EN

JS

EPUB/PDF

→ Browser: Document, Events, Interfaces → Document and resource loading

21st October 2019

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 until the script downloads, execute it, and only after 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

<!-- 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?spe
</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 that it should go on working with the page, and load the script “in background”, then run the script when it loads.

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

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

- 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 that:

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

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

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

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

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

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

So, if we have a long script first, and then a smaller one, then the latter one waits.

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

The small script downloads first, runs second

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 makes it first.

But the specification requires scripts to execute in the document order, so it waits for `long.js` to execute.

The `defer` attribute is only for external scripts

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

async

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

- The page doesn't wait for async scripts, the contents are processed and displayed.
- `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)
- Other scripts don't wait for `async` scripts, and `async` scripts don't wait for them.

So, if we have several `async` scripts, they may execute in any order. Whatever loads first – 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 async src="https://javascript.info/article/script-async-defer/small.j

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

1. The page content shows up immediately: `async` doesn't block it.
2. `DOMContentLoaded` may happen both before and after `async` , no guarantees here.
3. Async scripts don't wait for each other. A smaller script `small.js` goes second, but probably loads before `long.js` , so runs first. That's called a "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>
```

Dynamic scripts

We can also add a script 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).

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

script.async = false;

document.body.append(script);
```

For example, here we add two scripts. Without `script.async=false` they would execute in load-first order (the `small.js` probably first). But with that flag the order is "as in the document":

```
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");
```

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	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`.

⚠ Page without scripts should be usable

Please note that if you're using `defer`, then the page is visible *before* the script loads.

So the user may read the page, but some graphical components are probably not ready yet.

There should be "loading" indications in the proper places, and disabled buttons should show as such, so the user can clearly see what's ready and what's not.

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.

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…)