October 25, 2021

Scripts: async, defer

In modern websites, scripts are often "heavier" than HTML: their download size is larger, and processing time is also longer.

When the browser loads HTML and comes across a `<script>...</script>` tag, it can't continue building the DOM. It must execute the script right now. The same happens for external scripts `<script src="..."></script>` : the browser must wait for the script to download, execute the downloaded script, and only then can it process the rest of the page.

That leads to two important issues:

1. Scripts can't see DOM elements below them, so they can't add handlers etc.
2. If there's a bulky script at the top of the page, it "blocks the page". Users can't see the page content till it downloads and runs:

```
<p>...content before script...</p>

<script src="https://javascript.info/article/script-async-defer/long.js?speed=1"></script>

<!-- This isn't visible until the script loads -->
<p>...content after script...</p>
```

There are some workarounds to that. For instance, we can put a script at the bottom of the page. Then it can see elements above it, and it doesn't block the page content from showing:

```
<body>
  ...all content is above the script...

  <script src="https://javascript.info/article/script-async-defer/long.js?speed=1"></script
</body>
```

But this solution is far from perfect. For example, the browser notices the script (and can start downloading it) only after it downloaded the full HTML document. For long HTML documents, that may be a noticeable delay.

Such things are invisible for people using very fast connections, but many people in the world still have slow internet speeds and use a far-from-perfect mobile internet connection.

Luckily, there are two `<script>` attributes that solve the problem for us: `defer` and `async` .

defer

The `defer` attribute tells the browser not to wait for the script. Instead, the browser will continue to process the HTML, build DOM. The script loads "in the background", and then runs when the DOM is fully built.

Here's the same example as above, but with `defer` :

```
<p>...content before script...</p>

<script defer src="https://javascript.info/article/script-async-defer/long.js?speed=1"></sc

<!-- visible immediately -->
<p>...content after script...</p>
```

In other words:

- Scripts with `defer` never block the page.
- Scripts with `defer` always execute when the DOM is ready (but before `DOMContentLoaded` event).

The following example demonstrates the second part:

```
<p>...content before scripts...</p>

<script>
```

```
  document.addEventListener('DOMContentLoaded', () => alert("DOM ready after defer!"));
</script>

<script defer src="https://javascript.info/article/script-async-defer/long.js?speed=1"></sc

<p>...content after scripts...</p>
```

1. The page content shows up immediately.
2. `DOMContentLoaded` event handler waits for the deferred script. It only triggers when the script is downloaded and executed.

Deferred scripts keep their relative order, just like regular scripts.

Let's say, we have two deferred scripts: the `long.js` and then `small.js`:

```
<script defer src="https://javascript.info/article/script-async-defer/long.js"></script>
<script defer src="https://javascript.info/article/script-async-defer/small.js"></script>
```

Browsers scan the page for scripts and download them in parallel, to improve performance. So in the example above both scripts download in parallel. The `small.js` probably finishes first.

...But the `defer` attribute, besides telling the browser "not to block", ensures that the relative order is kept. So even though `small.js` loads first, it still waits and runs after `long.js` executes.

That may be important for cases when we need to load a JavaScript library and then a script that depends on it.

The `defer` attribute is only for external scripts

The `defer` attribute is ignored if the `<script>` tag has no `src`.

async

The `async` attribute is somewhat like `defer`. It also makes the script non-blocking. But it has important differences in the behavior.

The `async` attribute means that a script is completely independent:

- The browser doesn't block on `async` scripts (like `defer`).
- Other scripts don't wait for `async` scripts, and `async` scripts don't wait for them.
- `DOMContentLoaded` and async scripts don't wait for each other:
  - `DOMContentLoaded` may happen both before an async script (if an async script finishes loading after the page is complete)
  - ...or after an async script (if an async script is short or was in HTTP-cache)

In other words, `async` scripts load in the background and run when ready. The DOM and other scripts don't wait for them, and they don't wait for anything. A fully independent script that runs when loaded. As simple, as it can get, right?

Here's an example similar to what we've seen with `defer`: two scripts `long.js` and `small.js`, but now with `async` instead of `defer`.

They don't wait for each other. Whatever loads first (probably `small.js`) – runs first:

```
<p>...content before scripts...</p>

<script>
  document.addEventListener('DOMContentLoaded', () => alert("DOM ready!"));
</script>

<script async src="https://javascript.info/article/script-async-defer/long.js"></script>
<script async src="https://javascript.info/article/script-async-defer/small.js"></script>

<p>...content after scripts...</p>
```

- The page content shows up immediately: `async` doesn't block it.
- `DOMContentLoaded` may happen both before and after `async`, no guarantees here.
- A smaller script `small.js` goes second, but probably loads before `long.js`, so `small.js` runs first. Although, it might be that `long.js` loads first, if cached, then it runs first. In other words, async scripts run in the "load-first" order.

Async scripts are great when we integrate an independent third-party script into the page: counters, ads and so on, as they don't depend on our scripts, and our scripts shouldn't wait for them:

```
<!-- Google Analytics is usually added like this -->
<script async src="https://google-analytics.com/analytics.js"></script>
```

The `async` attribute is only for external scripts

Just like `defer`, the `async` attribute is ignored if the `<script>` tag has no `src`.

Dynamic scripts

There's one more important way of adding a script to the page.

We can create a script and append it to the document dynamically using JavaScript:

```
let script = document.createElement('script');
script.src = "/article/script-async-defer/long.js";
document.body.append(script); // (*)
```

The script starts loading as soon as it's appended to the document `(*)`.

Dynamic scripts behave as "async" by default.

That is:

- They don't wait for anything, nothing waits for them.
- The script that loads first – runs first ("load-first" order).

This can be changed if we explicitly set `script.async=false`. Then scripts will be executed in the document order, just like `defer`.

In this example, `loadScript(src)` function adds a script and also sets `async` to `false`.

So `long.js` always runs first (as it's added first):

```
function loadScript(src) {
  let script = document.createElement('script');
  script.src = src;
  script.async = false;
  document.body.append(script);
}

// long.js runs first because of async=false
loadScript("/article/script-async-defer/long.js");
loadScript("/article/script-async-defer/small.js");
```

Without `script.async=false`, scripts would execute in default, load-first order (the `small.js` probably first).

Again, as with the `defer`, the order matters if we'd like to load a library and then another script that depends on it.

Summary

Both `async` and `defer` have one common thing: downloading of such scripts doesn't block page rendering. So the user can read page content and get acquainted with the page immediately.

But there are also essential differences between them:

	Order	`DOMContentLoaded`
`async`	*Load-first order*. Their document order doesn't matter – which loads first runs first	Irrelevant. May load and execute while the document has not yet been fully downloaded. That happens if scripts are small or cached, and the document is long enough.
`defer`	*Document order* (as they go in the document).	Execute after the document is loaded and parsed (they wait if needed), right before `DOMContentLoaded`.

In practice, `defer` is used for scripts that need the whole DOM and/or their relative execution order is important.

And `async` is used for independent scripts, like counters or ads. And their relative execution order does not matter.

Page without scripts should be usable

Please note: if you're using `defer` or `async`, then user will see the page *before* the script loads.

In such case, some graphical components are probably not initialized yet.

Don't forget to put "loading" indication and disable buttons that aren't functional yet. Let the user clearly see what he can do on the page, and what's still getting ready.

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 few words of code, use the `<code>` tag, for several lines – wrap them in `<pre>` tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen...)