Notes - JavaScript - ES Modules

- Dr Nick Hayward

A collection of notes &c. on plain JavaScript modules, in particular fundamental structural design for ES modules introduced with ES2015.

Contents

- Intro
- Variables in JavaScript
    - global issues
- Modular design
- ES modules
    - how they work
    - module record
    - module instance
    - asynchronous or synchronous
- Finding a module file
    - browser loading of module files
    - CommonJS blocking
- Parsing ES modules
- Instantiation of ES modules
    - instantiate module graph
    - ES module live bindings
- Evaluation
- Conclusion

Intro

Modules in JavaScript are not a new concept. For example, CommonJS is a popular option for modular development with Node.js.

However, with the introducion of ES2015 (ES6), we now have a built-in option for plain JavaScript, *ES Modules*.

Variables in JavaScript

As we develop with JavaScript, variables are a fundamental part of the way we build and manage the logic for our applications.

JavaScript helps us manage variables, and effectively limit their number in a given context, using *scope*. For a given app, we may only need to track and maintain a few variables per scope.

However, due to the nature of *scope*, functions in JavaScript are unable to access variables in other functions not in the same scope. A child function may access a parent function's variables, but not vice-versa.

This helps with context, structure, and logic. However, sharing variables may become an issue.

global issues

Resorting to *global* declaration is problematic, and inherently relies on scripts being run in the required order. jQuery's early reliance on *global* declarations is a familiar example.

Another issue with *global* scope is open access to these declared variables. Any code in the *global* scope now has access to these variables, including modification.

Modular design

Modules are an efficient option for organising such variables and functions into logical groupings per app.

We may *export* variables, functions &c. from a module, and then *import* as necessary in another module.

This simple option to import and export makes it easier to abstract and divide logic within an app. It can also help *decouple* app code, providing independent functionality and testing per module.

Then, we may combine modules as needed to create various patterns and usage within an app.

ES modules

ES modules are now supported by all major web browsers. Node.js has also started to add support.

how they work

As we develop apps with modular patterns, we define connections between dependencies using any of the available *import* statements. In effect, we create a graph of dependencies relative to each app.

In effect, a defined *import* statement provides an access point to the graph. The path and extent of the traversal is relative to this access point. It might be shallow or deep, thereby defining a module's access to functions and variables.

module record

We might define and save our code as modules in JavaScript files, but a browser will need to parse them into usable data structures. These are known as *module records*.

Such records inform the browser of the content and usage within an app's files.

A module record will then be turned into a *module instance*, which combines two things,

- code
    - a set of instructions - logic for the app...
- state
    - state defines variable values at a given snapshot
    - data relative to a given point in time for the execution of the app

Such variables, however, are merely named references to parts of the memory storing the values.

module instance

Module loading for each app requires an instance per module, and a full graph of all module instances.

ES modules reconcile this process as follows,

- construction
    - module records created for all files in the app
- instantiation
    - location in memory defined for all exported values
    - exports and imports point to these locations - known as *linking*
    - **n.b.** values are not yet added to these memory locations
- evaluation
    - code is run to fill memory locations with values...

asynchronous or synchronous

ES modules are often referred to as asynchronous. However, the above reconciliation steps are not necessarily asynchronous. Depending upon the loading mechanism for the modules, they may be synchronous.

The *ES module spec* defines how files should be parsed into module records, plus how they should be instantiated and evaluated.

However, it's interesting to note that this spec does not define how to initially load such files.

Each platform, such as browser, Node &c., may define a different loader spec.

Finding a module file

To work with ES modules in a browser platform, we need to tell the app where to find the file, e.g.

```
<script src="main.js" type="module">
```

This defines and loads the initial `main.js` file and module. To define dependencies, we may use `import` statements, e.g.

```
import { count } from './counter.js'
```

The file path, `./counter.js`, is known as the *module specifier*. This tells the loader where to find the next module, i.e. one of the defined dependencies.

n.b. each host and platform will have a specific way of handling such *module specifiers*. Each platform will reconcile such paths using a *module resolution algorithm*, which may mean some Node.js defined *specifiers* will not work in a browser.

browser loading of module files

Currently, a browser relies on URLs for the path of a *module specifier*.

However, this does not apply for the full graph of modules. In effect, the loader does not know the required dependencies until the file has been parsed. The file, of course, may not be parsed until it has been fetched.

So, the tree of app files needs to be processed layer by layer. A file is parsed, each dependency is noted, then found, and finally loaded.

To help ES modules handle this parsing and loading without blocking the single thread for JS, the ES module spec splits the algorithm into multiple phases. This allows the construction and instantiation to operate in separate phases.

A browser may fetch files, build up an understanding of the module graph, and then work on the instantiation.

CommonJS blocking

This split phase algorithm is a notable difference between ES and CommonJS modules.

This is because loading from the filesystem is inherently faster than loading from a URL - Node.js and CommonJS modules may, therefore, block the main thread whilst a file is loading. Also, ***instantiation*** and ***evaluation*** are not separate phases in CommonJS.

However, it also means that CommonJS is able to traverse the full module tree, load, instantiate, and evaluate any required dependencies before returning the module instance.

A benefit, in some contexts, to this approach is the option to use variables in the module specifier.

However, as ES modules need to build up the whole module graph beforehand, it is not possible to define variables as part of the URL for module specifiers.

Parsing ES modules

After fetching a file, we need to parse it into a ***module record***. As noted above, this helps the browser understand the required parts of a given module.

This module record is then added to the ***module map***. Subsequently, each request for this module will now be pulled from this map.

Each module is also parsed with `strict` mode enabled by default.

The value of `this` will be also be `undefined`.

To help browsers parse files, we may explicitly define a file as a module type, e.g.

```
<script src="main.js" type="module">
```

This also helps the browser know that any subsequent imports from this file will also be modules.

So, by the end of the loading process we now have module records from a given module file.

Instantiation

As noted above, an instance combines our app's ***code*** with its ***state***. The ***state*** will exist in memory, so instantiation is needed to link or connect things to memory.

The JavaScript will, firstly, create a *module environment record*, which manages the variables for the module record.

Then, it finds locations in the memory for all of the defined exports.

The *module environment record* tracks memory locations to exports.

As noted above, values will not be added to their respective memory location until *evaluation*.

However, there is an exception to this rule. If an export defines a function declaration, it will be initialised during this phase.

instantiate module graph

The JS engine will instantiate the module graph using a *depth first post-order traversal*.

So, it will start at the bottom of the graph with dependencies that do not depend on anything else, and setup their exports. It will setup all of the exports that a module relies upon.

The JS engine will then move back a level to wire up imports from that module.

n.b. both import and export will point to the same location in memory. So, by wiring up exports first, the JS engine guarantees all imports can be connected to their matching exports.

n.b.2 in CommonJS, the export object is copied on export, which means any exported values are copies. Therefore, if the exporting module subsequently changes a value, the importing module will not be aware of the change. An app's lifecycle will then need to restarted, for example, for the changes to appear in the app.

ES module live bindings

Another notable difference between CommonJS and ES modules is the use of *live bindings*.

With *live bindings* in ES modules, if a value changes all dependent modules will be informed as well. In effect, the export and import module point to the same location in memory for a given value &c.

In effect, if the exporting module updates a value, the importing module will also see this change.

However, only an exporting module will be able to update such a value. An importing module may not change the value of their imports. The only caveat is with standard object usage in JavaScript. An object's property may be modified, as expected, from an importing module.

Evaluation

In the final step, *evaluation*, we are now able to add these values to their defined locations in memory.

The JavaScript engine will complete this by executing all of the top-level code, i.e. code outside of any functions.

To avoid multiple evaluations of such code, which may include queries to servers, data stores &c, a module map will cache the module by URL. This ensures there is only one module record for each module. It also ensures that each module file is only executed once. This will follow the same pattern as *instantiation*, and use a depth first post-order traversal.

Potential side effects should be avoided using this process.

Conclusion

One of the inherent benefits of ES modules, and the above three phase design, is support for patterns such as cycles in module and code execution.

Live bindings, and the nature of this phased design, ensures values may be updated and used as needed within an app.