# 第八章 对象、类与面向对象编程

## 1、属性的类型

属性类型分为数据属性和访问器属性。

### 数据属性

数据属性有四个特性：

①[[Configurable]]:表示属性是否可以通过 delete 删除并重新定义，是否可以修改该属性的特性，以及是否可以把它改为访问器属性。

② [[Enumerable]]:表示属性是否可以通过 for-in 循环返回。

③[[Writable]]:表示属性的值是否可以被修改。

④[[Value]]:属性的值。

直接定义在对象上的属性，其[[Configurable]]、[[Enumerable]]、[[Writable]]默认为true。

要修改属性的默认特性，就必须使用 Object.defineProperty()方法。

let person = {};  
 Object.defineProperty(person, "name", {  
 writable: false,  
 value: "Nicholas"  
 });  
 console.log(person.name); // "Nicholas"  
 person.name = "Greg";  
 console.log(person.name); // "Nicholas"

创建了一个名为 name 的属性并给它赋予了一个只读的值"Nicholas"。这个属性的值就不能再修改了，在非严格模式下尝试给这个属性重新赋值会被忽略。在严格模式下，尝试修改只读属性的值会抛出错误。

let person = {};  
 Object.defineProperty(person, "name", {  
 configurable: false,  
 value: "Nicholas"  
 });  
  
//下面运行会报错  
 Object.defineProperty(person, "name", {  
 configurable: true,  
 value: "Nicholas"  
 });

定义了一个name属性，其[[configurable]]特性为false，则该属性的特性不允许被修改。因此一个属性被定义为不可配置之后，就不能再变回可配置的了。

在调用Object.defineProperty()时，configurable、enumerable和writable的值如果不指定，则都默认为 false。

let person = {};  
 Object.defineProperty(person, "name", {  
 configurable: false,  
 value: "Nicholas",  
 });  
  
//此时writable为false，person.name='asdas',无效

### 访问器属性

同样有四个特性

①[[Configurable]]，与数据属性一样

②[[Enumerable]]，一样

③[[Get]]:获取函数，在读取属性时调用。默认值为 undefined。

④[[Set]]:设置函数，在写入属性时调用。默认值为 undefined。

获取函数和设置函数不一定都要定义。只定义获取函数意味着属性是只读的，尝试修改属性会被忽 略。在严格模式下，尝试写入只定义了获取函数的属性会抛出错误。类似地，只有一个设置函数的属性 是不能读取的，非严格模式下读取会返回 undefined，严格模式下会抛出错误。

Object.defineProperties()定义多个属性。这里定义了edition、year\_数据属性和year访问器属性。

Object.getOwnPropertyDescriptor()则获取对象某个属性的描述

let book = {};  
 Object.defineProperties(book, {  
 edition:{  
 value: 1  
 },   
 year\_:{  
 value:2017  
 },  
 year: {  
 get: function() {  
 return this.year\_;  
 },  
 set: function(newValue){  
 if (newValue > 2017) {  
 this.year\_ = newValue;  
 this.edition += newValue - 2017;  
 }  
 }  
 }  
});  
  
let descriptor = Object.getOwnPropertyDescriptor(book, "year\_");  
console.log(descriptor.value);  
console.log(descriptor.configurable);  
console.log(typeof descriptor.get);  
 // 2017  
// false  
// "undefined"  
  
descriptor = Object.getOwnPropertyDescriptor(book, "year");  
console.log(descriptor.value);  
console.log(descriptor.enumerable);  
console.log(typeof descriptor.get);  
// undefined  
// false  
// "function"

Object.getOwnPropertyDescriptors(obj)则是在obj的每个属性上调用Object.getOwnPropertyDescriptor方法，然后返回一个对象。

## 2、合并对象

Object.assign()

如果多个源对象都有相同的属性，则使 用最后一个复制的值。

let dest, src, result;  
dest = { id: 'dest' };  
 result = Object.assign(dest, { id: 'src1', a: 'foo' }, { id: 'src2', b: 'bar' });  
// Object.assign 会覆盖重复的属性  
 console.log(result); // { id: src2, a: foo, b: bar }

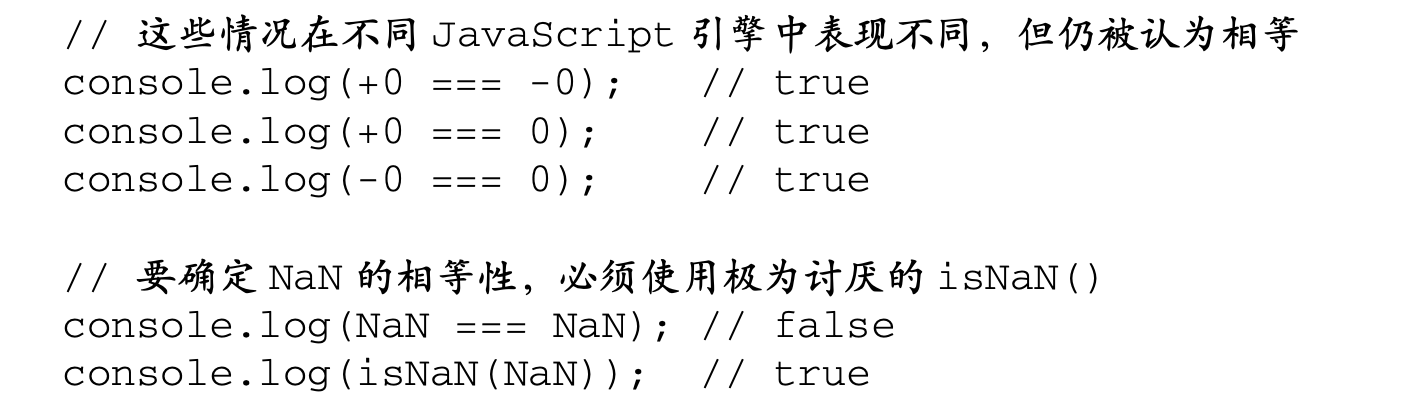
可以通过目标对象上的设置函数观察到覆盖的过程

dest = {   
set id(x) {  
 console.log(x);  
 }  
 };  
 Object.assign(dest,   
 { id: 'first' },  
 { id: 'second' },  
 { id: 'third' }  
 );  
  
 // first  
 // second  
 // third

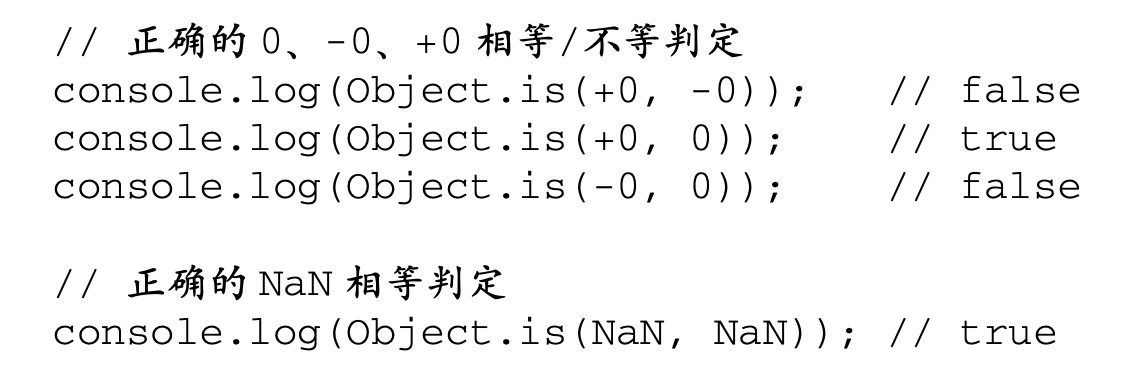
如果Object.assign()过程中出错，则操作会终止并退出，所以可能只会完成部分复制。

### 3、Object.is()

ES6之前“===”有这些边界情形



Object.is()有所改善



### 4、对象增强语法

中括号包围的计算属性键告诉运行时 将其作为 JavaScript 表达式而不是字符串来求值

const nameKey = 'name';  
 const ageKey = 'age';  
 const jobKey = 'job';  
let person = {  
 [nameKey]: 'Matt',  
 [ageKey]: 27,  
 [jobKey]: 'Software engineer'  
 };  
 console.log(person); // { name: 'Matt', age: 27, job: 'Software engineer' }

### 5、解构

解构是一个输出无关的顺序化操作。如果解构中途失败，则整个解构赋值只会完成一部分。

### 6、构造函数

任何函数只要使用 new 操作符调用就是构造函数，而不使用 new 操 作符调用的函数就是普通函数。

构造函数的问题：其定义的方法会在每个实例上都创建一遍。

要解决这个问题，可以把函数定义转移到构造函数外部:

function Person(name, age, job){  
 this.name = name;  
 this.age = age;  
 this.job = job;  
 this.sayName = sayName;  
 }  
 function sayName() {  
 console.log(this.name);  
 }  
   
 let person1 = new Person("Nicholas", 29, "Software Engineer");  
 let person2 = new Person("Greg", 27, "Doctor");  
 person1.sayName(); // Nicholas  
 person2.sayName(); // Greg

但这样会搞乱全局作用域

#### 7、原型

正常的原型链都会终止于 Object 的原型对象

Object 原型的原型是 null

function Person() {}  
  
console.log(Person.prototype.\_\_proto\_\_ === Object.prototype); //true   
console.log(Person.prototype.\_\_proto\_\_.constructor === Object);   
// true  
console.log(Person.prototype.\_\_proto\_\_.\_\_proto\_\_ === null);  
//true  
  
  
使用isPrototypeOf判断是不是实例的原型  
const person1 = new Person;  
console.log(Person.prototype.isPrototypeOf(person1)); // true  
  
  
//Object 类型有一个方法叫 Object.getPrototypeOf()，返回参数的内部特性 [[Prototype]]的值。  
  
console.log(Object.getPrototypeOf(person1) == Person.prototype);   
// true  
  
  
//Object 类型还有一个 setPrototypeOf()方法,可以向实例的私有特性[[Prototype]]写入一个新值。这样就可以重写一个对象的原型继承关系。  
let biped = {  
 numLegs: 2  
 };  
 let person = {  
 name: 'Matt'  
 };  
Object.setPrototypeOf(person, biped);  
console.log(person.name); //Matt  
console.log(person.numLegs);//2   
console.log(Object.getPrototypeOf(person) === biped); // true

hasOwnProperty()方法用于确定某个属性是在实例上还是在原型对象上。区分是实例属性还是原型属性。

in操作符：**console.log("name" in person1);**

只要通过对象可以访问，in 操作符就返回 true，而 hasOwnProperty()只有属性存在于实例上 时才返回 true。

如果要确定某个属性是否存在于原型上，则可以像下 面这样同时使用 hasOwnProperty()和 in 操作符:

function hasPrototypeProperty(object, name){  
 return !object.hasOwnProperty(name) && (name in object);  
 }

for in循环时，可以通过对象访问且可以被枚举的属性都会返回，包括实例 属性和原型属性。（遮蔽原型中不可枚举([[Enumerable]]特性被设置为 false)属性的实例属性也会 在 for-in 循环中返回）

Object.keys()获得对象上所有可枚举的实例属性

Object.getOwnPropertyNames()获得对象上所有属性，无论是否可枚举

Object.getOwnPropertySymbols()跟Object.getOwnPropertyNames()一样，只是针对符号而已

#### 8、对象迭代

ECMA2017增加了Object.values()和Object.entires()，Symbol属性会被他们忽略

#### 9、重写原型的例子

function Person() {}  
 let friend = new Person();  
 Person.prototype = {  
 constructor: Person,  
 name: "Nicholas",  
 age: 29,  
 job: "Software Engineer",  
 sayName() {  
 console.log(this.name);  
 }  
 };  
  
friend.sayName(); // 错误

Person 的新实例是在重写原型对象之前创建的。在调用 friend.sayName()的时 候，会导致错误。这是因为 firend 指向的原型还是最初的原型，而这个原型上并没有 sayName 属性。

#### 10、继承

实现继承是 ECMAScript 唯一支持的继承方式，而这主要是通过原型链实现的。

组合继承弥补了原型链和盗用构造函数的不足，是 JavaScript 中使用最多的继承模式。而且组合继 承也保留了 instanceof 操作符和 isPrototypeOf()方法识别合成对象的能力。

实现原型式继承的函数

function object(o) {  
 function F() {}  
 F.prototype = o;  
 return new F();  
 }

ECMAScript 5 通过增加 Object.create()方法将原型式继承的概念规范化了。这个方法接收两个 10 参数:作为新对象原型的对象，以及给新对象定义额外属性的对象(第二个可选)。在只有一个参数时， Object.create()与 object()方法效果相同

寄生式组合继承可以算是引用类型继承的最佳模式。

### 11、ES6 class

class受块级作用域限制

默认情况下，类定义中的代码都在严格模式下执行。

类构造函数没有什么特殊之处，实例化之后，它会成为普通的实例方法(但作为类构造函数，仍然 要使用 new 调用)。

class Person {}  
// 使用类创建一个新实例 let p1 = new Person();  
  
p1.constructor();  
 // TypeError: Class constructor Person cannot be invoked without 'new'  
  
// 使用对类构造函数的引用创建一个新实例   
let p2 = new p1.constructor();

es中没有class这种类型，类只是一种特殊的函数，typeof仍然是function

class Person {}  
 console.log(Person); // class Person {}  
 console.log(typeof Person); // function

与执行函数类似，类也可以立即实例化

// 因为是一个类表达式，所以类名是可选的  
let p = new class Foo {  
 constructor(x) {  
 console.log(x);  
 }  
 }('bar'); // bar  
 console.log(p); // Foo {}

class中的this

class Person {  
 constructor() {  
 // 添加到 this 的所有内容都会存在于不同的实例上  
 this.locate = () => console.log('instance', this);  
 }  
 // 定义在类的原型对象上   
 locate() {  
 console.log('prototype', this);  
 }  
 // 定义在类本身上   
 static locate() {   
 console.log('class', this);  
 }  
 }  
 let p = new Person();  
 p.locate(); // instance, Person {}  
 Person.prototype.locate(); // prototype, {constructor: ... }  
 Person.locate(); // class, class Person {}

在派生类中，super()相当于super.constructor()

在静态方法中可以通过 super 调用继承的类上定义的静态方法:

class Vehicle {  
 static identify() {  
 console.log('vehicle');  
 }  
}   
 class Bus extends Vehicle {  
 static identify() {  
 super.identify();  
 }  
}  
Bus.identify(); // vehicle

如果没有定义派生类的构造函数，在实例化派生类时会调用 super()，而且会传入所有传给派生类的参数。

class Vehicle {  
 constructor(licensePlate) {  
 this.licensePlate = licensePlate;  
 }  
}  
class Bus extends Vehicle {}  
console.log(new Bus('1337H4X')); // Bus { licensePlate: '1337H4X' }