Closure

# Cierre

La buena noticia sobre el ámbito es que las funciones internas tienen acceso a los parámetros y variables de las funciones dentro de las que están definidas (con la excepción de this y arguments). Esto es algo muy bueno.

Nuestra función getElementsByAttribute funcionaba porque declaraba una variable results, y la función interna que pasaba a walk\_the\_DOMtambién tenía acceso a la variable results.

Un caso más interesante es cuando la función interna tiene una vida más larga que su función externa.

Antes hicimos un myObject que tenía un método value y otro increment. Supongamos que queremos proteger el valor de cambios no autorizados.

En lugar de inicializar myObject con un literal de objeto, inicializaremos myObject llamando a una función que devuelve un literal de objeto. Esa función define una variable value. Esa variable está siempre disponible para los métodos increment y getValue, pero el ámbito de la función la mantiene oculta al resto del programa:

var myObject = (function () {  
 var value = 0;  
  
 return {  
 increment: function (inc) {  
 value += typeof inc === 'number' ? inc : 1;  
 },  
 getValue: function ( ) {  
 return value;  
 }  
 };  
}());

No estamos asignando una función a myObject. Estamos asignando el resultado de invocar esa función. Fíjate en el ( ) de la última línea. La función devuelve un objeto que contiene dos métodos, y esos métodos siguen gozando del privilegio de acceso a la variable value.

El constructor Quo de antes produjo un objeto con una propiedad status y un método get\_status. Pero eso no parece muy interesante. ¿Por qué llamarías a un método getter de una propiedad a la que puedes acceder directamente? Sería más útil que la propiedad statusfuera privada. Así que definamos otro tipo de función quo para hacer eso:

// Create a maker function called quo. It makes an  
// object with a get\_status method and a private  
// status property.  
  
var quo = function (status) {  
 return {  
 get\_status: function ( ) {  
 return status;  
 }  
 };  
};  
  
// Make an instance of quo.  
  
var myQuo = quo("amazed");  
  
document.writeln(myQuo.get\_status( ));

Esta función quo está diseñada para ser utilizada sin el prefijo new, por lo que su nombre no se escribe en mayúsculas. Cuando llamamos a quo, devuelve un nuevo objeto que contiene un método get\_status. Una referencia a ese objeto se almacena en myQuo. El método get\_status sigue teniendo acceso privilegiado a la propiedad status de quo, aunque quoya haya retornado. get\_status no tiene acceso a una copia del parámetro; tiene acceso al propio parámetro. Esto es posible porque la función tiene acceso al contexto en el que se creó, lo que se denomina *closure*.

Veamos un ejemplo más útil:

// Define a function that sets a DOM node's color  
// to yellow and then fades it to white.  
  
var fade = function (node) {  
 var level = 1;  
 var step = function ( ) {  
 var hex = level.toString(16);  
 node.style.backgroundColor = '#FFFF' + hex + hex;  
 if (level < 15) {  
 level += 1;  
 setTimeout(step, 100);  
 }  
 };  
 setTimeout(step, 100);  
};  
  
fade(document.body);

Llamamos a fade, pasándole document.body (el nodo creado por la etiqueta HTML <body> ). fade pone level a 1. Define una función step. Llama a setTimeout, pasándole la función step y un tiempo (100 milisegundos). A continuación, vuelve -fade ha terminado.

De repente, aproximadamente una décima de segundo después, se invoca a la función step. Crea un carácter de base 16 a partir de fade's level. A continuación, modifica el color de fondo del nodo fade. Después mira en fade's level. Si aún no ha llegado al blanco, incrementa fade's level y utiliza setTimeout para programarse para ejecutarse de nuevo.

De repente, se invoca de nuevo a la función step. Pero esta vez, fade 's level es 2. fade regresó hace un rato, pero sus variables siguen vivas mientras las necesiten una o más de las funciones internas de fade's.

Es importante entender que la función interna tiene acceso a las variables reales de las funciones externas y no a copias, para evitar el siguiente problema.

// BAD EXAMPLE  
  
// Make a function that assigns event handler functions to an array  
 of nodes the wrong way.  
// When you click on a node, an alert box is supposed to display the ordinal  
of the node.  
// But it always displays the number of nodes instead.  
  
var add\_the\_handlers = function (nodes) {  
 var i;  
 for (i = 0; i < nodes.length; i += 1) {  
 nodes[i].onclick = function (e) {  
 alert(i);  
 };  
 }  
};  
  
// END BAD EXAMPLE

La función add\_the\_handlers pretendía dar a cada manejador un número único i. Falla porque las funciones manejadoras están vinculadas a la variable i, no al valor de la variable ien el momento en que se hizo la función.

// BETTER EXAMPLE  
  
// Make a function that assigns event handler functions to an array of nodes.  
// When you click on a node, an alert box will display the ordinal of the node.  
  
var add\_the\_handlers = function (nodes) {  
 var helper = function (i) {  
 return function (e) {  
 alert(i);  
 };  
 };  
 var i;  
 for (i = 0; i < nodes.length; i += 1) {  
 modes[i].onclick = helper(i);  
 }  
};

Evita crear funciones dentro de un bucle. Puede ser un derroche computacional, y puede causar confusión, como vimos con el mal ejemplo. Evitamos la confusión creando una función de ayuda fuera del bucle que entregará una función que se vincula al valor actual de i.