作者：肖恩·埃隆·安德森 seander@cs。斯坦福大学

内容

- 计算奇偶校验（如果设置奇数位则为 1，否则为 0）
  - 以朴素的方式计算一个词的奇偶性
  - 通过查找表计算奇偶校验
  - 使用 64 位乘法和模除计算字节的奇偶校验
  - 用乘法计算单词的奇偶性
  - 并行计算奇偶校验
- 交换值
  - 用减法和加法交换值
  - 使用 XOR 交换值
  - 使用 XOR 交换单个位
- 反转位序列
  - 以明显的方式反转位
  - 通过查找表反转字中的位
  - 通过 3 次操作（64 位乘法和模除法）反转字节中的位
  - 通过 4 次操作（64 位乘法，无除法）反转字节中的位
  - 用 7 次操作反转一个字节中的位（没有 64 位，只有 32 位）
  - 通过 5 * lg(N) 操作并行反转 N 位数量
- 模数除法（又名计算
  余数
  ）
  - 计算模除以 1 << s 而无需除法运算（显而易见）
  - 计算模数除法 (1 << s) - 1 无需除法运算
  - 计算模数除以 (1 << s) - 1 并行，无需除法运算
- 查找整数的以 2 为底的整数对数（即最高位集的位置）
  - 在 O(N) 操作中找到设置为 MSB N 的整数的对数基数 2（显而易见的方法）
  - 查找具有 64 位 IEEE 浮点数的整数的以 2 为底的整数对数
  - 使用查找表查找整数的对数底数 2
  - 在 O(lg(N)) 操作中找到 N 位整数的对数基数 2
  - 使用乘法和查找在 O(lg(N)) 操作中找到 N 位整数的对数基数 2
- 查找整数的以 10 为底的整数对数
- 以显而易见的方式查找整数的以 10 为底的整数对数
- 查找 32 位 IEEE 浮点数的以 2 为底的整数对数
- 查找 32 位 IEEE 浮点数的 pow(2, r)-root 的整数对数基数 2（对于无符号整数 r）
- 计算连续的尾随零位（或查找位索引）
  - 线性计算右侧的连续零位（尾随）
  - 并行计算右侧连续的零位（尾随）
  - 通过二进制搜索计算右侧连续的零位（尾随）
  - 通过转换为浮点数来计算右侧的连续零位（尾随）

关于操作计数方法

在这里合计算法的操作数时，任何 C 运算符都算作一个操作。不需要写入 RAM 的中间赋值不计算在内。当然，这种操作计数方法只能作为实际机器指令数和 CPU 时间的近似值。假定所有操作都花费相同的时间，这在现实中并非如此，但随着时间的推移，CPU 越来越朝着这个方向发展。有许多细微差别决定系统运行给定代码示例的速度，例如缓存大小、内存带宽、指令集等。最后，基准测试是确定一种方法是否真的比另一种方法更快的最佳方法，因此请考虑将以下技术作为在目标架构上进行测试的可能性。

计算整数的符号

```
诠释五; // 我们想找到 v 的符号

符号; // 结果在这里
```

```
// CHAR_BIT 是每字节的位数（通常为 8）。

符号 = –(v < 0)；// 如果 v < 0 则 –1，否则 0。

// 或者，为了避免在带有标志寄存器 (IA32) 的 CPU 上发生分支：

sign = –(int)((unsigned int)((int)v) >> (sizeof(int) * CHAR_BIT – 1));

// 或者，少一条指令（但不可移植）：

sign = v >> (sizeof(int) * CHAR_BIT – 1);
```

对于 32 位整数，上面最后一个表达式的计算结果为 sign = v >> 31。这是一种比显而易见的方法 sign = -(v < 0) 更快的操作。这个技巧之所以有效，是因为当有符号整数右移时，最左边位的值被复制到其他位。当值为负时，最左边的位为 1，否则为 0；所有 1 位给出 -1。不幸的是，这种行为是特定于体系结构的。或者，如果您希望结果为 -1 或 +1，则使用：

```
符号 = +1 | (v >> (sizeof(int) * CHAR_BIT – 1)); // 如果 v < 0 则 –1，否则 +1
```

另一方面，如果您希望结果为 -1、0 或 +1，则使用：

```
符号 = (v != 0) | –(int)((unsigned int)((int)v) >> (sizeof(int) * CHAR_BIT – 1));

// 或者，为了更快的速度但更少的可移植性：

符号 = (v != 0) | (v >> (sizeof(int) * CHAR_BIT – 1)); // –1、0 或 +1

// 或者，为了便携性、简洁性和（可能）速度：

符号 = (v > 0) – (v < 0); // –1、0 或 +1
```

相反，如果您想知道某些东西是非负的，结果是 +1 还是 0，那么使用：

```
sign = 1 ^ ((unsigned int)v >> (sizeof(int) * CHAR_BIT – 1)); // 如果 v < 0 则 0，否则 1
```

警告：2003 年 3 月 7 日，Angus Duggan 指出 1989 年的 ANSI C 规范保留了已定义的签名右移实现的结果，因此在某些系统上此 hack 可能不起作用。为了更好的可移植性，Toby Speight 在 2005 年 9 月 28 日建议在这里和整个过程

中使用 CHAR_BIT 而不是假设字节是 8 位长。Angus 在 2006 年 3 月 4 日推荐了上述更便携的版本，涉及转换 。Rohit Garg 在 2009 年 9 月 12 日推荐了非负整数版本。

检测两个整数是否符号相反

```
整数 x, y; // 比较符号的输入值
```

```
布尔 f = ((x ^ y) < 0); // 真当且仅当 x 和 y 的符号相反
```

Manfred Weis 建议我在 2009 年 11 月 26 日添加此条目。

计算无分支的整数绝对值 (abs)

```
诠释五; // 我们想要找到 v 的绝对值
```

```
无符号整数 r; // 结果在这里
```

```
int const mask = v >> sizeof(int) * CHAR_BIT - 1;
```

```
r = (v + 面具) ^ 面具;
```

专利变体：

```
r = (v ^ 掩码) - 掩码;
```

某些 CPU 没有整数绝对值指令（或者编译器无法使用它们）。在分支成本很高的机器上，上面的表达式比显而易见的方法 r = (v < 0) ? -(unsigned)v : v，即使操作次数相同。
2003 年 3 月 7 日，Angus Duggan 指出 1989 年的 ANSI C 规范保留了已定义的带符号右移实现的结果，因此在某些系统上这种 hack 可能不起作用。我读过

ANSI C 不要求将值表示为二进制补码，因此它也可能因为这个原因而不起作用（在数量越来越少的旧机器上仍然使用一个补码）。2004 年 3 月 14 日，Keith H. Duggar 向我发送了上面的专利变体；它优于我最初想出的那个 `r=(+1|(v>>(sizeof(int)*CHAR_BIT-1)))*v`，因为没有使用乘法。不幸的是，这种方法已于 2000 年 6 月 6 日由 Vladimir Yu Volkonsky 在美国 获得专利并转让给 Sun Microsystems. 2006 年 8 月 13 日，尤里·卡明斯基 (Yuriy Kaminskiy) 告诉我，该专利可能无效，因为该方法甚至在专利申请之前就已发布，例如 Agner Fog 在 1996 年 11 月 9 日发表的如何优化奔腾处理器。尤里还提到，这份文件在 1997 年被翻译成俄文，弗拉基米尔本可以阅读的。此外，Internet Archive 也有一个旧 链接。2007 年 1 月 30 日，Peter Kankowski 与我分享了一个 abs 版本他发现这是受微软 Visual C++ 编译器输出的启发。它在这里作为主要解决方案。2007 年 12 月 6 日，海进抱怨结果是签了字，所以在计算最负值的 abs 时，还是负数。2008 年 4 月 15 日，安德鲁·夏皮拉 (Andrew Shapira) 指出，明显的方法可能会溢出，因为它当时缺少（未签名的）演员表；为了获得最大的便携性，他建议 `(v < 0) ? (1 + ((unsigned)(-1-v))) : (unsigned)v`. 但是 Vincent Lefèvre 在 2008 年 7 月 9 日引用 ISO C99 规范说服我删除它，因为即使在非 2s 补码机器上 -(unsigned)v 也会做正确的事情。-(unsigned)v 的计算首先通过添加 2N **将 v 的负值转换为无符号，产生 v 值的 2s 补码表示，我将其称为 U。然后，U 被取反，给出所需的结果，-U = 0 - U = 2**N - U = 2N **- (v+2**N) = -v = abs(v)。

计算两个整数的最小值 (min) 或最大值 (max)，无需分支

```
诠释 x; // 我们想找到 x 和 y 的最小值

诠释 y;

诠释; // 结果在这里


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

在一些罕见的机器上，分支非常昂贵并且不存在条件移动指令，上面的表达式可能比明显的方法更快，`r = (x < y) ? x : y`，即使它涉及另外两条指令。（不过，通

常情况下，显而易见的方法是最好的。）之所以有效，是因为如果 x < y，则 -(x < y) 将全部为 1，因此 r = y ^ (x ^ y) & ~0 = y ^ x ^ y = x。否则，如果 x >= y，则 -(x < y) 将全为零，因此 r = y ^ ((x ^ y) & 0) = y。在某些机器上，将 (x < y) 计算为 0 或 1 需要分支指令，因此可能没有优势。
要找到最大值，请使用：

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

快速和肮脏的版本：

如果你知道 INT_MIN <= x - y <= INT_MAX，那么你可以使用下面的，因为 (x - y) 只需要计算一次，所以速度更快。

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

```
r = x - ((x - y) & ((x - y) >> (sizeof(int) * CHAR_BIT - 1))); // 最大值(x, y)
```

请注意，1989 年的 ANSI C 规范没有指定带符号右移的结果，因此它们不可移植。如果在溢出时抛出异常，则 x 和 y 的值应该是无符号的或转换为无符号的以进行减法以避免不必要地抛出异常，但是右移需要一个带符号的操作数以在负数时产生所有位，因此转换在那里签名。
2003 年 3 月 7 日，Angus Duggan 指出了右移可移植性问题。2005 年 5 月 3 日，Randal E. Bryant 提醒我需要先决条件，INT_MIN <= x - y <= INT_MAX，并建议使用非快速和脏版本作为修复。这两个问题都只涉及快速和肮脏的版本。
Nigel Horspoon 于 2005 年 7 月 6 日观察到，gcc 在 Pentium 上生成的代码与明显的解决方案相同，因为它的计算方式 (x < y)。2008 年 7 月 9 日，Vincent Lef è vre 指出了 r = y + ((x - y) & -(x < y)) 中减法的溢出异常的可能性，这是以前的版本。Timothy B. Terriberry 在 2009 年 6 月 2 日建议使用 xor 而不是 add 和 subract 来避免转换和溢出的风险。

判断一个整数是否是 2 的幂

```
无符号整型 v; // 我们想看看 v 是否是 2 的幂
```

```
布尔 f; // 结果在这里
```

```
f = (v & (v – 1)) == 0;
```

请注意，此处 0 被错误地视为 2 的幂。要解决此问题，请使用：

```
f = v && !(v & (v – 1));
```

从恒定位宽扩展的符号

符号扩展对于内置类型是自动的，例如 chars 和 ints。但是假设您有一个带符号的二进制补码数 x，它仅使用 b 位存储。此外，假设你想将 x 转换为一个 int，它有超过 b 位。如果 x 为正，一个简单的副本就可以工作，但如果为负，则必须扩展符号。例如，如果我们只有 4 位来存储一个数，那么 -3 在二进制中表示为 1101。如果我们有 8 位，那么 -3 就是 11111101。当我们转换为具有更多位的表示时，4 位表示的最高有效位被向左复制以填充目的地；这是符号扩展。在 C 中，从常量位宽进行符号扩展是微不足道的，因为可以在结构或联合中指定位域。例如，要从 5 位转换为完整整数：

```
诠释 x; // 将其从使用 5 位转换为完整的 int

诠释; // 结果符号扩展数放在这里

结构 {signed int x:5;} s;

r = sx = x;
```

下面是一个 C++ 模板函数，它使用相同的语言特性在一次操作中从 B 位转换（当然，编译器会生成更多位）。

```
template <typename T, unsigned B>

内联 T signextend(const T x)

{

  结构 {T x:B;} s;
```

```
 返回 sx = x;

}


int r = signextend<signed int,5>(x); // 符号扩展 5 位数 x 到 r
```

John Byrd 于 2005 年 5 月 2 日在代码中发现了一个拼写错误（归因于 html 格式）。2006 年 3 月 4 日，Pat Wood 指出 ANSI C 标准要求位域具有要签名的关键字“signed”；否则，符号未定义。

从可变位宽扩展的符号

有时我们需要扩展一个数字的符号，但我们事先不知道表示它的位数 b。（或者我们可以使用 Java 等缺少位域的语言进行编程。）

```
无符号 b; // 表示 x 中数字的位数

诠释 x; // 将这个 b 位数符号扩展到 r

诠释; // 结果符号扩展数

int const m = 1U << (b – 1); // 如果 b 固定，掩码可以预先计算


x = x & ((1U << b) – 1); //（如果位置 b 上方的 x 中的位已经为零，则跳过此操作。）

r = (x^m) – m;
```

上面的代码需要四次操作，但是当位宽是常量而不是变量时，它只需要两次快速操作，假设高位已经为零。
一种不依赖于位置 b 上方 x 中的位为零的稍快但不太便携的方法是：

```
int const m = CHAR_BIT * sizeof(x) – b;

r = (x << m) >> m;
```

Sean A. Irvine 建议我在 2004 年 6 月 13 日向此页面添加符号扩展方法，他提供 m = (1 << (b – 1)) – 1; r = –(x & ~m) | x; 了一个起点，我从中优化得到 m = 1U << (b - 1); r = -(x & m) | X。但是后来在 2007 年 5 月 11 日，Shay Green 建议使用上面的版本，它比我的需要的操作少一次。Vipin Sharma 建议我添加一个步骤来处理 x 在 2008 年 10 月 15 日我们想要符号扩展的 b 位以外的位中有可能的情况。2009 年 12 月 31 日，Chris Pirazzi 建议我添加更快的版本，对于恒定位宽需要两个操作，对于可变宽度需要三个操作。

在 3 个操作中从可变位宽扩展符号

由于乘法和除法所需的努力，在某些机器上以下可能会很慢。这个版本是 4 个操作。如果您知道您的初始位宽 b 大于 1，您可以使用 r = (x * multipliers[b]) / multipliers[b] 在 3 个操作中执行这种类型的符号扩展，这只需要一个数组查找。

```
无符号 b; // 表示 x 中数字的位数

诠释 x; // 将这个 b 位数符号扩展到 r

诠释; // 结果符号扩展数

#define M(B) (1U << ((sizeof(x) * CHAR_BIT) – B)) // CHAR_BIT=位/字节

static int const 乘数[] =

{

 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),

 男(16)、男(17)、男(18)、男(19)、男(20)、男(21)、男(22)、男(23)、

 男(24), 男(25), 男(26), 男(27), 男(28), 男(29), 男(30), 男(31),

 男(32)

}; // (如果使用超过 64 位则添加更多)
```

```
static int 常量除数[] =

{

 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),

 男(16)、男(17)、男(18)、男(19)、男(20)、男(21)、男(22)、男(23)、

 男(24), 男(25), 男(26), 男(27), 男(28), 男(29), 男(30), 男(31),

 男(32)

}; //（为 64 位添加更多）

#undef M

r = (x * 乘数[b]) / 除数[b];
```

以下变体不可移植，但在采用算术右移并保持符号的体系结构上，它应该很快。

```
常量 s = –b; // 或：sizeof(x) * CHAR_BIT – b;

r = (x << s) >> s;
```

Randal E. Bryant 于 2005 年 5 月 3 日指出了早期版本中的一个错误（使用 multipliers[] 作为 divisors[]），它在 x=1 和 b=1 的情况下失败。

有条件地设置或清除位而不分支

```
布尔 f；// 条件标志

无符号整型；// 位掩码

无符号整数 w；// 要修改的词：if (f) w |= m; 否则 w &= ~m;



w ^= (–f ^ w) & m;
```

```
// 或者，对于超标量 CPU：

w = (w & ~m) | (–f & m);
```

在某些架构上，缺少分支可以弥补看似两倍的操作。例如，对 AMD Athlon™ XP 2100+ 进行的非正式速度测试表明它的速度提高了 5-10%。Intel Core 2 Duo 运行超标量版本比第一个版本快 16%。Glenn Slayden 在 2003 年 12 月 11 日告诉我第一个表达式。Marco Yu 在 2007 年 4 月 3 日与我分享了超标量版本，并在 2 天后提醒我注意打字错误。

在没有分支的情况下有条件地否定一个值

如果您只需要在标志为 false 时取反，则使用以下代码来避免分支：

```
布尔 fDontNegate；// 标志表明我们不应该否定 v。

诠释五; // 如果 fDontNegate 为假，输入值取反。

诠释; // 结果 = fDontNegate ? v : –v;


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

如果你只需要在标志为真时取反，那么使用这个：

```
布尔 f 否定；// 指示我们是否应该否定 v 的标志。

诠释五; // 如果 fNegate 为真，输入值取反。

诠释; // 结果 = fNegate ? –v：v；


r = (v ^ –fNegate) + fNegate;
```

Avraham Plotnitzky 建议我在 2009 年 6 月 2 日添加第一个版本。为了避免倍增，我在 2009 年 6 月 8 日提出了第二个版本。Alfonso De Gregorio 指出在 2009 年 11 月 26 日缺少一些括号，并且收到错误赏金。

根据掩码合并来自两个值的位

```
无符号整型；// 要合并到非屏蔽位中的值

无符号整数 b；// 要合并到屏蔽位中的值

无符号整数掩码；// 1 应该选择 b 中的位；0 从哪里来

无符号整数 r；// (a & ~mask) | 的结果 (b & mask) 在这里


r = a ^ ((a ^ b) & 面具);
```

这从根据位掩码组合两组位的明显方式中减少了一个操作。如果掩码是常数，则可能没有优势。
2006 年 2 月 9 日，罗恩·杰弗瑞 (Ron Jeffery) 将此寄给我。

计数位集（天真的方式）

```
无符号整型 v；// 计算 v 中设置的位数

无符号整数 c；// c 累加 v 中设置的总位数


对于（c = 0；v；v >>= 1）

{
```

```
  c += v & 1;

}
```

朴素的方法要求每位迭代一次，直到没有更多的位被设置。因此，对于只有高位集的 32 位字，它将经历 32 次迭代。

计算查找表设置的位

```
static const unsigned char BitsSetTable256[256] =

{

# 定义 B2(n) n, n+1, n+1, n+2

# 定义 B4(n) B2(n), B2(n+1), B2(n+1), B2(n+2)

# 定义 B6(n) B4(n), B4(n+1), B4(n+1), B4(n+2)

    B6(0), B6(1), B6(1), B6(2)

};



无符号整型 v；// 计算 32 位值 v 中设置的位数

无符号整数 c；// c 是 v 中设置的总位数



// 选项 1：

c = BitsSetTable256[v & 0xff] +

    BitsSetTable256[(v >> 8) & 0xff] +

    BitsSetTable256[(v >> 16) & 0xff] +

    BitsSetTable256[v >> 24];
```

```
// 选项 2：

无符号字符 * p = (无符号字符 *) &v;

c = BitsSetTable256[p[0]] +

   BitsSetTable256[p[1]] +

   BitsSetTable256[p[2]] +

   BitsSetTable256[p[3]];




// 最初通过算法生成表：

BitsSetTable256[0] = 0;

对于 (int i = 0; i < 256; i++)

{

  BitsSetTable256[i] = (i & 1) + BitsSetTable256[i / 2];

}
```

2009 年 7 月 14 日，Hallvard Furuseth 建议使用宏压缩表。

计数位集，Brian Kernighan 的方式

```
无符号整型 v；// 计算 v 中设置的位数

无符号整数 c；// c 累加 v 中设置的总位数

对于（c = 0；v；c++）
```

```
{
 v &= v - 1; // 清除最低有效位集
}
```

Brian Kernighan 的方法经历了与设置位一样多的迭代。所以如果我们有一个只设置了高位的 32 位字，那么它只会通过循环一次。
1988 年出版的 C Programming Language 第 2 版。(作者：Brian W. Kernighan 和 Dennis M. Ritchie)在练习 2-9 中提到了这一点。2006 年 4 月 19 日，Don Knuth 向我指出该方法“最初由 Peter Wegner 在 CACM 3 (1960), 322 中发表。 (也由 Derrick Lehmer 独立发现并于 1964 年在 Beckenbach 编辑的一本书中发表。)”

使用 64 位指令计算 14、24 或 32 位字中的位集

```
无符号整型 v；// 计算 v 中设置的位数
无符号整数 c；// c 累加 v 中设置的总位数

// 选项 1，v 中最多 14 位值：
c = (v * 0x200040008001ULL & 0x111111111111111ULL) % 0xf;

// 选项 2，v 中最多 24 位值：
c = ((v & 0xfff) * 0x1001001001001ULL & 0x84210842108421ULL) % 0x1f;
c += (((v & 0xfff000) >> 12) * 0x1001001001001ULL & 0x84210842108421ULL)
    % 0x1f;

// 选项 3，v 中最多 32 位值：
```

```
c = ((v & 0xfff) * 0x1001001001001ULL & 0x84210842108421ULL) % 0x1f;

c += (((v & 0xfff000) >> 12) * 0x1001001001001ULL & 0x84210842108421ULL) %
    0x1f;

c += ((v >> 24) * 0x1001001001001ULL & 0x84210842108421ULL) % 0x1f;
```

此方法需要具有快速模除法的 64 位 CPU 才能有效。第一个选项只需要 3 个操作；第二个选项取 10；第三个选项需要 15。
Rich Schroeppel 最初创建了一个 9 位版本，类似于选项 1；请参阅 Beeler, M.、Gosper, RW 和 Schroeppel, R. HAKMEM 的编程技巧部分。麻省理工学院 AI 备忘录 239，1972 年 2 月 29 日。 他的方法是上述变体的灵感来源，由 Sean Anderson 设计。Randal E. Bryant 在 2005 年 5 月 3 日提供了几个错误修复。Bruce Dawson 在 2007 年 2 月 1 日使用相同数量的操作调整了 12 位版本并使其适用于 14 位。

并行计数位集

```
无符号整型 v；// 计数在此设置的位（32 位值）

无符号整数 c；// 在这里存储总数

静态常数 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];
```

B 数组，用二进制表示为：

```
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
```

*我们可以通过继续使用二进制幻数*B 和 S 的模式来调整更大整数大小的方法。如果有 k 位，那么我们需要数组 S 和 B 的长度为 ceil(lg(k)) 个元素，并且我们必须为 c 计算与 S 或 B 相同数量的表达式。对于 32 位 v，使用了 16 个操作。

计算 32 位整数 v 中位数的最佳方法如下：

```
v = v – ((v >> 1) & 0x55555555); // 临时重用输入
```

```
v = (v & 0x33333333) + ((v >> 2) & 0x33333333); // 温度
```

```
c = ((v + (v >> 4) & 0xF0F0F0F) * 0x1010101) >> 24；// 数数
```

最好的位计数方法只需要 12 个操作，这与查找表方法相同，但避免了表的内存和潜在的缓存未命中。它是上述纯并行方法和早期使用乘法的方法（在使用 64 位指令计算位的部分中）的混合体，尽管它不使用 64 位指令。字节中设置的位计数是并行完成的，字节中设置的位的总和是通过乘以 0x1010101 并右移 24 位来计算的。

将最佳位计数方法概括为位宽最大为 128 的整数(由类型 T 参数化)是这样的：

```
v = v – ((v >> 1) & (T)~(T)0/3); // 温度
```

```
v = (v & (T)~(T)0/15*3) + ((v >> 2) & (T)~(T)0/15*3); // 温度
```

```
v = (v + (v >> 4)) & (T)~(T)0/255*15; // 温度
```

```
c = (T)(v * ((T)~(T)0/255)) >> (sizeof(T) – 1) * CHAR_BIT；// 数数
```

有关计算位数集（也称为*横向加法*）的更多信息， 请参阅 Ian Ashdown 的精彩新闻组帖子。2005 年 10 月 5 日，Andrew Shapira 引起了我的注意；他在

Software Optimization Guide for AMD Athlon™ 64 and Opteron™ Processors 的第 187-188 页找到了它. Charlie Gordon 在 2005 年 12 月 14 日提出了一种从纯并行版本中削减一个操作的方法，而 Don Clugston 在 2005 年 12 月 30 日又从中削减了三个操作。我对 Eric Cole 在 1 月 8 日发现的 Don 的建议打了个错字， 2006 年。Eric 后来在 2006 年 11 月 17 日建议将任意位宽泛化为最佳方法。2007 年 4 月 5 日，Al Williams 观察到我在第一个方法的顶部有一行死代码。

从最高有效位到给定位置计算位集（等级）

下面找到一个位的秩，这意味着它返回从最高有效位到给定位置的位设置为 1 的位的总和。

```
uint64_t v；// 计算 v 中从 MSB 到 pos 的排名（位集）。

无符号整型；// 计数位的位位置。

uint64_t r；// pos 位的结果排名在这里。



// 在给定位置后移出位。

r = v >> (sizeof(v) * CHAR_BIT – pos);

// 并行计算集合位。

// 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);
```

Juha Järvi 于 2009 年 11 月 21 日将其发送给我，作为计算具有给定秩的位位置的逆运算，如下所示。

选择具有给定计数（等级）的位位置（从最高有效位开始）

下面的 64 位代码选择从左数时第 r 个 1 位的位置。换句话说，如果我们从最高有效位开始并向右前进，计算设置为 1 的位数，直到达到所需的等级 r，然后返回我们停止的位置。如果请求的等级超过了设置的位数，则返回 64。代码可能会修改为 32 位或从右边数。

```
uint64_t v；// 输入值以查找等级为 r 的位置。

无符号整数 r；// 输入：位的期望等级 [1–64]。

无符号整数；// 输出：排名为 r [1–64] 的位的结果位置

uint64_t a, b, c, d；// 用于位计数的中间临时对象。

无符号整数 t；// 临时位计数。



// 对 64 位整数进行正常的并行位计数，

// 但存储所有中间步骤。

// 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);

// 现在进行无分支选择!

小号 = 64;

// 如果 (r > t) {s -= 32; r -= t;}

s -= ((t - r) & 256) >> 3; r -= (t & ((t - r) >> 8));

t = (d >> (s - 16)) & 0xff;

// 如果 (r > t) {s -= 16; r -= t;}

s -= ((t - r) & 256) >> 4; r -= (t & ((t - r) >> 8));

t = (c >> (s - 8)) & 0xf;

// 如果 (r > t) {s -= 8; r -= t;}

s -= ((t - r) & 256) >> 5; r -= (t & ((t - r) >> 8));

t = (b >> (s - 4)) & 0x7;

// 如果 (r > t) {s -= 4; r -= t;}

s -= ((t - r) & 256) >> 6; r -= (t & ((t - r) >> 8));

t = (a >> (s - 2)) & 0x3;

// 如果 (r > t) {s -= 2; r -= t;}

s -= ((t - r) & 256) >> 7; r -= (t & ((t - r) >> 8));

t = (v >> (s - 1)) & 0x1;

// 如果 (r > t) s--;
```

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

 s = 65 - s;
```

如果您的目标 CPU 上的分支速度很快，请考虑取消注释 if 语句并注释掉它们后面的行。

Juha Järvi 于 2009 年 11 月 21 日将此发送给我。

计算奇偶校验的天真方法

```
无符号整型 v；// 计算奇偶校验的字值

布尔奇偶校验=假；//奇偶校验将是 v 的奇偶校验


而（五）

{

 奇偶校验=! 奇偶校验；

 v = v & (v - 1);

}
```

上面的代码使用了一种类似于上面 Brian Kernigan 的位计数的方法。花费的时间与设置的位数成正比。

通过查找表计算奇偶校验

```
static const bool ParityTable256[256] =

{

# 定义 P2(n) n, n^1, n^1, n
```

```
# 定义 P4(n) P2(n), P2(n^1), P2(n^1), P2(n)

# 定义 P6(n) P4(n), P4(n^1), P4(n^1), P4(n)

   P6(0), P6(1), P6(1), P6(0)

};



无符号字符 b；// 计算奇偶校验的字节值

bool parity = ParityTable256[b];



// 或者，对于 32 位字：

无符号整型 v；

v ^= v >> 16；

v ^= v >> 8；

bool parity = ParityTable256[v & 0xff];



// 变化：

无符号字符 * p = (无符号字符 *) &v;

parity = ParityTable256[p[0] ^ p[1] ^ p[2] ^ p[3]];
```

Randal E. Bryant 在 2005 年 5 月 3 日鼓励使用变量 p 添加（公认的）明显的最后一个变体。Bruce Rawles 于 2005 年 9 月 27 日在表变量名称的实例中发现了一个拼写错误，他获得了 10 美元的错误赏金。2006 年 10 月 9 日，Fabrice Bellard 提出了上面的 32 位变体，它只需要一次查表；以前的版本有四次查找（每个字节一次）并且速度较慢。2009 年 7 月 14 日，Hallvard Furuseth 建议使用宏压缩表。

使用 64 位乘法和模除计算字节的奇偶校验

```
无符号字符 b；// 计算奇偶校验的字节值

布尔奇偶校验 =

 (((b * 0x0101010101010101ULL) & 0x8040201008040201ULL) % 0x1FF) & 1;
```

上面的方法需要大约 4 个操作，但只适用于字节。

用乘法计算单词的奇偶性

以下方法使用乘法仅用 8 次运算就计算了 32 位值的奇偶校验。

```
  无符号整型 v；// 32 位字

  v ^= v >> 1;

  v ^= v >> 2；

  v = (v & 0x11111111U) * 0x11111111U;

  返回 (v >> 28) & 1;
```

同样对于 64 位，8 个操作仍然足够。

```
  unsigned long long v; // 64 位字

  v ^= v >> 1;

  v ^= v >> 2；

  v = (v & 0x1111111111111111UL) * 0x1111111111111111UL;

  返回 (v >> 60) & 1;
```

Andrew Shapira 想出了这个，并于 2007 年 9 月 2 日寄给了我。

并行计算奇偶校验

```
无符号整型 v; // 计算奇偶校验的字值

v ^= v >> 16;

v ^= v >> 8;

v ^= v >> 4;

v &= 0xf;

返回 (0x6996 >> v) & 1;
```

上面的方法大约需要 9 次操作，适用于 32 位字。通过删除紧跟在“unsigned int v;”之后的两行，它可以被优化为仅在 5 个操作中处理字节。该方法首先将 32 位值的八个半字节移位和异或在一起，将结果留在 v 的最低半字节。接下来，将二进制数 0110 1001 1001 0110（十六进制为 0x6996）右移表示的值在 v 的最低半字节中。这个数字就像一个微型 16 位奇偶校验表，由 v 中的低四位索引。结果在位 1 中具有 v 的奇偶校验，它被屏蔽并返回。
感谢 Mathew Hendry 在 2002 年 12 月 15 日最后指出移位查找的想法。该优化减少了两个操作，仅使用移位和 XORing 来查找奇偶校验。

用减法和加法交换值

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

                    (((a) -= (b)), ((b) += (a)), ((a) = (b) - (a))))
```

这会在不使用临时变量的情况下 交换 a 和 b 的值。 当您知道这不可能发生时，可以省略对 a 和 b 在内存中相同位置的初始检查。（作为优化，编译器可能会忽略它。）如果启用溢出异常，则传递无符号值，这样就不会抛出异常。下面的 XOR 方法在某些机器上可能会稍微快一些。不要将其与浮点数一起使用（除非您对它们的原始整数表示形式进行操作）。
Sanjeev Sivasankaran 建议我在 2007 年 6 月 12 日添加此内容。Vincent Lef è vre 在 2008 年 7 月 9 日指出了溢出异常的可能性

使用 XOR 交换值

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

这是在不为临时变量使用额外空间的情况下 交换变量 a 和 b 的值的老技巧 。2005 年 1 月 20 日，Iain A. Fleming 指出，当您使用相同的内存位置交换时，上面的宏不起作用，例如 SWAP(a[i], a[j]) with i == j。因此，如果可能发生这种情况，请考虑将宏定义为 (((a) == (b)) || (((a) ^= (b)), ((b) ^= (a)), ((a) ^= (b))))。2009 年 7 月 14 日，Hallvard Furuseth 建议在某些机器上，(((a) ^ (b)) && ((b) ^= (a) ^= (b), (a) ^= (b)))可能会更快，因为 (a) ^ (b) 表达式被重用。

使用 XOR 交换单个位

```
无符号整型 i, j; // 要交换的位序列的位置

无符号整数 n；// 每个序列中连续的比特数

无符号整数 b；// 要交换的位驻留在 b 中

无符号整数 r；// 位交换结果放在这里



无符号整数 x = ((b >> i) ^ (b >> j)) & ((1U << n) – 1); // 临时异或

r = b ^ ((x << i) | (x << j));
```

作为交换位范围的示例，假设我们有 b = 001 0 111 1（以二进制表示）并且我们想要交换从 i = 1（右起第二位）开始的 n = 3 个连续位与 3 从 j = 5 开始的连续位；结果将是 r = 111 0 001 1（二进制）。
这种交换方法类似于通用 XOR 交换技巧，但用于对单个位进行操作。变量 x 存储我们要交换的位值对的异或结果，然后这些位被设置为它们自己与 x 异或的结果。当然，如果序列重叠，结果是不确定的。

2009 年 7 月 14 日，Hallvard Furuseth 建议我将 1 << n 更改为 1U << n，因为该值被分配给无符号位并避免移入符号位。

以明显的方式反转位

```
无符号整型 v；// 要反转的输入位

无符号整数 r = v；// r 将是 v 的反转位；首先得到 v 的 LSB

int s = sizeof(v) * CHAR_BIT – 1; // 最后需要额外的移位



对于 (v >>= 1; v; v >>= 1)

{

 r <<= 1;

 r |= v & 1;

 s--;

}

r <<= s；// 当 v 的最高位为零时移位
```

2004 年 10 月 15 日，Michael Hoisie 指出了原始版本中的一个错误。Randal E. Bryant 在 2005 年 5 月 3 日建议删除一个额外的操作。Behdad Esfabod 在 2005 年 5 月 18 日提出了一个小的更改，消除了循环的一次迭代。然后，在 2007 年 2 月 6 日，Liyong Zhou 建议了一个更好的循环版本而 v 不为 0，因此它不会遍历所有位，而是提前停止。

通过查找表反转字中的位

```
static const unsigned char BitReverseTable256[256] =

{

# 定义 R2(n) n, n + 2*64, n + 1*64, n + 3*64

# 定义 R4(n) R2(n), R2(n + 2*16), R2(n + 1*16), R2(n + 3*16)

# 定义 R6(n) R4(n), R4(n + 2*4), R4(n + 1*4), R4(n + 3*4)

    R6(0)、R6(2)、R6(1)、R6(3)

};



无符号整型 v；// 反转 32 位值，每次 8 位

无符号整数 c；// c 将使 v 反转



// 选项 1：

c = (BitReverseTable256[v & 0xff] << 24) |

    (BitReverseTable256[(v >> 8) & 0xff] << 16) |

    (BitReverseTable256[(v >> 16) & 0xff] << 8) |

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



// 选项 2：

无符号字符 * p = (无符号字符 *) &v;

无符号字符 * q = (无符号字符 *) &c;

q[3] = BitReverseTable256[p[0]];

q[2] = BitReverseTable256[p[1]];
```

```
q[1] = BitReverseTable256[p[2]];

q[0] = BitReverseTable256[p[3]];
```

第一种方法大约需要 17 次操作，第二种方法大约需要 12 次，假设您的 CPU 可以轻松加载和存储字节。
2009 年 7 月 14 日，Hallvard Furuseth 建议使用宏压缩表。

使用 3 个操作（64 位乘法和模除法）反转字节中的位：

```
无符号字符 b；// 反转这个（8 位）字节

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

乘法运算创建 8 位字节模式的五个单独副本，以扇出为 64 位值。AND 运算选择相对于每个 10 位位组处于正确（反转）位置的位。乘法和 AND 运算从原始字节复制位，因此它们每个都只出现在 10 位集中的一个中。来自原始字节的位的反向位置与它们在任何 10 位集合中的相对位置一致。最后一步涉及模数除以 2^10 - 1，其效果是将 64 位中的每组 10 位（从位置 0-9、10-19、20-29 .....）合并在一起价值。它们不重叠，因此模数除法的加法步骤表现得像 or 操作。
此方法归因于 Beeler, M.、Gosper, RW 和 Schroeppel, R. HAKMEM 的 Programming Hacks 部分中的 Rich Schroeppel。麻省理工学院 AI 备忘录 239，1972 年 2 月 29 日。

通过 4 次操作（64 位乘法，无除法）反转字节中的位：

```
无符号字符 b；//反转这个字节

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

下面显示了布尔变量 a, b, c, d, e, f, g,和的位值流 h，它包含一个 8 位字节。请注意第一个乘法如何将位模式扇出到多个副本，而最后一个乘法将它们组合在右数第五个字节中。 abcd efgh (–> hgfe dcba) * 1000 0000 0010 0000 0000 1000 0000 0010 (0x80200802)

---------- 0abc defg h00a bcde fgh0 0abc defg h00a bcde fgh0 & 0000 1000 1000 0100 0100 0010 0010 0001 0001 0000 (0x0884422110)

---------- 0000 d000 h000 0c00 0g00 00b0 00f0 000a 000e 0000 * 0000 0001 0000 0001 0000 0001 0000 0001 0000 0001 (0x0101010101)

---------- 0000 d000 h000 0c00 0g00 00b0 00f0 000a 000e 0000 0000 d000 h000 0c00 0g00 00b0 00f0 000a 000e 0000 0000 d000 h000 0c00 0g00 00b0 00f0 000a 000e 0000 0000 d000 h000 0c00 0g00 00b0 00f0 000a 000e 0000 0000 d000 h000 0c00 0g00 00b0 00f0 000a 000e 0000

---------- 0000 d000 h000 dc00 hg00 dcb0 hgf0 dcba hgfe dcba hgfe 0cba 0gfe 00ba 00fe 000a 000e 0000 >> 32

---------- 0000 d000 h000 dc00 hg00 dcb0 hgf0 dcba hgfe dcba & 1111 1111

---------- hgfe dcba 请注意，最后两个步骤可以在某些处理器上合并，因为寄存器可以按字节访问；只需乘以一个寄存器存储结果的高 32 位并取低字节。因此，它可能只需要 6 个操作。
由 Sean Anderson 于 2001 年 7 月 13 日设计。

使用 7 次操作（无 64 位）反转字节中的位：

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

确保将结果分配或转换为 unsigned char 以删除较高位中的垃圾。由 Sean Anderson 于 2001 年 7 月 13 日设计。Mike Keith 于 2002 年 1 月 3 日发现错别字并提供更正。

在 5 * lg(N) 操作中并行反转 N 位数量：

```
无符号整型 v; // 32 位字反转位序



//交换奇偶位

v = ((v >> 1) & 0x55555555) | ((v & 0x55555555) << 1);

// 交换连续的对

v = ((v >> 2) & 0x33333333) | ((v & 0x33333333) << 2);

// 交换半字节 ...

v = ((v >> 4) & 0x0F0F0F0F) | ((v & 0x0F0F0F0F) << 4);

// 交换字节
```

```
v = ((v >> 8) & 0x00FF00FF) | ((v & 0x00FF00FF) << 8);

// 交换 2 字节长的对

v = ( v >> 16 ) | ( v << 16);
```

下面的变体也是 O(lg(N))，但是它需要更多的操作来反转 v。它的优点是通过动态计算常数来占用更少的内存。

```
无符号整数 s = sizeof(v) * CHAR_BIT；// 位大小；必须是 2 的幂

无符号整数掩码 = ~0;

而 ((s >>= 1) > 0)

{

 掩码 ^=（掩码 << s）；

 v = ((v >> s) & 掩码) | ((v << s) & ~掩码);

}
```

上述这些方法最适用于 N 较大的情况。如果您将以上内容与 64 位整数（或更大）一起使用，则需要添加更多行（遵循模式）；否则只有低 32 位取反，结果在低 32 位。
有关详细信息，请参阅 Dr. Dobb's Journal 1983，Edwin Freed 关于二进制幻数的文章。Ken Raeburn 于 2005 年 9 月 13 日提出了第二个变体。Veldmeijer 在 2006 年 3 月 19 日的最后一行中提到第一个版本可以在没有 ANDS 的情况下运行。

在没有除法运算符的情况下计算模除以 1 << s

```
常量无符号整数 n；// 分子

常量无符号整数；

const unsigned int d = 1U << s; // 所以 d 将是以下之一：1, 2, 4, 8, 16, 32, ...
```

```
无符号整型；// m 将是 n % d

m = n & (d – 1);
```

大多数程序员很早就学会了这个技巧，但为了完整起见将其包括在内。

计算模数除法 (1 << s) - 1 没有除法运算符

```
无符号整数 n；// 分子

常量无符号整数；// 小号 > 0

const unsigned int d = (1 << s) – 1; // 所以 d 是 1, 3, 7, 15, 31, ...)。

无符号整型；// n % d 在这里。



对于（m = n；n > d；n = m）

{

 对于 (m = 0; n; n >>= s)

 {

   m += n & d;

 }

}

// 现在 m 是一个从 0 到 d 的值，但是因为有模数除法

// 当 m 为 d 时，我们希望 m 为 0。

m = m == d ? 0：米；
```

这种模数除以小于 2 的幂的整数的方法最多需要 5 + (4 + 5 * ceil(N / s)) * ceil(lg(N / s)) 操作，其中 N 是分子中的位数。换句话说，最多需要 O(N * lg(N)) 时间。
由 Sean Anderson 于 2001 年 8 月 15 日设计。在 Sean A. Irvine 于 2004 年 6 月 17 日纠正我之前，我错误地评论说我们可以 m = ((m + 1) & d) - 1;在最后选择分配。Michael Miller 于 2005 年 4 月 25 日在代码中发现了一个拼写错误。

通过 (1 << s) - 1 并行计算模数除法，无需除法运算符

```
// 以下是针对 32 位字长的!

静态常量无符号整数 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}
};


静态常量无符号整型 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}

};



无符号整数 n；// 分子

常量无符号整数；// 小号 > 0

const unsigned int d = (1 << s) – 1; // 所以 d 是 1, 3, 7, 15, 31, ...)。

无符号整型；// n % d 在这里。



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：米；// 或者，不太便携：m = m & –((signed)(m – d) >> s);
```

这种用小于 2 的幂的整数求模数除法的方法最多需要 O(lg(N)) 时间，其中 N 是分子中的位数 (32 位，对于上面的代码）。操作次数最多为 12 + 9 * ceil(lg(N))。如果您在编译时知道分母，则可以删除这些表；只需提取一些相关条目并展开循环即可。它可以很容易地扩展到更多位。

它通过并行求和 base (1 << s) 中的值来找到结果。首先，每隔一个基数 (1 << s) 值添加到前一个。想象一下，结果写在一张纸上。将纸张切成两半，以便每个切

块上有一半的值。对齐值并将它们相加到一张新纸上。重复将这张纸切成两半 (这将是前一张纸的四分之一) 并求和，直到不能进一步切割为止。在执行 lg(N/s/2) 次切割后，我们不再切割；只需继续添加值并将结果像以前一样放在一张新纸上，同时至少有两个 s 位值。
由 Sean Anderson 于 2001 年 8 月 20 日设计。Randy E. Bryant 于 2005 年 5 月 3 日发现了一个拼写错误 (粘贴代码后，我后来在变量声明中添加了“unsinged”) 。和之前的 hack 一样，我错误地评论说我们可以 m = ((m + 1) & d) – 1;在最后选择赋值，Don Knuth 在 2006 年 4 月 19 日纠正了我并建议 m = m & –((signed)(m – d) >> s). 2009 年 6 月 18 日，Sean Irvine 提出了一个使用 ((n >> s) & M[s])而不是 的 更改((n & ~M[s]) >> s)，这通常需要更少的操作，因为 M[s] 常量已经加载。

在 O(N) 操作中找到设置为 MSB N 的整数的对数基数 2（显而易见的方法）

```
无符号整型 v；// 32 位字查找以 2 为底的对数

无符号整数 r = 0；// r 将是 lg(v)


while (v >>= 1) // 展开以获得更快的速度...

{

 r++;

}
```

整数的对数基数 2 与最高位集 (或最高有效位集，MSB) 的位置相同。下面的 log base 2 方法比这个更快。

查找具有 64 位 IEEE 浮点数的整数的以 2 为底的整数对数

```
诠释五; // 32 位整数，用于查找以 2 为底的对数

诠释; // log_2(v) 的结果放在这里

union { unsigned int u[2]; 双 d; }吨; // 温度



图[__FLOAT_WORD_ORDER==LITTLE_ENDIAN] = 0x43300000；

tu[__FLOAT_WORD_ORDER!=LITTLE_ENDIAN] = v;

td -= 4503599627370496.0;

r = (tu[__FLOAT_WORD_ORDER==LITTLE_ENDIAN] >> 20) - 0x3FF;
```

上面的代码通过将整数存储在尾数中同时将指数设置为 2 52 来加载一个带有 32 位整数（没有填充位）的 64 位（IEEE-754 浮点）双精度数。从这个新创建的双精度数中减去 2 52（表示为双精度数），这将生成的指数设置为输入值 v 的对数基数 2。剩下的就是将指数位移动到位置（右 20 位）并减去偏差 0x3FF（十进制为 1023）。这种技术只需要 5 次操作，但许多 CPU 在处理双精度数时速度很慢，并且必须适应体系结构的字节顺序。
Eric Cole 于 2006 年 1 月 15 日发给我。Evan Felix 于 2006 年 4 月 4 日指出了一个打字错误。Vincent Lef è vre 于 2008 年 7 月 9 日告诉我更改字节序检查以使用浮点数的字节序，这可能与整数的字节序不同.

使用查找表查找整数的对数底数 2

```
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)
};

无符号整型 v；// 32 位字查找日志
无符号 r；// r 将是 lg(v)
注册无符号整型 t, tt; // 临时工

如果（tt = v >> 16）
{
 r = (t = tt >> 8) ? 24 + 日志表 256[t] : 16 + 日志表 256[tt];
}
别的
{
 r = (t = v >> 8) ? 8 + 日志表 256[t]：日志表 256[v];
}
```

查表法只需要 7 次左右的操作就可以找到一个 32 位值的 log。如果扩展为 64 位数量，大约需要 9 次操作。另一个操作可以通过使用四个表来减少，每个表都可以添加可能的内容。使用 int 表元素可能会更快，具体取决于您的体系结构。上面的代码被调整为均匀分布的*输出*值。如果您的*输入*均匀分布在所有 32 位值中，则考虑使用以下内容：

```
如果（tt = v >> 24）
{
 r = 24 + 日志表 256[tt];
```

```
}

否则如果 (tt = v >> 16)

{

 r = 16 + 日志表 256[tt];

}

否则如果 (tt = v >> 8)

{

 r = 8 + 日志表 256[tt];

}

别的

{

 r = LogTable256[v];

}
```

最初通过算法生成日志表：

```
LogTable256[0] = LogTable256[1] = 0;

对于 (int i = 2; i < 256; i++)

{

 LogTable256[i] = 1 + LogTable256[i / 2];

}

LogTable256[0] = -1; // 如果你想让 log(0) 返回 -1
```

Behdad Esfahbod 和我在 2005 年 5 月 18 日（平均）削减了一部分手术。2006 年 11 月 14 日，Emanuel Hoogeveen 又削减了一部分手术。David A. Butterfield 在 2008 年 9 月 19 日建议调整为均匀分布的输入值的变体。Venkat Reddy 在

2009 年 1 月 5 日告诉我 log(0) 应该返回 -1 以指示错误，所以我更改了表中的第一个条目。

在 O(lg(N)) 操作中找到 N 位整数的对数基数 2

```
无符号整型 v；// 查找 log2 的 32 位值

const unsigned int b[] = {0x2, 0xC, 0xF0, 0xFF00, 0xFFFF0000};

const unsigned int S[] = {1, 2, 4, 8, 16};

诠释我;



注册无符号整数 r = 0；// log2(v) 的结果会放在这里

for (i = 4; i >= 0; i--) // 展开速度...

{

  如果 (v & b[i])

  {

    v >>= S[i];

    r |= S[i];

  }

}



// 或者（如果你的 CPU 分支很慢）：
```

```
无符号整型 v；// 查找 log2 的 32 位值

注册无符号整数 r；// log2(v) 的结果会放在这里

注册无符号整数移位；


r = (v > 0xFFFF) << 4; v >>= r；

移位 = (v > 0xFF) << 3; v >>= 转移；r |= 转移；

移位 = (v > 0xF) << 2; v >>= 转移；r |= 转移；

移位 = (v > 0x3) << 1; v >>= 转移；r |= 转移；

                                    r |= (v >> 1);



// 或（如果您知道 v 是 2 的幂）：


无符号整型 v；// 查找 log2 的 32 位值

static const unsigned int b[] = {0xAAAAAAAA, 0xCCCCCCCC, 0xF0F0F0F0,

                                 0xFF00FF00, 0xFFFF0000};

注册无符号整数 r = (v & b[0]) != 0;

for (i = 4; i > 0; i--) // 展开速度...

{

  r |= ((v & b[i]) != 0) << i;

}
```

当然，为了扩展代码以查找 33 位到 64 位数字的日志，我们将在 b 追加另一个元素 0xFFFFFFFF00000000，在 S 追加 32，然后从 5 循环到 0。这种方法

比较早的表查找版本，但如果您不想要大表或者您的体系结构访问内存很慢，这是一个不错的选择。第二种变体涉及稍微多一些操作，但在分支成本高的机器（例如 PowerPC）上它可能更快。
第二个版本是 Eric Cole 于 2006 年 1 月 7 日发给我的。Andrew Shapira 随后对其进行了一些修改，并于 2007 年 9 月 1 日将他的变体（上图）发给我 。John 向我建议了第三个变体欧文斯 于 2002 年 4 月 24 日；它更快，但 *仅当输入已知为 2 的幂时才适用*。2003 年 5 月 25 日，Ken Raeburn 建议通过为 b[] 使用较小的数字来改进一般情况，这在某些体系结构上加载速度更快（例如，如果字长为 16 位，则可能只需要一条加载指令）。这些值适用于一般版本，但不适用于它下面的特殊版本，其中 v 是 2 的幂；2003 年 12 月 12 日，Glenn Slayden 提请我注意这一疏忽。

使用乘法和查找在 O(lg(N)) 操作中找到 N 位整数的对数基数 2

```
uint32_t v; // 找到 32 位 v 的对数基数 2

诠释; // 结果在这里


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; // 第一轮向下取一个小于 2 的幂

v |= v >> 2;
```

```
v |= v >> 4;

v |= v >> 8;

v |= v >> 16;



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

上面的代码使用小型表查找和乘法计算 32 位整数的以 2 为底的对数。它只需要 13 次操作，而之前的方法需要（最多）20 次。纯基于表的方法需要最少的操作，但它在表大小和速度之间提供了合理的折衷。
如果你知道 v 是 2 的幂，那么你只需要以下内容：

```
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];
```

Eric Cole 于 2006 年 1 月 8 日在阅读了以下关于向上舍入到 2 的幂的条目以及以下 使用 DeBruijn 序列通过乘法和查找计算尾随位数的方法后设计了这个。2009 年 12 月 10 日，马克·迪金森 (Mark Dickinson) 要求将 v 舍入为比下一个 2 的次方（而不是 2 的次方）小 1，从而削减了一些操作。

查找整数的以 10 为底的整数对数

```
无符号整型 v;  // 非零的 32 位整数值，用于计算以 10 为底的对数

诠释; // 结果在这里

诠释吨; // 暂时的
```

```
static unsigned int const PowersOf10[] =

   {1, 10, 100, 1000, 10000, 100000,

    1000000, 10000000, 100000000, 1000000000};



t = (IntegerLogBase2(v) + 1) * 1233 >> 12; //（使用上面的 lg2 方法）

r = t - (v < PowersOf10[t]);
```

以 10 为底的整数对数首先使用上述技术之一计算以 2 为底的对数。根据关系 log 10 (v) = log 2 (v) / log 2 (10)，我们需要将其乘以 1 /log 2 (10)，大约为 1233/4096，或 1233 后跟右移 12。需要加一，因为 IntegerLogBase2 向下舍入。最后，由于值 t 只是一个可能相差 1 的近似值，因此可以通过减去 v < PowersOf10[t] 的结果来找到精确值。

该方法比 IntegerLogBase2 多了 6 次操作。它可以通过修改上面的日志基 2 表查找方法来加速（在具有快速内存访问的机器上），以便条目包含为 t 计算的内容（即，预添加、-mulitply 和 -shift）。假设使用了 4 个表（v 的每个字节一个），这样做总共只需要 9 次操作即可找到以 10 为底的日志。

Eric Cole 建议我在 2006 年 1 月 7 日添加一个版本。

以显而易见的方式查找整数的以 10 为底的整数对数

```
无符号整型 v; // 非零的 32 位整数值，用于计算以 10 为底的对数

诠释; // 结果在这里



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;
```

当输入均匀分布在 32 位值上时，此方法效果很好，因为第一次比较捕获了 76% 的输入，第二次比较捕获了 21%，第三次比较捕获了 2%，依此类推（斩波其余的每次比较都减少了 90%）。因此，平均需要不到 2.6 次操作。
2007 年 4 月 18 日，Emanuel Hoogeveen 提出了一个变体，其中条件使用除法，这不如简单比较快。

查找 32 位 IEEE 浮点数的以 2 为底的整数对数

```
常量浮点数；// 找到 int(log2(v))，其中 v > 0.0 && finite(v) && isnormal(v)

诠释 c; // 32 位 int c 得到结果；



c = *(const int *) &v; // 或者，为了可移植性：memcpy(&c, &v, sizeof c);

c = (c >> 23) – 127;
```

上面的速度很快，但是符合 IEEE 754 的架构使用 *次正规*（也称为*非正规*）浮点数。它们的指数位设置为零（表示 pow(2,-127)），尾数未归一化，因此它包含前导零，因此 log2 必须根据尾数计算。要适应次正规数，请使用以下内容：

```
常量浮点数；// 找到 int(log2(v))，其中 v > 0.0 && finite(v)

诠释 c; // 32 位 int c 得到结果；

int x = *(const int *) &v; // 或者，为了可移植性：memcpy(&x, &v, sizeof x);



c = x >> 23；


如果（三）
{
 c–=127；
```

```
}
别的
{ // 次正规，所以使用尾数重新计算：c = intlog2(x) – 149;
 注册无符号整数 t；// 暂时的
 // 注意前面定义了 LogTable256
 如果（t = x >> 16）
 {
   c = LogTable256[t] – 133；
 }
 别的
 {
   c = (t = x >> 8) ? LogTable256[t] – 141 : LogTable256[x] – 149;
 }
}
```

2004 年 6 月 20 日，Sean A. Irvine 建议我加入处理次正规数的代码。2005 年 6 月 11 日，Falk H ü ffner 指出 ISO C99 6.5/7 为常见类型双关习语 *(int *)& 指定了未定义的行为，尽管它已经在 99.9% 的 C 编译器上工作。他建议使用 memcpy 来获得最大的可移植性，或者在某些编译器上使用 float 和 int 的联合来生成比 memcpy 更好的代码。

查找 32 位 IEEE 浮点数的 pow(2, r)-root 的整数对数基数 2（对于无符号整数 r）

```
常量解释器；
常量浮点数；// 找到 int(log2(pow((double) v, 1. / pow(2, r)))),
```

```
            // 其中 isnormal(v) 和 v > 0

诠释 c; // 32 位 int c 得到结果;



c = *(const int *) &v; // 或者，为了可移植性：memcpy(&c, &v, sizeof c);

c = ((((c - 0x3f800000) >> r) + 0x3f800000) >> 23) - 127;
```

因此，如果 r 为 0，例如，我们有 c = int(log2((double) v))。如果 r 为 1，则我们有 c = int(log2(sqrt((double) v)))。如果 r 是 2，那么我们有 c = int(log2(pow((double) v, 1./4)))。
2005 年 6 月 11 日，Falk H ü ffner 指出 ISO C99 6.5/7 留下了类型双关习语 *(int *)& 未定义，他建议使用 memcpy。

线性计算右侧的连续零位（尾随）

```
无符号整型 v; // 输入以计算尾随零位

诠释 c; // 输出：c 将计算 v 的尾随零位,

     // 所以如果 v 是 1101000（基数 2），那么 c 就是 3
如果（五）
{
 v = (v ^ (v - 1)) >> 1; // 将 v 的尾随 0 设置为 1，其余为零
 对于（c = 0; v; c++）
 {
   v >>= 1;
 }
}
别的
```

```
{
  c = CHAR_BIT * sizeof(v);
}
```

（均匀分布的）随机二进制数中尾随零位的平均数量是一个，因此与下面的更快方法相比，这个 O（尾随零）解决方案并没有那么糟糕。
2007 年 8 月 15 日，Jim Cole 建议我添加一个线性时间方法来计算尾随零。
2007 年 10 月 22 日，Jason Cunningham 指出我忽略了为 v 粘贴无符号修饰符。

并行计算右侧连续的零位（尾随）

```
无符号整型 v；// 32 位字输入以计算右边的零位
无符号整数 c = 32；// c 将是右边零位的数量
v &= -signed(v);
如果 (v) c--；
如果 (v & 0x0000FFFF) c -= 16;
如果 (v & 0x00FF00FF) c -= 8;
如果 (v & 0x0F0F0F0F) c -= 4;
如果 (v & 0x33333333) c -= 2;
如果 (v & 0x55555555) c -= 1;
```

在这里，我们基本上是在执行与并行查找对数基数 2 相同的操作，但我们首先隔离最低 1 位，然后从最大值开始递减的 c。对于 N 位字，操作次数最多为 3 * lg(N) + 4。
Bill Burdick 建议进行优化，在 2011 年 2 月 4 日将时间从 4 * lg(N) 减少。

通过二进制搜索计算右侧连续的零位（尾随）

```
无符号整型 v；// 32 位字输入以计算右边的零位

无符号整数 c；// c 将是右边零位的数量，

                // 所以如果 v 是 1101000（基数 2），那么 c 就是 3

// 注意：如果 0 == v，则 c = 31。

如果 (v & 0x1)

{

 // 奇数 v 的特例（假设有一半时间发生）

 c = 0;

}

别的

{

 c = 1;

 如果 ((v & 0xffff) == 0)

 {

   v >>= 16;

   c += 16;

 }

 如果 ((v & 0xff) == 0)

 {

   v >>= 8;

   c += 8;
```

```
}

如果 ((v & 0xf) == 0)

{

  v >>= 4;

  c += 4;

}

如果 ((v & 0x3) == 0)

{

  v >>= 2;

  c += 2;

}

c -= v & 0x1;

}
```

上面的代码与前面的方法类似，但它以类似于二进制搜索的方式通过累加 c 来计算尾随零的数量。在第一步中，它检查 v 的低 16 位是否为零，如果是，则将 v 右移 16 位并将 16 添加到 c，这将 v 中要考虑的位数减少了一半。每个后续条件步骤同样将位数减半，直到只有 1。此方法比上一个方法快 (大约 33%)，因为 if 语句的主体执行频率较低。
Matt Whitlock 在 2006 年 1 月 25 日提出了这个建议。Andrew Shapira 在 2007 年 9 月 5 日削减了几个操作（通过设置 c=1 并在末尾无条件地减去）。

通过转换为浮点数来计算右侧的连续零位（尾随）

```
无符号整型 v; // 找到 v 中尾随零的数量

诠释; // 结果在这里
```

```
浮动 f = (浮动)(v & –v); // 将 v 中的最低有效位转换为浮点数

r = (*(uint32_t *)&f >> 23) – 0x7f;
```

虽然这只需要大约 6 次操作，但在某些机器上将整数转换为浮点数的时间可能会很长。将 32 位 IEEE 浮点表示的指数向下移动，并减去偏差以给出 v 中设置的最低有效 1 位的位置。如果 v 为零，则结果为 -127。

使用模除法和查找计算右侧的连续零位（尾随）

```
无符号整型 v; // 找到 v 中尾随零的数量

诠释; // 将结果放入 r

static const int Mod37BitPosition[] = // 将位值 mod 37 映射到它的位置

{

 32, 0, 1, 26, 2, 23, 27, 0, 3, 16, 24, 30, 28, 11, 0, 13, 4,

 7, 17, 0, 25, 22, 31, 15, 29, 10, 12, 6, 0, 21, 14, 9, 5,

 20, 8, 19, 18

};

r = Mod37BitPosition[(–v & v) % 37];
```

上面的代码找到了右边尾随的零的数量，因此二进制 0100 将产生 2。它利用了前 32 位位置值与 37 互质的事实，因此执行与 37 的模除法得到每个唯一编号从 0 到 36。然后可以使用小型查找表将这些数字映射到零的数量。它仅使用 4 个操作，但是索引到表中并执行模除法可能使其不适用于某些情况。根据 Hacker's Delight 的说法，我独立想出了这个，然后搜索表值的子序列，发现它是 Reiser 早些时候发明 的。

使用乘法和查找计算右侧的连续零位（尾随）

```
无符号整型 v；// 查找 32 位 v 中尾随零的数量

诠释; // 结果在这里

static const int MultiplyDeBruijnBitPosition[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 = MultiplyDeBruijnBitPosition[((uint32_t)((v & -v) * 0x077CB531U)) >> 27];
```

将位向量转换为集合位的索引是一个示例。它比之前涉及模数除法的操作多了一项操作，但乘法可能更快。表达式 (v & -v) 从 v 中提取最低有效的 1 位。常量 0x077CB531UL 是一个 de Bruijn 序列，它会为每个可能的位位置产生一个独特的位模式，并将其乘以高 5 位。如果没有位设置，它返回 0。阅读 Charles E. Leiserson、Harald Prokof 和 Keith H. Randall 撰写 的论文 Using de Bruijn Sequences to Index 1 in a Computer Word 可以找到更多信息。
2005 年 10 月 8 日，Andrew Shapira 建议我添加这个。Dustin Spicuzza 在 2009 年 4 月 14 日要求我将乘法的结果转换为 32 位类型，以便在使用 64 位整数编译时它可以工作。

通过浮动铸造四舍五入到下一个最高的 2 次幂

```
无符号整数常量 v；// 将这个 32 位值四舍五入为下一个最高的 2 次方

无符号整数 r；// 把结果放在这里。（所以 v=3 -> r=4；v=8 -> r=8）



如果 (v > 1)

{
```

```
浮动 f = (浮动)v;

unsigned int const t = 1U << ((*(unsigned int *)&f >> 23) - 0x7f);

r = t << (t < v);

}

别的

{

r = 1;

}
```

上面的代码使用了 8 个操作，但适用于所有 v <= (1<<31)。
快速和肮脏的版本，对于 1 < v < (1<<25) 的域：

```
浮动 f = (浮动)(v - 1);

r = 1U << ((*(无符号整数*)(&f) >> 23) - 126);
```

虽然快速和肮脏的版本只使用大约 6 个操作，但在 Athlon™ XP 2100+ CPU 上进行基准测试时，它比下面的技术 （涉及 12 个操作） 慢大约三倍 。不过，有些 CPU 会更好。
2005 年 9 月 27 日，Andi Smithers 建议我包括一种转换为浮点数的技术，以查找数字的 lg 以四舍五入为 2 的幂。与此处快速而肮脏的版本类似，他的版本适用于小于 (1 <<25)，由于尾数四舍五入，但多用了一次操作。

四舍五入到下一个最高的 2 次方

```
无符号整型 v; // 计算 32 位 v 的下一个最高的 2 次方

在 - ;

v |= v >> 1;
```

```
v |= v >> 2;

v |= v >> 4;

v |= v >> 8;

v |= v >> 16;

在++;
```

在 12 次操作中，此代码计算 32 位整数的下一个最高的 2 次方。结果可用公式 1U << (lg(v - 1) + 1) 表示。请注意，在 v 为 0 的边缘情况下，它返回 0，这不是 2 的幂；如果重要的话，您可以附加表达式 v += (v == 0) 来解决这个问题。使用公式和使用查找表的 log base 2 方法进行 2 次操作会更快，但在某些情况下，查找表并不合适，所以上面的代码可能是最好的。（在 Athlon™ XP 2100+ 上，我发现上面的 shift-left 然后 OR 代码与使用单个 BSR 汇编语言指令一样快，它反向扫描以找到最高设置位。）它通过复制最高位设置为所有较低位，然后加一，

您也可以使用 floor(lg(v)) 的查找表，然后计算 1<<(1+floor(lg(v)))；仅在 8 或 9 次操作中计算 2 的下一个更高次幂。Atul Divekar 建议我在 2010 年 9 月 5 日提及此事。

由 Sean Anderson 于 2001 年 9 月 14 日设计。Pete Hart 向我指出 他和 William Lewis 在 1997 年 2 月 发布的几个新闻组帖子，他们在其中得出了相同的算法。

以明显的方式交错位

```
无符号短 x；// 交错 x 和 y 的位，这样所有的

无符号短 y；// x 的位在偶数位置，y 在奇数位置；

无符号整数 z = 0；// z 获取结果莫顿数。


for (int i = 0; i < sizeof(x) * CHAR_BIT; i++) // 展开以获得更快的速度...
```

```
{
  z |= (x & 1U << i) << i | (y & 1U << i) << (i + 1);
}
```

交错位（又名 Morton 数）对于线性化 2D 整数坐标很有用，因此 x 和 y 被组合成一个可以轻松比较的数字，并且具有这样的属性，即如果 x 和 y 值接近，则一个数字通常接近另一个数字.

通过查表交错位

```
static const unsigned short MortonTable256[256] =
{
  0x0000、0x0001、0x0004、0x0005、0x0010、0x0011、0x0014、0x0015、
  0x0040、0x0041、0x0044、0x0045、0x0050、0x0051、0x0054、0x0055、
  0x0100、0x0101、0x0104、0x0105、0x0110、0x0111、0x0114、0x0115、
  0x0140、0x0141、0x0144、0x0145、0x0150、0x0151、0x0154、0x0155、
  0x0400、0x0401、0x0404、0x0405、0x0410、0x0411、0x0414、0x0415、
  0x0440、0x0441、0x0444、0x0445、0x0450、0x0451、0x0454、0x0455、
  0x0500、0x0501、0x0504、0x0505、0x0510、0x0511、0x0514、0x0515、
  0x0540、0x0541、0x0544、0x0545、0x0550、0x0551、0x0554、0x0555、
  0x1000、0x1001、0x1004、0x1005、0x1010、0x1011、0x1014、0x1015、
  0x1040、0x1041、0x1044、0x1045、0x1050、0x1051、0x1054、0x1055、
  0x1100、0x1101、0x1104、0x1105、0x1110、0x1111、0x1114、0x1115、
  0x1140、0x1141、0x1144、0x1145、0x1150、0x1151、0x1154、0x1155、
```

0x1400、0x1401、0x1404、0x1405、0x1410、0x1411、0x1414、0x1415、

0x1440、0x1441、0x1444、0x1445、0x1450、0x1451、0x1454、0x1455、

0x1500、0x1501、0x1504、0x1505、0x1510、0x1511、0x1514、0x1515、

0x1540、0x1541、0x1544、0x1545、0x1550、0x1551、0x1554、0x1555、

0x4000、0x4001、0x4004、0x4005、0x4010、0x4011、0x4014、0x4015、

0x4040、0x4041、0x4044、0x4045、0x4050、0x4051、0x4054、0x4055、

0x4100、0x4101、0x4104、0x4105、0x4110、0x4111、0x4114、0x4115、

0x4140、0x4141、0x4144、0x4145、0x4150、0x4151、0x4154、0x4155、

0x4400、0x4401、0x4404、0x4405、0x4410、0x4411、0x4414、0x4415、

0x4440、0x4441、0x4444、0x4445、0x4450、0x4451、0x4454、0x4455、

0x4500、0x4501、0x4504、0x4505、0x4510、0x4511、0x4514、0x4515、

0x4540、0x4541、0x4544、0x4545、0x4550、0x4551、0x4554、0x4555、

0x5000、0x5001、0x5004、0x5005、0x5010、0x5011、0x5014、0x5015、

0x5040、0x5041、0x5044、0x5045、0x5050、0x5051、0x5054、0x5055、

0x5100、0x5101、0x5104、0x5105、0x5110、0x5111、0x5114、0x5115、

0x5140、0x5141、0x5144、0x5145、0x5150、0x5151、0x5154、0x5155、

0x5400、0x5401、0x5404、0x5405、0x5410、0x5411、0x5414、0x5415、

0x5440、0x5441、0x5444、0x5445、0x5450、0x5451、0x5454、0x5455、

0x5500、0x5501、0x5504、0x5505、0x5510、0x5511、0x5514、0x5515、

0x5540、0x5541、0x5544、0x5545、0x5550、0x5551、0x5554、0x5555

};

```
无符号短 x；// 交错 x 和 y 的位，这样所有的

无符号短 y；// x 的位在偶数位置，y 在奇数位置；

无符号整数 z；// z 获取生成的 32 位莫顿数。


z = MortonTable256[y >> 8] << 17 |

    MortonTable256[x >> 8] << 16 |

    MortonTable256[y & 0xFF] << 1 |

    MortonTable256[x & 0xFF];
```

为了提高速度，使用一个额外的表，其中包含 MortonTable256 预向左移动一位的值。然后可以将第二个表用于 y 查找，从而将操作减少两次，但所需的内存几乎加倍。扩展同样的想法，可以使用四个表，其中两个表在前两个表的左侧预移位 16，因此我们总共只需要 11 个操作。

使用 64 位乘法交错位

在 11 次操作中，此版本交织两个字节的位（而不是像其他版本中的短操作），但许多操作是 64 位乘法，因此它并不适合所有机器。输入参数 x 和 y 应小于 256。

```
无符号字符 x；// 交错（8 位）x 和 y 的位，这样所有的

无符号字符；// x 的位在偶数位置，y 在奇数位置；

无符号短 z；// z 得到最终的 16 位莫顿数。


z = ((x * 0x0101010101010101ULL & 0x8040201008040201ULL) *

    0x0102040810204081ULL >> 49) & 0x5555 |
```

```
  ((y * 0x0101010101010101ULL & 0x8040201008040201ULL) *
   0x0102040810204081ULL >> 48) & 0xAAAA;
```

2004 年 10 月 10 日，Holger Bettag 在阅读了此处的基于乘法的位反转后受到启发，提出了这项技术。

通过二进制幻数交错位

```
static const unsigned int B[] = {0x55555555, 0x33333333, 0x0F0F0F0F, 0x00FF00FF};
static const unsigned int S[] = {1, 2, 4, 8};

无符号整型 x；// 交错 x 和 y 的低 16 位，所以 x 的位
无符号整型；// 位于奇数中的偶数位置和来自 y 的位；
无符号整数 z；// z 获取生成的 32 位莫顿数。
            // x 和 y 最初必须小于 65536。

x = (x | (x << S[3])) & B[3];
x = (x | (x << S[2])) & B[2];
x = (x | (x << S[1])) & B[1];
x = (x | (x << S[0])) & B[0];

y = (y | (y << S[3])) & B[3];
y = (y | (y << S[2])) & B[2];
y = (y | (y << S[1])) & B[1];
```

```
y = (y | (y << S[0])) & B[0];
```

```
z = x | (y << 1);
```

确定一个字是否有一个零字节

```
// 更少的操作：
```

```
无符号整型 v；// 32 位字检查其中是否有任何 8 位字节为 0
```

```
bool hasZeroByte = ~((((v & 0x7F7F7F7F) + 0x7F7F7F7F) | v) | 0x7F7F7F7F);
```

上面的代码在进行一次复制一个单词的快速字符串复制时可能很有用；它使用 5 个操作。另一方面，以明显的方式（随后）测试空字节至少有 7 次操作（以最节省的方式计算时），最多 12 次。

```
// 更多操作：
```

```
bool hasNoZeroByte = ((v & 0xff) && (v & 0xff00) && (v & 0xff0000) && (v & 0xff000000))
```

```
// 或者：
```

```
无符号字符 * p = (无符号字符 *) &v;
```

```
bool hasNoZeroBytes = *p && *(p + 1) && *(p + 2) && *(p + 3);
```

本节开头的代码（标记为“更少的操作”）首先将字中 4 个字节的高位清零。随后，它添加一个数字，如果任何低位被初始设置，该数字将导致溢出到字节的高位。接下来将原始单词的高位与这些值进行或运算；因此，如果设置了字节中的任何位，则设置字节的高位。最后，我们确定这些高位中是否有任何一个为零，方法是与除高位以外的所有位置的 OR 运算并将结果取反。扩展到 64 位是微不足道的；只需将常量增加为 0x7F7F7F7F7F7F7F7F。
为了进一步改进，可以执行仅需要 4 次操作的快速预测试以确定该字是否*可能*具有零字节。如果高字节为 0x80，测试也会返回 true，因此偶尔会出现误报，

但上面较慢且更可靠的版本可能会用于候选人，以便在正确输出的情况下整体提高速度。

```
bool hasZeroByte = ((v + 0x7efefeff) ^ ~v) & 0x81010100;

if (hasZeroByte) // 或者高字节可能只有 0x80

{

  hasZeroByte = ~((((v & 0x7F7F7F7F) + 0x7F7F7F7F) | v) | 0x7F7F7F7F);

}
```

还有一种更快的方法——使用 **hasless**(v, 1)，定义如下；它在 4 个操作中工作，不需要后续验证。它简化为

```
#define haszero(v) (((v) - 0x01010101UL) & ~(v) & 0x80808080UL)
```

只要 v 中的相应字节为零或大于 0x80，子表达式 (v - 0x01010101UL) 的计算结果为任何字节中设置的高位。子表达式 ~v & 0x80808080UL 的计算结果为以字节为单位设置的高位，其中 v 的字节未设置其高位（因此该字节小于 0x80）。最后，通过对这两个子表达式进行与操作，结果是高位设置，其中 v 中的字节为零，因为由于第一个子表达式中的值大于 0x80 而设置的高位被第二个子表达式屏蔽掉。

Paul Messmer 在 2004 年 10 月 2 日建议快速预测试改进。Juha Järvi 后来 hasless(v, 1) 在 2005 年 4 月 6 日建议，这是他在 Paul Hsieh 的装配实验室发现的；此前它是由 Alan Mycroft 于 1987 年 4 月 27 日在新闻组帖子中撰写的。

确定一个单词是否有一个字节等于 n

我们可能想知道一个字中的任何字节是否具有特定值。为此，我们可以对要测试的值与一个已填充了我们感兴趣的字节值的字进行异或。因为对一个值与自身进行异或运算会产生零字节，否则会产生非零字节，因此我们可以将结果传递给 haszero.

```
#define 有值（x，n）\

(有零((x) ^ (~0UL/255 * (n))))
```

Stephen M Bennet 在阅读 的条目后于 2009 年 12 月 13 日提出了这个建议 haszero。

确定一个单词是否有一个字节小于 n

测试单词 x 是否包含值 < n 的无符号字节。特别是对于 n=1，它可用于通过一次检查一个 long 来查找 0 字节，或者通过首先对 x 与掩码进行异或来查找任何字节。当 n 为常量时，使用 4 次算术/逻辑运算。
要求：x>=0；0<=n<=128

```
#define hasless(x,n) (((x)-~0UL/255*(n))&~(x)&~0UL/255*128)
```

要计算 7 次操作中 x 中小于 n 的字节数，请使用

```
#定义无数（x，n）\

(((~0UL/255*(127+(n))-((x)&~0UL/255*127))&~(x)&~0UL/255*128)/128%255)
```

Juha Järvi 于 2005 年 4 月 6 日向我发送了这项巧妙的技术。该 countless 宏是 Sean Anderson 于 2005 年 4 月 10 日添加的，灵感来自 Juha 的 countmore，如下所示。

确定一个单词是否有一个字节大于 n

测试单词 x 是否包含值 > n 的无符号字节。当 n 为常量时，使用 3 个算术/逻辑运算。
要求：x>=0；0<=n<=127

```
#define hasmore(x,n) (((x)+~0UL/255*(127-(n))|(x))&~0UL/255*128)
```

要计算在 6 次操作中 x 中大于 n 的字节数，请使用：

```
#define countmore(x,n) \
(((((x)&~0UL/255*127)+~0UL/255*(127-(n))|(x))&~0UL/255*128)/128%255)
```

该宏 hasmore 由 Juha Järvi 于 2005 年 4 月 6 日提出，他 countmore 于 2005 年 4 月 8 日添加。

确定一个单词是否有一个字节在 m 和 n 之间

当 m < n 时，此技术测试单词 x 是否包含无符号字节值，例如 m < value < n。当 n 和 m 为常数时，它使用 7 个算术/逻辑运算。

注意：等于 n 的字节可能会被报告 likelyhasbetween 为误报，因此如果需要某个结果，应按字符进行检查。

要求：x>=0；0<=米<=127；0<=n<=128

```
#define likelyhasbetween(x,m,n) \
((((x)-~0UL/255*(n))&~(x)&((x)&~0UL/255*127)+~0UL/255*(127-(m)))&~ 0UL/255*128)
```

这种技术适用于快速预测试。需要更多操作（常量 m 和 n 总共 8 个）但提供准确答案的变体是：

```
#define hasbetween(x,m,n) \
((~0UL/255*(127+(n))-((x)&~0UL/255*127)&~(x)&((x)&~0UL/255*127)+~0UL/255 *(127-(m)))&~0UL/255*128)
```

要计算 10 次操作中 x 中介于 m 和 n（不包括）之间的字节数，请使用：

```
#define countbetween(x,m,n) (hasbetween(x,m,n)/128%255)
```

Juha Järvilikelyhasbetween 于 2005 年 4 月 6 日提出建议。Sean Anderson 于 2005 年 4 月 10 日从那里创建了 hasbetween 和 countbetween。

计算字典顺序的下一位排列

假设我们有一个整数模式，其中 N 位设置为 1，并且我们希望在字典意义上进行 N 1 位的下一个排列。例如，如果 N 为 3 且位模式为 00010011，则下一个模式将为 00010101、00010110、00011001、00011010、00011100、00100011 等。以下是计算下一个排列的快速方法。

```
无符号整型 v；// 当前位排列

无符号整数 w；// 下一个比特排列



无符号整数 t = v | (v–1)；// t 将 v 的最低有效位 0 设置为 1

// 接下来将要更改的最高有效位设置为 1，

// 将最低有效位设置为 0，并添加必要的 1 位。

w = (t + 1) | (((~t & –~t) – 1) >> (__builtin_ctz(v) + 1));
```

x86 CPU 的 __builtin_ctz(v) GNU C 编译器内部函数返回尾随零的数量。如果您使用的是适用于 x86 的 Microsoft 编译器，则内在函数是 _BitScanForward。它们都发出 bsf 指令，但等效指令可能适用于其他架构。如果不是，则考虑使用前面提到的计算连续零位的方法之一。
这是另一个版本，由于其除法运算符而往往较慢，但它不需要计算尾随零。

```
无符号整型 t = (v | (v – 1)) + 1;

w = t | ((((t & –t) / (v & –v)) >> 1) – 1);
```

感谢阿根廷的 Dario Sneidermanis，他于 2009 年 11 月 28 日提供了此信息。
白俄罗斯语翻译（由 Webhostingrating 提供）可用。