Helephant.com

Javascript closures

Javascript closures are a really powerful feature of the javascript language. Closures are created when a function that's nested inside another function accesses a variable from its parent's scope. This is really useful for passing state around your application when the inner function is called after the outer function has exited.

Javascript supports functions nested inside other functions

Nested functions are possible because the function operator that makes it possible to create anonymous functions anywhere that it's possible to have any other type of statement. Even if you've never heard of closures before, you've probably done this without thinking about it too much when you set up an event handler:

```
function myFunction(){
     var button = document.getElementById("Button1");
     button.addEventListener("click", function()
     {
         alert("I am a function nested inside another function");
     }, false);
 }
```

Complete example

A closure is created when an inner function accesses variables from the outer function

In javascript an inner function can access variables from the outer function's scope. When this happens a closure is created. The closure is just a link from the inner function to the outer function from the time when the outer function exited. It's created to save the state of the variables from the outer function so they are still available when the inner function is run.

The best way to understand this is to look at an example. Take a look at this piece of code which is very similar to the last example:

```
function outerFunction(){
     var button = document.getElementById("Button1");
     var importantPieceOfState = "I love fish";
     // declare the inner event
     button.addEventListener("click", function()
     {
         alert(importantPieceOfState);
     }, false);
 }
```

Complete example

When you run the button click handler code, the alert will say “I love fish” even though the importantPieceOfState variable is only declared in the outerFunction()’s scope.

Notice that importantPieceOfState isn’t declared anywhere inside the click event handler and that it isn’t a global variable. It is only declared as a local variable to the outerFunction() function. The variable is available because when we run outerFunction() the javascript engine notices that the click handler has a reference to one of the variables in outerFunction(). It creates a closure to save the current state of outerFunction() for when the click event is run. If the inner function didn’t access any of the outer function’s variables the closure would not be created.

All the variables from the outer function are available to the inner function. This includes the parameters. The only exception is any variables from the outer function that are overridden by variables from the inner function that have the same name.

A new closure is created each time the function runs

In the last example outerFunction() is only run once so only one closure is created. If we ran the outer function multiple times a new closure would be created each time.

Here’s another example where the outer function is called multiple times with different parameters:

```
  function setClickColor(button, color){
        // declare the inner event
        button.addEventListener("click", function()
        {
            button.style.backgroundColor = color;
        }, false);
    }

   window.onload = function(){
        setClickColor(document.getElementById("Button1"), "red");
        setClickColor(document.getElementById("Button2"), "blue");
    }
```

Complete example

Notice that the parameter has a different value for each inner function that is created. The color parameter is set to red for the red button click and blue for the blue button click. This happens because a new closure is created each time the function is run.

Closures are created when the outer function exits

The closure is actually created when the outer function exits, not when the inner function is created. This means the values of any variables will be saved as they are when the outer function exits.

This can be a bit confusing if you're creating functions inside a loop. All of the functions you create will point to the same closure so the variable values will be the same for them all.

Imagine you had a list of buttons and you wanted to set up a click event for each. For some reason the order of the buttons is important so you want to access the button's index in the array inside the loop:

```
window.onload = function(){
     var buttons = ["Button1", "Button2"];
     for(var x=0; x&lt;buttons.length; x++)
     {
          var button = document.getElementById(buttons[x]);
          button.addEventListener("click", function()
          {
               alert("I am button " + x);
          }, false);
     }
 }
```

Complete example

Looking at the code you might expect x to be zero for the first button and one for the second button but it isn't. It's two for both buttons because after the second event handler has been set up, the x variable is incremented to two and the loop's condition fails. Then the function exits and the value of two is stored in the closure that's used by both button click events.

Closures reduce the need to pass around state

Closures reduce the need to pass state around the application. The inner function has access to the variables in the outer function so there is no need to store the information somewhere that the inner function can get it.

This is important when the inner function will be called after the outer function has exited. The most common example of this is when the inner function is being used to handle an event. In this case you get no control over the arguments that are passed to the function so using a closure to keep track of state can be very convenient.

For example imagine you're using the setTimeout function to do a mouseover effect where the element changes color when you mouse in for a few seconds and then changes back to the original color. You need to pass the setTimeout callback function the original color of the element but the IE version of setTimeout() method doesn't allow you to pass any parameters to the callback function.

Closures make passing information like this simple. We just need to store the original background color in a variable in the outer function and then the inner function can just use the variable to reset the background color:

```
var divs = document.getElementsByTagName(“div”);
 for(var x=0; x&lt;divs.length; x++)
 {
     divs[x].addEventListener("mouseover", function(e)
     {
         var element = e.currentTarget;
         var backgroundColor = element.style.backgroundColor;
         element.style.backgroundColor = "yellow";

         window.setTimeout(function()
         {
             element.style.backgroundColor = backgroundColor;
         }, 1500)
     }, false);
 }
```

Complete example

Some things closures make possible

Closures can be used to achieve some pretty neat stuff in javascript. Here are a link to a few really useful javascript patterns that use closures (please leave a comment if you know another interesting one):

- Module pattern – the most flexible way to do namespaces for javascript.
- Private and protected members – Douglas Crockford’s private and protected members for javascript objects.
- Curried javascript functions – a functional programming technique.

Potential problems

Closures are a really useful tool to have in your javascript coding arsenal but you have to know about the two potential pitfalls. The first is that it makes it easier to introduce javascript memory leaks into your code. The second is that it can use a lot of memory if the outer function is called very often or the data that is saved is very large.

Potential for memory leaks

Unfortunately some browsers (IE in particular) have problems disposing of DOM elements and javascript objects that have a circular reference between them. This can be a big problem because the affected objects will stay in memory until the browser is restarted, even if another page is loaded. This isn’t a problem for closures exclusively but closures can sometimes make the circular references hard to see.

To solve this you just need to break all the links from DOM to javascript when the page unloads and then the circular references are broken. This will mean unhooking any DOM events that have been set up during the page’s life and setting any attributes of the DOM elements that point to javascript objects to null. It's really important to do this even if you're not using closures because you're really likely to run into memory leaks if you don't.

Most javascript frameworks will automatically unhook all of the event handlers if you register the events using the framework's methods. In the ASP.NET Ajax framework you need to make sure that you call $clearHandlers() on any html element to which you added a handler.

Javascript memory leaks are my least favourite type of webpage program to debug because it can be really hard to track down what's leaking. However there are a few really great articles you can read to understand what happens so you can prevent your code ever leaking in the first place:

- Javascript memory leaks – why memory leaks happen and how to break the circular references.
- DHTML Leaks Like a Sieve – a good general explanation of the problem
- QuirksBlog memory leak links – links to a heap of useful articles about memory leaks

Could potentially use a lot of memory

When a closure is created the execution context of the current function call is saved with a reference to all the variables in the function's scope. This can use a lot of memory if a lot of frames get saved or if each execution context is very large.

Outer functions that get called a lot of time aren't good candidates to be used for closures because each time the function is called a new closure will be saved. The inner function can be called as many times as you want. It's only when the outer function gets called that a new closure gets created.

It could also be a problem if the amount of data inside the outer function are very large. If you had a ten thousand character string in the outer function it probably would not be a good candidate to be used for closures. Again this only affects the outer function. The inner function does not affect the amount of data to be saved.

In most cases using closures will be fine, but if you do find that your application is using more memory than you'd like closures are a possible cause of the problem.

More information

Closures are a bit of a tricky topic to get your head around if you've never seen them before but there are a few really great resources about them available to help.

- Closures tutorial – John Resig (of JQuery fame) has written a helpful interactive tutorial for helping people understand how closures work.
- Javascript closures for dummies is a fantastic reference that takes you through a heap of different code examples about closures and explains the most important concepts.
- Javascript Closures is the complete reference on closures. It goes into very extensive technical detail. It's not the easiest read but it explains a lot of useful things about how closures and javascript functions in general really work.

What's next?

Although constructor functions look a lot like classes, javascript does not have a class based object system like C# or Java. Instead it has a prototype based object system.

This article is part of a set of related posts about How javascript objects work.

Posted on 17 Oct 08 by Helen Emerson (last updated on 16 Nov 11).
Filed under Javascript, Web development