Preloading modules

By Sérgio Gomes
Web DevRel @ Google

Browsers are finally starting to natively support JavaScript modules, both with static and dynamic import support. This means it's now possible to write module-based JavaScript that runs natively in the browser, without transpilers or bundlers.

Module-based development offers some real advantages in terms of cacheability, helping you reduce the number of bytes you need to ship to your users. The finer granularity of the code also helps with the loading story, by letting you prioritize the critical code in your application.

However, module dependencies introduce a loading problem, in that the browser needs to wait for a module to load before it finds out what its dependencies are. One way around this is by preloading the dependencies, so that the browser knows about all the files ahead of time and can keep the connection busy.

Until now, there wasn't really a good way of declaratively preloading modules. Chrome 64 ships with `<link rel="modulepreload">` behind the "Experimental Web Platform Features" flag. `<link rel="modulepreload">` is a module-specific version of `<link rel="preload">` that solves a number of the latter's problems.

Warning: It's still very much early days for modules in the browser, so while we encourage experimentation, we advise caution when using this technology in production for now!

Wait, what's `<link rel="preload">`?

Chrome added support for `<link rel="preload">` back in version 50, as a way of declaratively requesting resources ahead of time, before the browser needs them.

```
<head>
  <link rel="preload" as="style" href="critical-styles.css">
  <link rel="preload" as="font" crossorigin type="font/woff2" href="myfont.woff2"
</head>
```

This works particularly well with resources such as fonts, which are often hidden inside CSS files, sometimes several levels deep. In that situation, the browser would have to wait for multiple roundtrips before finding out that it needs to fetch a large font file, when it could have used that time to start the download and take advantage of the full connection bandwidth.

`<link rel="preload">` and its HTTP header equivalent provide a simple, declarative way of letting the browser know straight away about critical files that will be needed as part of the current navigation. When the browser sees the preload, it starts a high priority download for the resource, so that by the time it's actually needed it's either already fetched or partly there.

OK, so why doesn't `<link rel="preload">` work for modules?

This is where things get tricky. There are several credentials modes for resources, and in order to get a cache hit they must match, otherwise you end up fetching the resource twice. Needless to say, double-fetching is bad, because it wastes the user's bandwidth and makes them wait longer, for no good reason.

For `<script>` and `<link>` tags, you can set the credentials mode with the `crossorigin` attribute. However, it turns out that a `<script type="module">` with no `crossorigin` attribute indicates a credentials mode of `omit`, which doesn't exist for `<link rel="preload">`. This means that you would have to change the `crossorigin` attribute in both your `<script>` and `<link>` to one of the other values, and you might not have an easy way of doing so if what you're trying to preload is a dependency of other modules.

Furthermore, fetching the file is only the first step in actually running the code. First, the browser has to parse and compile it. Ideally, this should happen ahead of time as well, so that when the module is needed, the code is ready to run. However, V8 (Chrome's JavaScript engine) parses and compiles modules differently from other JavaScript. `<link rel="preload">` doesn't provide any way of indicating that the file being loaded is a module, so all the browser can do is load the file and put it in the cache. Once the script is loaded using a `<script type="module">` tag (or it's loaded by another module), the browser parses and compiles the code as a JavaScript module.

So is `<link rel="modulepreload">` just `<link rel="preload">` for modules?

In a nutshell, yes. By having a specific `link` type for preloading modules, we can write simple HTML without worrying about what credentials mode we're using. The defaults just

work.

```
<head>
  <link rel="modulepreload" href="super-critical-stuff.mjs">
</head>
[...]
<script type="module" src="super-critical-stuff.mjs">
```

And since Chrome now knows that what you're preloading is a module, it can be smart and parse and compile the module as soon as it's done fetching, instead of waiting until it tries to run.

But what about modules' dependencies?

Funny you should ask! There is indeed something we haven't talked about: recursion.

The `<link rel="modulepreload">` spec actually allows for optionally loading not just the requested module, but all of its dependency tree as well. Browsers don't have to do this, but they can.

So what would be the best cross-browser solution for preloading a module and its dependency tree, since you'll need the full dependency tree to run the app?

Browsers that choose to preload dependencies recursively should have robust deduplication of modules, so in general the best practice would be to declare the module and the flat list of its dependencies, and trust the browser not to fetch the same module twice.

```
<head>
  <!-- dog.js imports dog-head.js, which in turn imports
       dog-head-mouth.js, which imports dog-head-mouth-tongue.js. -->
  <link rel="modulepreload" href="dog-head-mouth-tongue.mjs">
  <link rel="modulepreload" href="dog-head-mouth.mjs">
  <link rel="modulepreload" href="dog-head.mjs">
  <link rel="modulepreload" href="dog.mjs">
</head>
```

Does preloading modules help performance?

Preloading can help in maximizing bandwidth usage, by telling the browser about what it needs to fetch so that it's not stuck with nothing to do during those long roundtrips. If you're

experimenting with modules and running into performance issues due to deep dependency trees, creating a flat list of preloads can definitely help!

That said, module performance is still being worked on, so make sure you take a close look at what's happening in your application with Developer Tools, and consider bundling your application into several chunks in the meantime. There's plenty of ongoing module work happening in Chrome, though, so we're getting closer to giving bundlers their well-earned rest!

Last updated July 2, 2018.