8th February 2020

Modules, introduction

As our application grows bigger, we want to split it into multiple files, so called “modules”. A module usually contains a class or a library of functions.

For a long time, JavaScript existed without a language-level module syntax. That wasn’t a problem, because initially scripts were small and simple, so there was no need.

But eventually scripts became more and more complex, so the community invented a variety of ways to organize code into modules, special libraries to load modules on demand.

For instance:

- AMD – one of the most ancient module systems, initially implemented by the library require.js.
- CommonJS – the module system created for Node.js server.
- UMD – one more module system, suggested as a universal one, compatible with AMD and CommonJS.

Now all these slowly become a part of history, but we still can find them in old scripts.

The language-level module system appeared in the standard in 2015, gradually evolved since then, and is now supported by all major browsers and in Node.js. So we’ll study it from now on.

What is a module?

A module is just a file. One script is one module.

Modules can load each other and use special directives `export` and `import` to interchange functionality, call functions of one module from another one:

- `export` keyword labels variables and functions that should be accessible from outside the current module.
- `import` allows the import of functionality from other modules.

For instance, if we have a file `sayHi.js` exporting a function:

```
1  // 🗂 sayHi.js
2  export function sayHi(user) {
3    alert(`Hello, ${user}!`);
4  }
```

…Then another file may import and use it:

```
1  // 🗂 main.js
2  import {sayHi} from './sayHi.js';
```

```
3
4  alert(sayHi); // function...
5  sayHi('John'); // Hello, John!
```

The `import` directive loads the module by path `./sayHi.js` relative to the current file, and assigns exported function `sayHi` to the corresponding variable.

Let's run the example in-browser.

As modules support special keywords and features, we must tell the browser that a script should be treated as a module, by using the attribute `<script type="module">` .

Like this:

Result say.js index.html

```
1  <!doctype html>
2  <script type="module">
3    import {sayHi} from './say.js';
4
5    document.body.innerHTML = sayHi('John');
6  </script>
```

The browser automatically fetches and evaluates the imported module (and its imports if needed), and then runs the script.

Core module features

What's different in modules, compared to "regular" scripts?

There are core features, valid both for browser and server-side JavaScript.

Always "use strict"

Modules always `use strict` , by default. E.g. assigning to an undeclared variable will give an error.

```
1  <script type="module">
2    a = 5; // error
3  </script>
```

Module-level scope

Each module has its own top-level scope. In other words, top-level variables and functions from a module are not seen in other scripts.

In the example below, two scripts are imported, and `hello.js` tries to use `user` variable declared in `user.js` , and fails:

Result hello.js user.js index.html

```
<!doctype html>
<script type="module" src="user.js"></script>
<script type="module" src="hello.js"></script>
```

Modules are expected to `export` what they want to be accessible from outside and `import` what they need.

So we should import `user.js` into `hello.js` and get the required functionality from it instead of relying on global variables.

This is the correct variant:

Result hello.js user.js index.html

```
import {user} from './user.js';

document.body.innerHTML = user; // John
```

In the browser, independent top-level scope also exists for each `<script type="module">`:

```
<script type="module">
  // The variable is only visible in this module script
  let user = "John";
</script>

<script type="module">
  alert(user); // Error: user is not defined
</script>
```

If we really need to make a window-level global variable, we can explicitly assign it to `window` and access as `window.user`. But that's an exception requiring a good reason.

A module code is evaluated only the first time when imported

If the same module is imported into multiple other places, its code is executed only the first time, then exports are given to all importers.

That has important consequences. Let's look at them using examples:

First, if executing a module code brings side-effects, like showing a message, then importing it multiple times will trigger it only once – the first time:

```
// 📁 alert.js
alert("Module is evaluated!");
```

```
// Import the same module from different files

// 📁 1.js
import `./alert.js`; // Module is evaluated!

// 📁 2.js
import `./alert.js`; // (shows nothing)
```

In practice, top-level module code is mostly used for initialization, creation of internal data structures, and if we want something to be reusable – export it.

Now, a more advanced example.

Let's say, a module exports an object:

```
// 📁 admin.js
export let admin = {
  name: "John"
};
```

If this module is imported from multiple files, the module is only evaluated the first time, `admin` object is created, and then passed to all further importers.

All importers get exactly the one and only `admin` object:

```
// 📁 1.js
import {admin} from './admin.js';
admin.name = "Pete";

// 📁 2.js
import {admin} from './admin.js';
alert(admin.name); // Pete

// Both 1.js and 2.js imported the same object
// Changes made in 1.js are visible in 2.js
```

So, let's reiterate – the module is executed only once. Exports are generated, and then they are shared between importers, so if something changes the `admin` object, other modules will see that.

Such behavior allows us to *configure* modules on first import. We can setup its properties once, and then in further imports it's ready.

For instance, the `admin.js` module may provide certain functionality, but expect the credentials to come into the `admin` object from outside:

```
// 📁 admin.js
export let admin = { };

```

```
export function sayHi() {
  alert(`Ready to serve, ${admin.name}!`);
}
```

In `init.js`, the first script of our app, we set `admin.name`. Then everyone will see it, including calls made from inside `admin.js` itself:

```
// 📁 init.js
import {admin} from './admin.js';
admin.name = "Pete";
```

Another module can also see `admin.name`:

```
// 📁 other.js
import {admin, sayHi} from './admin.js';

alert(admin.name); // Pete

sayHi(); // Ready to serve, Pete!
```

import.meta

The object `import.meta` contains the information about the current module.

Its content depends on the environment. In the browser, it contains the url of the script, or a current webpage url if inside HTML:

```
<script type="module">
  alert(import.meta.url); // script url (url of the html page for an inline s
</script>
```

In a module, “this” is undefined

That’s kind of a minor feature, but for completeness we should mention it.

In a module, top-level `this` is undefined.

Compare it to non-module scripts, where `this` is a global object:

```
<script>
  alert(this); // window
</script>

<script type="module">
  alert(this); // undefined
</script>
```

Browser-specific features

There are also several browser-specific differences of scripts with `type="module"` compared to regular ones.

You may want skip this section for now if you're reading for the first time, or if you don't use JavaScript in a browser.

Module scripts are deferred

Module scripts are *always* deferred, same effect as `defer` attribute (described in the chapter Scripts: async, defer), for both external and inline scripts.

In other words:

- downloading external module scripts `<script type="module" src="...">` doesn't block HTML processing, they load in parallel with other resources.
- module scripts wait until the HTML document is fully ready (even if they are tiny and load faster than HTML), and then run.
- relative order of scripts is maintained: scripts that go first in the document, execute first.

As a side-effect, module scripts always "see" the fully loaded HTML-page, including HTML elements below them.

For instance:

```
<script type="module">
  alert(typeof button); // object: the script can 'see' the button below
  // as modules are deferred, the script runs after the whole page is loaded
</script>

Compare to regular script below:

<script>
  alert(typeof button); // Error: button is undefined, the script can't see e
  // regular scripts run immediately, before the rest of the page is processe
</script>

<button id="button">Button</button>
```

Please note: the second script actually runs before the first! So we'll see `undefined` first, and then `object`.

That's because modules are deferred, so we wait for the document to be processed. The regular script runs immediately, so we see its output first.

When using modules, we should be aware that the HTML page shows up as it loads, and JavaScript modules run after that, so the user may see the page before the JavaScript application is ready. Some functionality may not work yet. We should put "loading indicators", or otherwise ensure that the visitor won't be confused by that.

Async works on inline scripts

For non-module scripts, the `async` attribute only works on external scripts. Async scripts run immediately when ready, independently of other scripts or the HTML document.

For module scripts, it works on inline scripts as well.

For example, the inline script below has `async`, so it doesn't wait for anything.

It performs the import (fetches `./analytics.js`) and runs when ready, even if the HTML document is not finished yet, or if other scripts are still pending.

That's good for functionality that doesn't depend on anything, like counters, ads, document-level event listeners.

```
<!-- all dependencies are fetched (analytics.js), and the script runs -->
<!-- doesn't wait for the document or other <script> tags -->
<script async type="module">
  import {counter} from './analytics.js';

  counter.count();
</script>
```

External scripts

External scripts that have `type="module"` are different in two aspects:

1. External scripts with the same `src` run only once:

```
<!-- the script my.js is fetched and executed only once -->
<script type="module" src="my.js"></script>
<script type="module" src="my.js"></script>
```

2. External scripts that are fetched from another origin (e.g. another site) require CORS headers, as described in the chapter Fetch: Cross-Origin Requests. In other words, if a module script is fetched from another origin, the remote server must supply a header `Access-Control-Allow-Origin` allowing the fetch.

```
<!-- another-site.com must supply Access-Control-Allow-Origin -->
<!-- otherwise, the script won't execute -->
<script type="module" src="http://another-site.com/their.js"></script>
```

That ensures better security by default.

No "bare" modules allowed

In the browser, `import` must get either a relative or absolute URL. Modules without any path are called "bare" modules. Such modules are not allowed in `import`.

For instance, this `import` is invalid:

```
import {sayHi} from 'sayHi'; // Error, "bare" module
// the module must have a path, e.g. './sayHi.js' or wherever the module is
```

Certain environments, like Node.js or bundle tools allow bare modules, without any path, as they have their own ways for finding modules and hooks to fine-tune them. But browsers do not support bare modules yet.

Compatibility, “nomodule”

Old browsers do not understand `type="module"`. Scripts of an unknown type are just ignored. For them, it’s possible to provide a fallback using the `nomodule` attribute:

```
<script type="module">
  alert("Runs in modern browsers");
</script>

<script nomodule>
  alert("Modern browsers know both type=module and nomodule, so skip this")
  alert("Old browsers ignore script with unknown type=module, but execute thi
</script>
```

Build tools

In real-life, browser modules are rarely used in their “raw” form. Usually, we bundle them together with a special tool such as Webpack and deploy to the production server.

One of the benefits of using bundlers – they give more control over how modules are resolved, allowing bare modules and much more, like CSS/HTML modules.

Build tools do the following:

1. Take a “main” module, the one intended to be put in `<script type="module">` in HTML.
2. Analyze its dependencies: imports and then imports of imports etc.
3. Build a single file with all modules (or multiple files, that’s tunable), replacing native `import` calls with bundler functions, so that it works. “Special” module types like HTML/CSS modules are also supported.
4. In the process, other transformations and optimizations may be applied:
   - Unreachable code removed.
   - Unused exports removed (“tree-shaking”).
   - Development-specific statements like `console` and `debugger` removed.
   - Modern, bleeding-edge JavaScript syntax may be transformed to older one with similar functionality using Babel.
   - The resulting file is minified (spaces removed, variables replaced with shorter names, etc).

If we use bundle tools, then as scripts are bundled together into a single file (or few files), `import/export` statements inside those scripts are replaced by special bundler functions. So the resulting “bundled” script does not contain any `import/export`, it doesn’t require `type="module"`, and we can put it into a regular script:

```
<!-- Assuming we got bundle.js from a tool like Webpack -->
<script src="bundle.js"></script>
```

That said, native modules are also usable. So we won’t be using Webpack here: you can configure it later.

Summary

To summarize, the core concepts are:

1. A module is a file. To make `import/export` work, browsers need `<script type="module">`. Modules have several differences:
   - Deferred by default.
   - Async works on inline scripts.
   - To load external scripts from another origin (domain/protocol/port), CORS headers are needed.
   - Duplicate external scripts are ignored.
2. Modules have their own, local top-level scope and interchange functionality via `import/export`.
3. Modules always `use strict`.
4. Module code is executed only once. Exports are created once and shared between importers.

When we use modules, each module implements the functionality and exports it. Then we use `import` to directly import it where it's needed. The browser loads and evaluates the scripts automatically.

In production, people often use bundlers such as Webpack to bundle modules together for performance and other reasons.

In the next chapter we'll see more examples of modules, and how things can be exported/imported.

Previous lesson Next lesson

Share

Tutorial map

Comments

- If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.
- If you can't understand something in the article – please elaborate.
- To insert a few words of code, use the `<code>` tag, for several lines – use `<pre>`, for more than 10 lines – use a sandbox (plnkr, JSBin, codepen…)