Menu

Understanding JavaScript Function Invocation and "this"

Yehuda Katz | 10 Aug 2011 | 5 min read

Over the years, I've seen a lot of confusion about JavaScript function invocation. In particular, a lot of people have complained that the semantics of `this` in function invocations is confusing.

In my opinion, a lot of this confusion is cleared up by understanding the core function invocation primitive, and then looking at all other ways of invoking a function as sugar on top of that primitive. In fact, this is exactly how the ECMAScript spec thinks about it. In some areas, this post is a simplification of the spec, but the basic idea is the same.

The Core Primitive

First, let's look at the core function invocation primitive, a Function's `call` method[1]. The call method is relatively straight forward.

1. Make an argument list (`argList`) out of parameters 1 through the end
2. The first parameter is `thisValue`

3. Invoke the function with `this` set to `thisValue` and the `argList` as its argument list

For example:

```
function hello(thing) {
  console.log(this + " says hello " + thing);
}

hello.call("Yehuda", "world") //=> Yehuda says hello world
```

As you can see, we invoked the `hello` method with `this` set to `"Yehuda"` and a single argument `"world"`. This is the core primitive of JavaScript function invocation. You can think of all other function calls as desugaring to this primitive. (to "desugar" is to take a convenient syntax and describe it in terms of a more basic core primitive).

[1] In the ES5 spec, the `call` method is described in terms of another, more low level primitive, but it's a very thin wrapper on top of that primitive, so I'm simplifying a bit here. See the end of this post for more information.

Simple Function Invocation

Obviously, invoking functions with `call` all the time would be pretty annoying. JavaScript allows us to invoke functions directly using the parens syntax (`hello("world")`. When we do that, the invocation desugars:

```
function hello(thing) {
  console.log("Hello " + thing);
}

```

```
// this:
hello("world")

// desugars to:
hello.call(window, "world");
```

This behavior has changed in ECMAScript 5 **only when using strict mode**[2]:

```
// this:
hello("world")

// desugars to:
hello.call(undefined, "world");
```

The short version is: **a function invocation like `fn(...args)` is the same as `fn.call(window [ES5-strict: undefined], ...args)`**.

Note that this is also true about functions declared inline: `(function() {})()` is the same as `(function() {}).call(window [ES5-strict: undefined)`.

[2] Actually, I lied a bit. The ECMAScript 5 spec says that `undefined` is (almost) always passed, but that the function being called should change its `thisValue` to the global object when not in strict mode. This allows strict mode callers to avoid breaking existing non-strict-mode libraries.

Member Functions

The next very common way to invoke a method is as a member of an object (`person.hello()`). In this case, the invocation desugars:

```
var person = {
  name: "Brendan Eich",
  hello: function(thing) {
    console.log(this + " says hello " + thing);
  }
}

// this:
person.hello("world")

// desugars to this:
person.hello.call(person, "world");
```

Note that it doesn't matter how the `hello` method becomes attached to the object in this form. Remember that we previously defined `hello` as a standalone function. Let's see what happens if we attach is to the object dynamically:

```
function hello(thing) {
  console.log(this + " says hello " + thing);
}

person = { name: "Brendan Eich" }
person.hello = hello;

person.hello("world") // still desugars to person.hello.call(perso

hello("world") // "[object DOMWindow]world"
```

Notice that the function doesn't have a persistent notion of its 'this'. It is always set at call time based upon the way it was invoked by its caller.

Using `Function.prototype.bind`

Because it can sometimes be convenient to have a reference to a function with a persistent `this` value, people have historically used a simple closure trick to convert a function into one with an unchanging `this`:

```javascript
var person = {
  name: "Brendan Eich",
  hello: function(thing) {
    console.log(this.name + " says hello " + thing);
  }
}

var boundHello = function(thing) { return person.hello.call(perso

boundHello("world");
```

Even though our `boundHello` call still desugars to `boundHello.call(window, "world")`, we turn right around and use our primitive `call` method to change the `this` value back to what we want it to be.

We can make this trick general-purpose with a few tweaks:

```javascript
var bind = function(func, thisValue) {
  return function() {
    return func.apply(thisValue, arguments);
  }
}

var boundHello = bind(person.hello, person);
boundHello("world") // "Brendan Eich says hello world"
```

In order to understand this, you just need two more pieces of information. First, `arguments` is an Array-like object that represents all of the arguments passed into a function. Second, the `apply` method works exactly like the `call` primitive, except that it takes an Array-like object instead of listing the arguments out one at a time.

Our `bind` method simply returns a new function. When it is invoked, our new function simply invokes the original function that was passed in, setting the original value as `this`. It also passes through the arguments.

Because this was a somewhat common idiom, ES5 introduced a new method `bind` on all `Function` objects that implements this behavior:

```
var boundHello = person.hello.bind(person);
boundHello("world") // "Brendan Eich says hello world"
```

This is most useful when you need a raw function to pass as a callback:

```
var person = {
  name: "Alex Russell",
  hello: function() { console.log(this.name + " says hello world")
}

$("#some-div").click(person.hello.bind(person));

// when the div is clicked, "Alex Russell says hello world" is pri
```

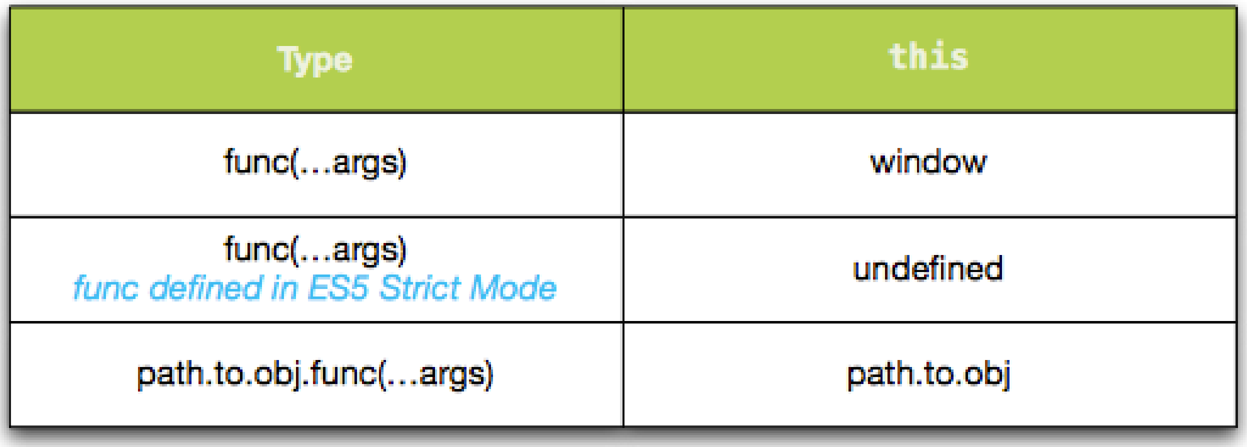
This is, of course, somewhat clunky, and TC39 (the committee that works on the next version(s) of ECMAScript) continues to work on a more elegant, still-backwards-compatible solution.

On jQuery

Because jQuery makes such heavy use of anonymous callback functions, it uses the `call` method internally to set the `this` value of those callbacks to a more useful value. For instance, instead of receiving `window` as `this` in all event handlers (as you would without special intervention), jQuery invokes `call` on the callback with the element that set up the event handler as its first parameter.

This is extremely useful, because the default value of `this` in anonymous callbacks is not particularly useful, but it can give beginners to JavaScript the impression that `this` is, **in general** a strange, often mutated concept that is hard to reason about.

If you understand the basic rules for converting a sugary function call into a desugared `func.call(thisValue, ...args)`, you should be able to navigate the not so treacherous waters of the JavaScript `this` value.



Type	`this`
func(...args)	window
func(...args) *func defined in ES5 Strict Mode*	undefined
path.to.obj.func(...args)	path.to.obj

PS: I Cheated

In several places, I simplified the reality a bit from the exact wording of the specification. Probably the most important cheat is the way I called `func.call` a "primitive". In reality, the spec has a primitive (internally referred to as `[[Call]]`) that both `func.call` and `[obj.]func()` use.

However, take a look at the definition of `func.call`:

1. If IsCallable(func) is false, then throw a TypeError exception.
2. Let argList be an empty List.
3. If this method was called with more than one argument then in left to right order starting with arg1 append each argument as the last element of argList
4. Return the result of calling the [[Call]] internal method of func, providing thisArg as the this value and argList as the list of arguments.

As you can see, this definition is essentially a very simple JavaScript language binding to the primitive `[[Call]]` operation.

If you look at the definition of invoking a function, the first seven steps set up `thisValue` and `argList`, and the last step is: "Return the result of calling the [[Call]] internal method on func, providing thisValue as the this value and providing the list argList as the argument values."

It's essentially identical wording, once the `argList` and `thisValue` have been determined.

I cheated a bit in calling `call` a primitive, but the meaning is essentially the same as had I pulled out the spec at the beginning of this article and quoted chapter and verse.

There are also some additional cases (most notably involving `with`) that I didn't cover here.

Share:

Yehuda Katz

Understanding "Prototypes" in JavaScript

For the purposes of this post, I will be talking about JavaScript…

What's Up With All These Changes in Rails?

Yesterday, there was a blog post entitled "What the Hell is…