Articles by Topic: Search Engine Optimization (SEO)
Digiwebs

Articles by Topic: RSS
Digiwebs

Articles by Topic: Liquid Web Design
Digiwebs

Scope In JavaScript

By **Digiwebs** Last updated **Sep 3, 2018** **ARTICLES**

```
$(function(){cards();});
$(window).on('resize', function(){cards();});
function cards(){
    var width = $(window).width();
    if(width < 750){
        cardssmallscreen();
    }else{
        cardsbigscreen();
    }
}
function cardssmallscreen(){
    var cards = $('.card').length;
    var height = 0;
    var card2 = 2;
```

Scope in JavaScript

Got something to say?

Share your comments on this topic with other web professionals
In: Articles

By Mike West

Published on September 11, 2006

Scope is one of the foundational aspects of the JavaScript language, and probably the one I've struggled with the most when building complex programs. I can't count the number of times I've lost track of what the `this` keyword refers to after passing control around from function to function, and I've often found myself contorting my code in all sorts of confusing ways, trying to retain some semblance of sanity in my understanding of which variables were accessible where.

This article will tackle the problem head-on, outlining definitions of context and scope, examining two JavaScript methods that allow us to manipulate context, and concluding with a deep dive into an *effective* solution to ninety percent of the problems I've run into.

Where Am I? And Who Are You?

Every bit of your JavaScript program is executed in one **execution context** or another. You can think of these contexts as your code's neighborhood, giving each line an understanding of where it comes from, and who its friends and neighbors are. As it turns out, this is important information, as JavaScript societies have fairly strict rules about who can associate with whom; execution contexts are better thought of as gated communities than as open subdivisions.

We can refer to these social boundaries generally as **scope**, and they're important enough to be codified in each neighborhood's charter, which we'll refer to as the context's **scope chain**. Code within a particular neighborhood can only access variables listed on its scope chain, and prefers interaction with locals to associations outside its neighborhood.

Practically speaking, evaluating a function establishes a distinct execution context that appends its local scope to the scope chain it was defined within. JavaScript resolves identifiers within a particular context by climbing up the scope chain, moving locally to globally. This means that local variables with the same name as variables higher up on the scope chain take precedence, which makes sense: If my good friends are talking together about "Mike West," it's pretty clear that they're talking about *me*, not about the bluegrass singer or the Duke professor, even though the latter two are (arguably) better known.

Let's walk through some example code to explore the implications:

```
 <script type="text/javascript">
var ima_celebrity = "Everyone can see me! I'm famous!",
the_president = "I'm the decider!";
```

```
function pleasantville() {
var the_mayor = “I rule Pleasantville with an iron fist!”,
ima_celebrity = “All my neighbors know who I am!”;

function lonely_house() {
var agoraphobic = “I fear the day star!”,
a_cat = “Meow.”;
}
}
</script>
```

Our global star, `ima_celebrity` , is recognized by everyone. She’s politically active, talking with `the_president` on a fairly frequent basis, and incredibly friendly; she’ll sign autographs and answer questions for anyone she runs into. That said, she doesn’t have a whole lot of personal contact with her fans. She’s pretty sure they *exist* and that they probably have lives of their own somewhere, but she certainly doesn’t know what they’re doing, or even their names.

Inside `pleasantville` , `the_mayor` is a well-known face. She’s always walking the streets of her town, chatting up her constituents, shaking hands, and kissing babies. As `pleasantville` is a big, important neighborhood, she’s got a big red phone in her office, giving her a direct line to the president (or at least a top aide) 24 hours a day, 7 days a week. She’s seen `lonely_house` up on a hill at the outskirts of town, but never really worried about who lives inside.

That `lonely_house` is a world unto itself. The `agoraphobic` stays inside most of the time, playing solitaire and feeding `a_cat` . He’s called `the_mayor` a few times to ask about local noise regulations, and even wrote `ima_celebrity` (Pleasantville’s `ima_celebrity` , that is) some fan mail after seeing her on the local news.

`this` ? What’s that?

In addition to establishing a scope chain, each execution context offers a keyword named `this` . In its most common usage, `this` serves as an identity function, providing our neighborhoods a way of referring to themselves. We can’t always rely on that behavior, however: Depending on how we get into a particular neighborhood, `this` might mean something else entirely. In fact, *how we get into the neighborhood* is itself exactly what `this` generally refers to. Four scenarios deserve special attention:

- **Calling an Object's Method**In typical object-oriented programming, we need a way of identifying and referring to the object that we're currently working with. `this` serves the purpose admirably, providing our objects the ability to examine themselves, and point at their own properties.

  ```
  <script
  type="text/javascript">
  var deep_thought = {
  the_answer: 42,
  ask_question: function () {
  return this.the_answer;
  }
  };
  ```

  var the_meaning = deep_thought.ask_question();
  </script>

  This example builds an object named `deep_thought`, sets its `the_answer` property to 42, and creates an `ask_question` method. When `deep_thought.ask_question()` is executed, JavaScript establishes an execution context for the function call, setting `this` to the object referenced by whatever came before the last ".", in this case: `deep_thought`. The method can then look in the mirror via `this` to examine its own properties, returning the value stored in `this.the_answer`: 42.
- **Constructor**Likewise, when defining a function to be used as a constructor with the `new` keyword, `this` can be used to refer to the object being created. Let's rewrite the example above to reflect that scenario:

  ```
  <script
  type="text/javascript">
  function BigComputer(answer) {
  this.the_answer = answer;
  this.ask_question = function () {
  return this.the_answer;
  }
  }
  ```

var deep_thought = new BigComputer(42);
var the_meaning = deep_thought.ask_question();
</script>

Instead of explicitly creating the `deep_thought` object, we'll write a function to create `BigComputer` objects, and *instantiate* `deep_thought` as an instance variable via the `new` keyword. When `new BigComputer()` is executed, a completely new object is created transparently in the background. `BigComputer` is called, and its `this` keyword is set to reference that new object. The function can set properties and methods on `this`, which is transparently returned at the end of `BigComputer`'s execution.

Notice, though, that `deep_thought.the_question()` still works just as it did before. What's going on there? Why does `this` mean something different inside `the_question` than it does inside `BigComputer`? Put simply, we *entered* `BigComputer` via `new`, so `this` meant "the new object." On the other hand, we *entered* `the_question` via `deep_thought`, so while we're executing that method, `this` means "whatever `deep_thought` refers to". `this` is not read from the scope chain as other variables are, but instead is *reset* on a context by context basis.

- **Function Call**What if we just call a normal, everyday function without any of this fancy object stuff? What does `this` mean in that scenario? `<script type="text/javascript">`
```
function test_this() {
return this;
}
var i_wonder_what_this_is = test_this();
</script>
```

  In this case, we weren't provided a context by `new`, nor were we given a context in the form of an object to piggyback off of. Here, `this` /files/includes/default.csss to reference the most global thing it can: for web pages, this is the `window` object.
- **Event Handler**For a more complicated twist on the normal function call, let's say that we're using a function to handle an `onclick` event. What does `this` mean when the event triggers our function's execution? Unfortunately, there's

not a simple answer to this question.If we write the event handler inline, `this` refers to the global `window` object:

```
  <script type="text/javascript">
function click_handler() {
alert(this); // alerts the window object
}
</script>
...
<button id='thebutton' onclick='click_handler()'>Click me!
</button>
```

However, when we add an event handler via JavaScript, `this` refers to the DOM element that generated the event. (Note: The event handling shown here is short and readable, but otherwise poor. Please use a real addEvent function instead.):

```
  <script type="text/javascript">
function click_handler() {
alert(this); // alerts the button DOM node
}
```

function addhandler() {
document.getElementById('thebutton').onclick = click_handler;
}

window.onload = addhandler;
</script>
…
<button id='thebutton'>Click me!</button>

Complications

Let's run with that last example for a moment longer. What if instead of running `click_handler`, we wanted to ask `deep_thought` a question every time we clicked the button? The code for that seems pretty straightforward; we might try this:

```
<script type="text/javascript">
function BigComputer(answer) {
this.the_answer = answer;
this.ask_question = function () {
alert(this.the_answer);
}
}

function addhandler() {
var deep_thought = new BigComputer(42),
the_button = document.getElementById(‘thebutton’);

the_button.onclick = deep_thought.ask_question;
}

window.onload = addhandler;
</script>
```

Perfect, right? We click on the button, `deep_thought.ask_question` is executed, and we get back “42.” So why is the browser giving us `undefined` instead? What did we do wrong?

The problem is simply this: We’ve passed off a reference to the `ask_question` method, which, when executed as an event handler, runs in a different context than when it’s executed as an object method. In short, the `this` keyword in `ask_question` is pointing at the DOM element that generated the event, not at a `BigComputer` object. The DOM element doesn’t have a `the_answer` property, so we’re getting back `undefined` instead of “42.” `setTimeout` exhibits similar behavior, delaying the execution of a function while at the same time moving it out into a global context.

This issue crops up all over the place in our programs, and it’s a terribly difficult problem to debug without keeping careful track of what’s going on in all the corners of your program, especially if your object has properties that *do* exist on DOM elements or the `window` object.

Manipulating Context With `.apply()` and `.call()`

We really *do* want to be able to ask `deep_thought` a question when we click the button, and more generally, we *do* want to be able to call object methods in their native context when responding to things like events and `setTimeout` calls. Two little-known JavaScript methods, `apply` and `call`, indirectly enable this functionality by allowing us to manually override the /files/includes/default.css value of `this` when we execute a function call. Let's look at `call` first:

```
 <script type="text/javascript">
var first_object = {
num: 42
};
var second_object = {
num: 24
};

function multiply(mult) {
return this.num * mult;
}

multiply.call(first_object, 5); // returns 42 * 5
multiply.call(second_object, 5); // returns 24 * 5
</script>
```

In this example, we first define two objects, `first_object` and `second_object`, each with a `num` property. Then we define a `multiply` function that accepts a single argument, and returns the product of that argument, and the `num` property of its `this` object. If we called that function by itself, the answer returned would almost certainly be `undefined`, since the global `window` object doesn't have a `num` property unless we explicitly set one. We need some way of telling `multiply` what its `this` keyword ought refer to; the `call` method of the `multiply` function is exactly what we're looking for.

The first argument to `call` defines what `this` means inside the executed function. The remaining arguments to `call` are passed into the executed function, just as if you'd called it yourself. So, when `multiply.call(first_object, 5)` is executed, the `multiply` function is called, `5` is passed in as the first argument, and the `this` keyword is set to refer to object `first_object`. Likewise, when `multiply.call(second_object, 5)` is executed, the `multiply` function is called, `5` is passed in as the first argument, and the `this` keyword is set to refer to object `second_object`.

`apply` works in exactly the same way as `call`, but allows you to wrap up the arguments to the called function in an array, which can be quite useful when programatically generating function calls. Replicating the functionality we just talked about using `apply` is trivial:

```
<script type="text/javascript">
...
```

multiply.apply(first_object, [5]); // returns 42 * 5
multiply.apply(second_object, [5]); // returns 24 * 5
</script>

`apply` and `call` are very useful on their own, and well worth keeping around in your toolkit, but they only get us halfway to solving the problem of context shifts for event handlers. It’s easy to think that we could solve the problem by simply using `call` to shift the meaning of `this` when we set up the handler:

```
function addhandler() {
var deep_thought = new BigComputer(42),
the_button = document.getElementById('thebutton');
```

the_button.onclick = deep_thought.ask_question.call(deep_thought);
}

The problem with this line of reasoning is simple: `call` executes the function *immediately*. Instead of providing a function reference to the `onclick` handler, we’re giving it *the result* of an executed function. We need to exploit another feature of JavaScript to really solve this problem.

The Beauty of `.bind()`

I’m not a *huge* fan of the Prototype JavaScript framework, but I am very much impressed with the quality of its code as a whole. In particular, one simple addition it makes to the `Function` object has had a hugely positive impact on my ability to manage the context in which function calls execute: `bind` performs the same general task as `call` , altering the context in which a function executes. The difference is that `bind` returns a function reference that can be used later, rather than the result of an immediate execution that we get with `call` .

If we simplify the `bind` function a bit to get at the key concepts, we can insert it into the multiplication example we discussed earlier to really dig into how it works; it's quite an elegant solution:

```
 <script type="text/javascript">
var first_object = {
num: 42
};
var second_object = {
num: 24
};

function multiply(mult) {
return this.num * mult;
}

Function.prototype.bind = function(obj) {
var method = this,
temp = function() {
return method.apply(obj, arguments);
};

return temp;
}

var first_multiply = multiply.bind(first_object);
first_multiply(5); // returns 42 * 5

var second_multiply = multiply.bind(second_object);
second_multiply(5); // returns 24 * 5
</script>
```

First, we define `first_object` , `second_object` , and the `multiply` function, just as before. With those taken care of, we move on to creating a `bind` method on the `Function` object's `prototype` , which has the effect of making `bind` available for all functions in our program. When `multiply.bind(first_object)` is called, JavaScript creates an execution context for the `bind` method, setting `this` to the `multiply` function, and setting the first argument, `obj`, to reference `first_object`. So far, so good.

The real genius of this solution is the creation of `method`, set equal to `this` (the `multiply` function itself). When the anonymous function is created on the next line, `method` is accessible via its scope chain, as is `obj` (`this` couldn't be used here, because when the newly created function is executed, `this` will be overwritten by a new, local context). This alias to `this` makes it possible to use `apply` to execute the `multiply` function, passing in `obj` to ensure that the context is set correctly. In computer-science-speak, `temp` is a *closure* that, when returned at the end of the `bind` call, can be used in any context whatsoever to execute `multiply` in the context of `first_object`.

This is exactly what we need for the event handler and `setTimeout` scenarios discussed above. The following code solves that problem completely, binding the `deep_thought.ask_question` method to the `deep_thought` context, so that it executes correctly whenever the event is triggered:

```
 function addhandler() {
var deep_thought = new BigComputer(42),
the_button = document.getElementById('thebutton');
```

the_button.onclick = deep_thought.ask_question.bind(deep_thought);
}

Beautiful.

References

- JavaScript Closures is the best resource on the net for a thorough discussion of closures: what they do, how they do it, and how to use them without going insane.
- The Protype JavaScript Framework is full of little nuggets like `bind`. The version available here not only allows the binding of a particular `this` value, but also of some or all of a function's arguments, which comes in handy all too often.
- Douglas Crockford's JavaScript essays are excellent resources for both basic and advanced JavaScript programmers. The man knows what he's talking

about, and explains difficult concepts in an easy-to-grasp manner.
- Variable Scope for New Programmers is a good article if you'd like more discussion of scope from a beginner's perspective. Written by Jonathan Snook, and published in this very magazine at the end of last year, it's still an informative and useful read.

Got something to say?

Share your comments with other professionals (2 comments)

- Add to Magnolia
- Add to My Yahoo!
- Add to Newsvine
- Digg this story
- Add to Del.icio.us

Related Topics: Scripting, Programming

Mike West abandoned suburban Texas' wide open plains in 2005 in favour of the Black Forest in Southern Germany where he currently lives and works. His musings about the web are periodically posted to his personal website, mikewest.org.

Digiwebs

Comments are closed, but trackbacks and pingbacks are open.