JavaScript: using closure space to create real private members

David Catuhe, 1 Jun 2015 CPOL

Rate: ☆☆☆☆☆ 0.00 (No votes)

In this tutorial, I want to share with you how to use this for your own projects and how performance and memory are impacted for the major browsers.

I recently developed Angular Cloud Data Connector, which enables Angular developers to use cloud data, specifically Azure mobile service, using web standards like indexed DB. I was trying to create a way for JavaScript developers to embed private members into an object. My technique for this specific case is to use what I call "closure space". In this tutorial, I want to share with you how to use this for your own projects and how performance and memory are impacted for the major browsers.

But before diving into it, let me share why you may need private members, as well as an alternate way to "simulate" private members.

Feel free to ping me on twitter if you want to discuss about this article: @deltakosh

Why use private members

When you create an object using JavaScript, you can define value members. If you want to control read/write access on them, you need accessors that can be defined like this:

Hide Copy Code

```
var entity = {};

entity._property = "hello world";
Object.defineProperty(entity, "property", {
    get: function () { return this._property; },
    set: function (value) {
        this._property = value;
    },
    enumerable: true,
    configurable: true
});
```

Doing this, you have full control over read and write operations. The problem is that the `_property` member is still accessible and can be modified directly.

This is exactly why you need a more robust way to define private members that can only be accessed by object's functions.

Using closure space

The solution is to use closure space. This memory space is built for you by the browser each time an inner function has access to variables from the scope of an outer function. This can be tricky sometimes, but for our topic this is a perfect solution.

So let's alter the previous code to use this feature:

Hide Copy Code

```
var createProperty = function (obj, prop, currentValue) {
    Object.defineProperty(obj, prop, {
        get: function () { return currentValue; },
        set: function (value) {
            currentValue = value;
        },
        enumerable: true,
        configurable: true
    });
}

var entity = {};

var myVar = "hello world";
createProperty(entity, "property", myVar);
```

In this example, the `createProperty` function has a `currentValue` variable that get and set functions can see. This variable is going to be saved in the closure space of get and set functions. Only these two functions can now see and update the `currentValue` variable! Mission accomplished!

The only caveat we have here is that the source value (`myVar`) is still accessible. So here comes another version for even more robust protection:

Hide Copy Code

```
var createProperty = function (obj, prop) {
    var currentValue = obj[prop];
    Object.defineProperty(obj, prop, {
        get: function () { return currentValue; },
        set: function (value) {
            currentValue = value;
        },
        enumerable: true,
        configurable: true
    });
}

var entity = {
    property: "hello world"
};

createProperty(entity, "property");
```

Using this method, even the source value is destructed. So mission fully accomplished!

Performance consideration

Let's now have a look at performance.

Obviously, closure spaces or even properties are slower and more expensive than just a plain variable. That's why this article focuses more on the difference between regular way and closure space technique.

To confirm the closure space approach is not too expensive compared to the standard way, I wrote this little benchmark:

Hide Copy Code

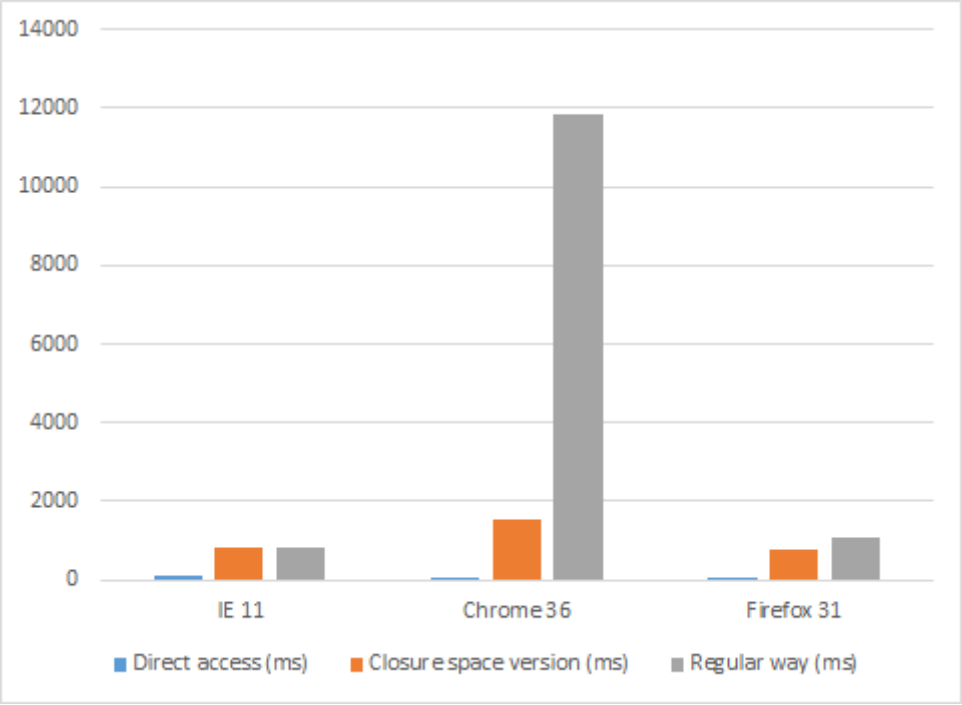
```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title></title>
```

I create 1 million objects, all with a property member. Then I do three tests:

- Do 1 million random accesses to the property
- Do 1 million random accesses to the "closure space" version
- Do 1 million random accesses to the regular get/set version

Here are a table and a chart about the result:

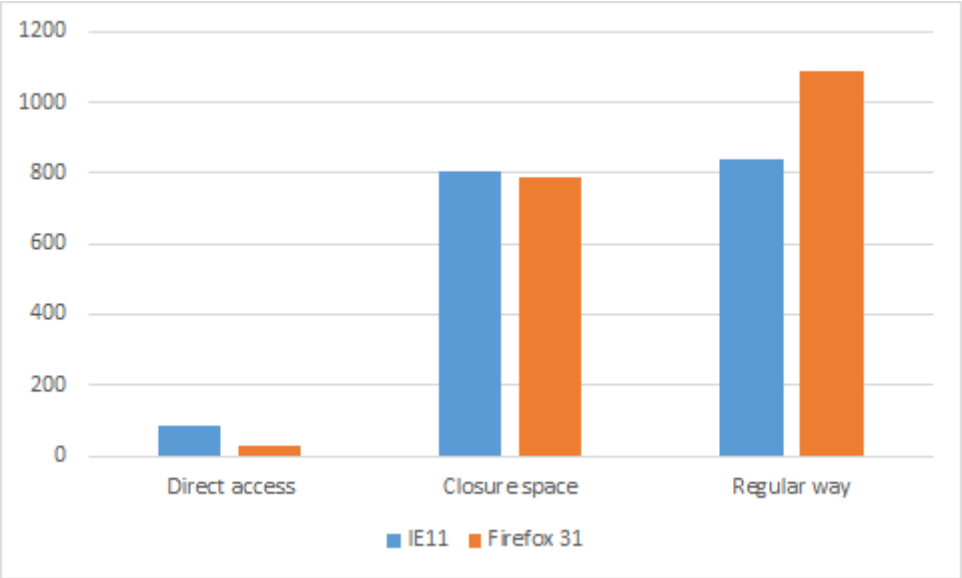
Browser	Direct access (ms)	Closure space version (ms)	Regular way
IE 11	84	806	840
Chrome 36	32	1550	11845
Firefox 31	31	788	1086



We can see that the closure space version is always faster than the regular version and depending on the browser, **it can be a really impressive optimization**.

Chrome performance is less than I expected. There may be a bug so to be sure, I contacted Google's team to figure out what's happening here. Also if you want to test how this performs in Microsoft Edge – Microsoft's new browser that will ship default with Windows 10 – you can download it here.

However, if we look closely we can find that using closure space or even a property can be ten times slower than direct access to a member. So be warned and use it wisely.



Memory footprint

We also have to check if this technique does not consume too much memory. To benchmark memory I wrote these three little pieces of code:

Reference code

Hide Copy Code

```
var sampleSize = 1000000;

var entities = [];

// Creating entities
for (var index = 0; index < sampleSize; index++) {
    entities.push({
        property: "hello world (" + index + ")"
    });
}
```

Regular way

Hide Copy Code

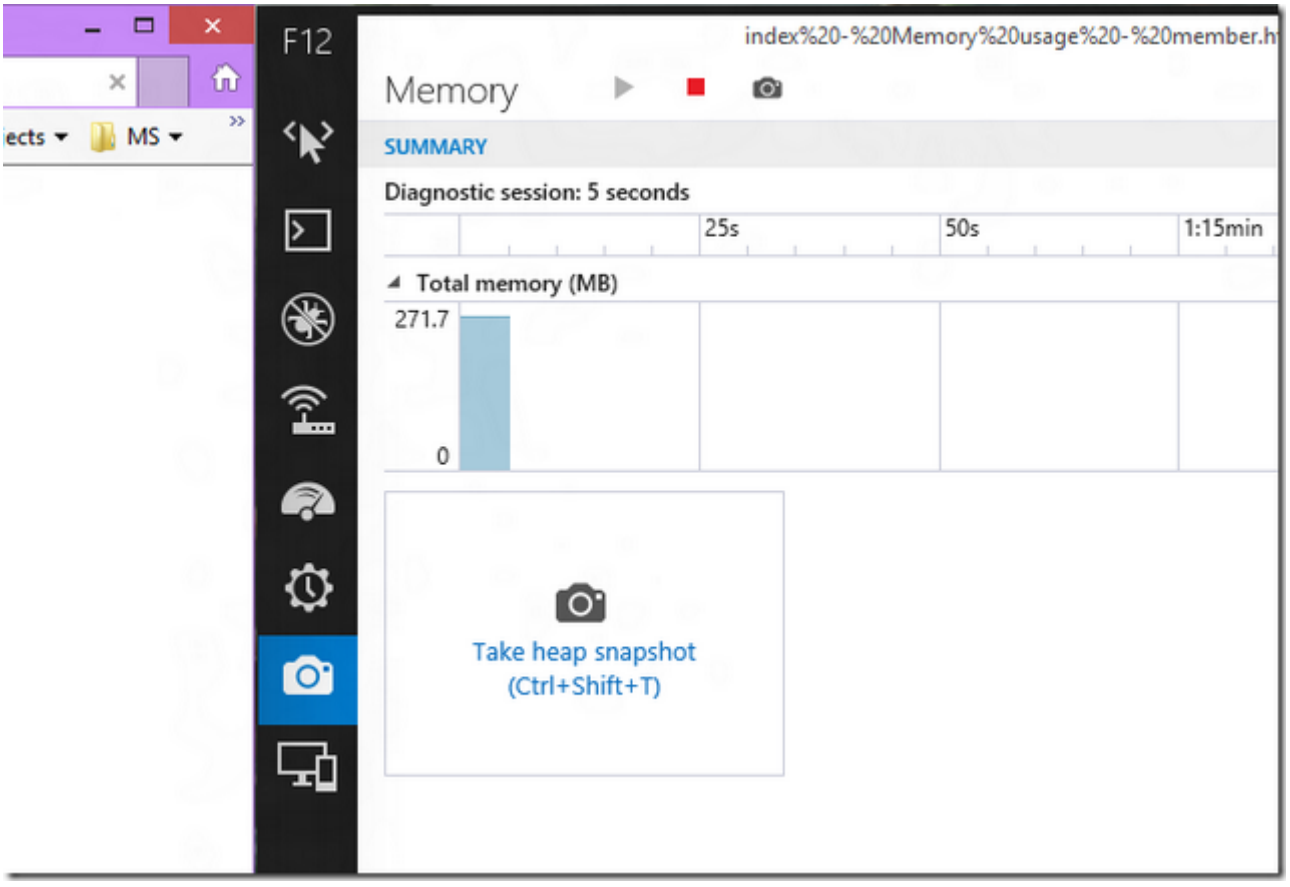
```
var sampleSize = 1000000;

var entities = [];

// Adding property and using local member to save private value
for (var index = 0; index < sampleSize; index++) {
    var entity = {};

    entity._property = "hello world (" + index + ")";
    Object.defineProperty(entity, "property", {
        get: function () { return this._property; },
        set: function (value) {
            this._property = value;
        },
        enumerable: true,
        configurable: true
    });

    entities.push(entity);
}
```

Closure space version

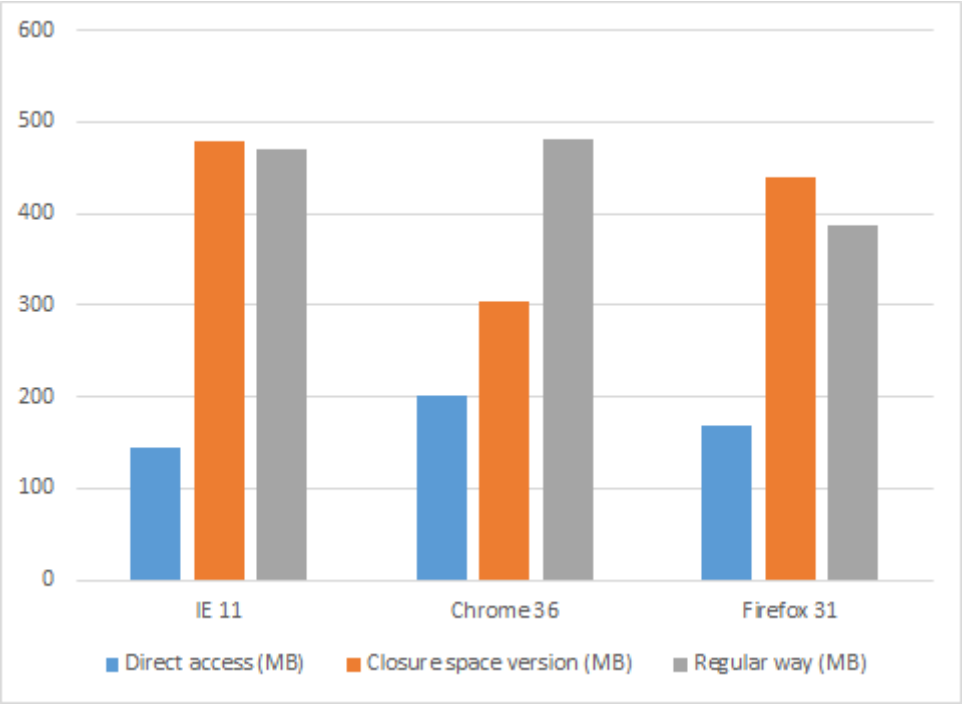
Hide Copy Code

```
var sampleSize = 1000000;

var entities = [];

var createProperty = function (obj, prop, currentValue) {
    Object.defineProperty(obj, prop, {
        get: function () { return currentValue; },
        set: function (value) {
            currentValue = value;
        },
        enumerable: true,
        configurable: true
    });
}

// Adding property and using closure space to save private value
for (var index = 0; index < sampleSize; index++) {
    var entity = {};

    var currentValue = "hello world (" + index + ")";
    createProperty(entity, "property", currentValue);

    entities.push(entity);
}
```

Then I ran all these three codes and I launched the embedded memory profiler (Example here using F12 tools):



Here are the results I got on my computer:



Between closure space and regular way, only Chrome has slightly better results for closure space version. IE11 and Firefox use a bit more memory but the browsers are relatively comparable – users probably won't notice a difference across the modern browsers.

More hands-on with JavaScript

It might surprise you a bit, but Microsoft has a bunch of free learning on many open source JavaScript topics and we're on a mission to create a lot more with Microsoft Edge coming. Check out my own:

- Introduction to WebGL 3D and HTML5 and Babylon.JS
- Building a Single Page Application with ASP.NET and AngularJS
- Cutting Edge Graphics in HTML

Or our team's learning series:

- Practical Performance Tips to Make your HTML/JavaScript Faster (a 7-part series from responsive design to casual games to performance optimization)
- The Modern Web Platform JumpStart (the fundamentals of HTML, CSS, and JS)
- Developing Universal Windows App with HTML and JavaScript JumpStart (use the JS you've already created to build an app)

And some free tools: Visual Studio Community, Azure Trial, and cross-browser testing tools for Mac, Linux, or Windows.

Conclusion

As you can see, closure space properties can be a **great way to create really private data**. You may have to deal with a small increase in memory consumption but from my point of view this is fairly reasonable (and at that price you can have a great performance improvement over using the regular way).

And by the way if you want to try it by yourself, please find all the code used here. There's a good "how-to" on Azure Mobile Services here.

This article is part of the web dev tech series from Microsoft. We're excited to share *Microsoft Edge* and its *new rendering engine* with you. Get free virtual machines or test remotely on your Mac, iOS, Android, or Windows device @ *modern.IE*.

License

Share

EMAIL TWITTER

About the Author

David Catuhe

United States

David Catuhe is a Principal Program Manager at Microsoft focusing on web development. He is author of the babylon.js framework for building 3D games with HTML5 and WebGL. Read his blog on MSDN or follow him @deltakosh on Twitter.

Comments and Discussions

Add a Comment or Question

Search Comments Go

-- There are no messages in this forum --