     return 0;
7 }

```

```

1 #include <stdio.h>
2 int main()
3 {
4     char a = 128;
5     printf("%u\n",a);
6     return 0;
7 }

```

### 2.3.4 练习4

```

1 #include <stdio.h>
2
3 int main()

```

```

4 {
5     char a[1000];
6     int i;
7     for(i=0; i<1000; i++)
8     {
9         a[i] = -1-i;
10    }
11    printf("%d",strlen(a));
12    return 0;
13 }

```

### 2.3.5 练习5

```

1 #include <stdio.h>
2
3 unsigned char i = 0;
4 int main()
5 {
6     for(i = 0; i<=255; i++)
7     {
8         printf("hello world\n");
9     }
10    return 0;
11 }

```

```

1 #include <stdio.h>
2
3 int main()
4 {
5     unsigned int i;
6     for(i = 9; i >= 0; i--)
7     {
8         printf("%u\n", i);
9     }
10    return 0;
11 }

```

### 2.3.6 练习6

```

1 #include <stdio.h>
2 //X86环境 小端字节序
3 int main()
4 {
5     int a[4] = { 1, 2, 3, 4 };
6     int *ptr1 = (int *)(&a + 1);
7     int *ptr2 = (int *)((int)a + 1);
8     printf("%x,%x", ptr1[-1], *ptr2);
9     return 0;
10 }

```

代码输出的结果是啥？

常见的浮点数: 3.14159、1E10等, 浮点数家族包括: `float`、`double`、`long double` 类型。

浮点数表示的范围: `float.h` 中定义

#### 3.1 练习

```
1 #include <stdio.h>
2
3 int main()
4 {
5     int n = 9;
6     float *pFloat = (float *)&n;
7     printf("n的值为: %d\n", n);
8     printf("*pFloat的值为: %f\n", *pFloat);
9
10    *pFloat = 9.0;
11    printf("num的值为: %d\n", n);
12    printf("*pFloat的值为: %f\n", *pFloat);
13    return 0;
14 }
```

输出什么?

#### 3.2 浮点数的存储

上面的代码中, `num` 和 `*pFloat` 在内存中明明是同一个数, 为什么浮点数和整数的解读结果会差别这么大?

要理解这个结果, 一定要搞懂浮点数在计算机内部的表示方法。

根据国际标准IEEE (电气和电子工程协会) 754, 任意一个二进制浮点数V可以表示成下面的形式:

$$V = (-1)^S * M * 2^E$$

- $(-1)^S$  表示符号位, 当S=0, V为正数; 当S=1, V为负数
- M 表示有效数字, M是大于等于1, 小于2的
- $2^E$  表示指数位

举例来说:

十进制的5.0, 写成二进制是 `101.0`, 相当于 `1.01×22`。

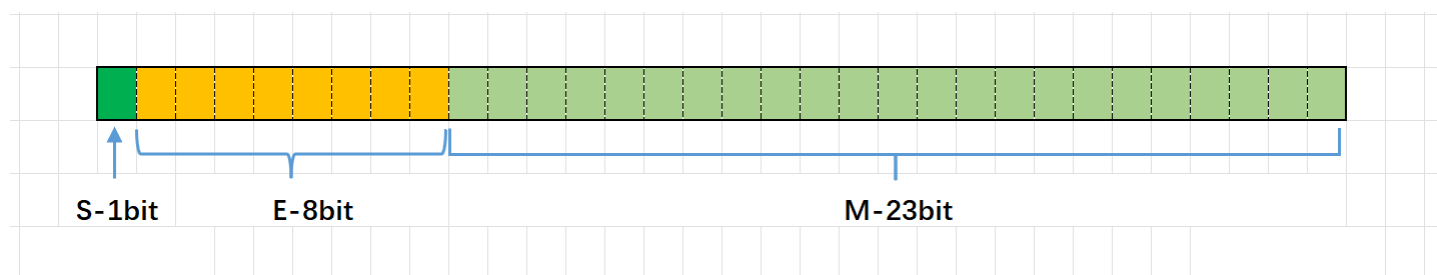
那么，按照上面V的格式，可以得出 $S=0$ ， $M=1.01$ ， $E=2$ 。

十进制的-5.0，写成二进制是  $-101.0$ ，相当于  $-1.01 \times 2^2$ 。那么， $S=1$ ， $M=1.01$ ， $E=2$ 。

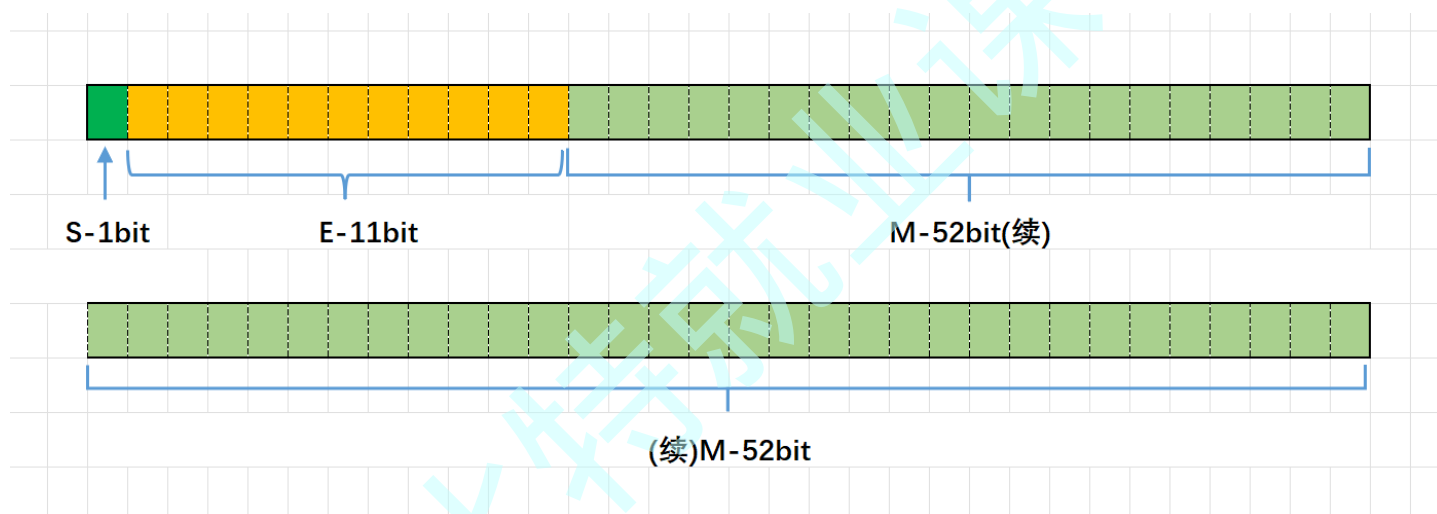
IEEE 754规定：

对于32位的浮点数，最高的1位存储符号位S，接着的8位存储指数E，剩下的23位存储有效数字M

对于64位的浮点数，最高的1位存储符号位S，接着的11位存储指数E，剩下的52位存储有效数字M



float类型浮点数内存分配



double类型浮点数内存分配

### 3.2.1 浮点数存的过程

IEEE 754 对有效数字M和指数E，还有一些特别规定。

前面说过， $1 \leq M < 2$ ，也就是说，M可以写成  $1.xxxxxx$  的形式，其中  $xxxxxx$  表示小数部分。

IEEE 754 规定，在计算机内部保存M时，默认这个数的第一位总是1，因此可以被舍去，只保存后面的xxxxxx部分。比如保存1.01的时候，只保存01，等到读取的时候，再把第一位的1加上去。这样做的目的，是节省1位有效数字。以32位浮点数为例，留给M只有23位，将第一位的1舍去以后，等于可以保存24位有效数字。

至于指数E，情况就比较复杂

首先，E为一个无符号整数（unsigned int）

这意味着，如果E为8位，它的取值范围为0~255；如果E为11位，它的取值范围为0~2047。但是，我们知道，科学计数法中的E是可以出现负数的，所以IEEE 754规定，存入内存时E的真实值必须再加上一个中间数，对于8位的E，这个中间数是127；对于11位的E，这个中间数是1023。比如， $2^{10}$ 的E是10，所以保存成32位浮点数时，必须保存成 $10+127=137$ ，即10001001。

### 3.2.2 浮点数取的过程

指数E从内存中取出还可以再分成三种情况：

#### E不全为0或不全为1

这时，浮点数就采用下面的规则表示，即指数E的计算值减去127（或1023），得到真实值，再将有效数字M前加上第一位的1。

比如：0.5 的二进制形式为0.1，由于规定正数部分必须为1，即将小数点右移1位，则为 $1.0 \times 2^{-1}$ ，其阶码为 $-1+127$ (中间值)=126，表示为01111110，而尾数1.0去掉整数部分为0，补齐0到23位00000000000000000000000，则其二进制表示形式为：

```
1 0 01111110 00000000000000000000000
```

#### E全为0

这时，浮点数的指数E等于1-127（或者1-1023）即为真实值，有效数字M不再加上第一位的1，而是还原为0.xxxxxx的小数。这样做是为了表示±0，以及接近于0的很小的数字。

```
1 0 00000000 00100000000000000000000
```

#### E全为1

这时，如果有效数字M全为0，表示±无穷大（正负取决于符号位s）；

```
1 0 11111111 00010000000000000000000
```

好了，关于浮点数的表示规则，就说到这里。

## 3.3 题目解析

下面，让我们回到一开始的练习



先看第1环节，为什么 9 还原成浮点数，就成了 0.000000？

9以整型的形式存储在内存中，得到如下二进制序列：

```
1 0000 0000 0000 0000 0000 0000 0000 1001
```

首先，将 9 的二进制序列按照浮点数的形式拆分，得到第一位符号位s=0，后面8位的指数E=00000000，

最后23位的有效数字M=000 0000 0000 0000 0000 1001。

由于指数E全为0，所以符合E为全0的情况。因此，浮点数V就写成：

$$V = (-1)^0 \times 0.000000000000000000000001001 \times 2^{(-126)} = 1.001 \times 2^{(-146)}$$

显然，V是一个很小的接近于0的正数，所以用十进制小数表示就是0.000000。

再看第2环节，浮点数9.0，为什么整数打印是 1091567616

首先，浮点数9.0 等于二进制的1001.0，即换算成科学计数法是： $1.001 \times 2^3$

$$\text{所以: } 9.0 = (-1)^0 * (1.001) * 2^3,$$

那么，第一位的符号位S=0，有效数字M等于001后面再加20个0，凑满23位，指数E等于3+127=130，即10000010

所以，写成二进制形式，应该是S+E+M，即

```
1 0 10000010 001 0000 0000 0000 0000 0000
```

这个32位的二进制数，被当做整数来解析的时候，就是整数在内存中的补码，原码正是 1091567616。

完