2nd December 2019

Mutation observer

`MutationObserver` is a built-in object that observes a DOM element and fires a callback in case of changes.

We'll first take a look at the syntax, and then explore a real-world use case, to see where such thing may be useful.

Syntax

`MutationObserver` is easy to use.

First, we create an observer with a callback-function:

```
let observer = new MutationObserver(callback);
```

And then attach it to a DOM node:

```
observer.observe(node, config);
```

`config` is an object with boolean options "what kind of changes to react on":

- `childList` – changes in the direct children of `node`,
- `subtree` – in all descendants of `node`,
- `attributes` – attributes of `node`,
- `attributeFilter` – an array of attribute names, to observe only selected ones.
- `characterData` – whether to observe `node.data` (text content),

Few other options:

- `attributeOldValue` – if `true`, pass both the old and the new value of attribute to callback (see below), otherwise only the new one (needs `attributes` option),
- `characterDataOldValue` – if `true`, pass both the old and the new value of `node.data` to callback (see below), otherwise only the new one (needs `characterData` option).

Then after any changes, the `callback` is executed: changes are passed in the first argument as a list of MutationRecord objects, and the observer itself as the second argument.

MutationRecord objects have properties:

- `type` – mutation type, one of

  - `"attributes"` : attribute modified
  - `"characterData"` : data modified, used for text nodes,
  - `"childList"` : child elements added/removed,
- `target` – where the change occurred: an element for `"attributes"` , or text node for `"characterData"` , or an element for a `"childList"` mutation,
- `addedNodes/removedNodes` – nodes that were added/removed,
- `previousSibling/nextSibling` – the previous and next sibling to added/removed nodes,
- `attributeName/attributeNamespace` – the name/namespace (for XML) of the changed attribute,
- `oldValue` – the previous value, only for attribute or text changes, if the corresponding option is set `attributeOldValue` / `characterDataOldValue` .

For example, here's a `<div>` with a `contentEditable` attribute. That attribute allows us to focus on it and edit.

```
<div contentEditable id="elem">Click and <b>edit</b>, please</div>

<script>
let observer = new MutationObserver(mutationRecords => {
  console.log(mutationRecords); // console.log(the changes)
});

// observe everything except attributes
observer.observe(elem, {
  childList: true, // observe direct children
  subtree: true, // and lower descendants too
  characterDataOldValue: true // pass old data to callback
});
</script>
```

If we run this code in the browser, then focus on the given `<div>` and change the text inside `<b>edit</b>` , `console.log` will show one mutation:

```
mutationRecords = [{
  type: "characterData",
  oldValue: "edit",
  target: <text node>,
  // other properties empty
}];
```

If we make more complex editing operations, e.g. remove the `<b>edit</b>` , the mutation event may contain multiple mutation records:

```
mutationRecords = [{
  type: "childList",
  target: <div#elem>,
  removedNodes: [<b>],
  nextSibling: <text node>,
  previousSibling: <text node>
  // other properties empty
}, {
  type: "characterData"
```

```
  target: <text node>
  // ...mutation details depend on how the browser handles such removal
  // it may coalesce two adjacent text nodes "edit " and ", please" into one
  // or it may leave them separate text nodes
}];
```

So, `MutationObserver` allows to react on any changes within DOM subtree.

Usage for integration

When such thing may be useful?

Imagine the situation when you need to add a third-party script that contains useful functionality, but also does something unwanted, e.g. shows ads `<div class="ads">Unwanted ads</div>` .

Naturally, the third-party script provides no mechanisms to remove it.

Using `MutationObserver` , we can detect when the unwanted element appears in our DOM and remove it.

There are other situations when a third-party script adds something into our document, and we'd like to detect, when it happens, to adapt our page, dynamically resize something etc.

`MutationObserver` allows to implement this.

Usage for architecture

There are also situations when `MutationObserver` is good from architectural standpoint.

Let's say we're making a website about programming. Naturally, articles and other materials may contain source code snippets.

Such snippet in an HTML markup looks like this:

```
...
<pre class="language-javascript"><code>
  // here's the code
  let hello = "world";
</code></pre>
...
```

Also we'll use a JavaScript highlighting library on our site, e.g. Prism.js. A call to `Prism.highlightElem(pre)` examines the contents of such `pre` elements and adds into them special tags and styles for colored syntax highlighting, similar to what you see in examples here, at this page.

When exactly to run that highlighting method? We can do it on `DOMContentLoaded` event, or at the bottom of the page. At that moment we have our DOM ready, can search for elements `pre[class*="language"]` and call `Prism.highlightElem` on them:

```
// highlight all code snippets on the page
document.querySelectorAll('pre[class*="language"]').forEach(Prism.highlightEl
```

Everything's simple so far, right? There are `<pre>` code snippets in HTML, we highlight them.

Now let's go on. Let's say we're going to dynamically fetch materials from a server. We'll study methods for that later in the tutorial. For now it only matters that we fetch an HTML article from a webserver and display it on demand:

```
let article = /* fetch new content from server */
articleElem.innerHTML = article;
```

The new `article` HTML may contain code snippets. We need to call `Prism.highlightElem` on them, otherwise they won't get highlighted.

Where and when to call `Prism.highlightElem` for a dynamically loaded article?

We could append that call to the code that loads an article, like this:

```
let article = /* fetch new content from server */
articleElem.innerHTML = article;

let snippets = articleElem.querySelectorAll('pre[class*="language-"]');
snippets.forEach(Prism.highlightElem);
```

…But imagine, we have many places in the code where we load contents: articles, quizzes, forum posts. Do we need to put the highlighting call everywhere? That's not very convenient, and also easy to forget.

And what if the content is loaded by a third-party module? E.g. we have a forum written by someone else, that loads contents dynamically, and we'd like to add syntax highlighting to it. No one likes to patch third-party scripts.

Luckily, there's another option.

We can use `MutationObserver` to automatically detect when code snippets are inserted in the page and highlight them.

So we'll handle the highlighting functionality in one place, relieving us from the need to integrate it.

Dynamic highlight demo

Here's the working example.

If you run this code, it starts observing the element below and highlighting any code snippets that appear there:

```
let observer = new MutationObserver(mutations => {

  for(let mutation of mutations) {
    // examine new nodes, is there anything to highlight?

    for(let node of mutation.addedNodes) {
      // we track only elements, skip other nodes (e.g. text nodes)
      if (!(node instanceof HTMLElement)) continue;

      // check the inserted element for being a code snippet
      if (node.matches('pre[class*="language-"]')) {
        Prism.highlightElement(node);
```

```
      }

      // or maybe there's a code snippet somewhere in its subtree?
      for(let elem of node.querySelectorAll('pre[class*="language-"]')) {
        Prism.highlightElement(elem);
      }
    }
  }

});

let demoElem = document.getElementById('highlight-demo');

observer.observe(demoElem, {childList: true, subtree: true});
```

Here, below, there's an HTML-element and JavaScript that dynamically fills it using `innerHTML`.

Please run the previous code (above, observes that element), and then the code below. You'll see how `MutationObserver` detects and highlights the snippet.

A demo-element with `id="highlight-demo"`, run the code above to observe it.

The following code populates its `innerHTML`, that causes the `MutationObserver` to react and highlight its contents:

```
let demoElem = document.getElementById('highlight-demo');

// dynamically insert content with code snippets
demoElem.innerHTML = `A code snippet is below:
  <pre class="language-javascript"><code> let hello = "world!"; </code></pre>
  <div>Another one:</div>
  <div>
    <pre class="language-css"><code>.class { margin: 5px; } </code></pre>
  </div>
`;
```

Now we have `MutationObserver` that can track all highlighting in observed elements or the whole `document`. We can add/remove code snippets in HTML without thinking about it.

Additional methods

There's a method to stop observing the node:

- `observer.disconnect()` – stops the observation.

When we stop the observing, it might be possible that some changes were not processed by the observer yet.

- `observer.takeRecords()` – gets a list of unprocessed mutation records, those that happened, but the callback did not handle them.

These methods can be used together, like this:

```
// we'd like to stop tracking changes
observer.disconnect();

// handle unprocessed some mutations
let mutationRecords = observer.takeRecords();
...
```

Garbage collection interaction

Observers use weak references to nodes internally. That is: if a node is removed from DOM, and becomes unreachable, then it becomes garbage collected.

The mere fact that a DOM node is observed doesn't prevent the garbage collection.

Summary

`MutationObserver` can react on changes in DOM: attributes, added/removed elements, text content.

We can use it to track changes introduced by other parts of our code, as well as to integrate with third-party scripts.

`MutationObserver` can track any changes. The config "what to observe" options are used for optimizations, not to spend resources on unneeded callback invocations.

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…)