Bit Hacks：关于一切位操作的魔法（上）

原文：https://graphics.stanford.edu/~seander/bithacks.html

欢迎在公众号，知乎，bilibili 上关注代码律动，一起学习魔法

计算整数的符号

```
int v;      // 我们需要找的符号
int sign;   // 结果存在这里

// CHAR_BIT 是一个字节中的比特位数，一般为8
// 方法1：
sign = -(v < 0);  // if v < 0 then -1, else 0.
// 方法2：这里不使用 if 以避免在 CPU 上的分支跳转
sign = -(int)((unsigned int)((int)v) >> (sizeof(int) * CHAR_BIT - 1));
// 方法3：更短，但是失去了可移植性
sign = v >> (sizeof(int) * CHAR_BIT - 1);
```

方法 3 在 32 位的机器上实际执行的是`sign = v >> 31` ，这显然比方法 1 `sign = -(v < 0)` 要快。它的作用原理是有符号整数最左边的比特位标记了它的符号，当符号位 1 的时候，把所有比特位右移 31 个单位，在 32 位计算机上，有符号整数的符号位就到了第一位，其它位变成 0。所以最后只要判断结果是不是 1 就好了。不过这个方法没有可移植性。

还有，如果你更喜欢用 -1 和 +1 来描述正负，可以这样写：

```
sign = +1 | (v >> (sizeof(int) * CHAR_BIT - 1));  // if v < 0 then -1, else +1
```

再假如你希望区分正数，负数，零，那么用：

```
sign = (v != 0) | -(int)((unsigned int)((int)v) >> (sizeof(int) * CHAR_BIT -
1));
// 更快但是没有可移植性:
sign = (v != 0) | (v >> (sizeof(int) * CHAR_BIT - 1));  // -1, 0, or +1
// 可能是最短的:
sign = (v > 0) - (v < 0); // -1, 0, or +1
```

如果你只是想知道是不是非负数，返回 1 或者 0，那么这么用:

```
sign = 1 ^ ((unsigned int)v >> (sizeof(int) * CHAR_BIT - 1)); // if v < 0 then
0, else 1
```

检查两数是否异号

```
int x, y;               // 输入的两个数
bool f = ((x ^ y) < 0); // 当且仅当 x y 异号为 true
```

不用分支来计算绝对值

```
int v;           // 我们希望计算 v 的绝对值
unsigned int r;  // 结果存在 r 中
int const mask = v >> sizeof(int) * CHAR_BIT - 1;
// 方法 1：
r = (v + mask) ^ mask;
// 方法 2：
r = (v ^ mask) - mask;
```

一些 CPU 并没有计算整数绝对值的指令（或者编译器并没有使用它们），在计算机中，分支语句的代价是比较大的，所以上述的方法 1、2 的运行速度都快于`r = (v < 0) ? -(unsigned)v : v`，即使操作符数量是一样的。

不用分支计算两个数之间的最小最大值

```
int x, y;  // 我们想找出 x 和 y 的最大最小值
int r;  // 结果存在 r 中

r = y ^ ((x ^ y) & -(x < y)); // min(x, y)
```

因为在少数 CPU 上缺少条件移动指令以及分支语句的较高代价，所以上面的方法要比 `r = (x < y) ? x : y` 这种直观的方法要更快，即使它使用了更多的操作符。这个方法的作用原理是，当 `x < y` 时，则 `-(x < y)` 而值为 1，所以 `r = y ^ (x ^ y) & ~0 = y ^ x ^ y = x` ，否则，`x >= y` 时，`-(x < y)` 将为 0，所以 `r = y ^ ((x ^ y) & 0) = y` 。

计算最大值也是这样：

```
r = x ^ ((x ^ y) & -(x < y)); // max(x, y)
```

一个更快但是更脏的实现：

如果你知道 `INT_MIN <= x - y <= INT_MAX`，那么你可以用下面的方法，它更快，因为 `(x - y)` 只需要计算一次。

```
r = y + ((x - y) & ((x - y) >> (sizeof(int) * CHAR_BIT - 1))); // min(x, y)
r = x - ((x - y) & ((x - y) >> (sizeof(int) * CHAR_BIT - 1))); // max(x, y)
```

值得注意的是 1989 ANSI C 规范没有明确有符号整数的右移策略，所以他们并不具有可移植性。如果溢出时会抛出异常，则 x 和 y 应该先转为无符号整型，再做减法，以避免可能的溢出，不过在计算右移操作 `>>` 时，是需要有符号位的，所以在进行无符号整形减法后要转回有符号整型。

判断一个整数是不是 **2** 的幂

```
unsigned int v; // 我们想要计算 v 是不是 2 的幂
bool f;         // 结果存在 f 中
f = (v & (v - 1)) == 0;
```

注意 0 不应该是 2 的幂，为了避免这个问题，可以使用：

```
f = v && !(v & (v - 1));
```

对固定位长数字进行符号扩展

对于内置类型（比如 char 和 int），符号扩展（Sign extension）都是自动进行的。但是假如你有一个用 b 比特位记录的补码表示的数字 x，并想将 x 转为比特位更长的 int 类型。如果 x 是正数，那么只要简单的复制就可以了，但是如果 x 是负数，符号位就必须进行扩展。比如我们有 4 bits 存储 -3，表示为 `1101`，如果我们对它进行符号位扩展为 8 位，则 -3 表示为 `11111101`。最左边的符号位被重复了四遍，使最终的长度变成 8 位，这就是符号扩展的意义。在 C 中，符号扩展需要一些特殊的技巧，因为自定义的比特位长度需要用 struct 或者 union 来描述。比如：

```
int x; // 用 x 的 5 bits 转成一个完整的 int
int r; // 储存符号扩展的结果
struct {signed int x:5;} s;
r = s.x = x;
```

下面用一个 C++ 模板函数进行符号扩展的例子：

```
template <typename T, unsigned B>
inline T signextend(const T x)
{
  struct {T x:B;} s;
  return s.x = x;
}

int r = signextend<signed int,5>(x);
```

对变长位数字进行符号扩展

有时我们并无法直接固定位长，而是要处理可能不同长度的位数。所以假设我们要扩展的数字位长是 b，那么可以这么进行符号扩展：

```
unsigned b; // 数字 x 的位长为 b 来表示
int x;      // 将 x 的 b 个位扩展到 int
int r;      // 符号扩展的结果
int const m = 1U << (b - 1); // 如果 b 是固定的，那么这个 mask 可以事先计算好

x = x & ((1U << b) - 1);  // 如果 x 是 0，那么就不用计算，直接得到 0
r = (x ^ m) - m;
```

以上的代码需要 4 个操作符，但是当位长是固定的值时，这个算法就只需要两个操作符了。

下面这个版本还会更快一点，但是可移植性较弱：

```
int const m = CHAR_BIT * sizeof(x) - b;
r = (x << m) >> m;
```

用三个操作符对变长位数字进行符号扩展

下面的方法由于涉及到了乘法和除法，在一些计算机上可能会比较慢。如果你能确定你的位长 b 大于 1，那么可以使用数组查询的方式：`r = (x * multipliers[b]) / multipliers[b]` 简化到三个操作符。

```
unsigned b; // 数字 x 的位长为 b 来表示
int x;      // 数字 x 的位长为 b 来表示
int r;      // 符号扩展的结果

#define M(B) (1U << ((sizeof(x) * CHAR_BIT) - B)) // CHAR_BIT=bits/byte
```

```
static int const multipliers[] =
{
  0,      M(1),  M(2),  M(3),  M(4),  M(5),  M(6),  M(7),
  M(8),  M(9),  M(10), M(11), M(12), M(13), M(14), M(15),
  M(16), M(17), M(18), M(19), M(20), M(21), M(22), M(23),
  M(24), M(25), M(26), M(27), M(28), M(29), M(30), M(31),
  M(32)
}; // 如果你需要写 64 位的程序的话，你需要更多的 M
static int const divisors[] =
{
  1,     ~M(1),  M(2),  M(3),  M(4),  M(5),  M(6),  M(7),
  M(8),  M(9),  M(10), M(11), M(12), M(13), M(14), M(15),
  M(16), M(17), M(18), M(19), M(20), M(21), M(22), M(23),
  M(24), M(25), M(26), M(27), M(28), M(29), M(30), M(31),
  M(32)
}; // 如果你需要写 64 位的程序的话，你需要更多的 M
#undef M
r = (x * multipliers[b]) / divisors[b];
```

下面的方法没有可移植性，但是由于只用了位操作符，所以速度更快。

```
const int s = -b; // OR:  sizeof(x) * CHAR_BIT - b;
r = (x << s) >> s;
```

不用分支，根据条件设置/清除比特位

```
bool f;         // 条件 flag
unsigned int m; // bit mask
unsigned int w; // 需要修改的的数字:  if (f) w |= m; else w &= ~m;

w ^= (-f ^ w) & m;

// 支持超标量技术 (superscalar) 的 CPU 还可以这样写:
w = (w & ~m) | (-f & m);
```

在一些处理器架构中，减少分支可能提升一倍的操作符计算能力。

不用分支，根据条件求相反数

```
bool fDontNegate;  // 标记我们是否应该对 v 求相反数
int v;             // 原数字
int r;             // 结果 result = fDontNegate ? v : -v;

r = (fDontNegate ^ (fDontNegate - 1)) * v;
```

如果你想要当 flag 为 true 才求相反数时，你可以这样写：

```
bool fNegate;  // 标记我们是否应该对 v 求相反数
int v;         // 原数字
int r;         // 结果 result = fNegate ? -v : v;

r = (v ^ -fNegate) + fNegate;
```

根据掩码来合并两个数

```
unsigned int a;    // 准备合并的数 a
unsigned int b;    // 准备合并的数 a
unsigned int mask; // 掩码，如果对应位为 0，则取 a 的值，否则取 b 的值
unsigned int r;    // 保存结果 (a & ~mask) | (b & mask)

r = a ^ ((a ^ b) & mask);
```

上面这个方法比直接合并的策略可以少一个操作符。

计算 **bit** 数字中有多少个 **1**（原始方法）

```
unsigned int v; // 需要计算的数字
unsigned int c; // c 用来保存结果

for (c = 0; v; v >>= 1)
{
  c += v & 1;
}
```

原始的方法计算每一个 bit 时，都需要一次循环，所以对于一个 32 位的数来说，需要 32 次循环。

计算 bit 数字中有多少个 1（查表法）

```
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; // 计算 v 中有多少个 1
unsigned int c; // 结果存储在 c 中

// 方法 1:
c = BitsSetTable256[v & 0xff] +
BitsSetTable256[(v >> 8) & 0xff] +
BitsSetTable256[(v >> 16) & 0xff] +
BitsSetTable256[v >> 24];

// 方法 2:
unsigned char * p = (unsigned char *) &v;
c = BitsSetTable256[p[0]] +
    BitsSetTable256[p[1]] +
    BitsSetTable256[p[2]] +
    BitsSetTable256[p[3]];


// 不使用宏来初始化的话，可以用下面方法来初始化表格
BitsSetTable256[0] = 0;
for (int i = 0; i < 256; i++)
{
  BitsSetTable256[i] = (i & 1) + BitsSetTable256[i / 2];
}
```

计算 bit 数字中有多少个 1 （另外的一种方法）

```
unsigned int v; // 计算 v 中有多少个 1
unsigned int c; // 结果存储在 c 中

for (c = 0; v; c++)
{
  v &= v - 1; // 清除一个最高有效位
}
```

这个方法在 1960 年由 Peter Wegner 第一次提出，而后陆续有其它研究者独立也发现了这个方法。这个方法的循环次数和 v 中有多少 1 有关。

使用 64 位的操作符对14、24、32 位的数字计算 1 的数量

```
unsigned int v; // 计算 v 中有多少个 1
unsigned int c; // 结果存储在 c 中

// 方法 1，处理 14 位数:
c = (v * 0x200040008001ULL & 0x111111111111111ULL) % 0xf;

// 方法 2，处理 24 位数:
c =  ((v & 0xfff) * 0x1001001001001ULL & 0x84210842108421ULL) % 0x1f;
c += (((v & 0xfff000) >> 12) * 0x1001001001001ULL & 0x84210842108421ULL)
     % 0x1f;

// 方法 3，处理 32 位数:
c =  ((v & 0xfff) * 0x1001001001001ULL & 0x84210842108421ULL) % 0x1f;
c += (((v & 0xfff000) >> 12) * 0x1001001001001ULL & 0x84210842108421ULL) %
     0x1f;
     c += ((v >> 24) * 0x1001001001001ULL & 0x84210842108421ULL) % 0x1f;
```

这个方法要求具有快速除法模块的 64 位 CPU。方法 1 只需要 3 个操作，方法 2 需要 10 个，方法 3 需要 15 个。

用并行方法计算 1 的数量

```
unsigned int v; // 计算 v 中有多少个 1
unsigned int c; // 结果存储在 c 中
static const int S[] = {1, 2, 4, 8, 16}; // 魔法二进制数
static const int B[] = {0x55555555, 0x33333333, 0x0F0F0F0F, 0x00FF00FF,
0x0000FFFF};

c = v - ((v >> 1) & B[0]);
c = ((c >> S[1]) & B[1]) + (c & B[1]);
c = ((c >> S[2]) + c) & B[2];
c = ((c >> S[3]) + c) & B[3];
c = ((c >> S[4]) + c) & B[4];
```

用二进制表示 B 数组可以看到：

```
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
```

如果要处理更大的整数，我们可以按规律扩展魔法二进制数组的长度。如果需要计算 k 比特数字，则 S 和 B 的长度要达到 `ceil(lg(k))` ，同时 c 的初始化也要和 S 与 B 的长度对应。对于 32 位数，需要 16 个操作符。如果固定是 32 位数的话，最优方案可以写成：

```
v = v - ((v >> 1) & 0x55555555);
v = (v & 0x33333333) + ((v >> 2) & 0x33333333);
c = ((v + (v >> 4) & 0xF0F0F0F) * 0x1010101) >> 24;
```

对于上面的最优方案中，只需要 12 个操作符，这个数量和查表法是一样的，不过查表法可能会出现缓存缺失的问题，所以这个方案实际上的运行速度是更优的。这个方法结合了并行操作和前面章节介绍的乘法方法（使用 64 位指令），虽然这里没有用 64 位指令。最后计算 1 的和时，是将其乘以 `0x1010101` 后向右移 24 位。当计算位宽为 128 的数据时（这个值可以使用 T 来代替），我们可以这样写这个最优方法：

```
v = v - ((v >> 1) & (T)~(T)0/3);
v = (v & (T)~(T)0/15*3) + ((v >> 2) & (T)~(T)0/15*3);
v = (v + (v >> 4)) & (T)~(T)0/255*15;
c = (T)(v * ((T)~(T)0/255)) >> (sizeof(T) - 1) * CHAR_BIT; // 结果
```

计算从最高有效位到指定位置的 1 的数量

下面的方法会返回指定位的“阶（Rank）”，意思就是从最高有效位到制定位置中 1 的数量。

```
uint64_t v;        // 计算的 bit 对象
unsigned int pos; // 以最高位为原点的相对位置（自右向左）
uint64_t r;        // 储存结果：指定位置的“阶”

// 将指定位置向右移至最低位
r = v >> (sizeof(v) * CHAR_BIT - pos);

// 并行地计算
// r = (r & 0x5555...) + ((r >> 1) & 0x5555...);
r = r - ((r >> 1) & ~0UL/3);

// r = (r & 0x3333...) + ((r >> 2) & 0x3333...);
r = (r & ~0UL/5) + ((r >> 2) & ~0UL/5);
```

```
// r = (r & 0x0f0f...) + ((r >> 4) & 0x0f0f...);
r = (r + (r >> 4)) & ~0UL/17;

// r = r % 255;
r = (r * (~0UL/255)) >> ((sizeof(v) - 1) * CHAR_BIT);
```

从最高有效位开始，找到给定的“阶（Rank）”的位置

下面的算法用于在 64 位的数据中，找到从最高有效位开始计数，向低位寻找指定"阶"的位置。如果给定的“阶”高于整个 64 位的数据所容纳的数据，则返回 64。你也可以把这个算法改成 32 位的版本。

```
uint64_t v;          // 目标数据
unsigned int r;      // 指定的阶，范围为： [1-64].
unsigned int s;      // 找到的位置，范围位： [1-64]
uint64_t a, b, c, d; // 计算的中间结果
unsigned int t;      // 统计位数的中间结果

// 使用之前提到的计算阶的方法，但是这次要保存中间结果

// a = (v & 0x5555...) + ((v >> 1) & 0x5555...);
a =  v - ((v >> 1) & ~0UL/3);

// b = (a & 0x3333...) + ((a >> 2) & 0x3333...);
b = (a & ~0UL/5) + ((a >> 2) & ~0UL/5);

// c = (b & 0x0f0f...) + ((b >> 4) & 0x0f0f...);
c = (b + (b >> 4)) & ~0UL/0x11;

// d = (c & 0x00ff...) + ((c >> 8) & 0x00ff...);
d = (c + (c >> 8)) & ~0UL/0x101;
t = (d >> 32) + (d >> 48);

// 现在开始选择位置了，但是我们不使用分支语句
s  = 64;

// if (r > t) {s -= 32; r -= t;}
s -= ((t - r) & 256) >> 3; r -= (t & ((t - r) >> 8));
t  = (d >> (s - 16)) & 0xff;

// if (r > t) {s -= 16; r -= t;}
s -= ((t - r) & 256) >> 4; r -= (t & ((t - r) >> 8));
t  = (c >> (s - 8)) & 0xf;

// if (r > t) {s -= 8; r -= t;}
s -= ((t - r) & 256) >> 5; r -= (t & ((t - r) >> 8));
t  = (b >> (s - 4)) & 0x7;
```

```
// if (r > t) {s -= 4; r -= t;}
s -= ((t - r) & 256) >> 6; r -= (t & ((t - r) >> 8));
t  = (a >> (s - 2)) & 0x3;

// if (r > t) {s -= 2; r -= t;}
s -= ((t - r) & 256) >> 7; r -= (t & ((t - r) >> 8));
t  = (v >> (s - 1)) & 0x1;

// if (r > t) s--;
s -= ((t - r) & 256) >> 8;
s = 65 - s;
```

如果在你的计算平台上分支不是瓶颈，可以考虑取消上面的 if 语句注释。