位运算的一些技巧

张坤

清华大学

May 27, 2017

简单介绍

当计算某一算法的总操作次数时，任何一个 CPP 语言的运算符被当做一条操作，写内存的中间操作不被计算在内。但这种操作数计算方法假设每一条不同操作耗费的时间是相同的，但事实并非如此。有很多细节会影响到在某个系统上一段代码的运行时间，比如如缓存大小, 内存带宽, 指令集等。

简单介绍

计算机中的数据都是以二进制的形式存储。位运算，就是直接对整数在内存中以二进制的形式进行计算，各位之间彼此独立，因此处理速度非常快。

位运算的基本操作

Table: 位运算的六种操作

名称	符号	作用
与	&	对于每个位，只有两个位都为 1 时，该位为 1
或	|	对于每个位，只有两个位都为 0 时，该位为 0
异或	∧	对于每个位，只有两个位不同时，该位为 1
取反	~	对于每个位，0 变 1,1 变 0
左移	«	各二进位全部左移若干位，高位丢弃，低位补 0
右移	»	各二进位全部右移若干位，低位丢弃，高位补 0 有符号数，各编译器处理方法不一样， 有的补符号位（算术右移），有的补 0（逻辑右移）

位运算的优先级

优先级	运算符	结合性
1	() [] .	从左到右
2	! +(正) -(负) ~ ++ --	从右向左
3	* / %	从左向右
4	+(加) -(减)	从左向右
5	<< >> >>>	从左向右
6	< <= > >= instanceof	从左向右
7	== !=	从左向右
8	&(按位与)	从左向右
9	^	从左向右
10	|	从左向右
11	&&	从左向右
12	||	从左向右
13	?:	从右向左
14	= += -= *= /= %= &= |= ^= ~= <<= >>= >>>=	从右向左

虽然掌握各个运算符的优先级并不困难，但是为了避免出错，增强程序的可读性，利于调试，我们还是在需要的地方添加括号来保证优先运算。

判断一个整数的符号

```
 1 int v;    // 判断v的符号
 2 int sign; // 判断的结果存在这里
 3
 4 // CHAR_BIT 一个字节(byte)包含的位数(通常是8).
 5 // 方法1
 6 sign = -(v < 0); // if v < 0 then -1, else 0.
 7 // 方法2 避免通过使用CPU寄存器的标志位进行分支(IA32):
 8 sign = -(int)((unsigned int)((int)v) >> (sizeof(int) * CHAR_BIT - 1));
 9 // 方法3 再减少一条指令 (但不具有可移植性):
10 sign = v >> (sizeof(int) * CHAR_BIT - 1);
11
```

上面最后一个表达式等效于 $sign = v >> 31$（用于计算 32 位整数的符号），这种方法只进行一次运算，比通常的 $sign = -(v < 0)$ 要快。

这种方法可以奏效是因为当有符号的整数 v 进行右移时，最左边的位被拷贝到了其他位。当 v 的值为负数时，最左边的位为 1，否则的话最左边的位为 0（v 为整数或者 0 时）；所有的位为 1 表示的数为-1（-1 在内存中为 0xffffffff）。

判断一个整数的符号

如果你想要的结果是-1 或 1，方法如下：

```
1 int v;    // 判断v的符号
2 int sign; // 判断的结果存在这里
3
4 // CHAR_BIT 一个字节(byte)包含的位数(通常是8).
5
6 // if v < 0 then -1, else +1;
7 sign = +1 | (v >> (sizeof(int) * CHAR_BIT - 1));
8
```

如果你想要的结果是-1,0 或 +1，那么方法如下：

```
1 int v;    // 判断v的符号
2 int sign; // 判断的结果存在这里
3
4 // CHAR_BIT 一个字节(byte)包含的位数(通常是8).
5
6 // if v < 0 then -1, else +1;
7 sign = +1 | (v >> (sizeof(int) * CHAR_BIT - 1));
8
```

比较两个整数的符号是否相反

比较两个整数的符号是否相反，若相反返回 1

```
1 int x, y; // 待比较的两变量
2
3 bool f = ((x ^ y) < 0); // 如果x和y的符号相反，结果为真
```

计算一个整数的绝对值

计算一个整数的绝对值，不使用分支语句

```
 1 int v; // 求v的绝对值
 2 unsigned int r; // 结果存在这里
 3 int const mask = v >> sizeof(int) * CHAR_BIT - 1;
 4
 5 //方法一
 6 r = (v + mask) ^ mask;
 7
 8 //方法二
 9 r = (v ^ mask) - mask;
10
```

求两个整数中的最小值或最大值

求两个整数中的最小值或最大值，不使用分支结构

```
1  int x; // 求x y中的最小值
2  int y;
3  int r; // 存放结果
4
5  //求最小值的方法为
6  r = y ^ ((x ^ y) & -(x < y)); // min(x, y)
7
8  //求最大值的方法为:
9  r = x ^ ((x ^ y) & -(x < y)); // max(x, y)
10
```

上面的方法**可能**会比方法 $r = (x < y)?x : y$ 高效，虽然上面的方法多两条指令。如果 x<y，那么 $-(x < y)$ 的每一位都是 1，那么 $r = y \wedge (x \wedge y)\&\sim 0 = y \wedge x \wedge y = x$。

判断一个整数是不是 2 的幂

一个整数是不是 2 的幂

```
1 unsigned int v; // 判断 v 是不是 2 的幂
2 bool f; // 结果放在这里
3
4 f = (v & (v - 1)) == 0;
```

在这里 0 被认为是 2 的幂
例：0001 1000 & 0001 0111 = 0001 0000
例：0010 0000 & 0001 1111 = 0000 0000

判断并进行设置位或清除位的运算

```
1  bool f; // conditional flag
2  unsigned int m; // the bit mask
3  unsigned int w; // the word to modify
4
5  //if (f) w |= m; else w &= ~m;
6
7  //方法一
8  w ^= (-f ^ w) & m;
9
10 // 方法二
11 w = (w & ~m) | (-f & m);
12
```

使用异或运算交换两个变量的值

```
1 int swap_nums(int *a, int *b) {
2     *a = *a ^ *b;
3     *b = *a ^ *b;
4     *a = *a ^ *b;
5 }
6
```

使用异或运算交换两个变量的值

```
1 int swap_nums(int *a, int *b) {
2     *a = *a ^ *b;
3     *b = *a ^ *b;
4     *a = *a ^ *b;
5 }
6
```

```
1 int a = 1;
2 swap_nums(&a, &a);
3 printf("%d\n", a);
4
```

用户可能传入两个相同的地址，交换以后，数字就变成 0。

枚举子集

在限定必须不取某些元素的前提下枚举子集
mask 的第 x 位为 0 表示 x 必须不在子集中

```
1 for (int mask1 = mask; mask1 >= 0; mask1 = (mask1 - 1) & mask)
2
```

例：01011 01010 01001 01000 00100 00010 00001

在限定必须取某些元素的前提下枚举子集
mask 的第 x 位为 1 表示 x 必须在子集中

```
1 for (int mask1 = mask; mask1 < (1 << m); mask1 = (mask1 + 1) | mask)
2
```

例：01011 01111 11011 11111

有效位统计 (暴力)

```
1 unsigned int v; // count the number of bits set in v
2 unsigned int c; // c accumulates the total bits set in v
3
4 for (c = 0; v; v >>= 1){
5   c += v & 1;
6 }
7
```

原始的方法每一位需要进行一次迭代，直到所有的位统计完成。所以如果一个 32 位字长只有最高位有效时，也会进行 32 次迭代。

Brian Kernighan 法进行有效位统计

```
1 unsigned int v; // count the number of bits set in v
2 unsigned int c; // c accumulates the total bits set in v
3 for (c = 0; v; c++)
4 {
5   v &= v - 1; // clear the least significant bit set
6 }
7
```

使用 Brian Kernighan 法，变量中有多少个有效位，就进行多少次迭代。所以说如果一个 32 位字长的变量最高位有效，那么也只需要进行一次循环即可得出结果。

查表法进行有效位统计

```
static const unsigned char BitsSetTable256[256] =
{
#   define B2(n) n,     n+1,     n+1,     n+2
#   define B4(n) B2(n), B2(n+1), B2(n+1), B2(n+2)
#   define B6(n) B4(n), B4(n+1), B4(n+1), B4(n+2)
    B6(0), B6(1), B6(1), B6(2)
};

unsigned int v; // count the number of bits set in 32-bit value v
unsigned int c; // c is the total bits set in v

c = BitsSetTable256[v & 0xff] +
    BitsSetTable256[(v >> 8) & 0xff] +
    BitsSetTable256[(v >> 16) & 0xff] +
    BitsSetTable256[v >> 24];

// To initially generate the table algorithmically:
BitsSetTable256[0] = 0;
for (int i = 0; i < 256; i++)
{
  BitsSetTable256[i] = (i & 1) + BitsSetTable256[i / 2];
}

```

查表法进行有效位统计

表里面真实样子

```
1
2 static const unsigned char BitsSetTable256[256] =
3 {
4 0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4, 1, 2, 2, 3, 2, 3, 3,
  4, 2, 3, 3, 4, 3, 4, 4, 5, 1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 3, 4,
  4, 5, 2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6, 1, 2, 2, 3, 2,
  3, 3, 4, 2, 3, 3, 4, 3, 4, 4, 5, 2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5,
  4, 5, 5, 6, 2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6, 3, 4, 4,
  5, 4, 5, 5, 6, 4, 5, 5, 6, 5, 6, 6, 7, 1, 2, 2, 3, 2, 3, 3, 4, 2, 3,
  3, 4, 3, 4, 4, 5, 2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6, 2,
  3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6, 3, 4, 4, 5, 4, 5, 5, 6,
  4, 5, 5, 6, 5, 6, 6, 7, 2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5,
  6, 3, 4, 4, 5, 4, 5, 5, 6, 4, 5, 5, 6, 5, 6, 6, 7, 3, 4, 4, 5, 4, 5,
  5, 6, 4, 5, 5, 6, 5, 6, 6, 7, 4, 5, 5, 6, 5, 6, 6, 7, 5, 6, 6, 7, 6,
  7, 7, 8 };
5
```

对 14，24，32 位变量使用 64 位指令进行有效位统计

Counting bits set in 14, 24, or 32-bit words using 64-bit instructions

```
unsigned int v; // count the number of bits set in v
unsigned int c; // c accumulates the total bits set in v

// option 1, for at most 14-bit values in v:
c = (v * 0x200040008001ULL & 0x111111111111111ULL) % 0xf;

// option 2, for at most 24-bit values in v:
c =  ((v & 0xfff) * 0x1001001001001ULL & 0x84210842108421ULL) % 0x1f;
c += (((v & 0xfff000) >> 12) * 0x1001001001001ULL & 0x84210842108421ULL)
     % 0x1f;

// option 3, for at most 32-bit values in v:
c =  ((v & 0xfff) * 0x1001001001001ULL & 0x84210842108421ULL) % 0x1f;
c += (((v & 0xfff000) >> 12) * 0x1001001001001ULL & 0x84210842108421ULL) %
     0x1f;
c += ((v >> 24) * 0x1001001001001ULL & 0x84210842108421ULL) % 0x1f;
```

这种方法需要使用能够进行快速取模和除法的 64 位 CPU 才能具有较高的效率。14 位只需要 3 步操作；24 位需要 10 部操作；32 位需要 15 步操作。

并行计算有效位

The B array, expressed as binary, is:

```
B[0] = 0x55555555 = 01010101 01010101 01010101 01010101
B[1] = 0x33333333 = 00110011 00110011 00110011 00110011
B[2] = 0x0F0F0F0F = 00001111 00001111 00001111 00001111
B[3] = 0x00FF00FF = 00000000 11111111 00000000 11111111
B[4] = 0x0000FFFF = 00000000 00000000 11111111 11111111
```

计算 32 位整数的有效位的最佳方法如下:

```
1 v = v - ((v >> 1) & 0x55555555); // 把输入值作为临时变量再次使用
2 v = (v & 0x33333333) + ((v >> 2) & 0x33333333); // temp
3 c = ((v + (v >> 4) & 0xF0F0F0F) * 0x1010101) >> 24; // count
4
```

二进制逆序的通常方法

```
unsigned int v; // 待进行二进制逆序的数
unsigned int r = v; // r 将会用来保存 v 的二进制逆序结果;
int s = sizeof(v) * CHAR_BIT - 1; // 最后要额外进行的位移数

for (v >>= 1; v; v >>= 1)
{
  r <<= 1;
  r |= v & 1;
  s--;
}
r <<= s; // 移动的位数s 即 v 的最高位为0的位的个数

```

查表法对二进制序列进行逆序

```
 1 static const unsigned char BitReverseTable256[256] =
 2 {
 3 # define R2(n) n, n + 2*64, n + 1*64, n + 3*64
 4 # define R4(n) R2(n), R2(n + 2*16), R2(n + 1*16), R2(n + 3*16)
 5 # define R6(n) R4(n), R4(n + 2*4 ), R4(n + 1*4 ), R4(n + 3*4 )
 6     R6(0), R6(2), R6(1), R6(3)
 7 };
 8
 9 unsigned int v; // 逆序 32 位变量，一次 8 位
10 unsigned int c; // c 用来得到 v 逆序后的结果。
11
12 // Option 1:
13 c = (BitReverseTable256[v & 0xff] << 24) |
14     (BitReverseTable256[(v >> 8) & 0xff] << 16) |
15     (BitReverseTable256[(v >> 16) & 0xff] << 8) |
16     (BitReverseTable256[(v >> 24) & 0xff]);
17
```

二进制位序列逆序

```
1 //3步运算实现一个字节的二进制位序列逆序
2 //64位乘法和除法取模运算
3
4 unsigned char b; // 逆序这个 (8位) 字节
5 b = (b * 0x0202020202ULL & 0x010884422010ULL) % 1023;
6
7 //4步运算实现一个字节的二进制位序列逆序
8 //64位乘法，不需要除法运算
9
10 unsigned char b; // 逆序这个字节
11 b = ((b * 0x80200802ULL) & 0x0884422110ULL) * 0x0101010101ULL >> 32;
12
13 //7步运算实现一个字节的二进制位序列逆序
14 //非64位
15 b = ((b * 0x0802LU & 0x22110LU) | (b * 0x8020LU & 0x88440LU)) * 0x10101LU >> 16;
16
```

10 billion 次操作下的时间分别是 16.536s、6.569s、3.844s。

补充

其他功能

将布尔数组压位，C++ 库：bitset

参考资料

http://graphics.stanford.edu/ seander/bithacks.html