ES6_04_扩展的对象功能

属性初始化器的速记法

ES5

```
function createPerson(name, age) {
    return {
name: name,
age: age };
}
```

ES6

```
function createPerson(name, age) {
    return {
name,
age };
}
```

当对象字面量中的属性只有名称时, JS 引擎会在周边作用域查找同名变量。若找到,该变量 的值将会被赋给对象字面量的同名属性。在本例中,局部变量 `name` 的值就被赋给了 `name` 属性。

方法简写

ES5

```
var person = {
    name: "Nicholas",
    sayName: function() {
        console.log(this.name);
    }
};
```

ES6

```
var person = {
    name: "Nicholas",
    sayName() {
        console.log(this.name);
    }
};
```

需计算属性名

在 ES6 中,需计算属性名是对象字面量语法的一部分,它用的也是方括号表示法,与此前在 对象实例上的用法一致。例如:

```
var lastName = "last name";
var person = {
    "first name": "Nicholas",
    [lastName]: "Zakas"
};
console.log(person["first name"]);
console.log(person[lastName]);
// "Nicholas"
// "Zakas"
```

对象字面量内的方括号表明该属性名需要计算,其结果是一个字符串。这意味着其中可以包含表达式,像下面这样:

```
var suffix = " name";
var person = {
    ["first" + suffix]: "Nicholas",
    ["last" + suffix]: "Zakas"
};
console.log(person["first name"]);
console.log(person["last name"]);
// "Nicholas"
// "Zakas"
```

新的方法

Object.is() 方法

当在 JS 中要比较两个值时,你可能会使用相等运算符(`==`)或严格相等运算符(`===`)。为了避免在比较时发生强制类型转换,许多开发者更倾向于使用后者。但严格相等运算 符也并不完全准确,例如,它认为 `+0` 与 `-0` 相等,即使这两者在 JS 引擎中有不同的表 示;另外 `NaN === NaN` 会返回 `false` ,因此有必要使用 `isNaN()` 函数来正确检测 `NaN` 。

ES6 引入了 `Object.is()` 方法来弥补严格相等运算符残留的怪异点。此方法接受两个参数, 并会在二者的值相等时返回 `true` ,此时要求二者类型相同并且值也相等。这有个例子:

```
console.log(+0 == -0);
console.log(+0 === -0);
console.log(Object.is(+0, -0));
console.log(NaN == NaN);
console.log(NaN === NaN);
console.log(Object.is(NaN, NaN));
console.log(5 == "5");
console.log(5 === "5");
console.log(5 == 5);
console.log(5 === 5);
console.log(Object.is(5, 5));
console.log(Object.is(5, "5"));
// true
// false
// false
// false
// true
// true
// true
// true
// false
// true
// false
```

在许多情况下, `Object.is()` 的结果与 `===` 运算符是相同的,仅有的例外是:它会认为 `+0` 与 `-0` 不相等,而且 `NaN` 等于 `NaN` 。不过仍然没必要停止使用严格相等运算符,选择 `Object.is()` ,还是选择 `==` 或 `===` ,取决于代码的实际情况。

Object.assign() 方法

混入(Mixin)是在 JS 中组合对象时最流行的模式。在一次混入中,一个对象会从另一个对 象中接收属性与方法。很多 JS 的库中都有类似下面的混入方法:

```
function mixin(a, b) {
    Object.keys(b).forEach(function(key) {
        a[key] = b[key];
    });
    return a;
}
```

`mixin()` 函数在 `supplier` 对象的自有属性上进行迭代,并将这些属性复制到 `receiver` 对 象(浅复制,当属性值为对象时,仅复制其引用)。这样 `receiver` 对象就能获得新的属性 而无须使用继承,正如下面代码:

```
function EventTarget() { /*...*/ }
EventTarget.prototype = {
    constructor: EventTarget,
    emit: function() { /*...*/ },
    on: function() { /*...*/ }
};
var myObject = {};
mixin(myObject, EventTarget.prototype);
myObject.emit("somethingChanged");
```

此处 `myObject` 对象接收了 `EventTarget.prototype` 对象的行为,这给了它分别使用 `emit()` 与 `on()` 方法来发布事件与订阅事件的能力。

`Object.assign()` 方法接受任意数量的供应者,而接收者会按照供应者在参数中的顺序来依次 接收它们的属性。这意味着在接收者中,第二个供应者的属性可能会覆盖第一个供应者的, 这在下面的代码片段中就发生了:

```
var a = {};
Object.assign(a,
    {
type: "js",
        name: "file.js"
    },
{
type: "css"
} );
console.log(a.type);
console.log(a.name);
// "css"
// "file.js"
```

操作访问器属性

需要记住 `Object.assign()` 并未在接收者上创建访问器属性,即使供应者拥有访问器属 性。由于 `Object.assign()` 使用赋值运算符,供应者的访问器属性就会转变成接收者的 数据属性,例如:

```
var receiver = {},
supplier = {
get name() {
        return "file.js"
    }
};
Object.assign(receiver, supplier);
var descriptor =
Object.getOwnPropertyDescriptor(receiver, "name");
console.log(descriptor.value);      // "file.js"
console.log(descriptor.get);        // undefined
```

此代码中的 `supplier` 对象拥有一个名为 `name` 的访问器属性。在使用了 `Object.assign()` 方法后, `receiver.name` 就作为一个数据属性存在了,其值为 `"file.js"` ,这是因为在调用 `Object.assign()` 时, `supplier.name` 返回的值是 `"file.js"` 。

重复的对象字面量属性

```
"use strict";
var person = {
name: "Nicholas",
name: "Greg" // 在 ES5 严格模式中是语法错误
};
```

在 ES5 严格模式下运行时,第二个 `name` 属性会造成语法错误。但 ES6 移除了重复属性的 检查,严格模式与非严格模式都不再检查重复的属性。当存在重复属性时,排在后面的属性 的值会成为该属性的实际值,如下所示:

```
"use strict";
var person = {
name: "Nicholas",
name: "Greg" // 在 ES6 严格模式中不会出错
};
console.log(person.name);       // "Greg"
```

自有属性的枚举顺序

ES5 并没有定义对象属性的枚举顺序,而是把该问题留给了 JS 引擎厂商。而 ES6 则严格定 义了对象自有属性在被枚举时返回的顺序。这对 `Object.getOwnPropertyNames()` 与 `Reflect.ownKeys` 如何返回属性造成了影响,还同样影响了 `Object.assign()` 处理属性的顺序。

自有属性枚举时基本顺序如下:

1. 所有的数字类型键,按升序排列。
2. 所有的字符串类型键,按被添加到对象的顺序排列。
3. 所有的符号类型(详见第六章)键,也按添加顺序排列。

```
var obj = {
    a: 1,
    0: 1,
    c: 1,
    2: 1,
    b: 1,
    1: 1
};
obj.d = 1;
console.log(Object.getOwnPropertyNames(obj).join(""))
;
// "012acbd"
```

`Object.getOwnPropertyNames()` 方法按 `0 、 1 、 2 、 a 、 c 、 b 、 d` 的顺序返 回了 `obj` 对象的属性。注意,数值类型的键会被合并并排序,即便这未遵循在对象字面量中 的顺序。字符串类型的键会跟在数值类型的键之后,按照被添加到 `obj` 对象的顺序,在对象 字面量中定义的键会首先出现,接下来是此后动态添加到对象的键。

更强大的原型

修改对象的原型

一般来说,对象的原型会在通过构造器或 `Object.create()` 方法创建该对象时被指定。直到 ES5 为止, JS 编程最重要的假定之一就是对象的原型在初始化完成后会保持不变。尽管 ES5 添加了 `Object.getPrototypeOf()` 方法来从任意指定对象中获取其原型,但仍然缺少在 初始化之后更改对象原型的标准方法。

ES6 通过添加 `Object.setPrototypeOf()` 方法而改变了这种假定,此方法允许你修改任意指 定对象的原型。它接受两个参数:需要被修改原型的对象,以及将会成为前者原型的对象。 例如:

```
let person = {
    getGreeting() {
        return "Hello";
    }
};
let dog = {
    getGreeting() {
        return "Woof";
    }
};
// 原型为 person
let friend = Object.create(person);
console.log(friend.getGreeting());
console.log(Object.getPrototypeOf(friend) ===
person);  // true
// 将原型设置为 dog
Object.setPrototypeOf(friend, dog);
console.log(friend.getGreeting());
// "Woof"
console.log(Object.getPrototypeOf(friend) ===
dog);     // true
```

对象原型的实际值被存储在一个内部属性 `[[Prototype]]` 上, `Object.getPrototypeOf()` 方 法会返回此属性存储的值,而 `Object.setPrototypeOf()` 方法则能够修改该值。不过,使用 `[[Prototype]]` 属性的方式还不止这些。