Chapter 5. Inheritance

# Capítulo 5. Herencia

Divide una cosa entera en muchos objetos; Como las perspectivas, que bien miradas No muestran más que confusión. . .

-WilliamShakespeare, *The Tragedy of King Richard the Second*

La herencia es un tema importante en la mayoría de los lenguajes de programación.

En los lenguajes clásicos (como Java), la herencia (o extends) proporciona dos servicios útiles. En primer lugar, es una forma de reutilización del código. Si una nueva clase es en su mayor parte similar a una clase existente, sólo tienes que especificar las diferencias. Los patrones de reutilización de código son extremadamente importantes porque tienen el potencial de reducir significativamente el coste del desarrollo de software. La otra ventaja de la herencia clásica es que incluye la especificación de un sistema de tipos. Esto libera en gran medida al programador de tener que escribir operaciones de casting explícitas, lo cual es muy bueno porque al hacer casting se pierden las ventajas de seguridad de un sistema de tipos.

JavaScript, al ser un lenguaje poco tipado, nunca hace castings. El linaje de un objeto es irrelevante. Lo que importa de un objeto es lo que puede hacer, no de qué desciende.

JavaScript proporciona un conjunto mucho más rico de patrones de reutilización de código. Puede imitar el patrón clásico, pero también admite otros patrones que son más expresivos. El conjunto de posibles patrones de herencia en JavaScript es enorme. En este capítulo, veremos algunos de los patrones más sencillos. Son posibles construcciones mucho más complicadas, pero normalmente es mejor mantener la sencillez.

En los lenguajes clásicos, los objetos son instancias de clases, y una clase puede heredar de otra clase. JavaScript es un lenguaje prototípico, lo que significa que los objetos heredan directamente de otros objetos.

# Pseudoclásico

JavaScript es conflictivo en cuanto a su naturaleza prototípica. Su mecanismo prototípico está oscurecido por un complicado negocio sintáctico que parece vagamente clásico. En lugar de hacer que los objetos hereden directamente de otros objetos, se inserta un nivel innecesario de indirección, de modo que los objetos se producen mediante funciones constructoras.

Cuando se crea un objeto función, el constructor Functionque produce el objeto función ejecuta algún código como éste:

this.prototype = {constructor: this};

Al nuevo objeto función se le da una propiedad prototypecuyo valor es un objeto que contiene una propiedad constructor cuyo valor es el nuevo objeto función. El objeto prototype es el lugar donde se depositarán los rasgos heredados. Cada función recibe un objeto prototype porque el lenguaje no proporciona una forma de determinar qué funciones están destinadas a ser utilizadas como constructoras. La propiedad constructor no es útil. Lo importante es el objeto prototype.

Cuando se invoca una función con el patrón de invocación a constructor utilizando el prefijo new, se modifica la forma en que se ejecuta la función. Si el operador new fuera un método en lugar de un operador, podría haberse implementado así:

Function.method('new', function ( ) {  
  
// Create a new object that inherits from the  
// constructor's prototype.  
  
 var that = Object.beget(this.prototype);  
  
// Invoke the constructor, binding -this- to  
// the new object.  
  
 var other = this.apply(that, arguments);  
  
// If its return value isn't an object,  
// substitute the new object.  
  
 return (typeof other === 'object' && other) || that;  
});

Podemos definir un constructor y aumentar su prototype:

var Mammal = function (name) {  
 this.name = name;  
};  
  
Mammal.prototype.get\_name = function ( ) {  
 return this.name;  
};  
  
Mammal.prototype.says = function ( ) {  
 return this.saying || '';  
};

Ahora, podemos crear una instancia:

var myMammal = new Mammal('Herb the Mammal');  
var name = myMammal.get\_name( ); // 'Herb the Mammal'

Podemos hacer otra pseudoclase que herede de Mammal definiendo su función constructory sustituyendo su prototype por una instancia de Mammal:

var Cat = function (name) {  
 this.name = name;  
 this.saying = 'meow';  
};  
  
// Replace Cat.prototype with a new instance of Mammal  
  
Cat.prototype = new Mammal( );  
  
// Augment the new prototype with  
// purr and get\_name methods.  
  
Cat.prototype.purr = function (n) {  
 var i, s = '';  
 for (i = 0; i < n; i += 1) {  
 if (s) {  
 s += '-';  
 }  
 s += 'r';  
 }  
 return s;  
};  
Cat.prototype.get\_name = function ( ) {  
 return this.says( ) + ' ' + this.name + ' ' + this.says( );  
};  
  
var myCat = new Cat('Henrietta');  
var says = myCat.says( ); // 'meow'  
var purr = myCat.purr(5); // 'r-r-r-r-r'  
var name = myCat.get\_name( );  
// 'meow Henrietta meow'

El patrón pseudoclásico pretendía parecer algo orientado a objetos, pero tiene un aspecto bastante extraño. Podemos ocultar parte de la fealdad utilizando el método method y definiendo un método inherits:

Function.method('inherits', function (Parent) {  
 this.prototype = new Parent( );  
 return this;  
});

Nuestros métodos inherits y method devuelven this, lo que nos permite programar en cascada. Ahora podemos hacer nuestro Cat con una sola declaración.

var Cat = function (name) {  
 this.name = name;  
 this.saying = 'meow';  
}.  
 inherits(Mammal).  
 method('purr', function (n) {  
 var i, s = '';  
 for (i = 0; i < n; i += 1) {  
 if (s) {  
 s += '-';  
 }  
 s += 'r';  
 }  
 return s;  
 }).  
 method('get\_name', function ( ) {  
 return this.says( ) + ' ' + this.name + ' ' + this.says( );  
 });

Al ocultar el prototype jazz, ahora parece un poco menos extraño. Pero, ¿hemos mejorado algo realmente? Ahora tenemos funciones constructoras que actúan como clases, pero en los bordes puede haber un comportamiento sorprendente. No hay privacidad; todas las propiedades son públicas. No hay acceso a los métodos de super.

Peor aún, existe un grave peligro con el uso de funciones constructoras. Si olvidas incluir el prefijo new al llamar a una función constructora, entonces this no estará vinculado a un nuevo objeto. Lamentablemente, this estará ligado al objeto global, así que en lugar de aumentar tu nuevo objeto, estarás destrozando variables globales. Eso está muy mal. No hay ninguna advertencia de compilación ni de ejecución.

Se trata de un grave error de diseño del lenguaje. Para mitigar este problema, existe la convención de que todas las funciones constructoras se nombren con mayúscula inicial, y que nada más se escriba con mayúscula inicial. Esto nos permite rezar para que la inspección visual pueda encontrar la falta de new. Una alternativa mucho mejor es no utilizar new en absoluto.

La forma pseudoclásica puede proporcionar comodidad a los programadores que no están familiarizados con JavaScript, pero también oculta la verdadera naturaleza del lenguaje. La notación de inspiración clásica puede inducir a los programadores a componer jerarquías innecesariamente profundas y complicadas. Gran parte de la complejidad de las jerarquías de clases está motivada por las restricciones de la comprobación estática de tipos. JavaScript está completamente libre de esas restricciones. En los lenguajes clásicos, la herencia de clases es la única forma de reutilización del código. JavaScript tiene más y mejores opciones.