### 3.5.2 位操作符

位操作符用于在最基本的层次上，即按内存中表示数值的位来操作数值。ECMAScript 中的所有数值都以 IEEE-754 64 位格式存储，但位操作符并不直接操作 64 位的值。而是先将 64 位的值转换成 32 位的整数，然后执行操作，最后再将结果转换回 64 位。对于开发人员来说，由于 64 位存储格式是透明的，因此整个过程就像是只存在 32 位的整数一样。

对于有符号的整数，32 位中的前 31 位用于表示整数的值。第 32 位用于表示数值的符号：0 表示正数，1 表示负数。这个表示符号的位叫做**符号位**，符号位的值决定了其他位数值的格式。其中，正数以纯二进制格式存储，31 位中的每一位都表示 2 的幂。第一位（叫做位 0）表示 $2^0$，第二位表示 $2^1$，以此类推。没有用到的位以 0 填充，即忽略不计。例如，数值 18 的二进制表示是 00000000000000000000000000010010，或者更简洁的 10010。这是 5 个有效位，这 5 位本身就决定了实际的值（如图 3-1 所示）。

| 1 | 0 | 0 | 1 | 0 |
|---|---|---|---|---|

$(2^4 \times 1) + (2^3 \times 0) + (2^2 \times 0) + (2^1 \times 1) + (2^0 \times 0)$

默认情况下，ECMAScript 中的所有整数都是有符号整数。不过，当然也存在无符号整数。对于无符号整数来说，第 32 位不再表示符号，因为无符号整数只能是正数。而且，无符号整数的值可以更大，因为多出的一位不再表示符号，可以用来表示数值。

负数同样以二进制码存储，但使用的格式是二进制补码。

#### 1. 按位非（NOT）

按位非操作符由一个波浪线（~）表示，执行按位非的结果就是返回数值的反码。按位非是 ECMAScript 操作符中少数几个与二进制计算有关的操作符之一。下面看一个例子：

```
var num1 = 25;              // 二进制 00000000000000000000000000011001
var num2 = ~num1;           // 二进制 11111111111111111111111111100110
alert(num2);                // -26
```

*BitwiseNotExample01.htm*





这里，对 25 执行按位非操作，结果得到了–26。这也验证了按位非操作的本质：操作数的负值减 1。因此，下面的代码也能得到相同的结果：

```
var num1 = 25;
var num2 = -num1 - 1;
alert(num2);              // "-26"
```

虽然以上代码也能返回同样的结果，但由于按位非是在数值表示的最底层执行操作，因此速度更快。

### 2. 按位与（AND）

按位与操作符由一个和号字符（&）表示，它有两个操作符数。从本质上讲，按位与操作就是将两个数值的每一位对齐，然后根据下表中的规则，对相同位置上的两个数执行 AND 操作：

| 第一个数值的位 | 第二个数值的位 | 结　果 |
|---|---|---|
| 1 | 1 | 1 |
| 1 | 0 | 0 |
| 0 | 1 | 0 |
| 0 | 0 | 0 |

简而言之，按位与操作只在两个数值的对应位都是 1 时才返回 1，任何一位是 0，结果都是 0。

下面看一个对 25 和 3 执行按位与操作的例子：

```
var result = 25 & 3;
alert(result);      //1
```

*BitwiseAndExample01.htm*

可见，对 25 和 3 执行按位与操作的结果是 1。为什么呢？请看其底层操作：

```
 25  = 0000 0000 0000 0000 0000 0000 0001 1001
  3  = 0000 0000 0000 0000 0000 0000 0000 0011
-------------------------------------------
AND = 0000 0000 0000 0000 0000 0000 0000 0001
```

原来，25 和 3 的二进制码对应位上只有一位同时是 1，而其他位的结果自然都是 0，因此最终结果等于 1。

原来，25 和 3 的二进制码对应位上只有一位同时是 1，而其他位的结果自然都是 0，因此最终结果等于 1。

### 3. 按位或（OR）

按位或操作符由一个竖线符号（|）表示，同样也有两个操作数。按位或操作遵循下面这个真值表。

| 第一个数值的位 | 第二个数值的位 | 结　果 |
|---|---|---|
| 1 | 1 | 1 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 0 | 0 | 0 |

由此可见，按位或操作在有一个位是 1 的情况下就返回 1，而只有在两个位都是 0 的情况下才返回 0。

如果在前面按位与的例子中对 25 和 3 执行按位或操作，则代码如下所示：

```
var result = 25 | 3;
alert(result);       //27
```

*BitwiseOrExample01.htm*

### 4. 按位异或（XOR）

按位异或操作符由一个插入符号（^）表示，也有两个操作数。以下是按位异或的真值表。

| 第一个数值的位 | 第二个数值的位 | 结　果 |
|---|---|---|
| 1 | 1 | 0 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 0 | 0 | 0 |

按位异或与按位或的不同之处在于，这个操作在两个数值对应位上只有一个 1 时才返回 1，如果对应的两位都是 1 或都是 0，则返回 0。

对 25 和 3 执行按位异或操作的代码如下所示：

```
var result = 25 ^ 3;
alert(result);    //26
```

### 5. 左移

左移操作符由两个小于号（<<）表示，这个操作符会将数值的所有位向左移动指定的位数。例如，如果将数值 2（二进制码为 10）向左移动 5 位，结果就是 64（二进制码为 1000000），代码如下所示：

```
var oldValue = 2;                    // 等于二进制的 10
var newValue = oldValue << 5;        // 等于二进制的 1000000，十进制的 64
```

*LeftShiftExample01.htm*

注意，在向左移位后，原数值的右侧多出了 5 个空位。左移操作会以 0 来填充这些空位，以便得到的结果是一个完整的 32 位二进制数（见图 3-2）。

### 6. 有符号的右移

有符号的右移操作符由两个大于号（>>）表示，这个操作符会将数值向右移动，但保留符号位（即正负号标记）。有符号的右移操作与左移操作恰好相反，即如果将 64 向右移动 5 位，结果将变回 2：

```
var oldValue = 64;                   // 等于二进制的 1000000
var newValue = oldValue >> 5;        // 等于二进制的 10 ，即十进制的 2
```

*SignedRightShiftExample01.htm*

图　3-3

### 7. 无符号右移

无符号右移操作符由 3 个大于号（>>>）表示，这个操作符会将数值的所有 32 位都向右移动。对正数来说，无符号右移的结果与有符号右移相同。仍以前面有符号右移的代码为例，如果将 64 无符号右移 5 位，结果仍然还是 2：

```
var oldValue = 64;                   // 等于二进制的 1000000
var newValue = oldValue >>> 5;       // 等于二进制的 10 ，即十进制的 2
```

*UnsignedRightShiftExample01.htm*