CSAPP 笔记

apl

2025 年 3 月 26 日

目录

1 信息的存储

1.1 使用比特表示信息

1.1.1 万物皆比特

由于信号易存储在双稳态单元中，并且可以在存在噪声和不准确的信道中可靠地传输。信息都可以使用二进制的编码进行表示，计算机通过二进制来发送指令，以及表示和处理各种数字、字符串等。

1.1.2 字节数据编码

1 Byte = 8 bits。二进制表示范围是 00000000_2 到 11111111_2；十进制表示范围是 0_{10} 到 255_{10}；十六进制表示范围是 00_{16} 到 FF_{16}，十六进制以 16 为基数，计数符号为 0 - 9 和 A - F 。在 C 语言中，FA1D37B_{16} 可表示为 0xFA1D37B 或 0xfa1d37b。

十进制	十六进制	二进制
0	00	00000000
1	01	00000001
2	02	00000010
3	03	00000011
4	04	00000100
5	05	00000101
6	06	00000110
7	07	00000111
8	08	00001000
9	09	00001001
10	0A	00001010
11	0B	00001011
12	0C	00001100
13	0D	00001101
14	0E	00001110
15	0F	00001111

表 1: 字节数据编码对应表

1.2 位运算

1.2.1 布尔代数

- And（与）：$A\&B = 1$ 当且仅当 $A = 1$ 且 $B = 1$。
- Or（或）：$A|B = 1$ 当 $A = 1$ 或者 $B = 1$。

- Not（非）：$\sim A = 1$ 当 $A = 0$。
- Exclusive - Or（Xor，异或）：$A\hat{}B = 1$ 当 $A = 1$ 或者 $B = 1$，但不同时为 1，即相同为 0，不同为 1。

1.2.2 C 语言中的位运算

C 语言定义了四个位运算符号：

C 表达式	二进制表达式	二进制结果	十六进制结果
~ 0x41	~[0100 0001]	[10111110]	0xBE
~ 0x00	~ [00000000]	[11111111]	0xFF
0x69&0x55	[01101001]&[01010101]	[01000001]	0x41
0x69 | 0x55	[01101001]|[01010101]	[01111101]	0x7D

表 2: C 语言位运算示例

1.2.3 异或运算的应用：数据交换

在 C 语言中，可以利用异或运算实现不使用额外变量交换两个数：

```
void funny(int *x, int *y)
{
    *x = *x ^ *y; /* #1 */
    *y = *x ^ *y; /* #2 */
    *x = *x ^ *y; /* #3 */
}
```

交换过程如下：

步骤	*x	*y
开始	A	B
1	A^B	B
2	A^B	(A^B)^B = A^(B^B) = A
3	(A^B)^A = (B^A)^A = B^(A^A) = B	A
结束	B	A

表 3: 异或运算交换数据过程

1.2.4 C 语言中的逻辑运算

C 语言定义了三种逻辑运算：||（逻辑或）、&&（逻辑与）、!（逻辑非），具有短路效应。例如：

- x && 5/x 可以用于避免除 0 运算。
- p && *p++ 可以避免空指针运算。

- 5 || x = y 赋值语句将不会被执行。

1.2.5 C 语言中的移位运算

C 语言中有逻辑移位和算术移位。右移运算有逻辑移位（左侧补 0）和算术移位（左侧补原最高位值）两种操作。对于无符号数，右移是逻辑的；对于有符号数，几乎所有的编译器针对有符号数的右移都采用的是算术右移

Operation	Values	
Argument x	[01100011]	[10010101]
x << 4	[00110000]	[01010000]
x >> 4 (logical)	[00000110]	[00001001]
x >> 4 (arithmetic)	[00000110]	[11111001]

图 1 移位运算

1.2.6 未定义行为

C 语言规范中没有被明确定义的行为称为未定义行为（UB），编程时应避免使用未定义行为，但有符号数算术右移除外。例如，移位 k，当 k 大于等于变量位长时，值直接变为 0，在 GCC 中的实现如下：

```
int aval = 0x0EDCBA98 >> 36;
movl $0, -8(%ebp)   // 值直接变为 0
unsigned uval = 0xFEDCBA98u << 40;
movl $0, -4(%ebp)   // 值直接变为 0
```

1.2.7 运算优先级

移位运算符的优先级低于加减乘除，例如 $-1 << 2+3 << 4$，正确的运算顺序为 $(1 << (2+3)) << 4$ 。

1.3 信息的存储和表示

1.3.1 字长

字长是指针数据的大小（虚拟地址宽度）。32 位 (4 字节) 计算机字长限制了地址空间为 4GiB（2^{32} 字节）；64 位字长 (8 字节) 寻址能力达到了 18EiB

1.3.2 C 语言中的各数据类型位宽

计算机和编译器支持多种数据类型，或是小于字长，或大于字长，但长度都是整数个字节。

C 语言数据类型	典型 32 位系统	典型 64 位系统	x86 - 64
char	1	1	1
short	2	2	2
int	4	4	4
long	4	8	8
float	4	4	4
double	8	8	8
pointer	4	8	8

表 4: C 语言数据类型位宽

1.3.3 字节序

有小端序（Little endian）和大端序（Big endian）两种。小端序如 Intel，低地址存放低位数据，高地址存放高位数据；大端序如 IBM、Sun Microsystem（Oracle），低地址存放高位数据，高地址存放低位数据。例如，对于 0x1234567 ：

	地址			
	低地址			高地址
大端序	$0x100 : 0x12$	$0x101 : 0x34$	$0x102 : 0x56$	$0x103 : 0x07$
小端序	$0x100 : 0x07$	$0x101 : 0x56$	$0x102 : 0x34$	$0x103 : 0x12$

表 5: 字节序示例

1.3.4 探索数据在存储器中的存储方式

通过以下代码可以打印各变量的字节表示形式：

```c
#include <stdio.h>
typedef unsigned char *byte_pointer;
void show_bytes(byte_pointer start,int len)
{
    int i;
    for(i = 0; i < len; i++)
        printf("%.2x ",start[i]);
    printf("\n");
}
void show_int(int x)
{
    show_bytes((byte_pointer)&x, sizeof(int));
}
void show_float(float x)
{
    show_bytes((byte_pointer)&x, sizeof(float));
}
```

```
18 void show_pointer(void *x)
19 {
20     show_bytes((byte_pointer)x, sizeof(void*));
21 }
```

在 Linux32/64（小端）、Win32（小端）和 Sun（32 位，大端）系统下的测试结果如下：

机器	值	类型	字节（十六进制）
Linux 32	12345	int	39 30 00 00
Windows	12345	int	39 30 00 00
Sun	12345	int	00 00 30 39
Linux 64	12345	int	39 30 00 00
Linux 32	12345.0	float	00 e4 40 46
Windows	12345.0	float	00 e4 40 46
Sun	12345.0	float	46 40 e4 00
Linux 64	12345.0	float	00 e4 40 46
Linux 32	&*ival*	int *	e4 f9 ff bf
Windows	&*ival*	int *	b4 cc 22 00
Sun	&*ival*	int *	ef ff fa 0c
Linux 64	&*ival*	int *	b8 11 e5 ff ff 7f 00 00

表 6: 不同系统下数据存储测试结果

1.3.5 指针的存储方法

不同的编译器和计算机可能会分配不同的地址，甚至每一次运行时得到的结果都不相同。例如，在 x86 - 64、Sun、IA32 环境下：

```
1 int B = -15213;
2 int *P = &B;
```

1.3.6 字符串的表示

C 语言的字符串使用 char 数组表示，每个字符都被编码成 ASCII 码，是一个 7 比特的字符编码集（扩展集为 8 比特）。字符“0”的编码是 $0x30$，数字字符 i 的编码是 $0x30 + i$ 。字符串的结尾应为空字符，即 ASCII 编码为 0 。字符串的表示与字节序无关，大小端兼容。

1.3.7 程序的表示

不同类型的机器使用不同的且不兼容的指令和指令编码。在相同处理器不同的操作系统中，由于编码规范存在差异，同样代码所生成的程序也不是二进制兼容的，程序很少能够在不同类型机器和不同操作系统中实现二进制水平上移植。

1.3.8 小知识：PE 和 ELF 格式

Windows 操作系统下常用的可执行文件格式是 PE（Portable Executable）；Unix 家族（含 Linux）操作系统下可执行文件格式为 ELF（Executable and Linkable Format）。

2 浮点数

2.1 传统小数的二进制表示

在二进制中，小数点左侧每一位的权重为 2^i，右侧为 $\frac{1}{2^i}$（位置计数法），一个二进制数 b 可以表示为 $b = \sum_{i=-n}^{m} 2^i \times b_i$。例如：$5\frac{3}{4} = 101.11_2$ $2\frac{7}{8} = 010.111_2$

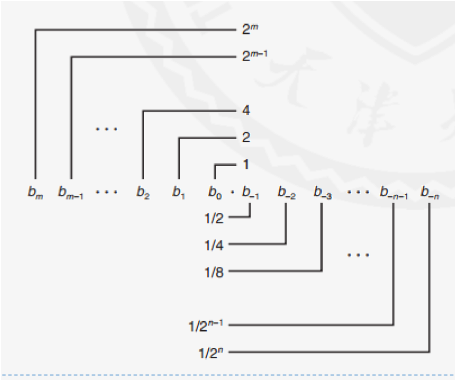

图 2 小数的二进制表示

并且，除以 2 可通过逻辑右移实现，乘以 2 可通过左移实现。像 $0.11111\cdots_2$ 这样的数字十分接近但小于 1，可表示为 $1.0-\varepsilon$。同时，并非所有有理数都能精确地用有限位二进制小数表示，例如 $\frac{1}{3} = 0.0101010101[01]\cdots_2$，只能精确表示 $\frac{x}{2^k}$ 这种形式的数字。

2.2 IEEE 浮点数的表示

浮点数的表示形式为 $(-1)^s \times M \times 2^E$，其中：

- 符号位 s 决定了数字的正负。
- 尾数 M 通常在 $[1.0, 2.0)$ 或 $[0.0, 1.0)$。
- 阶码 E 是浮点数的权重，为 2 的 E 次幂。

图 3 浮点数编码

编码方式为：最高位是符号位 s，exp 编码后得到 E（$\text{exp} \neq E$），frac 编码后得到 M（$\text{frac} \neq M$）。

2.2.1 几种精度的浮点数

常见的浮点数精度有：

精度类型	总位数	符号位	exp 位数	frac 位数
单精度	32 位	1 位	8 位	23 位
双精度	64 位	1 位	11 位	52 位
扩展精度（仅 Intel 支持）	80 位	1 位	15 位	63 或 64 位

表 7: 不同精度浮点数的位宽分布

2.2.2 规格化数

当 $\exp \neq 000\cdots0$ 且 $\exp \neq 111\cdots1$ 时：

- 尾数编码为包含一个隐式前置的 1，即 $M = 1.x_1x_2\cdots$，其中 $x_1x_2\cdots$ 为 frac 域的各位的编码。
- 阶码为一个有偏置的指数，$E = \text{Exp} - \text{Bias}$，Exp 为 exp 域无符号数编码值，$\text{bias} = 2^{k-1}-1$，$k$ 是 exp 的位宽。
- 例如，对于单精度浮点数，$\text{bias} = 127$（$\text{Exp} : 1\cdots254$，$E : -126\cdots127$）；双精度浮点数，$\text{bias} = 1023$（$\text{Exp} : 1\cdots2046$，$E : -1022\cdots1023$）。隐式前置的整数 1 始终存在，因此在 frac 中不需要包含。

以 float $f = 15213.0$ 为例：

$$
\begin{aligned}
f = 15213_{10} &= 11011011011010000000_2 \\
&= 1.1101101101101101_2 \times 2^{13} \\
\text{尾数}\quad M &= 1.\mathbf{1101101101101101}_2 \\
\text{frac} &= \mathbf{1101101101101}0000000000_2 \\
\text{阶码}\quad E &= 13, \\
\text{Bias} &= 2^{k-1} - 1 = 127 \quad (k = 8), \\
\text{Exp} &= 140 = 10001100_2
\end{aligned}
$$

结果为 0 10001100 11011011011010000000000。

2.2.3 非规格化数

当 $\exp = 000\cdots0$ 时：

- 阶码 $E = -\text{Bias} + 1$（而不是 $E = 0 - \text{Bias}$）。
- 尾数编码为包含一个隐式前置的 0，即 $M = 0.x_1x_2\cdots$。

- 当 $\exp = 000\cdots0$，且 $\text{frac} = 000\cdots0$ 时，表示 0，要注意 $+0$ 和 -0 的区别。
- 当 $\exp = 000\cdots0$，且 $\text{frac} \neq 000\cdots0$ 时，表示非常接近于 0.0 的数字，这些数字是等间距的。

2.2.4 特殊值

当 $\exp = 111\cdots1$ 时：

- 当 $\exp = 111\cdots1$ 且 $\text{frac} = 000\cdots0$ 时，表示无穷 ∞，意味着运算出现了溢出，有正向溢出和负向溢出，例如 $1.0/0.0 = -1.0/-0.0 = +\infty$，$1.0/-0.0 = -\infty$。
- 当 $\exp = 111\cdots1$ 且 $\text{frac} \neq 000\cdots0$ 时，表示不是一个数字（NaN），表示数值无法确定，例如 $\sqrt{-1}, \infty - \infty, \infty \times 0$。

2.2.5 IEEE 编码的特殊属性

1. 浮点数 0 和整数 0 编码相同，所有位都为 0
2. 几乎可以用无符号整数比较的方法实现浮点数的比较运算，但首先要比较符号位，同时要考虑 $-0 = 0$ 和 NaN 的问题 (NaN 比其他值都大)

2.2.6 需要关注的数字

不同类型的浮点数有一些特殊的数字：

描述	exp	frac	单精度值（十进制）	双精度值（十进制）
零	$00\cdots00$	$0\cdots00$	0.0	0.0
最小非规格化数	$00\cdots00$	$0\cdots01$	$2^{-23} \times 2^{-126}$	$2^{-52} \times 2^{-1022}$
最大非规格化数	$00\cdots00$	$1\cdots11$	$(1-\varepsilon) \times 2^{-126}$	$(1-\varepsilon) \times 2^{-1022}$
最小规格化数	$00\cdots01$	$0\cdots00$	1×2^{-126}	1×2^{-1022}
一	$01\cdots11$	$0\cdots00$	1.0	1.0
最大规格化数	$11\cdots10$	$1\cdots11$	$(2-\varepsilon) \times 2^{127}$	$(2-\varepsilon) \times 2^{1023}$

表 8: 不同精度浮点数的特殊值

2.2.7 举例：一个微型的浮点数编码系统

以 8 位浮点数编码为例，最高位为符号位，接下来是 4 位 exp，偏置 Bias 为 7，最后 3 位是 frac，与 IEEE 规范具有相同的形式，有规格化数、非规格化数，以及 0、NaN 和无穷的编码。

s	exp	frac	E	值
0	0000	000	-6	0
0	0000	001	-6	$\frac{1}{8} \times \frac{1}{64} = \frac{1}{512}$（最接近 0）
⋮	⋮	⋮	⋮	⋮
0	1111	000	n/a	inf

表 9: 8 位浮点数编码示例

2.2.8 总结

单精度浮点数可分为以下几类：

1. 规格化数：exp $\neq$ 0000000 且 exp $\neq$ 11111111。
2. 非规格化数：exp = 00000000，frac $\neq$ 0000000。
3. 无穷：exp = 11111111，frac = 00000000。
4. NaN：exp = 11111111，frac $\neq$ 00000000。

2.3 几种舍入模式

常见的舍入模式有：

1. 向下舍入：舍入结果接近但不会大于实际结果。
2. 向上舍入：舍入结果接近但不会小于实际结果。
3. 向 0 舍入：舍入结果向 0 的方向靠近，如果为正数，舍入结果不大于实际结果；如果为负数，舍入结果不小于实际结果。
4. 向偶数舍入：浮点数运算默认的舍入模式，其他的舍入模式都会统计偏差，一组正数的总和将始终被高估或低估。

向偶数舍入适用于舍入至小数点后任何位置，当数字正好处在四舍五入的中间时，向最低位为偶数的方向舍入。例如：

值	结果	说明
1.2349999	1.23	比中间值小，四舍
1.2350001	1.24	比中间值大，五入
1.2350000	1.24	中间，向上舍入（偶数方向）
1.2450000	1.24	中间，向下舍入（偶数方向）

表 10: 向偶数舍入示例

在二进制数中，偶数方向意味着舍入后最后一位为 0，中间意味着待舍入的部分为 $100\cdots_2$。

2.4 浮点数运算

浮点数运算的基本思想是先计算出精确的值，然后将结果调整至目标的精度。如果阶码值过大，可能会导致溢出，可能会进行舍入以满足尾数的位宽。例如：

$$x +^f y = \text{Round}(x + y)$$
$$x \times^f y = \text{Round}(x \times y)$$

2.4.1 浮点数乘法

对于浮点数 $(-1)^{s_1} M_1 2^{E_1} \times (-1)^{s_2} M_2 2^{E_2}$：

- 精确结果：$M_1 \times M_2$，阶码 $E = E_1 + E_2$，符号位 $s = s_1^{s_2}$。
- 修正：如果 $M \geq 2$，右移 M，并增大 E 的值；如果 E 超出范围，发生溢出；对 M 进行舍入以满足 frac 的位宽精度要求。在实际实现中，尾数相乘的细节较为繁琐。

2.4.2 浮点数加法

对于浮点数 $(-1)^{s_1} M_1 2^{E_1} + (-1)^{s_2} M_2 2^{E_2}$（假设 $E_1 > E_2$）：

- 精确结果：$(-1)^s M_2^E$，其中符号位 s 和尾数 M 是有符号数对齐后相加的结果，阶码 $E = E_2$。
- 修正：如果 $M \geq 2$，右移 M，并增大 E 的值；如果 $M < 1$，左移 M k 位，然后 E 减去 k；如果 E 超出范围，发生溢出；对 M 进行舍入以满足 frac 的位宽精度要求。

2.5 C 语言中的浮点数

C 语言标准确保支持两种精度的浮点数，即 float（单精度）和 double（双精度）。在进行类型转换时：

- double/float 转 int：截断尾数部分，向 0 舍入。标准中未定义越界和 NaN 的情况，通常设置为 T_{Min} 和 T_{Max}。
- int 转 double：只要 int 的位宽小于等于 53 位，就能精确转换。
- int 转 float：会根据舍入模式进行舍入。

以下是一些 C 语言中浮点数操作的示例（假设 d 和 f 分别是 float 和 double 且不是 NaN 和无穷）：

表达式	结果
`x == (int)(float) x`	否：有效数字为 24 位
`x == (int)(double) x`	是：有效数字为 53 位
`f == (float)(double) f`	是：提高精度
`d == ( float ) d`	否：丢失精度
`f == -(-f)`	是：仅改变符号位
`2/3 == 2/3.0`	否：$2/3 == 0$
`if(d < 0.0) ((d*2) < 0.0)`	是
`d * d >= 0.0`	是
`(d + f) - d == f`	否：不满足结合律

表 11: C 语言中浮点数操作示例结果

在浮点数运算中，存在精度和准确度的问题。例如 $0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1 \neq 1$，但 $0.2+0.2+0.2+0.2+0.2 = 1$。在使用浮点数进行比较时，如`if (d1 - d2 == 0)`，需要谨慎处理，因为浮点数的精度限制可能导致结果不准确。在进行浮点数运算时，要充分考虑精度损失对计算结果的影响。