30th December 2019

Page: DOMContentLoaded, load, beforeunload, unload

The lifecycle of an HTML page has three important events:

- `DOMContentLoaded` – the browser fully loaded HTML, and the DOM tree is built, but external resources like pictures `<img>` and stylesheets may be not yet loaded.
- `load` – not only HTML is loaded, but also all the external resources: images, styles etc.
- `beforeunload/unload` – the user is leaving the page.

Each event may be useful:

- `DOMContentLoaded` event – DOM is ready, so the handler can lookup DOM nodes, initialize the interface.
- `load` event – external resources are loaded, so styles are applied, image sizes are known etc.
- `beforeunload` event – the user is leaving: we can check if the user saved the changes and ask them whether they really want to leave.
- `unload` – the user almost left, but we still can initiate some operations, such as sending out statistics.

Let’s explore the details of these events.

DOMContentLoaded

The `DOMContentLoaded` event happens on the `document` object.

We must use `addEventListener` to catch it:

```
document.addEventListener("DOMContentLoaded", ready);
// not "document.onDOMContentLoaded = ..."
```

For instance:

```
<script>
  function ready() {
    alert('DOM is ready');

    // image is not yet loaded (unless was cached), so the size is 0x0
    alert(`Image size: ${img.offsetWidth}x${img.offsetHeight}`);
  }

  document.addEventListener("DOMContentLoaded", ready);
</script>
```

```html

<img id="img" src="https://en.js.cx/clipart/train.gif?speed=1&cache=0">
```

In the example the `DOMContentLoaded` handler runs when the document is loaded, so it can see all the elements, including `<img>` below.

But it doesn't wait for the image to load. So `alert` shows zero sizes.

At first sight, the `DOMContentLoaded` event is very simple. The DOM tree is ready – here's the event. There are few peculiarities though.

DOMContentLoaded and scripts

When the browser processes an HTML-document and comes across a `<script>` tag, it needs to execute before continuing building the DOM. That's a precaution, as scripts may want to modify DOM, and even `document.write` into it, so `DOMContentLoaded` has to wait.

So DOMContentLoaded definitely happens after such scripts:

```html
<script>
  document.addEventListener("DOMContentLoaded", () => {
    alert("DOM ready!");
  });
</script>

<script src="https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.3.0/lodash.js

<script>
  alert("Library loaded, inline script executed");
</script>
```

In the example above, we first see "Library loaded…", and then "DOM ready!" (all scripts are executed).

Scripts that don't block DOMContentLoaded

There are two exceptions from this rule:

1. Scripts with the `async` attribute, that we'll cover a bit later, don't block `DOMContentLoaded` .
2. Scripts that are generated dynamically with `document.createElement('script')` and then added to the webpage also don't block this event.

DOMContentLoaded and styles

External style sheets don't affect DOM, so `DOMContentLoaded` does not wait for them.

But there's a pitfall. If we have a script after the style, then that script must wait until the stylesheet loads:

```html
<link type="text/css" rel="stylesheet" href="style.css">
<script>
  // the script doesn't not execute until the stylesheet is loaded
  alert(getComputedStyle(document.body).marginTop);
</script>
```

The reason for this is that the script may want to get coordinates and other style-dependent properties of elements, like in the example above. Naturally, it has to wait for styles to load.

As `DOMContentLoaded` waits for scripts, it now waits for styles before them as well.

Built-in browser autofill

Firefox, Chrome and Opera autofill forms on `DOMContentLoaded` .

For instance, if the page has a form with login and password, and the browser remembered the values, then on `DOMContentLoaded` it may try to autofill them (if approved by the user).

So if `DOMContentLoaded` is postponed by long-loading scripts, then autofill also awaits. You probably saw that on some sites (if you use browser autofill) – the login/password fields don't get autofilled immediately, but there's a delay till the page fully loads. That's actually the delay until the `DOMContentLoaded` event.

window.onload

The `load` event on the `window` object triggers when the whole page is loaded including styles, images and other resources.

The example below correctly shows image sizes, because `window.onload` waits for all images:

```
<script>
  window.onload = function() {
    alert('Page loaded');

    // image is loaded at this time
    alert(`Image size: ${img.offsetWidth}x${img.offsetHeight}`);
  };
</script>

<img id="img" src="https://en.js.cx/clipart/train.gif?speed=1&cache=0">
```

window.onunload

When a visitor leaves the page, the `unload` event triggers on `window` . We can do something there that doesn't involve a delay, like closing related popup windows.

The notable exception is sending analytics.

Let's say we gather data about how the page is used: mouse clicks, scrolls, viewed page areas, and so on.

Naturally, `unload` event is when the user leaves us, and we'd like to save the data on our server.

There exists a special `navigator.sendBeacon(url, data)` method for such needs, described in the specification https://w3c.github.io/beacon/.

It sends the data in background. The transition to another page is not delayed: the browser leaves the page, but still performs `sendBeacon` .

Here's how to use it:

```
let analyticsData = { /* object with gathered data */ };

window.addEventListener("unload", function() {
  navigator.sendBeacon("/analytics", JSON.stringify(analyticsData));
};
```

- The request is sent as POST.
- We can send not only a string, but also forms and other formats, as described in the chapter Fetch, but usually it's a stringified object.
- The data is limited by 64kb.

When the `sendBeacon` request is finished, the browser probably has already left the document, so there's no way to get server response (which is usually empty for analytics).

There's also a `keepalive` flag for doing such "after-page-left" requests in fetch method for generic network requests. You can find more information in the chapter Fetch API.

If we want to cancel the transition to another page, we can't do it here. But we can use another event – `onbeforeunload`.

window.onbeforeunload

If a visitor initiated navigation away from the page or tries to close the window, the `beforeunload` handler asks for additional confirmation.

If we cancel the event, the browser may ask the visitor if they are sure.

You can try it by running this code and then reloading the page:

```
window.onbeforeunload = function() {
  return false;
};
```

For historical reasons, returning a non-empty string also counts as canceling the event. Some time ago browsers used to show it as a message, but as the modern specification says, they shouldn't.

Here's an example:

```
window.onbeforeunload = function() {
  return "There are unsaved changes. Leave now?";
};
```

The behavior was changed, because some webmasters abused this event handler by showing misleading and annoying messages. So right now old browsers still may show it as a message, but aside of that – there's no way to customize the message shown to the user.

readyState

What happens if we set the `DOMContentLoaded` handler after the document is loaded?

Naturally, it never runs.

There are cases when we are not sure whether the document is ready or not. We'd like our function to execute when the DOM is loaded, be it now or later.

The `document.readyState` property tells us about the current loading state.

There are 3 possible values:

- `"loading"` – the document is loading.
- `"interactive"` – the document was fully read.
- `"complete"` – the document was fully read and all resources (like images) are loaded too.

So we can check `document.readyState` and setup a handler or execute the code immediately if it's ready.

Like this:

```
function work() { /*...*/ }

if (document.readyState == 'loading') {
  // loading yet, wait for the event
  document.addEventListener('DOMContentLoaded', work);
} else {
  // DOM is ready!
  work();
}
```

There's also the `readystatechange` event that triggers when the state changes, so we can print all these states like this:

```
// current state
console.log(document.readyState);

// print state changes
document.addEventListener('readystatechange', () => console.log(document.read
```

The `readystatechange` event is an alternative mechanics of tracking the document loading state, it appeared long ago. Nowadays, it is rarely used.

Let's see the full events flow for the completeness.

Here's a document with `<iframe>`, `<img>` and handlers that log events:

```
<script>
  log('initial readyState:' + document.readyState);

  document.addEventListener('readystatechange', () => log('readyState:' + doc
  document.addEventListener('DOMContentLoaded', () => log('DOMContentLoaded')

  window.onload = () => log('window onload');
</script>
```

```

<iframe src="iframe.html" onload="log('iframe onload')"></iframe>

<img src="http://en.js.cx/clipart/train.gif" id="img">
<script>
  img.onload = () => log('img onload');
</script>
```

The working example is in the sandbox.

The typical output:

1. [1] initial readyState:loading
2. [2] readyState:interactive
3. [2] DOMContentLoaded
4. [3] iframe onload
5. [4] img onload
6. [4] readyState:complete
7. [4] window onload

The numbers in square brackets denote the approximate time of when it happens. Events labeled with the same digit happen approximately at the same time (± a few ms).

- `document.readyState` becomes `interactive` right before `DOMContentLoaded` . These two things actually mean the same.
- `document.readyState` becomes `complete` when all resources (`iframe` and `img`) are loaded. Here we can see that it happens in about the same time as `img.onload` (`img` is the last resource) and `window.onload` . Switching to `complete` state means the same as `window.onload` . The difference is that `window.onload` always works after all other `load` handlers.

Summary

Page load events:

- The `DOMContentLoaded` event triggers on `document` when the DOM is ready. We can apply JavaScript to elements at this stage.
  - Script such as `<script>...</script>` or `<script src="..."></script>` block DOMContentLoaded, the browser waits for them to execute.
  - Images and other resources may also still continue loading.
- The `load` event on `window` triggers when the page and all resources are loaded. We rarely use it, because there's usually no need to wait for so long.
- The `beforeunload` event on `window` triggers when the user wants to leave the page. If we cancel the event, browser asks whether the user really wants to leave (e.g we have unsaved changes).
- The `unload` event on `window` triggers when the user is finally leaving, in the handler we can only do simple things that do not involve delays or asking a user. Because of that limitation, it's rarely used. We can send out a network request with `navigator.sendBeacon` .
- `document.readyState` is the current state of the document, changes can be tracked in the `readystatechange` event:
  - `loading` – the document is loading.

- `interactive` – the document is parsed, happens at about the same time as `DOMContentLoaded`, but before it.
- `complete` – the document and resources are loaded, happens at about the same time as `window.onload`, but before it.

Previous lesson Next lesson

Share Tutorial map

Comments