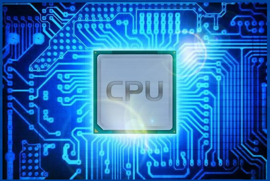
Datawhale

Datawhale 开源社区

DATAWHALE OPEN SOURCE COMMUNITY

深入理解计算机系统（4）

Computer Systems A Programmer's Perspective CSAPP



李岳昆、易远哲

realyurk@gmail.com、yuanzhe.yi@outlook.com

2021 年 12 月 14 日

第 I 部分

信息的表示和处理-II

整数运算

浮点数

无符号加法

Datawhale

两个无符号数相加代码如下：

无符号数加法

```c
unsigned char a = 255;
unsigned char b = 1;
unsigned char c = a + b;
printf("c=%d", c);
```

我们期望的结果是 256，但实际结果为 0。产生这个结果是因为 a 加 b 的和超过了 unsigned char 类型所能表示的最大值 255。

无符号数加法溢出

Datawhale

在 C 语言执行的过程中，对于溢出的情况并不会报错，但是我们希望判定运算结果是否发生了溢出。

溢出判断

```c
int uadd_ok(unsigned x, unsigned y){
    unsigned sum = x + y;
    return sum >= x;  // 溢出返回 0，没溢出返回 1
}
```

因为 x 和 y 都是大于 0 的，因此，两者之和大于其中任何一个。

补码加法溢出

与无符号数相加不同的是，有符号数的溢出分为正溢出和负溢出。

- 当 x 加 y 的和大于等于 2^{w-1} 时，发生正溢出，此时，得到的结果会减去 2^w。
- 当 x 加 y 的和小于 -2^{w-1} 时，发生负溢出，此时，得到的结果会加上 2^w。

补码加法溢出

```c
char x = 127;
char y = 1;
char z = x + y;
printf("z=%d", z);
```

运行结果为-128，发生了正溢出.

两种乘法

无符号数乘法

w 位的无符号数 x 和 y，二者的乘积可能需要 2w 位来表示。在 C 语言中，定义了无符号数乘法所产生的结果是 w 位，因此，运行结果会截取 2w 位中的低 w 位。截断采用取模的方式，因此，运行结果等于 x 与 y 乘积并对 2 的 w 次方取模

补码乘法

计算机的有符号数用补码表示，因此补码乘法就是有符号数乘法。无论是无符号数乘法，还是补码乘法，运算结果的位级表示都是一样的，只不过补码乘法比无符号数乘法多一步，需要将无符号数转换成补码（有符号数）。虽然完整的乘积结果的位级表示可能会不同，但是截断后的位级表示都是相同的

其它一些情况，如果乘以的是 2 的整数倍，那么可以通过位移进行快速运算.

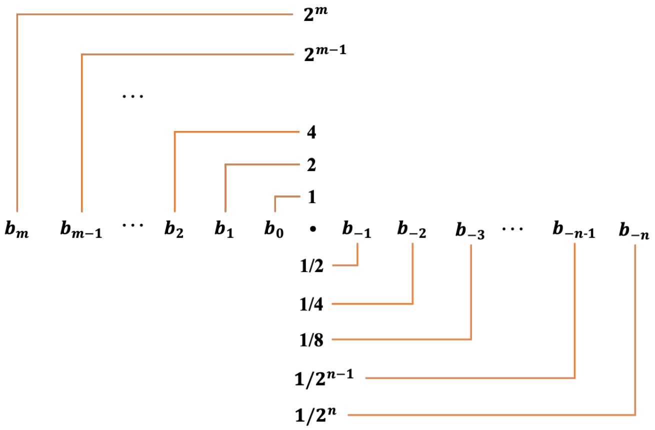
两种除法

我们以乘除 2 的倍数为例进行讨论补码乘除的运算规则:

- 原码运算:
  - 对于一般的以 2^w 为因子的乘法，我们只需要对原码进行移动，例如:$[5]_{原}=0101_2$，若将原码乘以 2，则相当于所有位数向左移动一个单位，即为 $1010_2=[10]_{原}$，除法只需要进行右移即可。即: 对于原码，不论正负，若某个数字乘 2^w 的倍数，则只需要对原码向左移动 w 个单位，空缺位补 0。
- 补码运算:
  - 对于补码，正数则仍然按照原码规则进行计算，而负数则需要保证符号位不变，在向左移动时补 0，向右移动时补 1。例如:$[-5]_{补}=1011_2$，将其乘以 2，则保持符号位最高的 1 不变，其余位置向左移动一个单位，空出来的最后侧加 0，则为 $10110_2=[-10]_{补}$

二进制小数

理解浮点数的第一步是考虑含有小数值的二进制数。



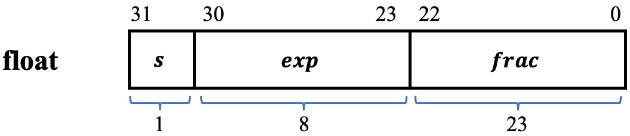
对于这种定点表示方法，并不能很有效的表示非常大的数。

IEEE 浮点表示

下面是 IEEE 的关于浮点数的表示：

$$V = (-1)^s \times M \times 2^E$$

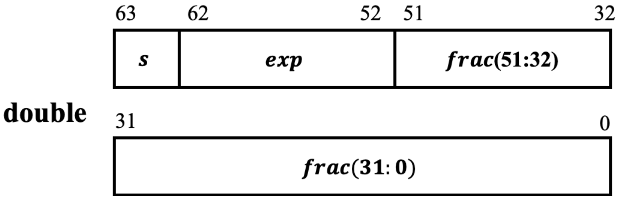
- 三个变量：符号 s、阶码 E 和尾数 M，下面以单精度浮点数为例。
- 例如 C 语言中 float 类型的变量占 4 个字节，32 个比特位，这 32 个比特位被划分成 3 个字段来解释。



其中最高位 31 位表示符号位 s。当 s=0 时，表示正数；s=1 时则表示负数。从第 23 位到 30 位，这 8 个二进制位与阶码的值 E 是相关的。剩余的 23 位与尾数 M 是相关的。

IEEE 浮点表示

对于 64 位双精度浮点数，其二进制位与浮点数的关系如图所示。



与单精度浮点数相比，双精度浮点数的符号位也是 1 位。但是阶码字段的长度为 11 位，小数字段的长度为 52 位。

浮点数的数值可以分为三类：第一类是规格化的值，第二类是非规格化的值，第三类是特殊值。其中阶码的值决定了这个数是属于其中哪一类。

IEEE 浮点表示

Datawhale

① 当阶码字段的二进制位不全为 0，且不全为 1 时，此时表示的是规格化的值。

s	$\neq 0$ 且 $\neq 255$	f

② 当阶码字段的二进制位全为 0 时，此时表示的数值是非规格化的值。

s	0	0	0	0	0	0	0	0	f

③ 当阶码字段的二进制位全为 1 时，表示的数值为特殊值。

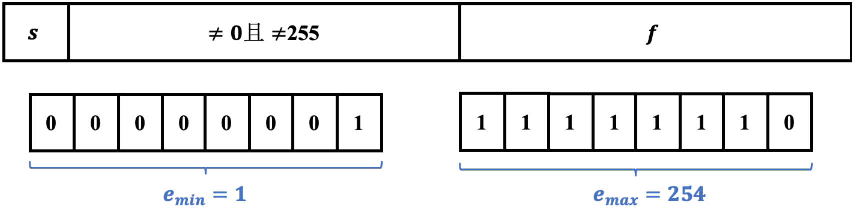
s	1	1	1	1	1	1	1	1	0	0	0	0	0	0	…	0	0	0	0	0	0

特殊值分类为两类，一类表示无穷大或者无穷小，另外一类表示“不是一个数”。

s	1	1	1	1	1	1	1	1	$\neq 0$

规格化的值

当表示规格化的值时，其中阶码字段的取值范围如图所示。



最小值是 1，最大值是 254。为了方便表述，我们用小写字母 e 来表示这个 8 位二进制数，需要注意的是阶码 E 的值并不等于 e (8 个二进制位) 所表示的值，而是 e 的值减去一个偏置量，偏置量的值与阶码字段的位数是相关的。

$$E = e - bias$$

规格化的值

Datawhale

1. 当表示单精度的值时，阶码字段的长度为 8，偏置量等于 127。
   - bias(float)=2^{8-1}-1=127
2. 当表示双精度的数时，阶码字段的长度为 11，偏置量等于 1023。
   - bias(double)=2^{11-1}-1=1023

因此，结合 e 的范围 [1,254] 对于单精度浮点数，阶码 E 的取值范围是 [-126,127]。

尾数 M 被定义为 1+f，尾数 M 的二进制表示如图所示。

f_{22}	f_{21}	…	f_1	f_0

$$M = 1.f_{22}f_{21}\cdots f_1 f_0 = 1 + f$$

非规格化的值

当阶码字段的二进制位全为 0 时，所表示的是非规格化的值，关于非规格化的数有两个用途：

1. 提供了表示数值 0 的方法，当符号位 s 等于 0，阶码字段全为 0，小数字段也全为 0 时，此时表示正零。当符号位 s 等于 1，阶码字段全为 0，小数字段也全为 0 时，此时表示负零。根据 IEEE 的浮点规则，正零和负零在某些方面被认为不同，而其他方面是相同的.
   - Case 0: s=0　M=f=0　V=+0.0
   - Case 1: s=1　M=f=0　V=-0.0
2. 非规格化的数是可以表示非常接近 0 的数。当阶码字段全为 0 的时，阶码 E 的值等于 1-bias，而尾数的值 M 等于 f，不包含隐藏的 1。这与规格化的值的解释方法不同，需要特别注意。

非规格化	规格化
E=1-bias M=f	E=e-bias M=1+f

特殊值

当阶码字段全为 1，且小数字段全为 0 时，表示无穷大的数。无穷大也分为两种，正无穷大和负无穷大。如果符号位 s 等于 0 时，表示正无穷大；符号位 s 等于 1，表示负无穷大。

- Case 0: s=0　f=0　V=+∞
- Case 1: s=1　f=0　V=-∞

此外，还会遇到一些运算结果不为实数或者用无穷也无法表示的情况。

NaN 的出现

这里引入一个新的概念："不是一个数"(Not a Number)。例如我们对-1 进行开方运算或者无穷减无穷的运算，此时得到的结果就会返回 NaN。当阶码字段全为 1，且小数字段不为 0 时，可以表示 NaN (Not a Number)。

整型转单精度浮点型

接下来我们将整型数 12345 转换成浮点数 12345.0，通过转换过程，我们将会了解这段匹配数位是如何产生的。整型数 12345，其二进制数的表示为：

- int 12,345
- 0000 0000 0000 0000 0011 0000 0011 1001

虽然 int 类型的变量占 32 个比特位，由于该数的高 18 位都等于 0，以将高 18 位忽略，只看低 14 位。

- 11 0000 0011 1001

根据规格化数的表示规则，我们可以将 12345 用下式表达：

$$12,345 = 1.1000000111001 \times 2^{13}$$

整型转单精度浮点型

根据我们 IEEE 浮点数的编码规则，我们将小数点左边的 1 丢弃，由于单精度的小数字段长度为 23，我们还需要在末端增加 10 个零：

- 1 0000 0011 1001 0000 0000 00

这样我们就得到了浮点数的小数字段，从 12345 的规格化表示可以发现阶码 E 的值等于 13，由于单精度浮点数的 bias 等于 127，因此根据公式 E=e-bias，可以计算出 e 的值等于 140，其二进制表示如下：

$$E = e - bias \quad \text{e=140 1000 1100}$$

整型转单精度浮点型

Datawhale

这样一来，得到了浮点数的阶码字段，再加上符号位的 0，整个单精度浮点数的二进制表示就构造完毕。

int 12,345

$11\ 0000\ 0011\ 1001$

$12{,}345 = 1.1\ 0000\ 0011\ 1001 \times 2^{13}$ $\quad bias_{float} = 127$

0	1000 1100	1 0000 0011 1001 0000 0000 00
	exp	frac

$E = e - bias \qquad e = 140$

由于表示方法的原因，限制了浮点数的范围和精度，所以浮点运算只能近似的表示实数运算。

舍入的概念

对于值 x，可能无法用浮点形式来精确的表示，因此我们希望可以找到“最接近的值 x’来代替 x，这就是舍入操作的任务。一个关键的问题就是在两个可能的值中间确定舍入方向，例如一个数值 1.5，想把该数舍入到最接近的整数，舍入结果应该是 1 还是 2 呢?

IEEE 浮点格式定义了四种不同的舍入方式，分别是：向偶数舍入、向零舍入、向下舍入以及向上舍入。

舍入

Datawhale

- 向下舍入和向上舍入的情况比较简单，向下舍入总是朝向小的方向进行舍入，而向上舍入总是朝向大的方向进行舍入。

Mode	1.40	1.60	1.50	2.50	-1.50
向下舍入	1	1	1	2	-2
向上舍入	2	2	2	3	-1

- 向零舍入就是把正数进行向下舍入，把负数进行向上舍入。将这种舍入规则映射到数轴上，可以发现舍入是朝向零的方向。
- 向偶数舍入，也被称为向最接近的值进行舍入。

Mode	1.40	1.60	1.50	2.50	-1.50
向偶舍入	1	2	2	2	-2

舍入

- 需要注意的是当遇到两个可能结果的中间数值时，舍入结果应该如何计算，向偶数舍入的结果要遵循最低有效数字是偶数的规则，因此 1.5 的舍入结果究竟是 1 还是 2，取决于 1 和 2 哪个数是偶数.
- 乍一看，向偶数舍入这种方式有点随意，为什么要偏向取偶数呢?
- 如果总是采用向上舍入，会导致结果的平均值相对于真实值略高；如果总是采用向下舍入，会导致结果的平均值相对于真实值略低。向偶数舍入就避免了这种统计偏差。使得有一半的情况需要向上舍入，有一半的情况需要向下舍入。
- 对于不想舍入到整数的情况，向偶数舍入的方法同样适用。我们只需要考虑最低有效位是偶数还是奇数即可。

舍入例子

例如，我们将下面的两个十进制小数精确到百分位。

- 1.2349999　1.2350001
- 由于这两个数并不在 1.23 和 1.24 正中间，所以两个数的舍入结果分别为 1.23 和 1.24，并不需要考虑百分位是否是偶数。
- 由于 1.235 在 1.23 与 1.24 中间，这时我们需要考虑百分位是否是偶数的情况，因此舍入结果是 1.24。
- 类似的情况，向偶数舍入也可以用在二进制小数上，将最低有效位的值 0 认为是偶数，1 认为是奇数。例如二进制小数 10.11100，当舍入需要精确到小数点右边 2 位时，由于这个数是两个可能值（11.00 和 10.11）的中间值，根据向偶数舍入的规则，舍入结果为 11.00。

浮点运算

Datawhale

$$(3.14 + 1e10) - 1e10 = 0.0 \tag{1}$$

$$3.14 + (1e10 - 1e10) = 3.14 \tag{2}$$

这是由于 (1) 对结果进行了舍入，值 3.14 会丢失，因此，对于浮点数的加法是不具有结合性的。

同样由于溢出或舍入而失去精度，导致浮点数的乘法也不具有结合性。

$$(1e20 * 1e20) * 1e - 20 = +\infty \tag{3}$$

$$1e20 * (1e20 * 1e - 20) = 1e20 \tag{4}$$

此外，浮点乘法在加法上不具备分配性。

$$1e20 * (1e20 - 1e20) = 0.0 \tag{5}$$

$$1e20 * 1e20 - 1e20 * 1e20 = NaN \tag{6}$$

浮点运算 tips

- 对于从事科学计算的程序员以及编译器的开发人员来说，缺乏结合性和分配性是一个比较严重的问题。
- C 语言提供了两种不同的浮点数据类型：单精度 float 类型和双精度 double 类型。当 int，float、double 不同数据类型之间进行强制类型转换时，得到的结果可能会超出我们的预期。
- 当 int 类型转换成 float 类型时，数字不会发生溢出，但是可能会被舍入。这是由于单精度浮点数的小数字段是 23 位，可能会出现无法保留精度的情况。

浮点运算 tips

Datawhale

- 从 double 类型转换成 float 类型，由于 float 类型所表示数值的范围更小，所以可能会发生溢出。
- 此外，float 类型的精度相对于 double 较小，转换后还可能被舍入。
- 将 float 类型或者 double 类型的浮点数转换成 int 类型，一种可能的情况是值会向零舍入，例如 1.9 将被转换成 1，-1.9 将被转换成-1；另外一种可能的情况是发生溢出。