Побитовые операторы

Побитовые операторы интерпретируют операнды как последовательность из 32 битов (нулей и единиц). Они производят операции, используя двоичное представление числа, и возвращают новую последовательность из 32 бит (число) в качестве результата.

Эта глава требует дополнительных знаний в программировании и не очень важная, при первом чтении вы можете пропустить её и вернуться потом, когда захотите понять, как побитовые операторы работают.

Формат 32-битного целого числа со знаком

Побитовые операторы в JavaScript работают с 32-битными целыми числами в их двоичном представлении.

Это представление называется «32-битное целое со знаком, старшим битом слева и дополнением до двойки».

Разберём, как устроены числа внутри подробнее, это необходимо знать для битовых операций с ними.

- Что такое двоичная система счисления, вам, надеюсь, уже известно. При разборе побитовых операций мы будем обсуждать именно двоичное представление чисел, из 32 бит.
- *Старший бит слева* – это научное название для самого обычного порядка записи цифр (от большего разряда к меньшему). При этом, если больший разряд отсутствует, то соответствующий бит равен нулю.

  Примеры представления чисел в двоичной системе:

```
a = 0;   // 00000000000000000000000000000000
a = 1;   // 00000000000000000000000000000001
a = 2;   // 00000000000000000000000000000010
a = 3;   // 00000000000000000000000000000011
a = 255;// 00000000000000000000000011111111
```

  Обратите внимание, каждое число состоит ровно из 32-битов.

- *Дополнение до двойки* – это название способа поддержки отрицательных чисел.

  Двоичный вид числа, обратного данному (например, `5` и `-5`) получается путём обращения всех битов с прибавлением 1.

  То есть, нули заменяются на единицы, единицы – на нули и к числу прибавляется `1`. Получается внутреннее представление того же числа, но со знаком минус.

  Например, вот число `314`:

```
00000000000000000000000100111010
```

  Чтобы получить `-314`, первый шаг – обратить биты числа: заменить `0` на `1`, а `1` на `0`:

```
11111111111111111111111011000101
```

  Второй шаг – к полученному двоичному числу прибавить единицу, обычным двоичным сложением: `11111111111111111111111011000101 + 1 = 11111111111111111111111011000110`.

  Итак, мы получили:

```
1   -314 = 11111111111111111111111011000110
```


Принцип дополнения до двойки делит все двоичные представления на два множества: если крайний-левый бит равен `0` – число положительное, если `1` – число отрицательное. Поэтому этот бит называется *знаковым битом*.

Список операторов

В следующей таблице перечислены все побитовые операторы. Далее операторы разобраны более подробно.

Оператор	Использование	Описание
Побитовое И (AND)	`a & b`	Ставит 1 на бит результата, для которого соответствующие биты операндов равны 1.
Побитовое ИЛИ (OR)	`a | b`	Ставит 1 на бит результата, для которого хотя бы один из соответствующих битов операндов равен 1.
Побитовое исключающее ИЛИ (XOR)	`a ^ b`	Ставит 1 на бит результата, для которого только один из соответствующих битов операндов равен 1 (но не оба).
Побитовое НЕ (NOT)	`~a`	Заменяет каждый бит операнда на противоположный.
Левый сдвиг	`a << b`	Сдвигает двоичное представление `a` на `b` битов влево, добавляя справа нули.
Правый сдвиг, переносящий знак	`a >> b`	Сдвигает двоичное представление `a` на `b` битов вправо, отбрасывая сдвигаемые биты.
Правый сдвиг с заполнением нулями	`a >>> b`	Сдвигает двоичное представление `a` на `b` битов вправо, отбрасывая сдвигаемые биты и добавляя нули слева.

Побитовые операторы работают следующим образом:

1. Операнды преобразуются в 32-битные целые числа, представленные последовательностью битов. Дробная часть, если она есть, отбрасывается.
2. Для бинарных операторов – каждый бит в первом операнде рассматривается вместе с соответствующим битом второго операнда: первый бит с первым, второй со вторым и т.п. Оператор применяется к каждой паре бит, давая соответствующий бит результата.
3. Получившаяся в результате последовательность бит интерпретируется как обычное число.

Посмотрим, как работают операторы, на примерах.

Вспомогательные функции parseInt, toString

Для удобной работы с примерами в этой статье, если вы захотите протестировать что-то в консоли, пригодятся две функции.

- `parseInt("11000", 2)` – переводит строку с двоичной записью числа в число.
- `n.toString(2)` – получает для числа `n` запись в 2-ной системе в виде строки.

Например:

```
var access = parseInt("11000", 2); // получаем чис

alert( access ); // 24, число с таким 2-ным предст

var access2 = access.toString(2); // обратно двои

alert( access2 ); // 11000
```

Без них перевод в двоичную систему и обратно был бы куда менее удобен. Более подробно они разбираются в главе Числа.

& (Побитовое И)

Выполняет операцию И над каждой парой бит.

Результат `a & b` равен единице только когда оба бита `a` и `b` равны единице.

Таблица истинности для `&`:

a	b	a & b
0	0	0
0	1	0
1	0	0
1	1	1

Пример:

```
 9 (по осн. 10)
   = 00000000000000000000000000001001 (по осн. 2)
14 (по осн. 10)
   = 00000000000000000000000000001110 (по осн. 2)
                    --------------------------------
14 & 9 (по осн. 10)
   = 00000000000000000000000000001000 (по осн. 2)
   = 8 (по осн. 10)
```

| (Побитовое ИЛИ)

Выполняет операцию ИЛИ над каждой парой бит. Результат `a | b` равен 1, если хотя бы один бит из `a`, `b` равен 1.

Таблица истинности для `|`:

a	b	a | b
0	0	0
0	1	1
1	0	1
1	1	1

Пример:

```
 9 (по осн. 10)
   = 00000000000000000000000000001001 (по осн. 2)
14 (по осн. 10)
   = 00000000000000000000000000001110 (по осн. 2)
                    --------------------------------
14 | 9 (по осн. 10)
   = 00000000000000000000000000001111 (по осн. 2)
   = 15 (по осн. 10)
```

^ (Исключающее ИЛИ)

Выполняет операцию «Исключающее ИЛИ» над каждой парой бит.

`a` Исключающее ИЛИ `b` равно 1, если только `a=1` или только `b=1`, но не оба одновременно `a=b=1`.

Таблица истинности для исключающего ИЛИ:

a	b	a ^ b
0	0	0
0	1	1
1	0	1
1	1	0

Как видно, оно даёт 1, если ЛИБО слева `1`, ЛИБО справа `1`, но не одновременно. Поэтому его и называют «исключающее ИЛИ».

Пример:

```
 9 (по осн. 10)
   = 00000000000000000000000000001001 (по осн. 2)
14 (по осн. 10)
   = 00000000000000000000000000001110 (по осн. 2)
                    --------------------------------
14 ^ 9 (по осн. 10)
   = 00000000000000000000000000000111 (по осн. 2)
   = 7 (по осн. 10)
```

Исключающее ИЛИ в шифровании

Исключающее или можно использовать для шифрования, так как эта операция полностью обратима. Двойное применение исключающего ИЛИ с тем же аргументом даёт исходное число.

Иначе говоря, верна формула: `a ^ b ^ b == a`.

Пусть Вася хочет передать Пете секретную информацию `data`. Эта информация заранее превращена в число, например строка интерпретируется как последовательность кодов символов.

Вася и Петя заранее договариваются о числовом ключе шифрования `key`.

Алгоритм:

- Вася берёт двоичное представление `data` и делает операцию `data ^ key`. При необходимости `data` бьётся на части, равные по длине `key`, чтобы можно было провести побитовое ИЛИ `^` для каждой части. В JavaScript оператор `^` работает с 32-битными целыми числами, так что `data` нужно разбить на последовательность таких чисел.
- Результат `data ^ key` отправляется Пете, это шифровка.

Например, пусть в `data` очередное число равно `9`, а ключ `key` равен `1220461917`.

```
Данные: 9 в двоичном виде
00000000000000000000000000001001

Ключ: 1220461917 в двоичном виде
01001000101111101100010101011101

Результат операции 9 ^ key:
01001000101111101100010101010100
Результат в 10-ной системе (шифровка):
1220461908
```

- Петя, получив очередное число шифровки `1220461908`, применяет к нему такую же операцию `^ key`.
- Результатом будет исходное число `data`.

В нашем случае:

```
Полученная шифровка в двоичной системе:
9 ^ key = 1220461908
01001000101111101100010101010100

Ключ: 1220461917 в двоичном виде:
01001000101111101100010101011101

Результат операции 1220461917 ^ key:
00000000000000000000000000001001
Результат в 10-ной системе (исходное сообщение):
9
```

Конечно, такое шифрование поддаётся частотному анализу и другим методам дешифровки, поэтому современные алгоритмы используют операцию XOR `^` как одну из важных частей более сложной многоступенчатой схемы.

~ (Побитовое НЕ)

Производит операцию НЕ над каждым битом, заменяя его на обратный ему.

Таблица истинности для НЕ:

a	~a
0	1
1	0

Пример:

```
 9 (по осн. 10)
  = 00000000000000000000000000001001 (по осн. 2)
                 --------------------------------
~9 (по осн. 10)
  = 11111111111111111111111111110110 (по осн. 2)
  = -10 (по осн. 10)
```

Из-за внутреннего представления отрицательных чисел получается так, что `~n == -(n+1)`.

Например:

```
alert( ~3 ); // -4
alert( ~-1 ); // 0
```

<< (Битовый сдвиг влево)

Операторы битового сдвига принимают два операнда. Первый – это число для сдвига, а второй – количество битов, которые нужно сдвинуть в первом операнде.

Оператор `<<` сдвигает первый операнд на указанное число битов влево. Лишние биты отбрасываются, справа добавляются нулевые биты.

Например, `9 << 2` даст `36`:

```
9 (по осн.10)
  = 00000000000000000000000000001001 (по осн.2)
                 --------------------------------
9 << 2 (по осн.10)
  = 00000000000000000000000000100100 (по осн.2)
  = 36 (по осн.10)
```

Операция `<< 2` сдвинула и отбросила два левых нулевых бита и добавила справа два новых нулевых.

Левый сдвиг почти равен умножению на 2

Битовый сдвиг `<< N` обычно имеет тот же эффект, что и умножение на два `N` раз, например:

```
alert( 3 << 1 ); // 6, умножение на 2
alert( 3 << 2 ); // 12, умножение на 2 два раза
alert( 3 << 3 ); // 24, умножение на 2 три раза
```

Конечно, следует иметь в виду, что побитовые операторы работают только с 32-битными числами, поэтому верхний порог такого «умножения» ограничен:

```
alert(10000000000 << 1); // -1474836480, отброшен
alert(10000000000 * 2); // 20000000000, обычное ум
```

>> (Правый битовый сдвиг, переносящий знак)

Этот оператор сдвигает биты вправо, отбрасывая лишние. При этом слева добавляется *копия* крайнего-левого бита.

Знак числа (представленный крайним-левым битом) при этом не меняется, так как новый крайний-левый бит имеет то же значение, что и исходном числе.

Поэтому он назван «переносящим знак».

Например, `9 >> 2` даст `2`:

```
9 (по осн.10)
  = 00000000000000000000000000001001 (по осн.2)
                 --------------------------------
9 >> 2 (по осн.10)
  = 00000000000000000000000000000010 (по осн.2)
  = 2 (по осн.10)
```

Операция `>> 2` сдвинула вправо и отбросила два правых бита `01` и добавила слева две копии первого бита `00`.

Аналогично, `-9 >> 2` даст `-3`:

```
-9 (по осн.10)
  = 11111111111111111111111111110111 (по осн.2)
                 --------------------------------
-9 >> 2 (по осн.10)
  = 11111111111111111111111111111101 (по осн.2) = -3 (п
```

Здесь операция `>> 2` сдвинула вправо и отбросила два правых бита `11` и добавила слева две копии первого бита `11`., Знак числа сохранён, так как крайний-левый (знаковый) бит сохранил значение `1`.

Правый сдвиг почти равен целочисленному делению на 2

Битовый сдвиг `>> N` обычно имеет тот же результат, что и целочисленное деление на два `N` раз:

```
alert( 100 >> 1 ); // 50, деление на 2
alert( 100 >> 2 ); // 25, деление на 2 два раза
alert( 100 >> 3 ); // 12, деление на 2 три раза, и
```

>>> (Правый сдвиг с заполнением нулями)

Этот оператор сдвигает биты первого операнда вправо. Лишние биты справа отбрасываются. Слева добавляются нулевые биты.

Знаковый бит становится равным 0, поэтому результат всегда положителен.

Для неотрицательных чисел правый сдвиг с заполнением нулями `>>>` и правый сдвиг с переносом знака `>>` дадут одинаковый результат, т.к. в обоих случаях слева добавятся нули.

Для отрицательных чисел – результат работы разный. Например, `-9 >>> 2` даст `1073741821`, в отличие от `-9 >> 2` (даёт `-3`):

```
-9 (по осн.10)
  = 11111111111111111111111111110111 (по осн.2)
                  --------------------------------
-9 >>> 2 (по осн.10)
  = 00111111111111111111111111111101 (по осн.2)
  = 1073741821 (по осн.10)
```

Применение побитовых операторов

Побитовые операторы используются редко, но всё же используются.

Случаи применения побитовых операторов, которые мы здесь разберём, составляют большинство всех использований в JavaScript.

> **Осторожно, приоритеты!**
>
> В JavaScript побитовые операторы `^`, `&`, `|` выполняются после сравнений `==`.
>
> Например, в сравнении `a == b^0` будет сначала выполнено сравнение `a == b`, а потом уже операция `^0`, как будто стоят скобки `(a == b)^0`.
>
> Обычно это не то, чего мы хотим. Чтобы гарантировать желаемый порядок, нужно ставить скобки: `a == (b^0)`.

Маска

Для этого примера представим, что наш скрипт работает с пользователями.

У них могут быть различные роли в проекте:

- `Гость`
- `Редактор`
- Админ

Каждой роли соответствует ряд доступов к статьям и функциональности сайта.

Например, `Гость` может лишь просматривать статьи сайта, а `Редактор` – ещё и редактировать их, и тому подобное.

Что-то в таком духе:

Пользователь	Просмотр статей	Изменение статей	Просмотр товаров	Изменение товаров	Управление правами
Гость	Да	Нет	Да	Нет	Нет
Редактор	Да	Да	Да	Да	Нет
Админ	Да	Да	Да	Да	Да

Если вместо «Да» поставить `1`, а вместо «Нет» – `0`, то каждый набор доступов описывается числом:

Пользователь	Просмотр статей	Изменение статей	Просмотр товаров	Изменение товаров	Управление правами	В 10-ной системе
Гость	1	0	1	0	0	= 20
Редактор	1	1	1	1	0	= 30
Админ	1	1	1	1	1	= 31

В последней колонке находится десятичное число, которое получится, если прочитать строку доступов в двоичном виде.

Например, доступ гостя `10100 = 20`.

Такая интерпретация доступов позволяет «упаковать» много информации в одно число. Это экономит память, а кроме этого – это удобно, поскольку в дополнение к экономии – по такому значению очень легко проверить, имеет ли посетитель заданную *комбинацию доступов*!

Для этого посмотрим, как в 2-ной системе представляется каждый доступ в отдельности.

- Доступ, соответствующий только управлению правами: `00001 (=1)` (все нули кроме `1` на позиции, соответствующей этому доступу).
- Доступ, соответствующий только изменению товаров: `00010 (=2)`.
- Доступ, соответствующий только просмотру товаров: `00100 (=4)`.
- Доступ, соответствующий только изменению статей: `01000 (=8)`.

- Доступ, соответствующий только просмотру статей: `10000` (=16).

Доступ одновременно на просмотр и изменение статей – это двоичное число с `1` на соответствующих позициях, то есть `access = 11000`.

Как правило, доступы задаются в виде констант:

```
var ACCESS_ADMIN = 1;          // 00001
var ACCESS_GOODS_EDIT = 2;   // 00010
var ACCESS_GOODS_VIEW = 4;     // 00100
var ACCESS_ARTICLE_EDIT = 8; // 01000
var ACCESS_ARTICLE_VIEW = 16;  // 10000
```

Из этих констант получить нужную комбинацию доступов можно при помощи операции `|`.

```
var guest = ACCESS_ARTICLE_VIEW | ACCESS_GOODS_VIEW; //
var editor = guest | ACCESS_ARTICLE_EDIT | ACCESS_GOODS
var admin = editor | ACCESS_ADMIN; // 11111
```

Теперь, чтобы понять, есть ли в доступе `editor` нужный доступ, например управление правами – достаточно применить к нему побитовый оператор И (`&`) с соответствующей константой.

Ненулевой результат будет означать, что доступ есть:

```
alert(editor & ACCESS_ADMIN); // 0, доступа нет
alert(editor & ACCESS_ARTICLE_EDIT); // 8, доступ есть
```

Такая проверка работает, потому что оператор И ставит `1` на те позиции результата, на которых в обоих операндах стоит `1`.

Можно проверить один из нескольких доступов.

Например, проверим, есть ли права на просмотр ИЛИ изменение товаров. Соответствующие права задаются битом `1` на втором и третьем месте с конца, что даёт число `00110` (= `6` в 10-ной системе).

```
var check = ACCESS_GOODS_VIEW | ACCESS_GOODS_EDIT; // 6

alert( admin & check ); // не 0, значит есть доступ к п
```

Битовой маской называют как раз комбинацию двоичных значений (`check` в примере выше), которая используется для проверки и выборки единиц на нужных позициях.

Маски могут быть весьма удобны.

В частности, их используют в функциях, чтобы одним параметром передать несколько «флагов», т.е. однобитных значений.

Пример вызова функции с маской:

```
// найти пользователей с правами на изменение товаров и
findUsers(ACCESS_GOODS_EDIT | ACCESS_ADMIN);
```

Это довольно-таки коротко и элегантно, но, вместе с тем, применение масок налагает определённые ограничения. В частности, побитовые операторы в JavaScript работают только с 32-битными числами, а значит, к примеру, 33 доступа уже в число не упакуешь. Да и работа с двоичной системой счисления – как ни крути, менее удобна, чем с десятичной или с обычными логическими значениями `true/false`.

Поэтому основная сфера применения масок – это быстрые вычисления, экономия памяти, низкоуровневые операции, связанные с рисованием из JavaScript (3d-графика), интеграция с некоторыми функциями ОС (для

серверного JavaScript), и другие ситуации, когда уже существуют функции, требующие битовую маску.

Округление

Так как битовые операции отбрасывают десятичную часть, то их можно использовать для округления. Достаточно взять любую операцию, которая не меняет значение числа.

Например, двойное НЕ (~):

```
1  alert( ~~12.345 ); // 12
```

Подойдёт и Исключающее ИЛИ (^) с нулём:

```
1  alert( 12.345 ^ 0 ); // 12
```

Последнее даже более удобно, поскольку отлично читается:

```
1  alert(12.3 * 14.5 ^ 0); // (=178) "12.3 умножить на 14.
```

У побитовых операторов достаточно низкий приоритет, он меньше чем у остальной арифметики:

```
1  alert( 1.1 + 1.2 ^ 0 ); // 2, сложение выполнится раньш
```

Проверка на −1

Внутренний формат 32-битных чисел устроен так, что для смены знака нужно все биты заменить на противоположные («обратить») и прибавить `1` .

Обращение битов – это побитовое НЕ (~). То есть, при таком формате представления числа `-n = ~n + 1` . Или, если перенести единицу: `~n = -(n+1)` .

Как видно из последнего равенства, `~n == 0` только если `n == -1` . Поэтому можно легко проверить равенство `n == -1` :

```
1  var n = 5;
2
3  if (~n) { // сработает, т.к. ~n = -(5+1) = -6
4    alert( "n не -1" ); // выведет!
5  }
```

```
1  var n = -1;
2
3  if (~n) { // не сработает, т.к. ~n = -(-1+1) = 0
4    alert( "...ничего не выведет..." );
5  }
```

Проверка на `-1` пригождается, например, при поиске символа в строке. Вызов `str.indexOf("подстрока")` возвращает позицию подстроки в `str` , или `-1` если не нашёл.

```
1  var str = "Проверка";
2
3  if (~str.indexOf("верка")) { // Сочетание "if (~...inde
4    alert( 'найдено!' );
5  }
```

Умножение и деление на степени 2

Оператор `a << b`, сдвигая биты, по сути умножает `a` на `2` в степени `b`.

Например:

```
alert( 1 << 2 ); // 1*(2*2) = 4
alert( 1 << 3 ); // 1*(2*2*2) = 8
alert( 3 << 3 ); // 3*(2*2*2) = 24
```

При этом следует иметь в виду, что максимальный верхний порог такого умножения меньше, чем обычно, так как побитовый оператор манипулирует 32-битными целыми, в то время как обычные операторы работают с числами длиной 64 бита.

Оператор сдвига в другую сторону `a >> b`, производит обратную операцию – целочисленное деление `a` на 2^b.

```
alert( 8 >> 2 ); // 2 = 8/4, убрали 2 нуля в двоичном п
alert( 11 >> 2 ); // 2, целочисленное деление (менее зн
```

Итого

- Бинарные побитовые операторы: `& | ^ << >> >>>`.
- Унарный побитовый оператор один: `~`.

Как правило, битовое представление числа используется для:

- Округления числа: `(12.34^0) = 12`.
- Проверки на равенство `-1`: `if (~n) { n не -1 }`.
- Упаковки нескольких битововых значений («флагов») в одно значение. Это экономит память и позволяет проверять наличие комбинации флагов одним оператором `&`.
- Других ситуаций, когда нужны битовые маски.

Задачи

Побитовый оператор и значение

важность: 5

Почему побитовые операции в примерах ниже не меняют число? Что они делают внутри?

```
alert( 123 ^ 0 ); // 123
alert( 0 ^ 123 ); // 123
alert( ~~123 ); // 123
```

решение

Проверка, целое ли число

важность: 3

Напишите функцию `isInteger(num)`, которая возвращает `true`, если `num` – целое число, иначе `false`.

Например:

```
alert( isInteger(1) ); // true
alert( isInteger(1.5) ); // false
alert( isInteger(-0.5) ); // false
```

решение

Симметричны ли операции ^, |, &?

важность: 5

Верно ли, что для любых `a` и `b` выполняются равенства ниже?

- `(a ^ b) == (b ^ a)`
- `(a & b) == (b & a)`
- `(a | b) == (b | a)`

Иными словами, при перемене мест – всегда ли результат останется тем же?

решение

Почему результат разный?

важность: 5

Почему результат второго `alert'a` такой странный?

```
1  alert( 123456789 ^ 0 ); // 123456789
2  alert( 12345678912345 ^ 0 ); // 1942903641
```

решение

Комментарии

перед тем как писать...