MutationObserver: наблюдатель за изменениями

`MutationObserver` – это встроенный объект, наблюдающий за DOM-элементом и запускающий колбэк в случае изменений.

Сначала мы познакомимся с синтаксисом, а затем разберём примеры использования.

Синтаксис

`MutationObserver` очень прост в использовании.

Сначала мы создаём наблюдатель за изменениями с помощью колбэк-функции:

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

Потом прикрепляем его к DOM-узлу:

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

`config` – это объект с булевыми параметрами «на какие изменения реагировать»:

- `childList` – изменения в непосредственных детях `node`,
- `subtree` – во всех потомках `node`,
- `attributes` – в атрибутах `node`,
- `attributeFilter` – массив имён атрибутов, чтобы наблюдать только за выбранными.
- `characterData` – наблюдать ли за `node.data` (текстовое содержимое),

И ещё пара опций:

- `characterDataOldValue` – если `true`, будет передавать и старое и новое значение `node.data` в колбэк (см далее), иначе только новое (также требуется опция `characterData`),
- `attributeOldValue` – если `true`, будет передавать и старое и новое старое значение атрибута в колбэк (см далее), иначе только новое (также требуется опция `attributes`).

Затем, после изменений, выполняется `callback`, в который изменения передаются первым аргументом как список объектов MutationRecord, а сам наблюдатель идёт вторым аргументом.

Объекты MutationRecord имеют следующие свойства:

- `type` – тип изменения, один из:
  - `"attributes"` изменён атрибут,
  - `"characterData"` изменены данные `elem.data`, это для текстовых узлов
  - `"childList"` добавлены/удалены дочерние элементы,
- `target` – где произошло изменение: элемент для `"attributes"`, текстовый узел для `"characterData"` или элемент для `"childList"`,
- `addedNodes/removedNodes` – добавленные/удалённые узлы,
- `previousSibling/nextSibling` – предыдущий или следующий одноуровневый элемент для добавленных/удалённых элементов,

- `attributeName/attributeNamespace` – имя/пространство имён (для XML) изменённого атрибута,
- `oldValue` – предыдущее значение, только для изменений атрибута или текста, если включена соответствующая опция `attributeOldValue` / `characterDataOldValue`.

Для примера возьмём `<div>` с атрибутом `contentEditable`. Этот атрибут позволяет нам сфокусироваться на элементе, например, кликнув, и отредактировать содержимое.

```
<div contentEditable id="elem">Отредактируй <b>меня</b>

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

// наблюдать за всем, кроме атрибутов
observer.observe(elem, {
  childList: true, // наблюдать за непосредственными де
  subtree: true, // и более глубокими потомками
  characterDataOldValue: true // передавать старое знач
});
</script>
```

Теперь, если мы изменим текст внутри `<b>меня</b>`, мы получим единичное изменение:

```
mutationRecords = [{
  type: "characterData",
  oldValue: "меня",
  target: <text node>,
  // другие свойства пусты
}];
```

Если мы выберем или удалим `<b>меня</b>` полностью, мы получим сразу несколько изменений:

```
mutationRecords = [{
  type: "childList",
  target: <div#elem>,
  removedNodes: [<b>],
  nextSibling: <text node>,
  previousSibling: <text node>
  // другие свойства пусты
}, {
  type: "characterData"
  target: <text node>
  // ...детали изменений зависят от того, как браузер о
  // он может соединить два соседних текстовых узла "От
  // или может оставить их разными текстовыми узлами
}];
```

Так что, `MutationObserver` позволяет реагировать на любые изменения в DOM-поддереве.

Использование для интеграции

Когда это может быть нужно?

Представим ситуацию, когда вы подключаете сторонний скрипт, который добавляет какую-то полезную функциональность на страницу, но при этом делает что-то лишнее, например, показывает рекламу `<div class="ads">Ненужная реклама</div>`.

Разумеется, сторонний скрипт не даёт каких-то механизмов её убрать.

Используя `MutationObserver`, мы можем отследить, когда в нашем DOM появится такой элемент и удалить его. А полезную функциональность оставить. Хотя, конечно, создатели стороннего скрипта вряд ли обрадуются, что вы их полезный скрипт взяли, а рекламу удалили.

Есть и другие ситуации, когда сторонний скрипт добавляет что-то в наш документ, и мы хотели бы отследить, когда это происходит, чтобы адаптировать нашу страницу, динамически поменять какие-то размеры и т.п.

`MutationObserver` для этого как раз отлично подходит.

Использование для архитектуры

Есть и ситуации, когда `MutationObserver` хорошо подходит с архитектурной точки зрения.

Представим, что мы создаём сайт о программировании. Естественно, статьи на нём и другие материалы могут содержать фрагменты с исходным кодом.

Такой фрагмент в HTML-разметке выглядит так:

```
...
<pre class="language-javascript"><code>
  // вот код
  let hello = "world";
</code></pre>
...
```

Также на нашем сайте мы будем использовать JavaScript-библиотеку для подсветки синтаксиса, например Prism.js. Вызов метода `Prism.highlightElem(pre)` ищет такие элементы `pre` и добавляет в них стили и теги, которые в итоге дают цветную подсветку синтаксиса, подобно той, которую вы видите в примерах здесь, на этой странице.

Когда конкретно нам вызвать этот метод подсветки? Можно по событию `DOMContentLoaded` или просто внизу страницы написать код, который будет искать все `pre[class*="language"]` и вызывать `Prism.highlightElem` для них:

```
// выделить все примеры кода на странице
document.querySelectorAll('pre[class*="language"]').for
```

Пока всё просто, правда? В HTML есть фрагменты кода в `<pre>`, и для них мы включаем подсветку синтаксиса.

Идём дальше. Представим, что мы собираемся динамически подгружать материалы с сервера. Позже в учебнике мы изучим способы для этого. На данный момент имеет значение только то, что мы получаем HTML-статью с веб-сервера и показываем её по запросу:

```
let article = /* получить новую статью с сервера */
articleElem.innerHTML = article;
```

HTML подгружённой статьи `article` может содержать примеры кода. Нам нужно вызвать `Prism.highlightElem` для них, чтобы подсветить синтаксис.

Кто и когда должен вызывать `Prism.highlightElem` для динамически загруженной статьи?

Мы можем добавить этот вызов к коду, который загружает статью, например, так:

```
let article = /* получить новую статью с сервера */
articleElem.innerHTML = article;

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

...Но представьте, что у нас есть много мест в коде, где мы загружаем что-либо: статьи, опросы, посты форума. Нужно ли нам в каждый такой вызов добавлять `Prism.highlightElem`? Получится не очень удобно, да и можно легко забыть сделать это.

А что, если содержимое загружается вообще сторонним кодом? Например, у нас есть форум, написанный другим человеком, загружающий содержимое динамически, и нам захотелось добавить к нему выделение синтаксиса. Никто не любит править чужие скрипты.

К счастью, есть другой вариант.

Мы можем использовать `MutationObserver`, чтобы автоматически определять момент, когда примеры кода появляются на странице, и подсвечивать их.

Тогда вся функциональность для подсветки синтаксиса будет в одном месте, а мы будем избавлены от необходимости интегрировать её.

Пример динамической подсветки синтаксиса

Вот работающий пример.

Если вы запустите этот код, он начнёт наблюдать за элементом ниже, подсвечивая код любого примера, который появляется там:

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

  for(let mutation of mutations) {
    // проверим новые узлы, есть ли что-то, что надо по
    for(let node of mutation.addedNodes) {
      // отслеживаем только узлы-элементы, другие (текс
      if (!(node instanceof HTMLElement)) continue;

      // проверить, не является ли вставленный элемент
      if (node.matches('pre[class*="language-"]')) {
        Prism.highlightElement(node);
      }

      // или, может быть, пример кода есть в его поддер
      for(let elem of node.querySelectorAll('pre[class*
        Prism.highlightElement(elem);
      }
    }
  }

});

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

observer.observe(demoElem, {childList: true, subtree: t
```

Ниже находится HTML-элемент и JavaScript, который его динамически заполнит примером кода через `innerHTML`.

Пожалуйста, запустите предыдущий код (он наблюдает за этим элементом), а затем код, расположенный ниже. Вы увидите как `MutationObserver` обнаружит и подсветит фрагменты кода.

Демо-элемент с `id="highlight-demo"`, за которым следит код примера выше.

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

// динамически вставить содержимое как фрагменты кода
demoElem.innerHTML = `Фрагмент кода ниже:
  <pre class="language-javascript"><code> let hello = "
  <div>Ещё один:</div>
  <div>
```

```
8       <pre class="language-css"><code>.class { margin: 5p
9     </div>
10  `;
```


Теперь у нас есть `MutationObserver`, который может отслеживать вставку кода в наблюдаемых элементах или во всём документе. Мы можем добавлять/удалять фрагменты кода в HTML, не задумываясь об их подсветке.

Дополнительные методы

Метод, останавливающий наблюдение за узлом:

- `observer.disconnect()` – останавливает наблюдение.

Вместе с ним используют метод:

- `mutationRecords = observer.takeRecords()` – получает список необработанных записей изменений, которые произошли, но колбэк для них ещё не выполнился.

```
1  // мы отключаем наблюдатель
2  observer.disconnect();
3
4  // он, возможно, не успел обработать некоторые изменени
5  let mutationRecords = observer.takeRecords();
6  // обработать mutationRecords
```

Сборка мусора

Объекты `MutationObserver` используют внутри себя так называемые «слабые ссылки» на узлы, за которыми смотрят. Так что если узел удалён из DOM и больше не достижим, то он будет удалён из памяти вне зависимости от наличия наблюдателя.

Итого

`MutationObserver` может реагировать на изменения в DOM: атрибуты, добавленные/удалённые элементы, текстовое содержимое.

Мы можем использовать его, чтобы отслеживать изменения, производимые другими частями нашего собственного кода, а также интегрироваться со сторонними библиотеками.

`MutationObserver` может отслеживать любые изменения. Разные опции конфигурации «что наблюдать» предназначены для оптимизации, чтобы не тратить ресурсы на лишние вызовы колбэка.

Комментарии

перед тем как писать...