# 对象

## 概述

### 生成方法

对象（object）是 JavaScript 语言的核心概念，也是最重要的数据类型。

什么是对象？简单说，对象就是一组“键值对”（key-value）的集合，是一种无序的复合数据集合。

```javascript
var obj = {
  foo: 'Hello',
  bar: 'World'
};
```

上面代码中，大括号就定义了一个对象，它被赋值给变量`obj`，所以变量`obj`就指向一个对象。该对象内部包含两个键值对（又称为两个“成员”），第一个键值对是`foo: 'Hello'`，其中`foo`是“键名”（成员的名称），字符串`Hello`是“键值”（成员的值）。键名与键值之间用冒号分隔。第二个键值对是`bar: 'World'`，`bar`是键名，`World`是键值。两个键值对之间用逗号分隔。

### 键名

对象的所有键名都是字符串（ES6 又引入了 Symbol 值也可以作为键名），所以加不加引号都可以。上面的代码也可以写成下面这样。

```javascript
var obj = {
  'foo': 'Hello',
  'bar': 'World'
};
```

如果键名是数值，会被自动转为字符串。

```javascript
var obj = {
  1: 'a',
  3.2: 'b',
  1e2: true,
  1e-2: true,
  .234: true,
  0xFF: true
};

obj
// Object {
```

```
//   1: "a",
//   3.2: "b",
//   100: true,
//   0.01: true,
//   0.234: true,
//   255: true
// }

obj['100'] // true
```

上面代码中，对象`obj`的所有键名虽然看上去像数值，实际上都被自动转成了字符串。

如果键名不符合标识名的条件（比如第一个字符为数字，或者含有空格或运算符），且也不是数字，则必须加上引号，否则会报错。

```javascript
// 报错
var obj = {
  1p: 'Hello World'
};

// 不报错
var obj = {
  '1p': 'Hello World',
  'h w': 'Hello World',
  'p+q': 'Hello World'
};
```

上面对象的三个键名，都不符合标识名的条件，所以必须加上引号。

对象的每一个键名又称为“属性”（property），它的“键值”可以是任何数据类型。如果一个属性的值为函数，通常把这个属性称为“方法”，它可以像函数那样调用。

```javascript
var obj = {
  p: function (x) {
    return 2 * x;
  }
};

obj.p(1) // 2
```

上面代码中，对象`obj`的属性`p`，就指向一个函数。

如果属性的值还是一个对象，就形成了链式引用。

```javascript
```

```
var o1 = {};
var o2 = { bar: 'hello' };

o1.foo = o2;
o1.foo.bar // "hello"
```

上面代码中，对象`o1`的属性`foo`指向对象`o2`，就可以链式引用`o2`的属性。

对象的属性之间用逗号分隔，最后一个属性后面可以加逗号（trailing comma），也可以不加。

```javascript
var obj = {
  p: 123,
  m: function () { ... },
}
```

上面的代码中，`m`属性后面的那个逗号，有没有都可以。

属性可以动态创建，不必在对象声明时就指定。

```javascript
var obj = {};
obj.foo = 123;
obj.foo // 123
```

上面代码中，直接对`obj`对象的`foo`属性赋值，结果就在运行时创建了`foo`属性。

### 对象的引用

如果不同的变量名指向同一个对象，那么它们都是这个对象的引用，也就是说指向同一个内存地址。修改其中一个变量，会影响到其他所有变量。

```javascript
var o1 = {};
var o2 = o1;

o1.a = 1;
o2.a // 1

o2.b = 2;
o1.b // 2
```

上面代码中，`o1`和`o2`指向同一个对象，因此为其中任何一个变量添加属性，另一个变量都可以读写该属性。

此时，如果取消某一个变量对于原对象的引用，不会影响到另一个变量。

```javascript
var o1 = {};
var o2 = o1;

o1 = 1;
o2 // {}
```

上面代码中，`o1`和`o2`指向同一个对象，然后`o1`的值变为1，这时不会对`o2`产生影响，`o2`还是指向原来的那个对象。

但是，这种引用只局限于对象，如果两个变量指向同一个原始类型的值。那么，变量这时都是值的拷贝。

```javascript
var x = 1;
var y = x;

x = 2;
y // 1
```

上面的代码中，当`x`的值发生变化后，`y`的值并不变，这就表示`y`和`x`并不是指向同一个内存地址。

### 表达式还是语句?

对象采用大括号表示，这导致了一个问题：如果行首是一个大括号，它到底是表达式还是语句?

```javascript
{ foo: 123 }
```

JavaScript 引擎读到上面这行代码，会发现可能有两种含义。第一种可能是，这是一个表达式，表示一个包含`foo`属性的对象；第二种可能是，这是一个语句，表示一个代码区块，里面有一个标签`foo`，指向表达式`123`。

为了避免这种歧义，JavaScript 引擎的做法是，如果遇到这种情况，无法确定是对象还是代码块，一律解释为代码块。

```javascript
{ console.log(123) } // 123
```

上面的语句是一个代码块，而且只有解释为代码块，才能执行。

如果要解释为对象，最好在大括号前加上圆括号。因为圆括号的里面，只能是表达式，所以确保大括号只能解释为对象。

```javascript
({ foo: 123 }) // 正确
({ console.log(123) }) // 报错
```

这种差异在`eval`语句（作用是对字符串求值）中反映得最明显。

```javascript
eval('{foo: 123}') // 123
eval('({foo: 123})') // {foo: 123}
```

上面代码中，如果没有圆括号，`eval`将其理解为一个代码块；加上圆括号以后，就理解成一个对象。

## 属性的操作

### 属性的读取

读取对象的属性，有两种方法，一种是使用点运算符，还有一种是使用方括号运算符。

```javascript
var obj = {
  p: 'Hello World'
};

obj.p // "Hello World"
obj['p'] // "Hello World"
```

上面代码分别采用点运算符和方括号运算符，读取属性`p`。

请注意，如果使用方括号运算符，键名必须放在引号里面，否则会被当作变量处理。

```javascript
var foo = 'bar';

var obj = {
  foo: 1,
  bar: 2
};

obj.foo  // 1
obj[foo]  // 2
```

上面代码中，引用对象`obj`的`foo`属性时，如果使用点运算符，`foo`就是字符串；如果使用方括号运算符，但是不使用引号，那么`foo`就是一个变量，指向字符串`bar`。

方括号运算符内部还可以使用表达式。

```javascript
obj['hello' + ' world']
obj[3 + 3]
```

数字键可以不加引号，因为会自动转成字符串。

```javascript
var obj = {
  0.7: 'Hello World'
};

obj['0.7'] // "Hello World"
obj[0.7] // "Hello World"
```

上面代码中，对象`obj`的数字键`0.7`，加不加引号都可以，因为会被自动转为字符串。

注意，数值键名不能使用点运算符（因为会被当成小数点），只能使用方括号运算符。

```javascript
var obj = {
  123: 'hello world'
};

obj.123 // 报错
obj[123] // "hello world"
```

上面代码的第一个表达式，对数值键名`123`使用点运算符，结果报错。第二个表达式使用方括号运算符，结果就是正确的。

### 属性的赋值

点运算符和方括号运算符，不仅可以用来读取值，还可以用来赋值。

```javascript
var obj = {};

obj.foo = 'Hello';
obj['bar'] = 'World';
```

上面代码中，分别使用点运算符和方括号运算符，对属性赋值。

JavaScript 允许属性的“后绑定”，也就是说，你可以在任意时刻新增属性，没必要在定义对象的时候，就定义好属性。

```javascript
var obj = { p: 1 };

// 等价于

var obj = {};
obj.p = 1;
```

### 属性的查看

查看一个对象本身的所有属性，可以使用`Object.keys`方法。

```javascript
var obj = {
  key1: 1,
  key2: 2
};

Object.keys(obj);
// ['key1', 'key2']
```

### 属性的删除：delete 命令

`delete`命令用于删除对象的属性，删除成功后返回`true`。

```javascript
var obj = { p: 1 };
Object.keys(obj) // ["p"]

delete obj.p // true
obj.p // undefined
Object.keys(obj) // []
```

上面代码中，`delete`命令删除对象`obj`的`p`属性。删除后，再读取`p`属性就会返回`undefined`，而且`Object.keys`方法的返回值也不再包括该属性。

注意，删除一个不存在的属性，`delete`不报错，而且返回`true`。

```javascript
var obj = {};
delete obj.p // true
```

上面代码中，对象`obj`并没有`p`属性，但是`delete`命令照样返回`true`。因此，不能根据`delete`命令的结果，认定某个属性是存在的。

只有一种情况，`delete`命令会返回`false`，那就是该属性存在，且不得删除。

```javascript
var obj = Object.defineProperty({}, 'p', {
  value: 123,
  configurable: false
});

obj.p // 123
delete obj.p // false
```

上面代码之中，对象`obj`的`p`属性是不能删除的，所以`delete`命令返回`false`（关于`Object.defineProperty`方法的介绍，请看《标准库》的 Object 对象一章）。

另外，需要注意的是，`delete`命令只能删除对象本身的属性，无法删除继承的属性（关于继承参见《面向对象编程》章节）。

```javascript
var obj = {};
delete obj.toString // true
obj.toString // function toString() { [native code] }
```

上面代码中，`toString`是对象`obj`继承的属性，虽然`delete`命令返回`true`，但该属性并没有被删除，依然存在。这个例子还说明，即使`delete`返回`true`，该属性依然可能读取到值。

### 属性是否存在：in 运算符

`in`运算符用于检查对象是否包含某个属性（注意，检查的是键名，不是键值），如果包含就返回`true`，否则返回`false`。它的左边是一个字符串，表示属性名，右边是一个对象。

```javascript
var obj = { p: 1 };
'p' in obj // true
'toString' in obj // true
```

`in`运算符的一个问题是，它不能识别哪些属性是对象自身的，哪些属性是继承的。就像上面代码中，对象`obj`本身并没有`toString`属性，但是`in`运算符会返回`true`，因为这个属性是继承的。

这时，可以使用对象的`hasOwnProperty`方法判断一下，是否为对象自身的属性。

```javascript
var obj = {};
```

```
if ('toString' in obj) {
  console.log(obj.hasOwnProperty('toString')) // false
}
```

### 属性的遍历：for...in 循环

`for...in`循环用来遍历一个对象的全部属性。

```javascript
var obj = {a: 1, b: 2, c: 3};

for (var i in obj) {
  console.log('键名：', i);
  console.log('键值：', obj[i]);
}
// 键名： a
// 键值： 1
// 键名： b
// 键值： 2
// 键名： c
// 键值： 3
```

`for...in`循环有两个使用注意点。

- 它遍历的是对象所有可遍历（enumerable）的属性，会跳过不可遍历的属性。
- 它不仅遍历对象自身的属性，还遍历继承的属性。

举例来说，对象都继承了`toString`属性，但是`for...in`循环不会遍历到这个属性。

```javascript
var obj = {};

// toString 属性是存在的
obj.toString // toString() { [native code] }

for (var p in obj) {
  console.log(p);
} // 没有任何输出
```

上面代码中，对象`obj`继承了`toString`属性，该属性不会被`for...in`循环遍历到，因为它默认是“不可遍历”的。关于对象属性的可遍历性，参见《标准库》章节中 Object 一章的介绍。

如果继承的属性是可遍历的，那么就会被`for...in`循环遍历到。但是，一般情况下，都是只想遍历对象自身的属性，所以使用`for...in`的时候，应该结合使用`hasOwnProperty`方法，在循环内部判断一下，某个属性是否为对象自身的属性。

```javascript
var person = { name: '老张' };

for (var key in person) {
  if (person.hasOwnProperty(key)) {
    console.log(key);
  }
}
// name
```

## with 语句

`with`语句的格式如下：

```javascript
with (对象) {
  语句;
}
```

它的作用是操作同一个对象的多个属性时，提供一些书写的方便。

```javascript
// 例一
var obj = {
  p1: 1,
  p2: 2,
};
with (obj) {
  p1 = 4;
  p2 = 5;
}
// 等同于
obj.p1 = 4;
obj.p2 = 5;

// 例二
with (document.links[0]){
  console.log(href);
  console.log(title);
  console.log(style);
}
// 等同于
console.log(document.links[0].href);
```

```
console.log(document.links[0].title);
console.log(document.links[0].style);
```

注意，如果`with`区块内部有变量的赋值操作，必须是当前对象已经存在的属性，否则会创造一个当前作用域的全局变量。

```javascript
var obj = {};
with (obj) {
  p1 = 4;
  p2 = 5;
}

obj.p1 // undefined
p1 // 4
```

上面代码中，对象`obj`并没有`p1`属性，对`p1`赋值等于创造了一个全局变量`p1`。正确的写法应该是，先定义对象`obj`的属性`p1`，然后在`with`区块内操作它。

这是因为`with`区块没有改变作用域，它的内部依然是当前作用域。这造成了`with`语句的一个很大的弊病，就是绑定对象不明确。

```javascript
with (obj) {
  console.log(x);
}
```

单纯从上面的代码块，根本无法判断`x`到底是全局变量，还是对象`obj`的一个属性。这非常不利于代码的除错和模块化，编译器也无法对这段代码进行优化，只能留到运行时判断，这就拖慢了运行速度。因此，建议不要使用`with`语句，可以考虑用一个临时变量代替`with`。

```javascript
with(obj1.obj2.obj3) {
  console.log(p1 + p2);
}

// 可以写成
var temp = obj1.obj2.obj3;
console.log(temp.p1 + temp.p2);
```

## 参考链接

- Dr. Axel Rauschmayer，[Object properties in JavaScript](http://www.2ality.com/2012/10/javascript-properties.html)

- Lakshan Perera, [Revisiting JavaScript Objects](http://www.laktek.com/2012/12/29/revisiting-javascript-objects/)
- Angus Croll, [The Secret Life of JavaScript Primitives](http://javascriptweblog.wordpress.com/2010/09/27/the-secret-life-of-javascript-primitives/)i
- Dr. Axel Rauschmayer, [JavaScript's with statement and why it's deprecated](http://www.2ality.com/2011/06/with-statement.html)