Bit Hacks：关于一切位操作的魔法（中）

欢迎关注代码律动 codingwave 知乎、Bilibili、微信公众号

计算奇偶性

由于求余操作的性能代价比较高，如果希望追求性能，可以使用位操作：

```
unsigned int v;         // 需要计算奇偶的数字
bool parity = false;    // 结果

while (v)
{
  parity = !parity;
  v = v & (v - 1);
}
```

这段代码运行时间和输入数字的有效比特位有关，当 parity 为 true 时，表示 v 为奇数。

查表法计算奇偶性

```
static const bool ParityTable256[256] =
{
#   define P2(n) n, n^1, n^1, n
#   define P4(n) P2(n), P2(n^1), P2(n^1), P2(n)
#   define P6(n) P4(n), P4(n^1), P4(n^1), P4(n)
    P6(0), P6(1), P6(1), P6(0)
};

// 方法 1：
unsigned char b;   // 需要计算奇偶性的数字
bool parity = ParityTable256[b];

// 方法 2（32 位数字）：
unsigned int v;
v ^= v >> 16;
v ^= v >> 8;

bool parity = ParityTable256[v & 0xff];
```

```
// 方法 3 (64 位数字):
unsigned char * p = (unsigned char *) &v;
parity = ParityTable256[p[0] ^ p[1] ^ p[2] ^ p[3]];
```

使用 64 位运算符计算奇偶性（仅限 byte 类型）

```
unsigned char b;  // 需要计算的数
bool parity =
  (((b * 0x0101010101010101ULL) & 0x8040201008040201ULL) % 0x1FF) & 1;
```

以上的计算只需要 4 个运算符，但是只对单字节类型有效。

使用乘法计算奇偶性

下面的方法用于计算 32 位数的奇偶性，只需要 8 个运算符：

```
unsigned int v; // 32-bit 数
v ^= v >> 1;
v ^= v >> 2;
v = (v & 0x11111111U) * 0x11111111U;
return (v >> 28) & 1;
```

对于 64 位数，8 个运算符也足够了：

```
unsigned long long v; // 64-bit 数字
v ^= v >> 1;
v ^= v >> 2;
v = (v & 0x1111111111111111UL) * 0x1111111111111111UL;
return (v >> 60) & 1;
```

并行计算奇偶性

```
unsigned int v;  // 需要计算奇偶性的数
v ^= v >> 16;
v ^= v >> 8;
v ^= v >> 4;
v &= 0xf;
return (0x6996 >> v) & 1;
```

上面的方法用于计算 32 位数的奇偶性，需要 9 个操作符，并且对于单字节长度的数据的话，可以进一步优化为 5 个操作符，你需要做的仅仅是删掉 `unsigned int v;` 后面的两行即可。这个方法一步一步地进行右移和异或操作，在最后一步，二进制数 `0110 1001 1001 0110`（使用 16 进制表示为 `0x6996`）根据上面的结果进行右移。这个右移的数有点像是微缩版的 16 位奇偶表，提供对 4 bit 数的奇偶性查询。

只用加减交换两个数字

```
#define SWAP(a, b) ((&(a) == &(b)) || \
                    (((a) -= (b)), ((b) += (a)), ((a) = (b) - (a))))
```

此方法不需要建立临时变量，不需要任何额外的空间。要注意的是实际上真正起作用的是 `||` 操作符的右边部分，但是如果两个数的地址相同时，这个方法会出错，所以有前面的前置判断 `&(a) == &(b)`。

使用异或交换两个数字

```
#define SWAP(a, b) (((a) ^= (b)), ((b) ^= (a)), ((a) ^= (b)))
```

这也是一个很常用的方法，也是不需要额外空间。这个方法在两个数内存地址相同时也会出错，所以如果可能出现这种情况的话，考虑定义宏为 `(((a) == (b)) || (((a) ^= (b)), ((b) ^= (a)), ((a) ^= (b))))`，注意，这个方法还可以写成下面形式，`(((a) ^ (b)) && ((b) ^= (a) ^= (b), (a) ^= (b)))`，因为 `(a) ^ (b)` 表达式可以被复用，所以可能会更快。

交换指定位置与长度的比特序列

```
unsigned int i, j; // 需要交换的两个位置 i, j
unsigned int n;    // 需要交换的长度
unsigned int b;    // 需要交换的序列
unsigned int r;    // 交换结果

unsigned int x = ((b >> i) ^ (b >> j)) & ((1U << n) - 1); // XOR temporary
r = b ^ ((x << i) | (x << j));
```

假如我们需要交换的序列是 b = **0010111**（用二进制表示），从位置 i = 1 开始（由右向左数），交换连续 n = 3 个比特到位置 j = 5，最后得到的结果就是 r = **1110001**。

将比特序列反向

```
unsigned int v;     // 输入的比特序列
unsigned int r = v; // 保存结果
int s = sizeof(v) * CHAR_BIT - 1; // 计算要额外移位的次数

for (v >>= 1; v; v >>= 1)
{
  r <<= 1;
  r |= v & 1;
  s--;
}
r <<= s; // 当最高位为 0 时, r 的结果要额外移位相应次数
```

查表法实现反向比特序列

```
static const unsigned char BitReverseTable256[256] =
{
#   define R2(n)     n,     n + 2*64,     n + 1*64,     n + 3*64
#   define R4(n) R2(n), R2(n + 2*16), R2(n + 1*16), R2(n + 3*16)
#   define R6(n) R4(n), R4(n + 2*4 ), R4(n + 1*4 ), R4(n + 3*4 )
    R6(0), R6(2), R6(1), R6(3)
};

unsigned int v; // 反向一个 32 位数字, 一次转 8 位
unsigned int c; // 结果

// 方法 1:
c = (BitReverseTable256[v & 0xff] << 24) |
    (BitReverseTable256[(v >> 8) & 0xff] << 16) |
    (BitReverseTable256[(v >> 16) & 0xff] << 8) |
```

```
    (BitReverseTable256[(v >> 24) & 0xff]);

// 方法 2:
unsigned char * p = (unsigned char *) &v;
unsigned char * q = (unsigned char *) &c;
q[3] = BitReverseTable256[p[0]];
q[2] = BitReverseTable256[p[1]];
q[1] = BitReverseTable256[p[2]];
q[0] = BitReverseTable256[p[3]];
```

第一种方法需要进行 17 个操作，第二种的操作数更少，只要 12 个操作，但是这是建立在你的 CPU 对单字节存储和读取足够快的基础上的。

只用 3 个操作符反向比特序列（使用 64 位乘法与取模）

```
unsigned char b; // 反向一个 8 位比特序列

b = (b * 0x0202020202ULL & 0x010884422010ULL) % 1023;
```

上面的乘法操作将会在一个 64 位的数字中创建原 8 位序列的 5 个副本。AND 操作符在每个副本正确的位置（反向位置）选取比特位，并按照 10 个一组进行分组。最后一步，通过取 2^10-1 的模，可以使 64 位整数按照每 10 位结合为一组的形式合并 (0-9, 10-19, 20-29, ...)。

只用 4 个操作符反向比特序列（使用 64 位乘法）

```
unsigned char b; // 反向这个序列

b = ((b * 0x80200802ULL) & 0x0884422110ULL) * 0x0101010101ULL >> 32;
```

用 7 个操作符反向比特序列

```
b = ((b * 0x0802LU & 0x22110LU) | (b * 0x8020LU & 0x88440LU)) * 0x10101LU >>
16;
```

记得将你的结果转为 unsigned 类型，以免由于最高位为 1 而被识别成了负数。

使用 5 × lg(N) 个操作符反向 N 个比特

```c
unsigned int v; // 32 bit 数字

// 交换奇数和偶数比特位
v = ((v >> 1) & 0x55555555) | ((v & 0x55555555) << 1);
// 交换连续比特对
v = ((v >> 2) & 0x33333333) | ((v & 0x33333333) << 2);
// 交换 4 个比特位
v = ((v >> 4) & 0x0F0F0F0F) | ((v & 0x0F0F0F0F) << 4);
// 交换单字节
v = ((v >> 8) & 0x00FF00FF) | ((v & 0x00FF00FF) << 8);
// 交换双字节
v = ( v >> 16            ) | ( v               << 16);
```

下面的方法时间复杂度是 O(lg(N))，虽然它需要更多的运算，但是其占用空间更小。

```c
unsigned int s = sizeof(v) * CHAR_BIT; // bit 位数，必须是 2 的幂次
unsigned int mask = ~0;
while ((s >>= 1) > 0)
{
  mask ^= (mask << s);
  v = ((v >> s) & mask) | ((v << s) & ~mask);
}
```

以上方法适合 N 比较大的场景。如果你对 64 位整型（或者更大）使用这个方法，你需要按照以上规律添加更多的处理代码，不然的话只有最低的 32 位会被反向。

不用除法，使用左移实现取余操作

```c
const unsigned int n;          // 被除数
const unsigned int s;
const unsigned int d = 1U << s; // d 的可能值是: 1, 2, 4, 8, 16, 32, ...
unsigned int m;                // m = n % d
m = n & (d - 1);
```

这个操作只对 d 是 2 的幂的情况有效。

不用除法，计算某数对 (1 << s) - 1 的余数

```c
unsigned int n;                      // 被除数
```

```
const unsigned int s;                 // s > 0
const unsigned int d = (1 << s) - 1; // d 的可能值是 1, 3, 7, 15, 31, ...).
unsigned int m;                       // m = n % d

for (m = n; n > d; n = m)
{
  for (m = 0; n; n >>= s)
  {
    m += n & d;
  }
}
// 现在 m 的取值范围是 [0, d]，因为这是一个求模方法，所以当 m == d 时，结果应该是 0
m = m == d ? 0 : m;
```

这个求余方法最多需要 5 + (4 + 5 * ceil(N / s)) * ceil(lg(N / s)) 次操作，其中 N 是要求模的数的比特数。换句话说这个方法的时间复杂度是 O(N * lg(N)) .

计算某数对 (1 << s) - 1 的余数

```
// 以下的代码针对 32 bit 数字

static const unsigned int M[] =
{
  0x00000000, 0x55555555, 0x33333333, 0xc71c71c7,
  0x0f0f0f0f, 0xc1f07c1f, 0x3f03f03f, 0xf01fc07f,
  0x00ff00ff, 0x07fc01ff, 0x3ff003ff, 0xffc007ff,
  0xff000fff, 0xfc001fff, 0xf0003fff, 0xc0007fff,
  0x0000ffff, 0x0001ffff, 0x0003ffff, 0x0007ffff,
  0x000fffff, 0x001fffff, 0x003fffff, 0x007fffff,
  0x00ffffff, 0x01ffffff, 0x03ffffff, 0x07ffffff,
  0x0fffffff, 0x1fffffff, 0x3fffffff, 0x7fffffff
};

static const unsigned int Q[][6] =
{
  { 0,  0,  0,  0,  0,  0}, {16,  8,  4,  2,  1,  1}, {16,  8,  4,  2,  2,
 2},
  {15,  6,  3,  3,  3,  3}, {16,  8,  4,  4,  4,  4}, {15,  5,  5,  5,  5,
 5},
  {12,  6,  6,  6 , 6,  6}, {14,  7,  7,  7,  7,  7}, {16,  8,  8,  8,  8,
 8},
  { 9,  9,  9,  9,  9,  9}, {10, 10, 10, 10, 10, 10}, {11, 11, 11, 11, 11,
11},
  {12, 12, 12, 12, 12, 12}, {13, 13, 13, 13, 13, 13}, {14, 14, 14, 14, 14,
14},
  {15, 15, 15, 15, 15, 15}, {16, 16, 16, 16, 16, 16}, {17, 17, 17, 17, 17,
```

```
17},
  {18, 18, 18, 18, 18, 18}, {19, 19, 19, 19, 19, 19}, {20, 20, 20, 20, 20,
20},
  {21, 21, 21, 21, 21, 21}, {22, 22, 22, 22, 22, 22}, {23, 23, 23, 23, 23,
23},
  {24, 24, 24, 24, 24, 24}, {25, 25, 25, 25, 25, 25}, {26, 26, 26, 26, 26,
26},
  {27, 27, 27, 27, 27, 27}, {28, 28, 28, 28, 28, 28}, {29, 29, 29, 29, 29,
29},
  {30, 30, 30, 30, 30, 30}, {31, 31, 31, 31, 31, 31}
};

static const unsigned int R[][6] =
{
  {0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000},
  {0x0000ffff, 0x000000ff, 0x0000000f, 0x00000003, 0x00000001, 0x00000001},
  {0x0000ffff, 0x000000ff, 0x0000000f, 0x00000003, 0x00000003, 0x00000003},
  {0x00007fff, 0x0000003f, 0x00000007, 0x00000007, 0x00000007, 0x00000007},
  {0x0000ffff, 0x000000ff, 0x0000000f, 0x0000000f, 0x0000000f, 0x0000000f},
  {0x00007fff, 0x0000001f, 0x0000001f, 0x0000001f, 0x0000001f, 0x0000001f},
  {0x00000fff, 0x0000003f, 0x0000003f, 0x0000003f, 0x0000003f, 0x0000003f},
  {0x00003fff, 0x0000007f, 0x0000007f, 0x0000007f, 0x0000007f, 0x0000007f},
  {0x0000ffff, 0x000000ff, 0x000000ff, 0x000000ff, 0x000000ff, 0x000000ff},
  {0x000001ff, 0x000001ff, 0x000001ff, 0x000001ff, 0x000001ff, 0x000001ff},
  {0x000003ff, 0x000003ff, 0x000003ff, 0x000003ff, 0x000003ff, 0x000003ff},
  {0x000007ff, 0x000007ff, 0x000007ff, 0x000007ff, 0x000007ff, 0x000007ff},
  {0x00000fff, 0x00000fff, 0x00000fff, 0x00000fff, 0x00000fff, 0x00000fff},
  {0x00001fff, 0x00001fff, 0x00001fff, 0x00001fff, 0x00001fff, 0x00001fff},
  {0x00003fff, 0x00003fff, 0x00003fff, 0x00003fff, 0x00003fff, 0x00003fff},
  {0x00007fff, 0x00007fff, 0x00007fff, 0x00007fff, 0x00007fff, 0x00007fff},
  {0x0000ffff, 0x0000ffff, 0x0000ffff, 0x0000ffff, 0x0000ffff, 0x0000ffff},
  {0x0001ffff, 0x0001ffff, 0x0001ffff, 0x0001ffff, 0x0001ffff, 0x0001ffff},
  {0x0003ffff, 0x0003ffff, 0x0003ffff, 0x0003ffff, 0x0003ffff, 0x0003ffff},
  {0x0007ffff, 0x0007ffff, 0x0007ffff, 0x0007ffff, 0x0007ffff, 0x0007ffff},
  {0x000fffff, 0x000fffff, 0x000fffff, 0x000fffff, 0x000fffff, 0x000fffff},
  {0x001fffff, 0x001fffff, 0x001fffff, 0x001fffff, 0x001fffff, 0x001fffff},
  {0x003fffff, 0x003fffff, 0x003fffff, 0x003fffff, 0x003fffff, 0x003fffff},
  {0x007fffff, 0x007fffff, 0x007fffff, 0x007fffff, 0x007fffff, 0x007fffff},
  {0x00ffffff, 0x00ffffff, 0x00ffffff, 0x00ffffff, 0x00ffffff, 0x00ffffff},
  {0x01ffffff, 0x01ffffff, 0x01ffffff, 0x01ffffff, 0x01ffffff, 0x01ffffff},
  {0x03ffffff, 0x03ffffff, 0x03ffffff, 0x03ffffff, 0x03ffffff, 0x03ffffff},
  {0x07ffffff, 0x07ffffff, 0x07ffffff, 0x07ffffff, 0x07ffffff, 0x07ffffff},
  {0x0fffffff, 0x0fffffff, 0x0fffffff, 0x0fffffff, 0x0fffffff, 0x0fffffff},
  {0x1fffffff, 0x1fffffff, 0x1fffffff, 0x1fffffff, 0x1fffffff, 0x1fffffff},
  {0x3fffffff, 0x3fffffff, 0x3fffffff, 0x3fffffff, 0x3fffffff, 0x3fffffff},
  {0x7fffffff, 0x7fffffff, 0x7fffffff, 0x7fffffff, 0x7fffffff, 0x7fffffff}
};

unsigned int n;        // 要计算的数字
const unsigned int s; // s > 0
const unsigned int d = (1 << s) - 1; // d 属于 1, 3, 7, 15, 31, ...
```

```
unsigned int m;         // m = n % d

m = (n & M[s]) + ((n >> s) & M[s]);

for (const unsigned int * q = &Q[s][0], * r = &R[s][0]; m > d; q++, r++)
{
  m = (m >> *q) + (m & *r);
}
m = m == d ? 0 : m; // 或者可移植性差一点的方法： m = m & -((signed)(m - d) >> s);
```

上面算法的操作复杂度是 O(lg(N)) N 是数字的位数 (在上面的例子中是 32 位) ，最高达到 12 + 9 * ceil(lg(N)). 如果在编译时知道分母的值（2的n次幂减1的值），那么就不需要上面的表了：只需要取表中相应的值放入循环进行计算即可. 这个方法也可以方便扩展到 64 位。

这种方法通过以(1 << s)为基数并行求和的方式求得结果。 首先所有(1 << s)的值加在之前的数上. 举个例子，想象一下结果写在一张纸上，把这张纸剪成两半，被剪开的两片纸上分别记录了这个值的一部分， 把这两个值排在一起，把他们的和写在一张新的纸上.然后重复这个把这纸剪成两半(得到的是上一次数据宽度的四分之一) 并求和的过程, 重复这个过程直到你无法再剪开这张纸了， 经过 lg(N/s/2) 次的剪纸, 无法再剪了。 当剪下来的值的宽度是s时，把剪出来的两个值相加.

计算一个整数以 2 为底的对数

```
unsigned int v; // 要计算的 32 位数
unsigned int r = 0; // r = log2(v)

while (v >>= 1) // 可以将循环展开以增加效率
{
  r++;
}
```

计算以 2 为低的对数和找最高有效位是一样的。

使用一个 64 位浮点数计算整型数字以 2 为底的对数

```
int v; // 要计算的 32 位整数
int r; // 结果为 log2(v)
union { unsigned int u[2]; double d; } t; // 临时变量

t.u[__FLOAT_WORD_ORDER==LITTLE_ENDIAN] = 0x43300000;
t.u[__FLOAT_WORD_ORDER!=LITTLE_ENDIAN] = v;
t.d -= 4503599627370496.0;
r = (t.u[__FLOAT_WORD_ORDER==LITTLE_ENDIAN] >> 20) - 0x3FF;
```

上面的代码加载 64 位双精度浮点数（IEEE-784），并将一个 32 位整数存在尾数中，指数设置为 2^52。 在构造好新的双精度结构后，减去 2^52 (用双精度浮点表达)，剩下就是将指数右移 20 位，并减去偏差值 0x3FF （十进制表示为 1023）。这个方法用了 5 个操作符，但是 CPU 操作双精度数的代价很大，而且还要考虑平台的大端小端问题。

使用查表法计算一个整数以 2 为底的对数

```
static const char LogTable256[256] =
{
#define LT(n) n, n, n, n, n, n, n, n, n, n, n, n, n, n, n, n
    -1, 0, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3,
    LT(4), LT(5), LT(5), LT(6), LT(6), LT(6), LT(6),
    LT(7), LT(7), LT(7), LT(7), LT(7), LT(7), LT(7), LT(7)
};

unsigned int v; // 要计算的 32 bit 数字
unsigned r;     // 结果 r = lg(v)
register unsigned int t, tt; // 临时变量

if (tt = v >> 16)
{
  r = (t = tt >> 8) ? 24 + LogTable256[t] : 16 + LogTable256[tt];
}
else
{
  r = (t = v >> 8) ? 8 + LogTable256[t] : LogTable256[v];
}
```

上面的查找过程只需要 7 次操作就可以找到 32 位数的对数，如果需要扩展到 64 位，则需要 9 次操作。其它的操作可以用 4 个表来代替，使用整数的 table 可能更快，当然这取决于你的 CPU 架构。

以上代码针对的正太分布的数据，而对于平均分布的 32 位输入来说，下面的代码效果会更好：

```
if (tt = v >> 24)
{
  r = 24 + LogTable256[tt];
```

```
}
else if (tt = v >> 16)
{
  r = 16 + LogTable256[tt];
}
else if (tt = v >> 8)
{
  r = 8 + LogTable256[tt];
}
else
{
  r = LogTable256[v];
}
```

如果要用算法生成 LogTable256，可以这么做：

```
LogTable256[0] = LogTable256[1] = 0;
for (int i = 2; i < 256; i++)
{
  LogTable256[i] = 1 + LogTable256[i / 2];
}
LogTable256[0] = -1; // 0 是不存在的，所以我们用 -1 来表示出错了
```

计算N比特整数以 2 为底的对数（O(lg(N)) 复杂度）

```
unsigned int v;  // 需要计算 log2 的 32 bit 数
const unsigned int b[] = {0x2, 0xC, 0xF0, 0xFF00, 0xFFFF0000};
const unsigned int S[] = {1, 2, 4, 8, 16};
int i;

register unsigned int r = 0; // log2 的结果
for (i = 4; i >= 0; i--) // 可以展开以提升性能
{
  if (v & b[i])
  {
    v >>= S[i];
    r |= S[i];
  }
}


// 如果你的 CPU 分支语句比较慢的话：

unsigned int v;          // 需要计算 log2 的 32 bit 数

register unsigned int r; // 结果
```

```
register unsigned int shift;

r =     (v > 0xFFFF) << 4; v >>= r;
shift = (v > 0xFF  ) << 3; v >>= shift; r |= shift;
shift = (v > 0xF   ) << 2; v >>= shift; r |= shift;
shift = (v > 0x3   ) << 1; v >>= shift; r |= shift;
                                        r |= (v >> 1);


// 或者你知道 v 是 2 的幂：

unsigned int v;  // 需要计算 log2 的 32 bit 数
static const unsigned int b[] = {0xAAAAAAAA, 0xCCCCCCCC, 0xF0F0F0F0,
                                 0xFF00FF00, 0xFFFF0000};
register unsigned int r = (v & b[0]) != 0;
for (i = 4; i > 0; i--) // 可以展开以提升性能
{
  r |= ((v & b[i]) != 0) << i;
}
```

如果我们需要计算 33 到 64 bit 的数，可以再添加一个元素 `0xFFFFFFFF00000000` 到 b 后，添加 32 到 S，并从 5 循环到 0。第一个方法稍微有一点慢，但是如果你不想让内存放置一大片区域用于查找表，这个方法就很值得一试。第二种方法需要略多一点的操作，但是在一些分支操作很慢的平台上很有用 (比如：PowerPC)。

用乘法与查表法来计算 N 比特整数以 2 为底的对数（O(lg(N)) 复杂度）

```
uint32_t v; // 需要计算的数字
int r;      // 结果

static const int MultiplyDeBruijnBitPosition[32] =
{
  0, 9, 1, 10, 13, 21, 2, 29, 11, 14, 16, 18, 22, 25, 3, 30,
  8, 12, 20, 28, 15, 17, 24, 7, 19, 27, 23, 6, 26, 5, 4, 31
};

v |= v >> 1;
v |= v >> 2;
v |= v >> 4;
v |= v >> 8;
v |= v >> 16;

r = MultiplyDeBruijnBitPosition[(uint32_t)(v * 0x07C4ACDDU) >> 27];
```

上面的代码用一个小的表查找与乘法来计算 32 位的整型数以 2 为底的对数，总共需要 13 个操作符，vi只需要 13 个操作符。

如果你已经知道 v 是 2 的幂，你可以这样写：

```
static const int MultiplyDeBruijnBitPosition2[32] =
{
  0, 1, 28, 2, 29, 14, 24, 3, 30, 22, 20, 15, 25, 17, 4, 8,
  31, 27, 13, 23, 21, 19, 16, 7, 26, 12, 18, 6, 11, 5, 10, 9
};
r = MultiplyDeBruijnBitPosition2[(uint32_t)(v * 0x077CB531U) >> 27];
```

计算一个整数以 10 为底的对数

```
unsigned int v; // 非 0 的 32 位输入
int r;          // 结果
int t;          // 临时变量

static unsigned int const PowersOf10[] =
    {1, 10, 100, 1000, 10000, 100000,
     1000000, 10000000, 100000000, 1000000000};

t = (IntegerLogBase2(v) + 1) * 1233 >> 12; // 使用前面提到的 log2 方法
r = t - (v < PowersOf10[t]);
```

计算以 10 为底的对数前，先要计算好以 2 为底的对数，通过等式 log10(v) = log2(v) / log2(10)，我们只要将以 2 为低的对数乘以 1/log2(10)（约等于 1233/4096，或者乘以 1233 并向右移 12 位）。最后，由于 t 只是一个偏离 1 的近似值，精确的值应该是将 t 减去 v < PowersOf10[t]。

这个方法相比计算整型数字的 log2 多用了 6 个操作符，当然这个计算 log2 的过程还可以通过使用查表法加速（这主要针对的是那些内存访问更快的平台）。这样的话计算 log10 只需要总共 9 个操作（假设对于 v 的每个 byte 都用一个表，总共 4 个表）。

计算到一个整数的以 10 为底的对数

```
unsigned int v; // 非 0 的 32 位输入
int r;          // 结果

r = (v >= 1000000000) ? 9 : (v >= 100000000) ? 8 : (v >= 10000000) ? 7 :
    (v >= 1000000) ? 6 : (v >= 100000) ? 5 : (v >= 10000) ? 4 :
    (v >= 1000) ? 3 : (v >= 100) ? 2 : (v >= 10) ? 1 : 0;
```

这个方法对于均匀分布（正太）的输入非常有效，因为 32 位的数字有 76% 会只需要一次比较，21% 的数字有两次比较，2 % 会有三次比较（每一轮降低 90%），所以平均有 2.6 次操作。当然，因为其中比较多的分支条件，运算速度有可能慢于简单的方法。

计算一个浮点数的以 2 为底的对数

```
const float v; // 目标是找到 int(log2(v))，浮点数 v 要大于 0
int c;         // 返回是 32 bit 整型

c = *(const int *) &v;  // 或者： memcpy(&c, &v, sizeof c);
c = (c >> 23) - 127;
```

上面的方法尽管速度很快，但是 IEEE 754-2008 标准定义了非规格化浮点数（subnormal floating point numbers）。这种数有前导0（最小非零小数可以表示到2^-127），为了兼容非规格化的浮点数，可以用下面的方法：

```
const float v;              // 目标是找到 int(log2(v))，浮点数 v 要大于 0
int c;                      // 返回是 32 bit 整型
int x = *(const int *) &v;  // 或者： memcpy(&x, &v, sizeof x);

c = x >> 23;

if (c)
{
  c -= 127;
}
else
{ // 非规格化浮点数: c = intlog2(x) - 149;
  register unsigned int t; // 临时变量
  // 注意 LogTable256 已经被定义了
  if (t = x >> 16)
  {
    c = LogTable256[t] - 133;
  }
  else
  {
    c = (t = x >> 8) ? LogTable256[t] - 141 : LogTable256[x] - 149;
  }
}
```

由于 ISO C99 6.5/7 对于 *(int *)& 的类型转换是一个未定义行为，尽管这个方法在 99.9% 的 C 编译器上是有效的。当然，可以使用 memcpy，或者是包含 float 和 int 的 union 类型。

计算 32 位浮点型数 v 的 int(log2(pow(v, 1. / pow(2, r)))) 的运算结果

```
const int r;
const float v;
int c;          // c 保存结果

c = *(const int *) &v;  // 或者使用： memcpy(&c, &v, sizeof c);
c = ((((c - 0x3f800000) >> r) + 0x3f800000) >> 23) - 127;
```

比如，如果 r 是 0，那么 c 的结果实际上就是 int(log2((double) v))。如果 r 是 1 的话，int(log2(sqrt((double) v)))。