

Object 对象的相关方法

JavaScript 在`Object`对象上面，提供了很多相关方法，处理面向对象编程的相关操作。本章介绍这些方法。

Object.getPrototypeOf()

`Object.getPrototypeOf`方法返回参数对象的原型。这是获取原型对象的标准方法。

```
```javascript
var F = function () {};
var f = new F();
Object.getPrototypeOf(f) === F.prototype // true
```
```

上面代码中，实例对象`f`的原型是`F.prototype`。

下面是几种特殊对象的原型。

```
```javascript
// 空对象的原型是 Object.prototype
Object.getPrototypeOf({}) === Object.prototype // true

// Object.prototype 的原型是 null
Object.getPrototypeOf(Object.prototype) === null // true

// 函数的原型是 Function.prototype
function f() {}
Object.getPrototypeOf(f) === Function.prototype // true
```
```

Object.setPrototypeOf()

`Object.setPrototypeOf`方法为参数对象设置原型，返回该参数对象。它接受两个参数，第一个是现有对象，第二个是原型对象。

```
```javascript
var a = {};
var b = {x: 1};
Object.setPrototypeOf(a, b);

Object.getPrototypeOf(a) === b // true
a.x // 1
```
```

上面代码中，`Object.setPrototypeOf`方法将对象`a`的原型，设置为对象`b`，因此`a`可以共享`b`的属性。

`new`命令可以使用`Object.setPrototypeOf`方法模拟。

```

```javascript
var F = function () {
 this.foo = 'bar';
};

var f = new F();
// 等同于
var f = Object.setPrototypeOf({}, F.prototype);
F.call(f);
```

```

上面代码中，`new`命令新建实例对象，其实可以分成两步。第一步，将一个空对象的原型设为构造函数的`prototype`属性（上例是`F.prototype`）；第二步，将构造函数内部的`this`绑定这个空对象，然后执行构造函数，使得定义在`this`上面的方法和属性（上例是`this.foo`），都转移到这个空对象上。

Object.create()

生成实例对象的常用方法是，使用`new`命令让构造函数返回一个实例。但是很多时候，只能拿到一个实例对象，它可能根本不是由构造函数生成的，那么能不能从一个实例对象，生成另一个实例对象呢？

JavaScript 提供了`Object.create`方法，用来满足这种需求。该方法接受一个对象作为参数，然后以它为原型，返回一个实例对象。该实例完全继承原型对象的属性。

```

```javascript
// 原型对象
var A = {
 print: function () {
 console.log('hello');
 }
};

// 实例对象
var B = Object.create(A);

Object.getPrototypeOf(B) === A // true
B.print() // hello
B.print === A.print // true
```

```

上面代码中，`Object.create`方法以`A`对象为原型，生成了`B`对象。`B`继承了`A`的所有属性和方法。

实际上，`Object.create`方法可以用下面的代码代替。

```

```javascript

```

```

if (typeof Object.create !== 'function') {
 Object.create = function (obj) {
 function F() {}
 F.prototype = obj;
 return new F();
 };
}

```

上面代码表明，`Object.create`方法的实质是新建一个空的构造函数`F`，然后让`F.prototype`属性指向参数对象`obj`，最后返回一个`F`的实例，从而实现让该实例继承`obj`的属性。

下面三种方式生成的新对象是等价的。

```

```javascript
var obj1 = Object.create({});
var obj2 = Object.create(Object.prototype);
var obj3 = new Object();

```

如果想要生成一个不继承任何属性（比如没有`toString`和`valueOf`方法）的对象，可以将`Object.create`的参数设为`null`。

```

```javascript
var obj = Object.create(null);

obj.valueOf()
// TypeError: Object [object Object] has no method 'valueOf'

```

上面代码中，对象`obj`的原型是`null`，它就不具备一些定义在`Object.prototype`对象上面的属性，比如`valueOf`方法。

使用`Object.create`方法的时候，必须提供对象原型，即参数不能为空，或者不是对象，否则会报错。

```

```javascript
Object.create()
// TypeError: Object prototype may only be an Object or null
Object.create(123)
// TypeError: Object prototype may only be an Object or null

```

`Object.create`方法生成的新对象，动态继承了原型。在原型上添加或修改任何方法，会立刻反映在新对象之上。

```

```javascript
var obj1 = { p: 1 };
var obj2 = Object.create(obj1);

```

```
obj1.p = 2;
obj2.p // 2
````
```

上面代码中，修改对象原型`obj1`会影响到实例对象`obj2`。

除了对象的原型，`Object.create`方法还可以接受第二个参数。该参数是一个属性描述对象，它所描述的对象属性，会添加到实例对象，作为该对象自身的属性。

```
````javascript  
var obj = Object.create({}, {
 p1: {
 value: 123,
 enumerable: true,
 configurable: true,
 writable: true,
 },
 p2: {
 value: 'abc',
 enumerable: true,
 configurable: true,
 writable: true,
 }
});
````
```

```
// 等同于  
var obj = Object.create({});  
obj.p1 = 123;  
obj.p2 = 'abc';  
````
```

`Object.create`方法生成的对象，继承了它的原型对象的构造函数。

```
````javascript  
function A() {}  
var a = new A();  
var b = Object.create(a);  
  
b.constructor === A // true  
b instanceof A // true  
````
```

上面代码中，`b`对象的原型是`a`对象，因此继承了`a`对象的构造函数`A`。

## Object.prototype.isPrototypeOf()

实例对象的`isPrototypeOf`方法，用来判断该对象是否为参数对象的原型。

```
````javascript  
var o1 = {};  
var o2 = Object.create(o1);  
````
```

```

var o3 = Object.create(o2);

o2.isPrototypeOf(o3) // true
o1.isPrototypeOf(o3) // true

```

上面代码中，`o1`和`o2`都是`o3`的原型。这表明只要实例对象处在参数对象的原型链上，`isPrototypeOf`方法都返回`true`。

```

```javascript
Object.prototype.isPrototypeOf({}) // true
Object.prototype.isPrototypeOf([]) // true
Object.prototype.isPrototypeOf(/xyz/) // true
Object.prototype.isPrototypeOf(Object.create(null)) // false

```

上面代码中，由于`Object.prototype`处于原型链的最顶端，所以对各种实例都返回`true`，只有直接继承自`null`的对象除外。

Object.prototype.__proto__

实例对象的`__proto__`属性（前后各两个下划线），返回该对象的原型。该属性可读写。

```

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

obj.__proto__ = p;
Object.getPrototypeOf(obj) === p // true

```

上面代码通过`\_\_proto\_\_`属性，将`p`对象设为`obj`对象的原型。

根据语言标准，`\_\_proto\_\_`属性只有浏览器才需要部署，其他环境可以没有这个属性。它前后的两根下划线，表明它本质是一个内部属性，不应该对使用者暴露。因此，应该尽量少用这个属性，而是用`Object.getPrototypeOf()`和`Object.setPrototypeOf()`，进行原型对象的读写操作。

原型链可以用`\_\_proto\_\_`很直观地表示。

```

```javascript
var A = {
  name: '张三'
};
var B = {
  name: '李四'
};

var proto = {
  print: function () {

```

```

    console.log(this.name);
  }
};

A.__proto__ = proto;
B.__proto__ = proto;

A.print() // 张三
B.print() // 李四

A.print === B.print // true
A.print === proto.print // true
B.print === proto.print // true

```

上面代码中，`A`对象和`B`对象的原型都是`proto`对象，它们都共享`proto`对象的`print`方法。也就是说，`A`和`B`的`print`方法，都是在调用`proto`对象的`print`方法。

获取原型对象方法的比较

如前所述，`__proto__`属性指向当前对象的原型对象，即构造函数的`prototype`属性。

```

```javascript
var obj = new Object();

obj.__proto__ === Object.prototype
// true
obj.__proto__ === obj.constructor.prototype
// true

```

上面代码首先新建了一个对象`obj`，它的`\_\_proto\_\_`属性，指向构造函数（`Object`或`obj.constructor`）的`prototype`属性。

因此，获取实例对象`obj`的原型对象，有三种方法。

```

- `obj.__proto__`
- `obj.constructor.prototype`
- `Object.getPrototypeOf(obj)`

```

上面三种方法之中，前两种都不是很可靠。`\_\_proto\_\_`属性只有浏览器才需要部署，其他环境可以不部署。而`obj.constructor.prototype`在手动改变原型对象时，可能会失效。

```

```javascript
var P = function () {};
var p = new P();

var C = function () {};
C.prototype = p;
var c = new C();

```

```
c.constructor.prototype === p // false
```
```

上面代码中，构造函数`C`的原型对象被改成了`p`，但是实例对象的`c.constructor.prototype`却没有指向`p`。所以，在改变原型对象时，一般要同时设置`constructor`属性。

```
```javascript
C.prototype = p;
C.prototype.constructor = C;

var c = new C();
c.constructor.prototype === p // true
```
```

因此，推荐使用第三种`Object.getPrototypeOf`方法，获取原型对象。

```
Object.getPrototypeOf()
```

`Object.getPrototypeOf`方法返回一个数组，成员是参数对象本身的所有属性的键名，不包含继承的属性键名。

```
```javascript
Object.getPrototypeOf(Date)
// ["parse", "arguments", "UTC", "caller", "name", "prototype", "now", "length"]
```
```

上面代码中，`Object.getPrototypeOf`方法返回`Date`所有自身的属性名。

对象本身的属性之中，有的是可以遍历的（enumerable），有的是不可以遍历的。

`Object.getPrototypeOf`方法返回所有键名，不管是否可以遍历。只获取那些可以遍历的属性，使用`Object.keys`方法。

```
```javascript
Object.keys(Date) // []
```
```

上面代码表明，`Date`对象所有自身的属性，都是不可以遍历的。

```
Object.prototype.hasOwnProperty()
```

对象实例的`hasOwnProperty`方法返回一个布尔值，用于判断某个属性定义在对象自身，还是定义在原型链上。

```
```javascript
Date.hasOwnProperty('length') // true
Date.hasOwnProperty('toString') // false
```
```

上面代码表明，`Date.length`（构造函数`Date`可以接受多少个参数）是`Date`自身的属性，`Date.toString`是继承的属性。

另外，`hasOwnProperty`方法是 JavaScript 之中唯一一个处理对象属性时，不会遍历原型链的方法。

## in 运算符和 for...in 循环

`in`运算符返回一个布尔值，表示一个对象是否具有某个属性。它不区分该属性是对象自身的属性，还是继承的属性。

```
```javascript
'length' in Date // true
'toString' in Date // true
```
```

`in`运算符常用于检查一个属性是否存在。

获得对象的所有可遍历属性（不管是自身的还是继承的），可以使用`for...in`循环。

```
```javascript
var o1 = { p1: 123 };

var o2 = Object.create(o1, {
  p2: { value: "abc", enumerable: true }
});

for (p in o2) {
  console.info(p);
}
// p2
// p1
```
```

上面代码中，对象`o2`的`p2`属性是自身的，`p1`属性是继承的。这两个属性都会被`for...in`循环遍历。

为了在`for...in`循环中获得对象自身的属性，可以采用`hasOwnProperty`方法判断一下。

```
```javascript
for ( var name in object ) {
  if ( object.hasOwnProperty(name) ) {
    /* loop code */
  }
}
```
```

获得对象的所有属性（不管是自身的还是继承的，也不管是否可枚举），可以使用下面的函数。



```

```javascript
function inheritedPropertyNames(obj) {
  var props = {};
  while(obj) {
    Object.getOwnPropertyNames(obj).forEach(function(p) {
      props[p] = true;
    });
    obj = Object.getPrototypeOf(obj);
  }
  return Object.getOwnPropertyNames(props);
}
```

```

上面代码依次获取`obj`对象的每一级原型对象“自身”的属性，从而获取`obj`对象的“所有”属性，不管是否可遍历。

下面是一个例子，列出`Date`对象的所有属性。

```

```javascript
inheritedPropertyNames(Date)
// [
//   "caller",
//   "constructor",
//   "toString",
//   "UTC",
//   ...
// ]
```

```

## ## 对象的拷贝

如果要拷贝一个对象，需要做到下面两件事情。

- 确保拷贝后的对象，与原对象具有同样的原型。
- 确保拷贝后的对象，与原对象具有同样的实例属性。

下面就是根据上面两点，实现的对象拷贝函数。

```

```javascript
function copyObject(orig) {
  var copy = Object.create(Object.getPrototypeOf(orig));
  copyOwnPropertiesFrom(copy, orig);
  return copy;
}

function copyOwnPropertiesFrom(target, source) {
  Object
    .getOwnPropertyNames(source)
    .forEach(function (propKey) {
      var desc = Object.getOwnPropertyDescriptor(source, propKey);
      Object.defineProperty(target, propKey, desc);
    });
}
```

```

```
});
return target;
}
```

另一种更简单的写法，是利用 ES2017 才引入标准的`Object.getOwnPropertyDescriptors`方法。

```
```javascript  
function copyObject(orig) {  
  return Object.create(  
    Object.getPrototypeOf(orig),  
    Object.getOwnPropertyDescriptors(orig)  
  );  
}
```

参考链接

- Dr. Axel Rauschmayer, [JavaScript properties: inheritance and enumerability](<http://www.2ality.com/2011/07/js-properties.html>)