JavaScript Module Pattern InDepth

http://www.adequatelygood.com/JavaScript-Module-Pattern-In-Depth.html
2015-09-21 11:30 AM

adequately good(/)

decent programming advice

written by ben cherry(http://twitter.com/bcherry)

2010-03-12

JavaScript Module Pattern: In-Depth(/JavaScript-Module-Pattern-In-Depth.html)

The module pattern is a common JavaScript coding pattern. It's generally well understood, but there are a number of advanced uses that have not gotten a lot of attention. In this article, I'll review the basics and cover some truly remarkable advanced topics, including one which I think is original.

The Basics

We'll start out with a simple overview of the module pattern, which has been well-known since Eric Miraglia (of YUI) first blogged about it(http://yuiblog.com/blog/2007/06/12/module-pattern/) three years ago. If you're already familiar with the module pattern, feel free to skip ahead to "Advanced Patterns".

Anonymous Closures

This is the fundamental construct that makes it all possible, and really is the single **best feature of JavaScript**. We'll simply create an anonymous function, and execute it immediately. All of the code that runs inside the function lives in a **closure**, which provides **privacy** and **state** throughout the lifetime of our application.

```
(function () {
        // ... all vars and functions are in this scope only
        // still maintains access to all globals
```

```
}());
```

Notice the `()` around the anonymous function. This is required by the language, since statements that begin with the token `function` are always considered to be **function declarations**. Including `()` creates a **function expression** instead.

Global Import

JavaScript has a feature known as **implied globals**. Whenever a name is used, the interpreter walks the scope chain backwards looking for a `var` statement for that name. If none is found, that variable is assumed to be global. If it's used in an assignment, the global is created if it doesn't already exist. This means that using or creating global variables in an anonymous closure is easy. Unfortunately, this leads to hard-to-manage code, as it's not obvious (to humans) which variables are global in a given file.

Luckily, our anonymous function provides an easy alternative. By passing globals as parameters to our anonymous function, we **import** them into our code, which is both **clearer** and **faster** than implied globals. Here's an example:

```
(function ($, YAHOO) {
        // now have access to globals jQuery (as $) and YAHOO in this code
}(jQuery, YAHOO));
```

Module Export

Sometimes you don't just want to *use* globals, but you want to *declare* them. We can easily do this by exporting them, using the anonymous function's **return value**. Doing so will complete the basic module pattern, so here's a complete example:

```
var MODULE = (function () {
        var my = {},
                privateVariable = 1;

        function privateMethod() {
                // ...
        }

        my.moduleProperty = 1;
        my.moduleMethod = function () {
                // ...
        };

        return my;
}());
```

Notice that we've declared a global module named `MODULE`, with two public properties: a method named `MODULE.moduleMethod` and a variable named `MODULE.moduleProperty`. In addition, it maintains **private internal state** using the closure of the anonymous function. Also, we can easily import needed globals, using the pattern we learned above.

Advanced Patterns

While the above is enough for many uses, we can take this pattern farther and create some very powerful, extensible constructs. Lets work through them one-by-one, continuing with our module named `MODULE`.

Augmentation

One limitation of the module pattern so far is that the entire module must be in one file. Anyone who has worked in a large code-base understands the value of splitting among multiple files. Luckily, we have a nice solution to **augment modules**. First, we import the module, then we add properties, then we export it. Here's an example, augmenting our `MODULE` from above:

```
var MODULE = (function (my) {
        my.anotherMethod = function () {
                // added method...
        };

        return my;
}(MODULE));
```

We use the `var` keyword again for consistency, even though it's not necessary. After this code has run, our module will have gained a new public method named `MODULE.anotherMethod`. This augmentation file will also maintain its own private internal state and imports.

Loose Augmentation

While our example above requires our initial module creation to be first, and the augmentation to happen second, that isn't always necessary. One of the best things a JavaScript application can do for performance is to load scripts asynchronously. We can create flexible multi-part modules that can load themselves in any order with **loose augmentation**. Each file should have the following structure:

```
var MODULE = (function (my) {
        // add capabilities...

        return my;
}(MODULE || {}));
```

In this pattern, the `var` statement is always necessary. Note that the import will create the module if it does not already exist. This means you can use a tool like **LABjs(http://labjs.com/)** and load all of your module files in parallel, without needing to block.

Tight Augmentation

While loose augmentation is great, it does place some limitations on your module. Most importantly, you cannot override module properties safely. You also cannot use module properties from other files during initialization (but you can at run-time after intialization). **Tight augmentation** implies a set loading order, but allows **overrides**. Here is a simple example (augmenting our original `MODULE`):

```
var MODULE = (function (my) {
	var old_moduleMethod = my.moduleMethod;

	my.moduleMethod = function () {
		// method override, has access to old through old_moduleMethod...
	};

	return my;
}(MODULE));
```

Here we've overridden `MODULE.moduleMethod`, but maintain a reference to the original method, if needed.

Cloning and Inheritance

```
var MODULE_TWO = (function (old) {
	var my = {},
		key;

	for (key in old) {
		if (old.hasOwnProperty(key)) {
			my[key] = old[key];
		}
	}

	var super_moduleMethod = old.moduleMethod;
	my.moduleMethod = function () {
		// override method on the clone, access to super through super_moduleMethod
	};

	return my;
}(MODULE));
```

This pattern is perhaps the **least flexible** option. It does allow some neat compositions, but that comes at the expense of flexibility. As I've written it, properties which are objects or functions will *not* be duplicated, they will exist as one object with two references. Changing one will change the other. This could be fixed for objects with a recursive cloning process, but probably cannot be fixed for functions, except perhaps with `eval`. Nevertheless, I've included it for completeness.

Cross-File Private State

One severe limitation of splitting a module across multiple files is that each file maintains its own private state, and does not get access to the private state of the other files. This can be fixed. Here is an example of a loosely augmented module that will **maintain private state** across all augmentations:

```
var MODULE = (function (my) {
	var _private = my._private = my._private || {},
		_seal = my._seal = my._seal || function () {
			delete my._private;
			delete my._seal;
```

```
                delete my._unseal;
        },
        _unseal = my._unseal = my._unseal || function () {
                my._private = _private;
                my._seal = _seal;
                my._unseal = _unseal;
        };

    // permanent access to _private, _seal, and _unseal

    return my;
}(MODULE || {}));
```

Any file can set properties on their local variable `_private`, and it will be immediately available to the others. Once this module has loaded completely, the application should call `MODULE._seal()`, which will prevent external access to the internal `_private`. If this module were to be augmented again, further in the application's lifetime, one of the internal methods, in any file, can call `_unseal()` before loading the new file, and call `_seal()` again after it has been executed. This pattern occurred to me today while I was at work, I have not seen this elsewhere. I think this is a very useful pattern, and would have been worth writing about all on its own.

Sub-modules

Our final advanced pattern is actually the simplest. There are many good cases for creating sub-modules. It is just like creating regular modules:

```
MODULE.sub = (function () {
    var my = {};
    // ...

    return my;
}());
```

While this may have been obvious, I thought it worth including. Sub-modules have all the advanced capabilities of normal modules, including augmentation and private state.

Conclusions

Most of the advanced patterns can be combined with each other to create more useful patterns. If I had to advocate a route to take in designing a complex application, I'd combine **loose augmentation**, **private state**, and **sub-modules**.

I haven't touched on performance here at all, but I'd like to put in one quick note: The module pattern is **good for performance**. It minifies really well, which makes downloading the code faster. Using **loose augmentation** allows easy non-blocking parallel downloads, which also speeds up download speeds. Initialization time is probably a bit slower than other methods, but worth the trade-off. Run-time performance should suffer no penalties so long as globals are imported correctly, and will probably gain speed in sub-modules by shortening the reference chain with local variables.

To close, here's an example of a sub-module that loads itself dynamically to its parent (creating it if it does not exist). I've left out private state for brevity, but including it would be simple. This code pattern allows an entire complex heirarchical

code-base to be loaded completely in parallel with itself, sub-modules and all.

```
var UTIL = (function (parent, $) {
        var my = parent.ajax = parent.ajax || {};

        my.get = function (url, params, callback) {
                // ok, so I'm cheating a bit :)
                return $.getJSON(url, params, callback);
        };

        // etc...

        return parent;
}(UTIL || {}, jQuery));
```

I hope this has been useful, and please leave a comment to share your thoughts. Now, go forth and write better, more modular JavaScript!

This post was featured on Ajaxian.com(http://ajaxian.com/archives/a-deep-dive-and-analysis-of-the-javascript-module-pattern), and there is a little bit more discussion going on there as well, which is worth reading in addition to the comments below.

filed under javascript(/tag/javascript) and module pattern(/tag/module pattern)

So do I (3 years later!), and also recomend this one (which brought me here): http://benalman.com/news/2010/...

34 Reply

Matt Langston → Jeisson Guevara
3 years ago

As do I. This article was amazing.

3 Reply

Shaun Walters → Matt Langston
2 years ago

So do I (4 years later!) :)

11 Reply

CJ Johnson → Shaun Walters
10 months ago

And so do I, 5 years later! I knew about the format, but this goes it to far more detail! I had no idea just how amazing it was.

Although I am confused, whats the difference between these two as far as privacy, speed, etc?

(function(){})();

(function(){}());

0 Reply

Colin Richardson → CJ Johnson
10 months ago

Nothing, I am not sure why he states () are required by the language to make the closure. It is just a user technique people use to denote that this function that is being created is going to call itself.

1 Reply

skube → CJ Johnson
7 months ago

AFAIK there is no difference in neither privacy nor speed. They are essentially the same. Though Crockford (and therefore JSLint) would prefer the latter over the former.

1 Reply

Ben Cherry ↗ skube
7 months ago

it doesn't matter whether you put the calling parentheses in or outside the wrapping ones, but the leading paren before `function` is absolutely necessary since otherwise the lexer tries to parse the line as a function declaration. Some people use a single `!` in front of the leading `function` to avoid having a closing paren at the end. But yeah, I wrote this article while firmly in the Crockford School of Thought so I advocated his beliefs on proper syntax here.

0 Reply

Elvis Attro ↗ Shaun Walters
9 months ago

Yet another " So do I (4 years later) :) " ... Just awesome !

0 Reply

David ↗ Shaun Walters
8 months ago

Me too! (5 years later whaaat)

0 Reply

evanhutomo ↗ Shaun Walters
6 months ago

So do I (5 years later!) :))

0 Reply

Tiantian Gao ↗ evanhutomo
3 months ago

So do I (5 years later)

0 Reply

Brennan Lawrence ↗ Jeisson Guevara
3 years ago

I agree with the time traveller from the year 2013 (glad to know the world doesn't end)!

11 Reply

AlexZ ↗ Brennan Lawrence
3 years ago

So do I, and hope to see more examples for each of them...if the world doesn't end

1 Reply

Vikram Bodicherla → Brennan Lawrence
3 years ago

Amazing article! Thank you.

This is the time traveller from 2013. A time-machine has been invented and I travelled back 15 days to ask you guys to run for your lives, since the world is going to end in 2013!

1 Reply

Guest → Brennan Lawrence
3 years ago

This article saved the world from apocalypse in 2013.

24 Reply

Guest → Brennan Lawrence
a year ago

2014 signing in!

3 Reply

Gavin Elster → Jeisson Guevara
3 years ago

+1. Super helpful and still relevant!

4 Reply

Zarne Dravitzki → Jeisson Guevara
2 years ago

Likewise.. 4 years later!

4 Reply

Load more comments

the author

Ben is a 25 year-old software engineer. He lives and works in San Francisco. Many people think he invented the term "hoisting" in JavaScript, but this is untrue.

- Twitter(http://twitter.com/bcherry) - @bcherry(http://twitter.com/bcherry)
- GitHub(http://github.com/bcherry) - My Code
- LinkedIn(http://www.linkedin.com/in/bcherryprogrammer) - Professional Profile
- Facebook(http://www.facebook.com/bcherry) - That Other Social Network
- Presentations(http://www.bcherry.net/talks/) - Slides From My Talks

---

categories

- javascript(/tag/javascript) (21)
- social gaming(/tag/social%20gaming) (1)
- css(/tag/css) (1)
- jquery(/tag/jquery) (2)
- performance(/tag/performance) (5)
- tools(/tag/tools) (2)
- html5(/tag/html5) (3)
- adequatelygood(/tag/adequatelygood) (1)
- timers(/tag/timers) (2)
- module pattern(/tag/module%20pattern) (3)
- talks(/tag/talks) (1)
- slide(/tag/slide) (1)
- python(/tag/python) (1)
- debugging(/tag/debugging) (1)
- testing(/tag/testing) (2)
- hashbang(/tag/hashbang) (1)
