Section 1.5 Binary Numbers 二进制算法

(我狂晕，这个……明明是不知道谁直接在线翻译，根本不通顺就放上来的，奈何鄙人也不通鸟语，只能将一些我看得懂的句子捋顺一下，望哪位大虾费神放个好的翻译上来) （额……我上学期刚学过口译，试着来完成前辈未竟的事业）

---二进制数转换(基础)---

电脑以 1 和 0 为基础操作的，这些被称为'二进制位'(bit)。 1 字节（Byte）内含有 8 位，例如：00110101。一个整型数据（int）在我的计算机上是 4 个字节，32 位：10011010 11010101 10100010 10101011。其他计算机的字节大小可能不同。

如你所见，32 个 1 和 0 记录下来或阅读起来有点麻烦。 因此，按照惯例人们把数字分成几组，每组 3 或 4 位：

```
1001.1010.1101.0101.1010.0010.1010.1011
10.011.010.110.101.011.010.001.010.101.011 < - (注意，从右往左开始计数 3)
```

这些分组，映射到数字，要么每 4 位表示一个 16 进制数（这里指个位数）或每 3 位表示一个 8 进制数。 很明显，需要一些新的十六进制数字（小数点后数字只去 0 .. 9，我们还需要 6 个数字）。现在，字母'A'..'F'被用来表示这些新的数字：10 .. 15。 下面的这一张表，显而易见地表现了它们的转化方式：

替换：十六进制：

```
000 - > 0  100 - > 4  0000 - > 0  0100 - > 4  1000 - > 8  1100 - > C
001 - > 1  101 - > 5  0001 - > 1  0101 - > 5  1001 - > 9  1101 - > D
010 - > 2  110 - > 6  0010 - > 2  0110 - > 6  1010 - > A  1110 - > E
011 - > 3  111 - > 7  0011 - > 3  0111 - > 7  1011 - > B  1111 - > F
```

所以现在我们很快地用 C 语言或其他语言表示以上陈述那些十六进制和八进制整数：

```
1001.1010.1101.0101.1010.0010.1010.1011
   - >  9    A    D    5    A    2    A    B- > 0x9AD5A2AB
```

（0x 为在十六进制数前面的标志）

```
10.011.010.110.101.011.010.001.010.101.011
       2 3 2 6 5 3 2 1 2 5 3 - > 023265321253
```

（这是前一个数字'0'）八进制的优点是可以比较容易、快速地写下来，但十六进制字节的更容易分离（这是因为数字是成对的）。

---位运算(进阶)---

有时为了效率将数字储存为位数字，而不是存储为整型。 比如在数组中记录选择（每个元素只可以是一个'是'或'不'的状态），用数组对选项进行标记（相同的状态，即'真'，每个位是'是'就是'假'），或者记录一些小的连续整数（例如对位元素 0 .. 3）。 当然，有时问题其实包含'位串'。

在 C/C++和其语言中，如果你知道它的八进制或十六进制表示形式，指定一个二进制数是很容易的： i= 0x9AD5A2AB;或 i= 023265321253; 更常见的是我们会用一个整数的权来记录状态。 比如下面这个例子：

i= 0x10000 + 0x100; 直到同一位上都是 1 之前它都是符合要求的： i= 0x100 + 0x100; 在这种情况下会发生进位，然后就得到了 0x200 而不是我们所希望得到的 0x100。(在此接着前辈的工作翻译下去，修改了前面一些翻译的和原文对比不准确的地方）而 C/C++等语言中的'|'，即“按位或”操作，却能达到我们所希望的要求。“按位或”操作的规则如下：

```
0 | 0 - > 0
0 | 1 - > 1
1 | 0 - > 1
1 | 1 - > 1
```

在 C 语言里'|'的操作称为'按位或'，以免与它的表兄'||'，即所谓的'逻辑或'或'or',混淆。 '||'运算符会计算其左侧的数，如果假（在 C 语言中为 0），再判断其右侧的数。 如果任意一个不为零，那么'||'的结果为真（为 1）。这是将'|'和'+'操作区分开来的最终规则。有时候这样的操作符运算以如下真值表的形式给出：

```
| | 0  1
```

```
---+------
 0 | 0  1
 1 | 1  1
```

显而易见，'按位或'的操作方式可以用来设置记录状态的整数。当任何一方或双方都是'1'时，输出结果为'1'。
最简单的查询方法是'逻辑与'（也称为'andif'）算子，记为'&'。真值表如下：

```
 & | 0 1
---+------
 0 | 0 0
 1 | 0 1
```

只有当输入的两个值均为'1'时，输出值才为'1'。 因此，如果你想知道一个整数的 0x100 位是否为'1'，语句很简单：

```
if (a & 0x100) { printf("yes, 0x100 is on\n"); }
```

C/C++以及其他语言还包含其他的操作符，比如“异或”，用'^'表示，真值表如下：

```
 ^ | 0 1
---+------
 0 | 0 1
 1 | 1 0
```

“异或”有时表示成'xor'，为了打字时比较轻松。 当且仅当输入的两个值之一是'1'时，输出结果才为 1。这个操作符能够很方便的来控制“开关”，即将数字的某一位由'1'变成'0'，或反之亦然。 例如以下这句代码：

```
a = a ^ 0x100; /* same as a ^= 0x100; */
```

在 0x100 位处将从 0 - > 1 或从 1 - > 0，根据其当前的值。
将某个数置零等同于两个基本操作符的运算（译者按：事实上下面是在讲异或的置零功能，即一个数和它本身进行异或操作得到结果是 0，例如 xor ax,ax 是将 ax 寄存器置零）。 我们新介绍一个一元运算符，它将一个数的每一位翻转，以创造一个数的“按位补”或者简称“补码”。这个运算符称为“按位取反”或者简称为“取反”，记为波浪符'~'。 下面是一个简单的例子：

```
char a, b; /* eight bits, not 32 */
 a = 0x4A;   /* 0100.1010 */
 b = ~a;     /* flip every bit: 1011.0101 */
 printf("b is 0x%X\n", b);
```

最终得出了这样的结果：

```
b == 0xB5
```

所以，如果一个数我们只有一位是 1（例如，0x100），那么~0x100 将所有其他的'0'位置'1'而将该位置'0'，得到：0xFFFFFEFF（注意'E'处于右起第三位，和原数的'1'位相同）。
以下两个操作符将一个数完全置零：

```
a = a & (~0x100); /* swtch off the 0x100 bit */
 /* same as a &= ~0x100;
```

因为取反后，原本所有为'0'的都变成了'1'，而所有为'1'的都变成了'0'，所以每一位都保证有 0 的存在再进行按位与操作后，所有的位数都变成了'0'。
总结
总之，这些操作符能够设置，清除，转换和查找整数中的任意一位二进制位：

```
 a |= 0x20;    /* turn on bit 0x20 */
a &= ~0x20;    /* turn off bit 0x20 */
a ^= 0x20;     /* toggle bit 0x20 */
if (a & 0x20) {
                    /* then the 0x20 bit is on */
}
```

（鉴于本人水平有限，如果有译得不准确的地方，欢迎大牛们予以辅正！）