Overview of Bit Operations

Computers use binary, which consists of two numbers: 0 and 1. All the basic operations are realized through bit operations. There are six types of bit operations, namely: AND, OR, XOR, negation, left shift, and right shift. Left shift and right shift are collectively referred to as shift operations, and shift operations are further divided into arithmetic shift and logical shift. Among these bit operations, only the negation is a unary operation, and the rest are binary operations.

AND, OR, XOR, and Negation

The symbol of the AND operation is `&`, and the operation rule is:

> For each binary bit, when the corresponding bits of both numbers are 1, the result is 1; Otherwise, the result is 0.

- 0 & 0 = 0
- 0 & 1 = 0
- 1 & 0 = 0
- 1 & 1 = 1

The symbol of the OR operation is `|`, and the operation rule is:

> For each binary bit, when the corresponding bits of both numbers are 0, the result is 0; Otherwise, the result is 1.

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

The symbol of the XOR operation is ⊕ (in the code, we usually use ^ to represent XOR), and the operation rule is:

> For each binary bit, when the corresponding bits of the two numbers are the same, the result is 0; Otherwise, the result is 1.

- $0 \oplus 0 = 0$
- $0 \oplus 1 = 1$
- $1 \oplus 0 = 1$
- $1 \oplus 1 = 0$

The symbol of the negation operation is ~, and the operation rule is:

> Flip each binary bit of a number: 0 becomes 1, and 1 becomes 0.

- $\sim 0 = 1$
- $\sim 1 = 0$

The following examples show the results of the four bitwise operators, and the numbers involved in the operation are all represented by signed **8-bit** binary numbers. The binary representation of 46 is 00101110, and the binary representation of 51 is 00110011. Try it yourself to get the result of the following bitwise operations.

- The result of 46 & 51 is 34, which corresponds to 00100010 in binary.
- The result of 46 | 51 is 63, which corresponds to 00111111 in binary.
- The result of $46 \oplus 51$ is 29, which corresponds to 00011101 in binary.
- The result of ~ 46 is -47, which corresponds to 11010001 in binary.
- The result of ~ 51 is -52, which corresponds to 11001100 in binary.

Shift operation

The shift operation can be divided into the left shift and the right shift according to the shift direction and can be divided into arithmetic shift and logical shift according to whether it is signed.

The symbol for the left shift operation is <<. In a left shift operation, all binary bits are shifted to the left by several bits, the high bits are discarded, and the low bits are filled with 0. For left shift operations, arithmetic shift and logical shift are the same.

The symbol for the right shift operator is >>. In a right shift operation, all binary bits are shifted to the right by several bits, the low bits are discarded, and how the high bits get filled differs between arithmetic shift and logical shift:

- When shifting right arithmetically, the high bits are filled with the highest bit;
- When shifting right logically, the high bits are filled with 0.

The following example shows the results of shift operations. The numbers involved in the operations are all represented in signed 8-bit binary numbers.

The binary representation of 29 is 00011101. Left shift of 29 by 2 bits results in 116, and the corresponding binary representation is 01110100; Shifting 29 to the left by 3 bits results in −24, which corresponds to 11101000 in binary.

The binary representation of 50 is 00110010. The result of 1-bit right shift of 11 bits is 25, which corresponds to 00011001 in binary; Shifting 50 to the right by 2 bits is 12, which corresponds to 00001100. For non-negative numbers, the arithmetic right shift and logical right shift are identical.

The binary representation of −50 is 11001110. The result of an arithmetic right shift of −50 by 2 bits is −13, and the corresponding binary representation is 11110011; the result of a logical right shift of −50 by 2 bits is 51, and the corresponding binary representation is 00110011. Results of arithmetic right shift and logical right shift are different. Which one is the right shift operation used in computers?

In C/C++, there are both signed and unsigned data types, where signed types are declared using the keyword `signed`, and the unsigned types are declared using the keyword `unsigned`. When neither keyword is used, the default is a signed type. For signed types, the right shift operation is an arithmetic right shift; for unsigned types, the right shift operation is a logical right shift. In Java, there is no unsigned data type, integers are always represented as signed types, so it is necessary to distinguish between arithmetic right shift and logical right shift. In Java, the symbol for an arithmetic right shift is >>, and the symbol for a logical right shift is >>>.

Relationship between shift operations and multiplication/division

By observing the examples above, we can see that shift operations are closely related to multiplication and division. Since all calculations in computers are implemented with bit operations, the efficiency of multiplication and division using shift operations is significantly higher than that of directly using multiplication and division.

The left shift operation corresponds to the multiplication. Shifting a number to the left by k bits is equivalent to multiplying the number by `2^k`. For example, left shifting 29 by 2 bits results in 116, which is equivalent to `29 × 4`. When the multiplier is not an integer power of 2, the multiplier can be split into the sum of the integer powers of 2. For example, `a × 6` is equivalent to `( a << 2 ) + ( a << 1 )`. For any integer, multiplication can be implemented by the left shift operation, but you need to be careful with the overflow situation. For example, in the 8-bit binary representation, 29 will overflow by shifting 3 bits to the left.

The arithmetic right shift operation corresponds to the division operation. Shifting a number to the right by k bits is equivalent to dividing the number by `2^k`. For example, right shifting 50 by 2 bits results in 12, which is equivalent to `50 / 4`, rounded down.

For implementation, could we conclude that, the arithmetic right shift of a number by k digit is equivalent to performing integer division by `2^k`? For non-negative numbers, the above statement is true. Integer division is rounded to 0, and right shift operation is rounded down, which is also rounded to 0. But for negative numbers, the above statement no longer holds. Integer division is rounded to 0, while the right shift is rounded down. These two are not the same. For example, the result of `(−50) >> 2` is −13, and the result of `(−50) / 4` is −12, which are not equal. Therefore, the arithmetic right shift of a number by k bits is not equivalent to dividing the number by `2^k`.

Properties of bitwise operations

Bit operations have many properties. Here, we will list common properties of AND, OR, XOR, and negation in bit operations. We assume that the variables below are all signed integers.

- Idempotent law: a & a = a, a | a = a (note that XOR does not satisfy the idempotent law);
- Commutative law: a & b = b & a, a | b = b | a, $a \oplus b = b \oplus a$;
- Associativity: (a & b) & c = a & (b & c), (a | b) | c = a | (b | c), $(a \oplus b) \oplus c = a \oplus (b \oplus c)$;
- Distributive Law: (a & b) | c = (a | c) & (b | c), (a | b) & c = (a & c) | (b & c), $(a \oplus b)$ & $c = (a$ & $c) \oplus (b$ & $c)$;
- De Morgan's Law: ~ (a & b) = (~a) | (~b), ~ (a | b) = (~a) & (~b);
- Negative operation properties: -1 = ~0, -a = ~(a–1);
- AND operation properties: a & 0 = 0, a & (-1) = a, a & (~a)=0;
- OR operation properties: a | 0 = a, a | (~a) = –1;
- XOR operation properties: $a \oplus 0 = a, \quad a \oplus a = 0$;
- Other properties: The result of a & (a–1) is to change the last 1 in the binary representation of `a` to 0; The result of a & (-a) (equivalent to a & (~(a–1))) is to keep only the last 1 of the binary representation of `a`,

and set the remaining 1s to 0. Using these properties, many bit operation problems can be solved strategically.