属性描述对象

概述

JavaScript 提供了一个内部数据结构，用来描述对象的属性，控制它的行为，比如该属性是否可写、可遍历等等。这个内部数据结构称为“属性描述对象”（attributes object）。每个属性都有自己对应的属性描述对象，保存该属性的一些元信息。

下面是属性描述对象的一个例子。

```javascript
{
  value: 123,
  writable: false,
  enumerable: true,
  configurable: false,
  get: undefined,
  set: undefined
}
```

属性描述对象提供6个元属性。

（1）`value`

`value`是该属性的属性值，默认为`undefined`。

（2）`writable`

`writable`是一个布尔值，表示属性值（value）是否可改变（即是否可写），默认为`true`。

（3）`enumerable`

`enumerable`是一个布尔值，表示该属性是否可遍历，默认为`true`。如果设为`false`，会使得某些操作（比如`for...in`循环、`Object.keys()`）跳过该属性。

（4）`configurable`

`configurable`是一个布尔值，表示可配置性，默认为`true`。如果设为`false`，将阻止某些操作改写该属性，比如无法删除该属性，也不得改变该属性的属性描述对象（`value`属性除外）。也就是说，`configurable`属性控制了属性描述对象的可写性。

（5）`get`

`get`是一个函数，表示该属性的取值函数（getter），默认为`undefined`。

（6）`set`

`set`是一个函数，表示该属性的存值函数（setter），默认为`undefined`。

Object.getOwnPropertyDescriptor()

`Object.getOwnPropertyDescriptor()`方法可以获取属性描述对象。它的第一个参数是目标对象，第二个参数是一个字符串，对应目标对象的某个属性名。

```javascript
var obj = { p: 'a' };

Object.getOwnPropertyDescriptor(obj, 'p')
// Object { value: "a",
//   writable: true,
//   enumerable: true,
//   configurable: true
// }
```

上面代码中，`Object.getOwnPropertyDescriptor()`方法获取`obj.p`的属性描述对象。

注意，`Object.getOwnPropertyDescriptor()`方法只能用于对象自身的属性，不能用于继承的属性。

```javascript
var obj = { p: 'a' };

Object.getOwnPropertyDescriptor(obj, 'toString')
// undefined
```

上面代码中，`toString`是`obj`对象继承的属性，`Object.getOwnPropertyDescriptor()`无法获取。

Object.getOwnPropertyNames()

`Object.getOwnPropertyNames`方法返回一个数组，成员是参数对象自身的全部属性的属性名，不管该属性是否可遍历。

```javascript
var obj = Object.defineProperties({}, {
  p1: { value: 1, enumerable: true },
  p2: { value: 2, enumerable: false }
});

Object.getOwnPropertyNames(obj)
// ["p1", "p2"]
```

上面代码中，`obj.p1`是可遍历的，`obj.p2`是不可遍历的。`Object.getOwnPropertyNames`会将它们都返回。

这跟`Object.keys`的行为不同，`Object.keys`只返回对象自身的可遍历属性的全部属性名。

```javascript
Object.keys([]) // []
Object.getOwnPropertyNames([]) // [ 'length' ]

Object.keys(Object.prototype) // []
Object.getOwnPropertyNames(Object.prototype)
// ['hasOwnProperty',
//  'valueOf',
//  'constructor',
//  'toLocaleString',
//  'isPrototypeOf',
//  'propertyIsEnumerable',
//  'toString']
```

上面代码中，数组自身的`length`属性是不可遍历的，`Object.keys`不会返回该属性。第二个例子的`Object.prototype`也是一个对象，所有实例对象都会继承它，它自身的属性都是不可遍历的。

Object.defineProperty()，Object.defineProperties()

`Object.defineProperty()`方法允许通过属性描述对象，定义或修改一个属性，然后返回修改后的对象，它的用法如下。

```javascript
Object.defineProperty(object, propertyName, attributesObject)
```

`Object.defineProperty`方法接受三个参数，依次如下。

- object：属性所在的对象
- propertyName：字符串，表示属性名
- attributesObject：属性描述对象

举例来说，定义`obj.p`可以写成下面这样。

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

obj.p // 123
```

```
obj.p = 246;
obj.p // 123
```

上面代码中，`Object.defineProperty()`方法定义了`obj.p`属性。由于属性描述对象的`writable`属性为`false`，所以`obj.p`属性不可写。注意，这里的`Object.defineProperty`方法的第一个参数是`{}`（一个新建的空对象），`p`属性直接定义在这个空对象上面，然后返回这个对象，这是`Object.defineProperty()`的常见用法。

如果属性已经存在，`Object.defineProperty()`方法相当于更新该属性的属性描述对象。

如果一次性定义或修改多个属性，可以使用`Object.defineProperties()`方法。

```javascript
var obj = Object.defineProperties({}, {
  p1: { value: 123, enumerable: true },
  p2: { value: 'abc', enumerable: true },
  p3: { get: function () { return this.p1 + this.p2 },
    enumerable:true,
    configurable:true
  }
});

obj.p1 // 123
obj.p2 // "abc"
obj.p3 // "123abc"
```

上面代码中，`Object.defineProperties()`同时定义了`obj`对象的三个属性。其中，`p3`属性定义了取值函数`get`，即每次读取该属性，都会调用这个取值函数。

注意，一旦定义了取值函数`get`（或存值函数`set`），就不能将`writable`属性设为`true`，或者同时定义`value`属性，否则会报错。

```javascript
var obj = {};

Object.defineProperty(obj, 'p', {
  value: 123,
  get: function() { return 456; }
});
// TypeError: Invalid property.
// A property cannot both have accessors and be writable or have a value

Object.defineProperty(obj, 'p', {
  writable: true,
  get: function() { return 456; }
});
// TypeError: Invalid property descriptor.
```

```
// Cannot both specify accessors and a value or writable attribute
```

上面代码中，同时定义了`get`属性和`value`属性，以及将`writable`属性设为`true`，就会报错。

`Object.defineProperty()`和`Object.defineProperties()`参数里面的属性描述对象，`writable`、`configurable`、`enumerable`这三个属性的默认值都为`false`。

```javascript
var obj = {};
Object.defineProperty(obj, 'foo', {});
Object.getOwnPropertyDescriptor(obj, 'foo')
// {
//   value: undefined,
//   writable: false,
//   enumerable: false,
//   configurable: false
// }
```

上面代码中，定义`obj.foo`时用了一个空的属性描述对象，就可以看到各个元属性的默认值。

Object.prototype.propertyIsEnumerable()

实例对象的`propertyIsEnumerable()`方法返回一个布尔值，用来判断某个属性是否可遍历。注意，这个方法只能用于判断对象自身的属性，对于继承的属性一律返回`false`。

```javascript
var obj = {};
obj.p = 123;

obj.propertyIsEnumerable('p') // true
obj.propertyIsEnumerable('toString') // false
```

上面代码中，`obj.p`是可遍历的，而`obj.toString`是继承的属性。

元属性

属性描述对象的各个属性称为“元属性”，因为它们可以看作是控制属性的属性。

value

`value`属性是目标属性的值。

```javascript
var obj = {};
obj.p = 123;

Object.getOwnPropertyDescriptor(obj, 'p').value
```

```
// 123

Object.defineProperty(obj, 'p', { value: 246 });
obj.p // 246
```

上面代码是通过`value`属性，读取或改写`obj.p`的例子。

writable

`writable`属性是一个布尔值，决定了目标属性的值（value）是否可以被改变。

```javascript
var obj = {};

Object.defineProperty(obj, 'a', {
  value: 37,
  writable: false
});

obj.a // 37
obj.a = 25;
obj.a // 37
```

上面代码中，`obj.a`的`writable`属性是`false`。然后，改变`obj.a`的值，不会有任何效果。

注意，正常模式下，对`writable`为`false`的属性赋值不会报错，只会默默失败。但是，严格模式下会报错，即使对`a`属性重新赋予一个同样的值。

```javascript
'use strict';
var obj = {};

Object.defineProperty(obj, 'a', {
  value: 37,
  writable: false
});

obj.a = 37;
// Uncaught TypeError: Cannot assign to read only property 'a' of object
```

上面代码是严格模式，对`obj.a`任何赋值行为都会报错。

如果原型对象的某个属性的`writable`为`false`，那么子对象将无法自定义这个属性。

```javascript
var proto = Object.defineProperty({}, 'foo', {
  value: 'a',
  writable: false
```

```
});

var obj = Object.create(proto);

obj.foo = 'b';
obj.foo // 'a'
```

上面代码中，`proto`是原型对象，它的`foo`属性不可写。`obj`对象继承`proto`，也不可以再自定义这个属性了。如果是严格模式，这样做还会抛出一个错误。

但是，有一个规避方法，就是通过覆盖属性描述对象，绕过这个限制。原因是这种情况下，原型链会被完全忽视。

```javascript
var proto = Object.defineProperty({}, 'foo', {
  value: 'a',
  writable: false
});

var obj = Object.create(proto);
Object.defineProperty(obj, 'foo', {
  value: 'b'
});

obj.foo // "b"
```

enumerable

`enumerable`（可遍历性）返回一个布尔值，表示目标属性是否可遍历。

JavaScript 的早期版本，`for...in`循环是基于`in`运算符的。我们知道，`in`运算符不管某个属性是对象自身的还是继承的，都会返回`true`。

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

上面代码中，`toString`不是`obj`对象自身的属性，但是`in`运算符也返回`true`，这导致了`toString`属性也会被`for...in`循环遍历。

这显然不太合理，后来就引入了“可遍历性”这个概念。只有可遍历的属性，才会被`for...in`循环遍历，同时还规定`toString`这一类实例对象继承的原生属性，都是不可遍历的，这样就保证了`for...in`循环的可用性。

具体来说，如果一个属性的`enumerable`为`false`，下面三个操作不会取到该属性。

- `for..in`循环
- `Object.keys`方法
- `JSON.stringify`方法

因此，`enumerable`可以用来设置“秘密”属性。

```javascript
var obj = {};

Object.defineProperty(obj, 'x', {
  value: 123,
  enumerable: false
});

obj.x // 123

for (var key in obj) {
  console.log(key);
}
// undefined

Object.keys(obj)  // []
JSON.stringify(obj) // "{}"
```

上面代码中，`obj.x`属性的`enumerable`为`false`，所以一般的遍历操作都无法获取该属性，使得它有点像“秘密”属性，但不是真正的私有属性，还是可以直接获取它的值。

注意，`for...in`循环包括继承的属性，`Object.keys`方法不包括继承的属性。如果需要获取对象自身的所有属性，不管是否可遍历，可以使用`Object.getOwnPropertyNames`方法。

另外，`JSON.stringify`方法会排除`enumerable`为`false`的属性，有时可以利用这一点。如果对象的 JSON 格式输出要排除某些属性，就可以把这些属性的`enumerable`设为`false`。

configurable

`configurable`(可配置性）返回一个布尔值，决定了是否可以修改属性描述对象。也就是说，`configurable`为`false`时，`value`、`writable`、`enumerable`和`configurable`都不能被修改了。

```javascript
var obj = Object.defineProperty({}, 'p', {
  value: 1,
  writable: false,
  enumerable: false,
  configurable: false
});

Object.defineProperty(obj, 'p', {value: 2})
// TypeError: Cannot redefine property: p
```

```
Object.defineProperty(obj, 'p', {writable: true})
// TypeError: Cannot redefine property: p

Object.defineProperty(obj, 'p', {enumerable: true})
// TypeError: Cannot redefine property: p

Object.defineProperty(obj, 'p', {configurable: true})
// TypeError: Cannot redefine property: p
```

上面代码中，`obj.p`的`configurable`为`false`。然后，改动`value`、`writable`、`enumerable`、`configurable`，结果都报错。

注意，`writable`只有在`false`改为`true`会报错，`true`改为`false`是允许的。

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

Object.defineProperty(obj, 'p', {writable: false})
// 修改成功
```

至于`value`，只要`writable`和`configurable`有一个为`true`，就允许改动。

```javascript
var o1 = Object.defineProperty({}, 'p', {
  value: 1,
  writable: true,
  configurable: false
});

Object.defineProperty(o1, 'p', {value: 2})
// 修改成功

var o2 = Object.defineProperty({}, 'p', {
  value: 1,
  writable: false,
  configurable: true
});

Object.defineProperty(o2, 'p', {value: 2})
// 修改成功
```

另外，`writable`为`false`时，直接目标属性赋值，不报错，但不会成功。

```javascript
```

```
var obj = Object.defineProperty({}, 'p', {
  value: 1,
  writable: false,
  configurable: false
});

obj.p = 2;
obj.p // 1
```

上面代码中，`obj.p`的`writable`为`false`，对`obj.p`直接赋值不会生效。如果是严格模式，还会报错。

可配置性决定了目标属性是否可以被删除（delete）。

```javascript
var obj = Object.defineProperties({}, {
  p1: { value: 1, configurable: true },
  p2: { value: 2, configurable: false }
});

delete obj.p1 // true
delete obj.p2 // false

obj.p1 // undefined
obj.p2 // 2
```

上面代码中，`obj.p1`的`configurable`是`true`，所以可以被删除，`obj.p2`就无法删除。

存取器

除了直接定义以外，属性还可以用存取器（accessor）定义。其中，存值函数称为`setter`，使用属性描述对象的`set`属性；取值函数称为`getter`，使用属性描述对象的`get`属性。

一旦对目标属性定义了存取器，那么存取的时候，都将执行对应的函数。利用这个功能，可以实现许多高级特性，比如某个属性禁止赋值。

```javascript
var obj = Object.defineProperty({}, 'p', {
  get: function () {
    return 'getter';
  },
  set: function (value) {
    console.log('setter: ' + value);
  }
});

obj.p // "getter"
obj.p = 123 // "setter: 123"
```

```

上面代码中，`obj.p`定义了`get`和`set`属性。`obj.p`取值时，就会调用`get`；赋值时，就会调用`set`。

JavaScript 还提供了存取器的另一种写法。

```javascript
var obj = {
  get p() {
    return 'getter';
  },
  set p(value) {
    console.log('setter: ' + value);
  }
};
```

上面的写法与定义属性描述对象是等价的，而且使用更广泛。

注意，取值函数`get`不能接受参数，存值函数`set`只能接受一个参数（即属性的值）。

存取器往往用于，属性的值依赖对象内部数据的场合。

```javascript
var obj ={
  $n : 5,
  get next() { return this.$n++ },
  set next(n) {
    if (n >= this.$n) this.$n = n;
    else throw new Error('新的值必须大于当前值');
  }
};

obj.next // 5

obj.next = 10;
obj.next // 10

obj.next = 5;
// Uncaught Error: 新的值必须大于当前值
```

上面代码中，`next`属性的存值函数和取值函数，都依赖于内部属性`$n`。

对象的拷贝

有时，我们需要将一个对象的所有属性，拷贝到另一个对象，可以用下面的方法实现。

```javascript
```
```

```
var extend = function (to, from) {
  for (var property in from) {
    to[property] = from[property];
  }

  return to;
}

extend({}, {
  a: 1
})
// {a: 1}
```

上面这个方法的问题在于，如果遇到存取器定义的属性，会只拷贝值。

```javascript
extend({}, {
  get a() { return 1 }
})
// {a: 1}
```

为了解决这个问题，我们可以通过`Object.defineProperty`方法来拷贝属性。

```javascript
var extend = function (to, from) {
  for (var property in from) {
    if (!from.hasOwnProperty(property)) continue;
    Object.defineProperty(
      to,
      property,
      Object.getOwnPropertyDescriptor(from, property)
    );
  }

  return to;
}

extend({}, { get a(){ return 1 } })
// { get a(){ return 1 } })
```

上面代码中，`hasOwnProperty`那一行用来过滤掉继承的属性，否则可能会报错，因为`Object.getOwnPropertyDescriptor`读不到继承属性的属性描述对象。

控制对象状态

有时需要冻结对象的读写状态，防止对象被改变。JavaScript 提供了三种冻结方法，最弱的一种是`Object.preventExtensions`，其次是`Object.seal`，最强的是`Object.freeze`。

Object.preventExtensions()

`Object.preventExtensions`方法可以使得一个对象无法再添加新的属性。

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

Object.defineProperty(obj, 'p', {
  value: 'hello'
});
// TypeError: Cannot define property:p, object is not extensible.

obj.p = 1;
obj.p // undefined
```

上面代码中，`obj`对象经过`Object.preventExtensions`以后，就无法添加新属性了。

Object.isExtensible()

`Object.isExtensible`方法用于检查一个对象是否使用了`Object.preventExtensions`方法。也就是说，检查是否可以为一个对象添加属性。

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

Object.isExtensible(obj) // true
Object.preventExtensions(obj);
Object.isExtensible(obj) // false
```

上面代码中，对`obj`对象使用`Object.preventExtensions`方法以后，再使用`Object.isExtensible`方法，返回`false`，表示已经不能添加新属性了。

Object.seal()

`Object.seal`方法使得一个对象既无法添加新属性，也无法删除旧属性。

```javascript
var obj = { p: 'hello' };
Object.seal(obj);

delete obj.p;
obj.p // "hello"

obj.x = 'world';
obj.x // undefined
```

上面代码中，`obj`对象执行`Object.seal`方法以后，就无法添加新属性和删除旧属性了。

`Object.seal`实质是把属性描述对象的`configurable`属性设为`false`，因此属性描述对象不再能改变了。

```javascript
var obj = {
  p: 'a'
};

// seal方法之前
Object.getOwnPropertyDescriptor(obj, 'p')
// Object {
//   value: "a",
//   writable: true,
//   enumerable: true,
//   configurable: true
// }

Object.seal(obj);

// seal方法之后
Object.getOwnPropertyDescriptor(obj, 'p')
// Object {
//   value: "a",
//   writable: true,
//   enumerable: true,
//   configurable: false
// }

Object.defineProperty(o, 'p', {
  enumerable: false
})
// TypeError: Cannot redefine property: p
```

上面代码中，使用`Object.seal`方法之后，属性描述对象的`configurable`属性就变成了`false`，然后改变`enumerable`属性就会报错。

`Object.seal`只是禁止新增或删除属性，并不影响修改某个属性的值。

```javascript
var obj = { p: 'a' };
Object.seal(obj);
obj.p = 'b';
obj.p // 'b'
```

上面代码中，`Object.seal`方法对`p`属性的`value`无效，是因为此时`p`属性的可写性由`writable`决定。

Object.isSealed()

`Object.isSealed`方法用于检查一个对象是否使用了`Object.seal`方法。

```javascript
var obj = { p: 'a' };

Object.seal(obj);
Object.isSealed(obj) // true
```

这时，`Object.isExtensible`方法也返回`false`。

```javascript
var obj = { p: 'a' };

Object.seal(obj);
Object.isExtensible(obj) // false
```

Object.freeze()

`Object.freeze`方法可以使得一个对象无法添加新属性、无法删除旧属性、也无法改变属性的值，使得这个对象实际上变成了常量。

```javascript
var obj = {
  p: 'hello'
};

Object.freeze(obj);

obj.p = 'world';
obj.p // "hello"

obj.t = 'hello';
obj.t // undefined

delete obj.p // false
obj.p // "hello"
```

上面代码中，对`obj`对象进行`Object.freeze()`以后，修改属性、新增属性、删除属性都无效了。这些操作并不报错，只是默默地失败。如果在严格模式下，则会报错。

Object.isFrozen()

`Object.isFrozen`方法用于检查一个对象是否使用了`Object.freeze`方法。

```javascript
var obj = {
```

```
  p: 'hello'
};

Object.freeze(obj);
Object.isFrozen(obj) // true
```

使用`Object.freeze`方法以后，`Object.isSealed`将会返回`true`，`Object.isExtensible`返回`false`。

```javascript
var obj = {
  p: 'hello'
};

Object.freeze(obj);

Object.isSealed(obj) // true
Object.isExtensible(obj) // false
```

`Object.isFrozen`的一个用途是，确认某个对象没有被冻结后，再对它的属性赋值。

```javascript
var obj = {
  p: 'hello'
};

Object.freeze(obj);

if (!Object.isFrozen(obj)) {
  obj.p = 'world';
}
```

上面代码中，确认`obj`没有被冻结后，再对它的属性赋值，就不会报错了。

局限性

上面的三个方法锁定对象的可写性有一个漏洞：可以通过改变原型对象，来为对象增加属性。

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

var proto = Object.getPrototypeOf(obj);
proto.t = 'hello';
obj.t
// hello
```

上面代码中，对象`obj`本身不能新增属性，但是可以在它的原型对象上新增属性，就依然能够在`obj`上读到。

一种解决方案是，把`obj`的原型也冻结住。

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

var proto = Object.getPrototypeOf(obj);
Object.preventExtensions(proto);

proto.t = 'hello';
obj.t // undefined
```

另外一个局限是，如果属性值是对象，上面这些方法只能冻结属性指向的对象，而不能冻结对象本身的内容。

```javascript
var obj = {
  foo: 1,
  bar: ['a', 'b']
};
Object.freeze(obj);

obj.bar.push('c');
obj.bar // ["a", "b", "c"]
```

上面代码中，`obj.bar`属性指向一个数组，`obj`对象被冻结以后，这个指向无法改变，即无法指向其他值，但是所指向的数组是可以改变的。