

Mutation Observer API

概述

Mutation Observer API 用来监视 DOM 变动。DOM 的任何变动，比如节点的增减、属性的变动、文本内容的变动，这个 API 都可以得到通知。

概念上，它很接近事件，可以理解为 DOM 发生变动就会触发 Mutation Observer 事件。但是，它与事件有一个本质不同：事件是同步触发，也就是说，DOM 的变动立刻会触发相应的事件；Mutation Observer 则是异步触发，DOM 的变动并不会马上触发，而是要等到当前所有 DOM 操作都结束才触发。

这样设计是为了应付 DOM 变动频繁的特点。举例来说，如果文档中连续插入1000个`<p>`元素，就会连续触发1000个插入事件，执行每个事件的回调函数，这很可能造成浏览器的卡顿；而 Mutation Observer 完全不同，只在1000个段落都插入结束后才会触发，而且只触发一次。

Mutation Observer 有以下特点。

- 它等待所有脚本任务完成后，才会运行（即异步触发方式）。
- 它把 DOM 变动记录封装成一个数组进行处理，而不是一条条个别处理 DOM 变动。
- 它既可以观察 DOM 的所有类型变动，也可以指定只观察某一类变动。

MutationObserver 构造函数

使用时，首先使用`MutationObserver`构造函数，新建一个观察器实例，同时指定这个实例的回调函数。

```
```javascript
var observer = new MutationObserver(callback);
```
```

上面代码中的回调函数，会在每次 DOM 变动后调用。该回调函数接受两个参数，第一个是变动数组，第二个是观察器实例，下面是一个例子。

```
```javascript
var observer = new MutationObserver(function (mutations, observer) {
 mutations.forEach(function(mutation) {
 console.log(mutation);
 });
});
```
```

MutationObserver 的实例方法

observe()

`observe`方法用来启动监听，它接受两个参数。

- 第一个参数：所要观察的 DOM 节点
- 第二个参数：一个配置对象，指定所要观察的特定变动

```
```javascript
var article = document.querySelector('article');

var options = {
 'childList': true,
 'attributes': true
};

observer.observe(article, options);
```
```

上面代码中，`observe`方法接受两个参数，第一个是所要观察的DOM元素是`article`，第二个是所要观察的变动类型（子节点变动和属性变动）。

观察器所能观察的 DOM 变动类型（即上面代码的`options`对象），有以下几种。

- **childList**：子节点的变动（指新增，删除或者更改）。
- **attributes**：属性的变动。
- **characterData**：节点内容或节点文本的变动。

想要观察哪一种变动类型，就在`option`对象中指定它的值为`true`。需要注意的是，必须同时指定`childList`、`attributes`和`characterData`中的一种或多种，若未均指定将报错。

除了变动类型，`options`对象还可以设定以下属性：

- `subtree`：布尔值，表示是否将该观察器应用于该节点的所有后代节点。
- `attributeOldValue`：布尔值，表示观察`attributes`变动时，是否需要记录变动前的属性值。
- `characterDataOldValue`：布尔值，表示观察`characterData`变动时，是否需要记录变动前的值。
- `attributeFilter`：数组，表示需要观察的特定属性（比如`['class', 'src']`）。

```
```javascript
// 开始监听文档根节点（即<html>标签）的变动
mutationObserver.observe(document.documentElement, {
 attributes: true,
 characterData: true,
 childList: true,
 subtree: true,
 attributeOldValue: true,
 characterDataOldValue: true
});
```
```

对一个节点添加观察器，就像使用`addEventListener`方法一样，多次添加同一个观察器是无效的，回调函数依然只会触发一次。如果指定不同的`options`对象，以后面添加的那个为准，类似覆盖。

下面的例子是观察新增的子节点。

```
```javascript
var insertedNodes = [];
var observer = new MutationObserver(function(mutations) {
 mutations.forEach(function(mutation) {
 for (var i = 0; i < mutation.addedNodes.length; i++) {
 insertedNodes.push(mutation.addedNodes[i]);
 }
 });
 console.log(insertedNodes);
});
observer.observe(document, { childList: true, subtree: true });
```
```

disconnect(), takeRecords ()

`disconnect`方法用来停止观察。调用该方法后，DOM 再发生变动，也不会触发观察器。

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

`takeRecords`方法用来清除变动记录，即不再处理未处理的变动。该方法返回变动记录的数组。

```
```javascript
observer.takeRecords();
```
```

下面是一个例子。

```
```javascript
// 保存所有没有被观察器处理的变动
var changes = mutationObserver.takeRecords();

// 停止观察
mutationObserver.disconnect();
```
```

MutationRecord 对象

DOM 每次发生变化，就会生成一条变动记录（MutationRecord 实例）。该实例包含了与变动相关的所有信息。Mutation Observer 处理的就是一个个`MutationRecord`实例所组成的数组。

`MutationRecord`对象包含了DOM的相关信息，有如下属性：

- `type`：观察的变动类型（`attributes`、`characterData`或者`childList`）。
- `target`：发生变动的DOM节点。
- `addedNodes`：新增的DOM节点。
- `removedNodes`：删除的DOM节点。
- `previousSibling`：前一个同级节点，如果没有则返回`null`。
- `nextSibling`：下一个同级节点，如果没有则返回`null`。
- `attributeName`：发生变动的属性。如果设置了`attributeFilter`，则只返回预先指定的属性。
- `oldValue`：变动前的值。这个属性只对`attribute`和`characterData`变动有效，如果发生`childList`变动，则返回`null`。

应用示例

子元素的变动

下面的例子说明如何读取变动记录。

```
```\javascript
var callback = function (records){
 records.map(function(record){
 console.log('Mutation type: ' + record.type);
 console.log('Mutation target: ' + record.target);
 });
};

var mo = new MutationObserver(callback);

var option = {
 'childList': true,
 'subtree': true
};

mo.observe(document.body, option);
```\
```

上面代码的观察器，观察`<body>`的所有下级节点（`childList`表示观察子节点，`subtree`表示观察后代节点）的变动。回调函数会在控制台显示所有变动的类型和目标节点。

属性的变动

下面的例子说明如何追踪属性的变动。

```
```\javascript
var callback = function (records) {
 records.map(function (record) {
 console.log('Previous attribute value: ' + record.oldValue);
 });
};
```

```

 });
};

var mo = new MutationObserver(callback);

var element = document.getElementById('#my_element');

var options = {
 'attributes': true,
 'attributeOldValue': true
}

mo.observe(element, options);
```

```

上面代码先设定追踪属性变动（`'attributes': true`），然后设定记录变动前的值。实际发生变动时，会将变动前的值显示在控制台。

取代 DOMContentLoaded 事件

网页加载的时候，DOM 节点的生成会产生变动记录，因此只要观察 DOM 的变动，就能在第一时间触发相关事件，也就没有必要使用`DOMContentLoaded`事件。

```

```javascript
var observer = new MutationObserver(callback);
observer.observe(document.documentElement, {
 childList: true,
 subtree: true
});
```

```

上面代码中，监听`document.documentElement`（即网页的`<html>`HTML 节点）的子节点的变动，`subtree`属性指定监听还包括后代节点。因此，任意一个网页元素一旦生成，就能立刻被监听到。

下面的代码，使用`MutationObserver`对象封装一个监听 DOM 生成的函数。

```

```javascript
(function(win){
 'use strict';

 var listeners = [];
 var doc = win.document;
 var MutationObserver = win.MutationObserver || win.WebKitMutationObserver;
 var observer;

 function ready(selector, fn){
 // 储存选择器和回调函数
 listeners.push({
 selector: selector,

```

```

 fn: fn
 });
 if(!observer){
 // 监听document变化
 observer = new MutationObserver(check);
 observer.observe(doc.documentElement, {
 childList: true,
 subtree: true
 });
 }
 // 检查该节点是否已经在DOM中
 check();
}

function check(){
 // 检查是否匹配已储存的节点
 for(var i = 0; i < listeners.length; i++){
 var listener = listeners[i];
 // 检查指定节点是否有匹配
 var elements = doc.querySelectorAll(listener.selector);
 for(var j = 0; j < elements.length; j++){
 var element = elements[j];
 // 确保回调函数只会对该元素调用一次
 if(!element.ready){
 element.ready = true;
 // 对该节点调用回调函数
 listener.fn.call(element, element);
 }
 }
 }
}

// 对外暴露ready
win.ready = ready;

})(this);

// 使用方法
ready('.foo', function(element){
 // ...
});

```

### ## 参考链接

- Paul Kinlan, [Detect DOM changes with Mutation Observers](https://developers.google.com/web/updates/2012/02/Detect-DOM-changes-with-Mutation-Observers)
- Tiffany Brown, [Getting to know mutation observers](http://dev.opera.com/articles/view/mutation-observers-tutorial/)

- Michal Budzynski, [JavaScript: The less known parts. DOM Mutations](<http://michalbe.blogspot.com/2013/04/javascript-less-known-parts-dom.html>)
- Jeff Griffiths, [DOM MutationObserver – reacting to DOM changes without killing browser performance](<https://hacks.mozilla.org/2012/05/dom-mutationobserver-reacting-to-dom-changes-without-killing-browser-performance/>)
- Addy Osmani, [Detect, Undo And Redo DOM Changes With Mutation Observers](<http://addyosmani.com/blog/mutation-observers/>)
- Ryan Morr, [Using Mutation Observers to Watch for Element Availability](<http://ryanmorr.com/using-mutation-observers-to-watch-for-element-availability/>)