JavaScript MutationObserver

Summary: in this tutorial, you will learn how to use the JavaScript `MutationObserver` API to watch for changes being made to the DOM tree.

Introduction to the JavaScript MutationObserver API

The `MutationObserver` API allows you to monitor for changes being made to the DOM tree. When the DOM nodes change, you can invoke a callback function to react to the changes.

The basic steps for using the `MutationObserver` API are:

First, define the callback function that will execute when the DOM changes:

```
function callback(mutations) {
    //
}
```

Second, create a `MutationObserver` object and pass the callback to the `MutationObserver()` constructor:

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

Third, call the `observe()` method to start observing the DOM changes.

```
observer.observe(targetNode, observerOptions);
```

The `observe()` method has two parameters. The `target` is the root of the subtree of nodes to monitor for changes. The `observerOptions` parameter contains properties that specify what DOM changes should be reported to the observer's callback.

Finally, stop observing the DOM changes by calling the `disconnect()` method:

```
observer.disconnect();
```

The MutationObserver options

The second argument of the `observe()` method allows you to specify options to describe the `MutationObserver`:

```
let options = {
    childList: true,
    attributes: true,
    characterData: false,
    subtree: false,
    attributeFilter: ['attr1', 'attr2'],
    attributeOldValue: false,
    characterDataOldValue: false
};
```

You don't need to use all the options. However, to make the `MutationObserver` works, at least one of `childList`, `attributes`, or `characterData` needs to be set to `true`, otherwise the `observer()` method will throw an error.

Observing changes to child elements

Assuming that you have the following list:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>MutationObserver Demo: ChildList</title>
</head>
<body>
    <ul id="language">
        <li>HTML</li>
        <li>CSS</li>
```

```
        <li>JavaScript</li>
        <li>TypeScript</li>
    </ul>

    <button id="btnStart">Start Observing</button>
    <button id="btnStop">Stop Observing</button>
    <button id="btnAdd">Add</button>
    <button id="btnRemove">Remove the Last Child</button>

    <script src="app.js"></script>
</body>
</html>
```

The following example illustrates how to use the `childList` property of the mutation `options` object to monitor for the child node changes.

First, select the elements like the `list` and `buttons` using the `querySelector()` method. By default, the `Stop Observing` button is `disabled`.

```
// selecting list
let list = document.querySelector('#language');

// selecting buttons
let btnAdd = document.querySelector('#btnAdd');
let btnRemove = document.querySelector('#btnRemove');
let btnStart = document.querySelector('#btnStart');

let btnStop = document.querySelector('#btnStop');
btnStop.disabled = true;
```

Second, declare a `log()` function that will be used as a callback for the `MutationObserver`:

```
function log(mutations) {
    for (let mutation of mutations) {
        if (mutation.type === 'childList') {
            console.log(mutation);
        }
```

```
    }
}
```

Third, create a new `MutationObserver` object:

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

Fourth, start observing the DOM changes to the child nodes of the list element when the `Start Observing` button is clicked by calling the `observe()` method with the `childList` of the `options` object is set to `true` :

```
btnStart.addEventListener('click', function () {
    observer.observe(list, {
        childList: true
    });

    btnStart.disabled = true;
    btnStop.disabled = false;
});
```

Fifth, add a new list item when the `add` button is clicked:

```
let counter = 1;
btnAdd.addEventListener('click', function () {
    // create a new item element
    let item = document.createElement('li');
    item.textContent = `Item ${counter++}`;

    // append it to the child nodes of list
    list.appendChild(item);
});
```

Sixth, remove the last child of the `list` when the `Remove` button is clicked:

```
btnRemove.addEventListener('click', function () {
    list.lastElementChild ?
```

```
        list.removeChild(list.lastElementChild) :
        console.log('No more child node to remove');
});
```

Finally, stop observing DOM changes when the `Stop Observing` button is clicked by calling the `disconnect()` method of the `MutationObserver` object:

```
btnStop.addEventListener('click', function () {
    observer.disconnect();
    // set button states
    btnStart.disabled = false;
    btnStop.disabled = true;
});
```

Put it all together:

```
(function () {
    // selecting the list
    let list = document.querySelector('#language');

    // selecting the buttons
    let btnAdd = document.querySelector('#btnAdd');
    let btnRemove = document.querySelector('#btnRemove');
    let btnStart = document.querySelector('#btnStart');

    // disable the stop button
    let btnStop = document.querySelector('#btnStop');
    btnStop.disabled = true;

    function log(mutations) {
        for (let mutation of mutations) {
            if (mutation.type === 'childList') {
                console.log(mutation);
            }
        }
    }

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

```
    btnStart.addEventListener('click', function () {
        observer.observe(list, {
            childList: true
        });

        btnStart.disabled = true;
        btnStop.disabled = false;
    });

    btnStop.addEventListener('click', function () {
        observer.disconnect();

        // Set the button state
        btnStart.disabled = false;
        btnStop.disabled = true;
    });

    let counter = 1;
    btnAdd.addEventListener('click', function () {
        // create a new item element
        let item = document.createElement('li');
        item.textContent = `Item ${counter++}`;

        // append it to the child nodes of list
        list.appendChild(item);
    });

    btnRemove.addEventListener('click', function () {
        list.lastElementChild ?
            list.removeChild(list.lastElementChild) :
            console.log('No more child node to remove');
    });

})();
```

Output

- HTML
- CSS
- JavaScript
- TypeScript

Start Observing | Stop Observing | Add | Remove

Notice that we placed all code in an IIFE (Immediately Invoked Function Expression).

Observing for changes to attributes

To observe for changes to attributes, you use the following `attributes` property of the `options` object:

```
let options = {
  attributes: true
}
```

If you want to observe the changes to one or more specific `attributes` while ignoring the others, you can use the `attributeFilter` property:

```
let options = {
  attributes: true,
  attributeFilter: ['class', 'style']
}
```

In this example, the `MutationObserver` will invoke the callback each time the `class` or `style` attribute changes.

Observing for changes to a subtree

To monitor the target node and its subtree of nodes, you set the `subtree` property of the `options` object to `true` :

```
let options = {
    subtree: true
}
```

Observing for changes to character data

To monitor the node for changes to its textual contents, you set the `characterData` property of the `options` object to `true`:

```
let options = {
    characterData: true
}
```

Accessing old values

To access the old values of attributes, you set the `attributeOldValue` property of the `options` object to `true`:

```
let options = {
    attributes: true,
    attributeOldValue: true
}
```

Similarly, you can access the old value of character data by setting the `characterDataOldValue` property of the `options` object to `true`:

```
let options = {
    characterData: true,
    subtree: true,
    characterDataOldValue: true
}
```

A practical example of MutationObserver

In JavaScript applications, the elements on the page are typically dynamically generated. To wait for a dynamic element, you need to use `MutationObserver`.

The following `waitForElement()` function waits for one or more elements specified by a selector using `MutationObserver` .

```
function waitForElement(selector) {
  return new Promise((resolve) => {
    if (document.querySelector(selector)) {
      return resolve(element);
    }
    const observer = new MutationObserver(() => {
      const element = document.querySelector(selector);
      if (element) {
        resolve(element);
        observer.disconnect();
      }
    });
    observer.observe(document.body, {
      childList: true,
      subtree: true,
    });
  });
}
```

How it works.

The `waitForElement()` function returns a promise. The promise will be resolved once the element is available.

First, resolve the element if it is available:

```
if (document.querySelector(selector)) {
    return resolve(element);
}
```

Second, create a new `MutationObserver` object to observe the DOM tree if the element is not available:

```
const observer = new MutationObserver(() => {
  const element = document.querySelector(selector);
  if (element) {
    resolve(element);
    observer.disconnect();
  }
});
```

The observer object will call the resolve() function once the element is available and stop observing the DOM tree.

Third, observe elements of the whole DOM tree:

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

Because the `waitForElement()` returns a `Promise`, you can use the `then()` method like this:

```
waitForElement()('.a-class').then((element) => {
    console.log('Element is ready');
    console.log(element.textContent);
});
```

Or you can use `await` syntax:

```
const element = await waitForElement()('.a-class');
console.log(element.textContent);
```

In this tutorial, you have learned about the JavaScript `MutationObserver` API that monitors the DOM changes and executes a callback every time the change occurs.