密码学中常用的位运算 （一）

密码学中，有很多运算量非常大的算法，如果不做大量的优化，可能就无法满足性能要求。今天我们介绍一个非常 简单的算法，就是计算一个uint64 二进制数字中1的个数。

先来看最容易实现的一个算法:

算法1

```
func counOne1(n uint64) int {
    count := 0
    for n != 0 {
        count += int(n & 1)
        n >>= 1
    }
    return count
}
```

这个算法的复杂度是n的位数，比如数字有64位，那么就需要64次循环。 对一个n复杂度的问题，等效的算法是分治算法。

算法2

```
func counOne2(x uint64) int {
    if x == 0 || x == 1 {
        return int(x)
    } else if x == 3 || x == 4 {
        // x= 2 -> 1 , x = 3 -> 2
        return int(x - 1)
    }
    if x >= 1<<32 {
        return counOne2(x>>32) + counOne2(x&0xffffffff)
    } else if x >= 1<<16 {
        return counOne2(x>>16) + counOne2(x&0xffff)
    } else if x >= 1<<8 {
        return counOne2(x>>8) + counOne2(x&0xff)
    } else if x >= 1<<4 {
        return counOne2(x>>4) + counOne2(x&0xf)
    } else if x >= 1<<2 {
        return counOne2(x>>2) + counOne2(x&0x3)
    }
}
```

这个函数理论上来说，和上面的算法复杂度是一样的，但是因为有大量的函数调用，实际上应该会更加的慢. 那么有没有办法不通过递归来实现这个算法呢。我们知道，递归的原因是有大量的临时变量的产生，所以我们要设计 一种方法，把临时变量写到x中，这样就不需要递归函数。我们知道uint64 一共就64位，实际上用

一个字节就能表示 64这个数字，所以，这个算法的本质是，要把长度信息从8个字节，压缩到一个字节。我们先来看两个bit的情况下怎么处理：

2个bit

00 -> 00(0+0)

01 -> 01(0+1)

10 -> 01(1+0)

11 -> 10(1+1)

这样长度的信息还在，但是从最多两个1，变成1个1。

4个bit

四个字节可以分成两个“两个字节”，然后再相加。我们列出一些情况看一下：

0000 -> 00 + 00 -> 0000

0001 -> 00 + 01 -> 0001

1001 -> 01 + 01 -> 0010

1111 -> 10 + 10 -> 0100

长度信息依然包含在数字中

8个bit

00010001 -> 0001 + 0001 -> 0010

11111111 -> 0100 + 0100 -> 1000

16个bit

1111111111111111 -> 1000 + 1000 -> 00010000

32个bit

11111111111111111111111111111111 -> 00010000 + 00010000 = 00100000

64个bit

1111111111111111111111111111111111111111111111111111111111111111->00100000+00100000 = 01000000

再来看一个32个bit的数据的情况下，64个bit的情况下就是两个32个字节相加

x	x	x	x	x	x	x	x	备注
1011	1100	0110	0011	0111	1110	1111	1111	原始数据
0110	1000	0101	0010	0110	1001	1010	1010	相邻2个bit相加

x	x	x	x	x	x	x	x	备注
0011	0010	0010	0010	0011	0011	0100	0100	相邻4个bit相加
0000	0101	0000	0100	0000	0110	0000	1000	相邻8个bit相加
0000	0000	0000	1001	0000	0000	0000	1110	相邻16个bit相加
0000	0000	0000	0000	0000	0000	0001	0111	相邻32个bit相加

算法3

```
func counOne3(x uint64) int {
    x = (x & 0x5555555555555555) + ((x >> 1) & 0x5555555555555555)
    x = (x & 0x3333333333333333) + ((x >> 2) & 0x3333333333333333)
    x = (x & 0x0f0f0f0f0f0f0f0f) + ((x >> 4) & 0x0f0f0f0f0f0f0f0f)
    x = (x & 0x00ff00ff00ff00ff) + ((x >> 8) & 0x00ff00ff00ff00ff)
    x = (x & 0x0000ffff0000ffff) + ((x >> 16) & 0x0000ffff0000ffff)
    x = (x & 0x00000000ffffffff) + ((x >> 32) & 0x00000000ffffffff)
    return int(x)
}
```

注意这里：

0x5555555555555555 // 01010101 ...

0x3333333333333333 // 00110011 ...

0x0f0f0f0f0f0f0f0f // 00001111 ...

0x00ff00ff00ff00ff // 0000000011111111 ...

0x0000ffff0000ffff // 00000000000000001111111111111111 ...

0x00000000ffffffff // 00000000000000000000000000000000111111111111111111111111111111111

通过位运算 0x5555555555555555 取出奇数位 (x >> 1) & 0x5555555555555555 取出偶数位对应我们2bit的情况。 同样的，0x3333333333333333 对应4bit的情况。

最后测试一下性能: 通过位运算可以实现 70倍左右的性能提升。