Functional

# Funcional

Uno de los puntos débiles de los patrones de herencia que hemos visto hasta ahora es que no tenemos privacidad. Todas las propiedades de un objeto son visibles. No obtenemos variables privadas ni métodos privados. A veces eso no importa, pero a veces importa mucho. En su frustración, algunos programadores desinformados han adoptado el patrón *pretend privacy*. Si tienen una propiedad que desean hacer privada, le dan un nombre de aspecto extraño, con la esperanza de que otros usuarios del código finjan que no pueden ver los miembros de aspecto extraño. Afortunadamente, tenemos una alternativa mucho mejor en una aplicación del patrón módulo.

Empezaremos creando una función que producirá objetos. Le daremos un nombre que empiece por minúscula porque no requerirá el uso del prefijo new. La función contiene cuatro pasos:

1. Crea un nuevo objeto. Hay muchas formas de crear un objeto. Puede hacer un objeto literal, o puede llamar a una función constructora con el prefijo new, o puede utilizar el método Object.create para hacer una nueva instancia a partir de un objeto existente, o puede llamar a cualquier función que devuelva un objeto.
2. Opcionalmente, define variables de instancia y métodos privados. Éstos no son más que vars ordinarios de la función.
3. Aumenta ese nuevo objeto con métodos. Esos métodos tendrán acceso privilegiado a los parámetros y a los vars definidos en el segundo paso.
4. Devuelve ese nuevo objeto.

He aquí una plantilla de pseudocódigo para un constructor funcional (el texto en negrita se ha añadido para resaltarlo):

var constructor = function (spec, my) {  
 var that, other private instance variables;  
 my = my || {};  
  
 Add shared variables and functions to my  
  
 that = a new object;  
  
 Add privileged methods to that  
  
 return that;  
};

El objeto spec contiene toda la información que el constructor necesita para crear una instancia. El contenido de spec puede copiarse en variables privadas o ser transformado por otras funciones. O los métodos pueden acceder a la información de spec cuando la necesiten. (Una simplificación es sustituir spec por un único valor. Esto es útil cuando el objeto que se está construyendo no necesita un objeto spec completo).

El objeto my es un contenedor de secretos que comparten los constructores de la cadena de herencia. El uso del objeto my es opcional. Si no se pasa un objeto my, entonces se construye un objeto my.

A continuación, declara las variables de instancia privadas y los métodos privados del objeto. Esto se hace simplemente declarando variables. Las variables y funciones internas del constructor se convierten en miembros privados de la instancia. Las funciones internas tienen acceso a spec y my y that y a las variables privadas.

A continuación, añade los secretos compartidos al objeto my. Esto se hace mediante asignación:

my.member = value;

Ahora, creamos un nuevo objeto y lo asignamos a that. Hay muchas formas de crear un nuevo objeto. Podemos utilizar un literal de objeto. Podemos llamar a un constructor pseudoclásico con el operador new. Podemos utilizar el método Object.create en un objeto prototipo. O podemos llamar a otro constructor funcional, pasándole un objeto spec (posiblemente el mismo objeto spec que se pasó a este constructor) y el objeto my. El objeto my permite al otro constructor compartir el material que pusimos en my. El otro constructor también puede poner sus propios secretos compartidos en my para que nuestro constructor pueda aprovecharlos.

A continuación, aumentamos that, añadiendo los métodos privilegiados que componen la interfaz del objeto. Podemos asignar nuevas funciones a miembros de that. O, de forma más segura, podemos definir las funciones primero como métodos privados, y luego asignarlas a that:

var methodical = function ( ) {  
 ...  
};  
that.methodical = methodical;

La ventaja de definir methodical en dos pasos es que si otros métodos quieren llamar a methodical, pueden llamar a methodical( ) en lugar de a that.methodical( ). Si la instancia se daña o se manipula de forma que that.methodical se sustituye, los métodos que llaman a methodical seguirán funcionando igual porque su methodical privado no se ve afectado por la modificación de la instancia.

Por último, devolvemos that.

Apliquemos este patrón a nuestro ejemplo de mamífero. Aquí no necesitamos my, así que lo omitiremos, pero utilizaremos un objeto spec.

Las propiedades name y saying son ahora completamente privadas. Sólo son accesibles a través de los métodos privilegiados get\_name y says:

var mammal = function (spec) {  
 var that = {};  
  
 that.get\_name = function ( ) {  
 return spec.name;  
 };  
  
 that.says = function ( ) {  
 return spec.saying || '';  
 };  
  
 return that;  
};  
  
var myMammal = mammal({name: 'Herb'});

En el patrón pseudoclásico, la función del constructor Cattenía que duplicar el trabajo realizado por el constructor Mammal. Eso no es necesario en el patrón funcional porque el constructor Cat llamará al constructor Mammal, dejando que Mammal haga la mayor parte del trabajo de creación del objeto, de modo que Cat sólo tiene que preocuparse de las diferencias:

var cat = function (spec) {  
 spec.saying = spec.saying || 'meow';  
 var that = mammal(spec);  
 that.purr = function (n) {  
 var i, s = '';  
 for (i = 0; i < n; i += 1) {  
 if (s) {  
 s += '-';  
 }  
 s += 'r';  
 }  
 return s;  
 };  
 that.get\_name = function () {  
 return that.says() + ' ' + spec.name + ' ' + that.says();  
 };  
 return that;  
};  
  
var myCat = cat({name: 'Henrietta'});

El patrón funcional también nos da una forma de tratar los supermétodos. Haremos un método superior que tome un nombre de método y devuelva una función que invoque a ese método. La función invocará al método original aunque se modifique la propiedad:

Object.method('superior', function (name) {  
 var that = this,  
 method = that[name];  
 return function ( ) {  
 return method.apply(that, arguments);  
 };  
});

Vamos a probarlo en un coolcat que es igual que cat excepto que tiene un método get\_name más fresco que llama al supermétodo. Sólo requiere un poco de preparación. Declararemos una variable super\_get\_name y le asignaremos el resultado de invocar al método superior:

var coolcat = function (spec) {  
 var that = cat(spec),  
 super\_get\_name = that.superior('get\_name');  
 that.get\_name = function (n) {  
 return 'like ' + super\_get\_name( ) + ' baby';  
 };  
 return that;  
};  
  
var myCoolCat = coolcat({name: 'Bix'});  
var name = myCoolCat.get\_name( );  
// 'like meow Bix meow baby'

El patrón funcional tiene una gran flexibilidad. Requiere menos esfuerzo que el patrón pseudoclásico, y nos proporciona una mejor encapsulación y ocultación de información y acceso a los métodos superiores.

Si todo el estado de un objeto es privado, entonces el objeto es a prueba de manipulaciones. Las propiedades del objeto se pueden sustituir o eliminar, pero la integridad del objeto no se ve comprometida. Si creamos un objeto en el estilo funcional, y si todos los métodos del objeto no hacen uso de this o that, entonces el objeto es *durable*. Un objeto duradero es simplemente una colección de funciones que actúan como *capabilities*.

Un objeto duradero no puede ser comprometido. El acceso a un objeto duradero no da a un atacante la capacidad de acceder al estado interno del objeto, excepto en lo permitido por los métodos.