Document 节点

概述

`document`节点对象代表整个文档，每张网页都有自己的`document`对象。`window.document`属性就指向这个对象。只要浏览器开始载入 HTML 文档，该对象就存在了，可以直接使用。

`document`对象有不同的办法可以获取。

- 正常的网页，直接使用`document`或`window.document`。
- `iframe`框架里面的网页，使用`iframe`节点的`contentDocument`属性。
- Ajax 操作返回的文档，使用`XMLHttpRequest`对象的`responseXML`属性。
- 内部节点的`ownerDocument`属性。

`document`对象继承了`EventTarget`接口、`Node`接口、`ParentNode`接口。这意味着，这些接口的方法都可以在`document`对象上调用。除此之外，`document`对象还有很多自己的属性和方法。

属性

快捷方式属性

以下属性是指向文档内部的某个节点的快捷方式。

（1）document.defaultView

`document.defaultView`属性返回`document`对象所属的`window`对象。如果当前文档不属于`window`对象，该属性返回`null`。

```javascript
document.defaultView === window // true
```

（2）document.doctype

对于 HTML 文档来说，`document`对象一般有两个子节点。第一个子节点是`document.doctype`，指向`<DOCTYPE>`节点，即文档类型（Document Type Declaration，简写DTD）节点。HTML 的文档类型节点，一般写成`<!DOCTYPE html>`。如果网页没有声明 DTD，该属性返回`null`。

```javascript
var doctype = document.doctype;
doctype // "<!DOCTYPE html>"
doctype.name // "html"
```

`document.firstChild`通常就返回这个节点。

（3）document.documentElement

`document.documentElement`属性返回当前文档的根元素节点（root）。它通常是`document`节点的第二个子节点，紧跟在`document.doctype`节点后面。HTML网页的该属性，一般是`<html>`节点。

（4）document.body，document.head

`document.body`属性指向`<body>`节点，`document.head`属性指向`<head>`节点。

这两个属性总是存在的，如果网页源码里面省略了`<head>`或`<body>`，浏览器会自动创建。另外，这两个属性是可写的，如果改写它们的值，相当于移除所有子节点。

（5）document.scrollingElement

`document.scrollingElement`属性返回文档的滚动元素。也就是说，当文档整体滚动时，到底是哪个元素在滚动。

标准模式下，这个属性返回的文档的根元素`document.documentElement`（即`<html>`）。兼容（quirk）模式下，返回的是`<body>`元素，如果该元素不存在，返回`null`。

```javascript
// 页面滚动到浏览器顶部
document.scrollingElement.scrollTop = 0;
```

（6）document.activeElement

`document.activeElement`属性返回获得当前焦点（focus）的 DOM 元素。通常，这个属性返回的是`<input>`、`<textarea>`、`<select>`等表单元素，如果当前没有焦点元素，返回`<body>`元素或`null`。

（7）document.fullscreenElement

`document.fullscreenElement`属性返回当前以全屏状态展示的 DOM 元素。如果不是全屏状态，该属性返回`null`。

```javascript
if (document.fullscreenElement.nodeName == 'VIDEO') {
  console.log('全屏播放视频');
}
```

上面代码中，通过`document.fullscreenElement`可以知道`<video>`元素有没有处在全屏状态，从而判断用户行为。

节点集合属性

以下属性返回一个`HTMLCollection`实例，表示文档内部特定元素的集合。这些集合都是动态的，原节点有任何变化，立刻会反映在集合中。

（1）document.links

`document.links`属性返回当前文档所有设定了`href`属性的`<a>`及`<area>`节点。

```javascript
// 打印文档所有的链接
var links = document.links;
for(var i = 0; i < links.length; i++) {
  console.log(links[i]);
}
```

（2）document.forms

`document.forms`属性返回所有`<form>`表单节点。

```javascript
var selectForm = document.forms[0];
```

上面代码获取文档第一个表单。

除了使用位置序号，`id`属性和`name`属性也可以用来引用表单。

```javascript
/* HTML 代码如下
  <form name="foo" id="bar"></form>
*/
document.forms[0] === document.forms.foo // true
document.forms.bar === document.forms.foo // true
```

（3）document.images

`document.images`属性返回页面所有`<img>`图片节点。

```javascript
var imglist = document.images;
```

```
for(var i = 0; i < imglist.length; i++) {
  if (imglist[i].src === 'banner.gif') {
    // ...
  }
}
```

上面代码在所有`img`标签中，寻找某张图片。

（4）document.embeds，document.plugins

`document.embeds`属性和`document.plugins`属性，都返回所有`<embed>`节点。

（5）document.scripts

`document.scripts`属性返回所有`<script>`节点。

```javascript
var scripts = document.scripts;
if (scripts.length !== 0 ) {
  console.log('当前网页有脚本');
}
```

（6）document.styleSheets

`document.styleSheets`属性返回文档内嵌或引入的样式表集合，详细介绍请看《CSS 对象模型》一章。

（7）小结

除了`document.styleSheets`，以上的集合属性返回的都是`HTMLCollection`实例。

```javascript
document.links instanceof HTMLCollection // true
document.images instanceof HTMLCollection // true
document.forms instanceof HTMLCollection // true
document.embeds instanceof HTMLCollection // true
document.scripts instanceof HTMLCollection // true
```

`HTMLCollection`实例是类似数组的对象，所以这些属性都有`length`属性，都可以使用方括号运算符引用成员。如果成员有`id`或`name`属性，还可以用这两个属性的值，在`HTMLCollection`实例上引用到这个成员。

```javascript
// HTML 代码如下
// <form name="myForm">
```

```
document.myForm === document.forms.myForm // true
```

文档静态信息属性

以下属性返回文档信息。

（1）document.documentURI，document.URL

`document.documentURI`属性和`document.URL`属性都返回一个字符串，表示当前文档的网址。不同之处是它们继承自不同的接口，`documentURI`继承自`Document`接口，可用于所有文档；`URL`继承自`HTMLDocument`接口，只能用于 HTML 文档。

```javascript
document.URL
// http://www.example.com/about

document.documentURI === document.URL
// true
```

如果文档的锚点（`#anchor`）变化，这两个属性都会跟着变化。

（2）document.domain

`document.domain`属性返回当前文档的域名，不包含协议和端口。比如，网页的网址是`http://www.example.com:80/hello.html`，那么`document.domain`属性就等于`www.example.com`。如果无法获取域名，该属性返回`null`。

`document.domain`基本上是一个只读属性，只有一种情况除外。次级域名的网页，可以把`document.domain`设为对应的上级域名。比如，当前域名是`a.sub.example.com`，则`document.domain`属性可以设置为`sub.example.com`，也可以设为`example.com`。修改后，`document.domain`相同的两个网页，可以读取对方的资源，比如设置的 Cookie。

另外，设置`document.domain`会导致端口被改成`null`。因此，如果通过设置`document.domain`来进行通信，双方网页都必须设置这个值，才能保证端口相同。

（3）document.location

`Location`对象是浏览器提供的原生对象，提供 URL 相关的信息和操作方法。通过`window.location`和`document.location`属性，可以拿到这个对象。

关于这个对象的详细介绍，请看《浏览器模型》部分的《Location 对象》章节。

（4）document.lastModified

`document.lastModified`属性返回一个字符串，表示当前文档最后修改的时间。不同浏览器的返回值，日期格式是不一样的。

```javascript
document.lastModified
// "03/07/2018 11:18:27"
```

注意，`document.lastModified`属性的值是字符串，所以不能直接用来比较。`Date.parse`方法将其转为`Date`实例，才能比较两个网页。

```javascript
var lastVisitedDate = Date.parse('01/01/2018');
if (Date.parse(document.lastModified) > lastVisitedDate) {
  console.log('网页已经变更');
}
```

如果页面上有 JavaScript 生成的内容，`document.lastModified`属性返回的总是当前时间。

（5）document.title

`document.title`属性返回当前文档的标题。默认情况下，返回`<title>`节点的值。但是该属性是可写的，一旦被修改，就返回修改后的值。

```javascript
document.title = '新标题';
document.title // "新标题"
```

（6）document.characterSet

`document.characterSet`属性返回当前文档的编码，比如`UTF-8`、`ISO-8859-1`等等。

（7）document.referrer

`document.referrer`属性返回一个字符串，表示当前文档的访问者来自哪里。

```javascript
document.referrer
// "https://example.com/path"
```

如果无法获取来源，或者用户直接键入网址而不是从其他网页点击进入，`document.referrer`返回一个空字符串。

`document.referrer`的值，总是与 HTTP 头信息的`Referer`字段保持一致。但是，`document.referrer`的拼写有两个`r`，而头信息的`Referer`字段只有一个`r`。

（8）document.dir

`document.dir`返回一个字符串，表示文字方向。它只有两个可能的值：`rtl`表示文字从右到左，阿拉伯文是这种方式；`ltr`表示文字从左到右，包括英语和汉语在内的大多数文字采用这种方式。

（9）document.compatMode

`compatMode`属性返回浏览器处理文档的模式，可能的值为`BackCompat`（向后兼容模式）和`CSS1Compat`（严格模式）。

一般来说，如果网页代码的第一行设置了明确的`DOCTYPE`（比如`<!doctype html>`），`document.compatMode`的值都为`CSS1Compat`。

文档状态属性

（1）document.hidden

`document.hidden`属性返回一个布尔值，表示当前页面是否可见。如果窗口最小化、浏览器切换了 Tab，都会导致导致页面不可见，使得`document.hidden`返回`true`。

这个属性是 Page Visibility API 引入的，一般都是配合这个 API 使用。

（2）document.visibilityState

`document.visibilityState`返回文档的可见状态。

它的值有四种可能。

> - `visible`：页面可见。注意，页面可能是部分可见，即不是焦点窗口，前面被其他窗口部分挡住了。
> - `hidden`：页面不可见，有可能窗口最小化，或者浏览器切换到了另一个 Tab。
> - `prerender`：页面处于正在渲染状态，对于用户来说，该页面不可见。
> - `unloaded`：页面从内存里面卸载了。

这个属性可以用在页面加载时，防止加载某些资源；或者页面不可见时，停掉一些页面功能。

（3）document.readyState

`document.readyState`属性返回当前文档的状态，共有三种可能的值。

- `loading`：加载 HTML 代码阶段（尚未完成解析）

- `interactive`：加载外部资源阶段
- `complete`：加载完成

这个属性变化的过程如下。

1. 浏览器开始解析 HTML 文档，`document.readyState`属性等于`loading`。
1. 浏览器遇到 HTML 文档中的`<script>`元素，并且没有`async`或`defer`属性，就暂停解析，开始执行脚本，这时`document.readyState`属性还是等于`loading`。
1. HTML 文档解析完成，`document.readyState`属性变成`interactive`。
1. 浏览器等待图片、样式表、字体文件等外部资源加载完成，一旦全部加载完成，`document.readyState`属性变成`complete`。

下面的代码用来检查网页是否加载成功。

```javascript
// 基本检查
if (document.readyState === 'complete') {
  // ...
}

// 轮询检查
var interval = setInterval(function() {
  if (document.readyState === 'complete') {
    clearInterval(interval);
    // ...
  }
}, 100);
```

另外，每次状态变化都会触发一个`readystatechange`事件。

document.cookie

`document.cookie`属性用来操作浏览器 Cookie，详见《浏览器模型》部分的《Cookie》章节。

document.designMode

`document.designMode`属性控制当前文档是否可编辑。该属性只有两个值`on`和`off`，默认值为`off`。一旦设为`on`，用户就可以编辑整个文档的内容。

下面代码打开`iframe`元素内部文档的`designMode`属性，就能将其变为一个所见即所得的编辑器。

```javascript
// HTML 代码如下
// <iframe id="editor" src="about:blank"></iframe>
var editor = document.getElementById('editor');
```

```
editor.contentDocument.designMode = 'on';
```

document.implementation

`document.implementation`属性返回一个`DOMImplementation`对象。该对象有三个方法，主要用于创建独立于当前文档的新的 Document 对象。

- `DOMImplementation.createDocument()`：创建一个 XML 文档。
- `DOMImplementation.createHTMLDocument()`：创建一个 HTML 文档。
- `DOMImplementation.createDocumentType()`：创建一个 DocumentType 对象。

下面是创建 HTML 文档的例子。

```javascript
var doc = document.implementation.createHTMLDocument('Title');
var p = doc.createElement('p');
p.innerHTML = 'hello world';
doc.body.appendChild(p);

document.replaceChild(
  doc.documentElement,
  document.documentElement
);
```

上面代码中，第一步生成一个新的 HTML 文档`doc`，然后用它的根元素`document.documentElement`替换掉`document.documentElement`。这会使得当前文档的内容全部消失，变成`hello world`。

方法

document.open()，document.close()

`document.open`方法清除当前文档所有内容，使得文档处于可写状态，供`document.write`方法写入内容。

`document.close`方法用来关闭`document.open()`打开的文档。

```javascript
document.open();
document.write('hello world');
document.close();
```

document.write()，document.writeln()

`document.write`方法用于向当前文档写入内容。

在网页的首次渲染阶段，只要页面没有关闭写入（即没有执行`document.close()`），`document.write`写入的内容就会追加在已有内容的后面。

```javascript
// 页面显示“helloworld”
document.open();
document.write('hello');
document.write('world');
document.close();
```

注意，`document.write`会当作 HTML 代码解析，不会转义。

```javascript
document.write('<p>hello world</p>');
```

上面代码中，`document.write`会将`<p>`当作 HTML 标签解释。

如果页面已经解析完成（`DOMContentLoaded`事件发生之后），再调用`write`方法，它会先调用`open`方法，擦除当前文档所有内容，然后再写入。

```javascript
document.addEventListener('DOMContentLoaded', function (event) {
  document.write('<p>Hello World!</p>');
});

// 等同于
document.addEventListener('DOMContentLoaded', function (event) {
  document.open();
  document.write('<p>Hello World!</p>');
  document.close();
});
```

如果在页面渲染过程中调用`write`方法，并不会自动调用`open`方法。（可以理解成，`open`方法已调用，但`close`方法还未调用。）

```html
<html>
<body>
hello
<script type="text/javascript">
  document.write("world")
</script>
</body>
</html>
```

在浏览器打开上面网页，将会显示`hello world`。

`document.write`是 JavaScript 语言标准化之前就存在的方法，现在完全有更符合标准的方法向文档写入内容（比如对`innerHTML`属性赋值）。所以，除了某些特殊情况，应该尽量避免使用`document.write`这个方法。

`document.writeln`方法与`write`方法完全一致，除了会在输出内容的尾部添加换行符。

```javascript
document.write(1);
document.write(2);
// 12

document.writeln(1);
document.writeln(2);
// 1
// 2
//
```

注意，`writeln`方法添加的是 ASCII 码的换行符，渲染成 HTML 网页时不起作用，即在网页上显示不出换行。网页上的换行，必须显式写入`<br>`。

document.querySelector()，document.querySelectorAll()

`document.querySelector`方法接受一个 CSS 选择器作为参数，返回匹配该选择器的元素节点。如果有多个节点满足匹配条件，则返回第一个匹配的节点。如果没有发现匹配的节点，则返回`null`。

```javascript
var el1 = document.querySelector('.myclass');
var el2 = document.querySelector('#myParent > [ng-click]');
```

`document.querySelectorAll`方法与`querySelector`用法类似，区别是返回一个`NodeList`对象，包含所有匹配给定选择器的节点。

```javascript
elementList = document.querySelectorAll('.myclass');
```

这两个方法的参数，可以是逗号分隔的多个 CSS 选择器，返回匹配其中一个选择器的元素节点，这与 CSS 选择器的规则是一致的。

```javascript
var matches = document.querySelectorAll('div.note, div.alert');
```

上面代码返回`class`属性是`note`或`alert`的`div`元素。

这两个方法都支持复杂的 CSS 选择器。

```javascript
// 选中 data-foo-bar 属性等于 someval 的元素
document.querySelectorAll('[data-foo-bar="someval"]');

// 选中 myForm 表单中所有不通过验证的元素
document.querySelectorAll('#myForm :invalid');

// 选中div元素，那些 class 含 ignore 的除外
document.querySelectorAll('DIV:not(.ignore)');

// 同时选中 div，a，script 三类元素
document.querySelectorAll('DIV, A, SCRIPT');
```

但是，它们不支持 CSS 伪元素的选择器（比如`:first-line`和`:first-letter`）和伪类的选择器（比如`:link`和`:visited`），即无法选中伪元素和伪类。

如果`querySelectorAll`方法的参数是字符串`*`，则会返回文档中的所有元素节点。另外，`querySelectorAll`的返回结果不是动态集合，不会实时反映元素节点的变化。

最后，这两个方法除了定义在`document`对象上，还定义在元素节点上，即在元素节点上也可以调用。

document.getElementsByTagName()

`document.getElementsByTagName`方法搜索 HTML 标签名，返回符合条件的元素。它的返回值是一个类似数组对象（`HTMLCollection`实例），可以实时反映 HTML 文档的变化。如果没有任何匹配的元素，就返回一个空集。

```javascript
var paras = document.getElementsByTagName('p');
paras instanceof HTMLCollection // true
```

上面代码返回当前文档的所有`p`元素节点。

HTML 标签名是大小写不敏感的，因此`getElementsByTagName`方法也是大小写不敏感的。另外，返回结果中，各个成员的顺序就是它们在文档中出现的顺序。

如果传入`*`，就可以返回文档中所有 HTML 元素。

```javascript
var allElements = document.getElementsByTagName('*');
```

```

注意，元素节点本身也定义了`getElementsByTagName`方法，返回该元素的后代元素中符合条件的元素。也就是说，这个方法不仅可以在`document`对象上调用，也可以在任何元素节点上调用。

```javascript
var firstPara = document.getElementsByTagName('p')[0];
var spans = firstPara.getElementsByTagName('span');
```

上面代码选中第一个`p`元素内部的所有`span`元素。

document.getElementsByClassName()

`document.getElementsByClassName`方法返回一个类似数组的对象（`HTMLCollection`实例），包括了所有`class`名字符合指定条件的元素，元素的变化实时反映在返回结果中。

```javascript
var elements = document.getElementsByClassName(names);
```

由于`class`是保留字，所以 JavaScript 一律使用`className`表示 CSS 的`class`。

参数可以是多个`class`，它们之间使用空格分隔。

```javascript
var elements = document.getElementsByClassName('foo bar');
```

上面代码返回同时具有`foo`和`bar`两个`class`的元素，`foo`和`bar`的顺序不重要。

注意，正常模式下，CSS 的`class`是大小写敏感的。（`quirks mode`下，大小写不敏感。）

与`getElementsByTagName`方法一样，`getElementsByClassName`方法不仅可以在`document`对象上调用，也可以在任何元素节点上调用。

```javascript
// 非document对象上调用
var elements = rootElement.getElementsByClassName(names);
```

document.getElementsByName()

`document.getElementsByName`方法用于选择拥有`name`属性的 HTML 元素（比如`<form>`、`<radio>`、`<img>`、`<frame>`、`<embed>`和`<object>`等），返回一个类似数组的的对象（`NodeList`实例），因为`name`属性相同的元素可能不止一个。
```

```javascript
// 表单为 <form name="x"></form>
var forms = document.getElementsByName('x');
forms[0].tagName // "FORM"
```

document.getElementById()

`document.getElementById`方法返回匹配指定`id`属性的元素节点。如果没有发现匹配的节点，则返回`null`。

```javascript
var elem = document.getElementById('para1');
```

注意，该方法的参数是大小写敏感的。比如，如果某个节点的`id`属性是`main`，那么`document.getElementById('Main')`将返回`null`。

`document.getElementById`方法与`document.querySelector`方法都能获取元素节点，不同之处是`document.querySelector`方法的参数使用 CSS 选择器语法，`document.getElementById`方法的参数是元素的`id`属性。

```javascript
document.getElementById('myElement')
document.querySelector('#myElement')
```

上面代码中，两个方法都能选中`id`为`myElement`的元素，但是`document.getElementById()`比`document.querySelector()`效率高得多。

另外，这个方法只能在`document`对象上使用，不能在其他元素节点上使用。

document.elementFromPoint()，document.elementsFromPoint()

`document.elementFromPoint`方法返回位于页面指定位置最上层的元素节点。

```javascript
var element = document.elementFromPoint(50, 50);
```

上面代码选中在`(50, 50)`这个坐标位置的最上层的那个 HTML 元素。

`elementFromPoint`方法的两个参数，依次是相对于当前视口左上角的横坐标和纵坐标，单位是像素。如果位于该位置的 HTML 元素不可返回（比如文本框的滚动条），则返回它的父元素（比如文本框）。如果坐标值无意义（比如负值或超过视口大小），则返回`null`。

`document.elementsFromPoint()`返回一个数组，成员是位于指定坐标（相对于视口）的所有元素。

```javascript
var elements = document.elementsFromPoint(x, y);
```

document.createElement()

`document.createElement`方法用来生成元素节点，并返回该节点。

```javascript
var newDiv = document.createElement('div');
```

`createElement`方法的参数为元素的标签名，即元素节点的`tagName`属性，对于 HTML 网页大小写不敏感，即参数为`div`或`DIV`返回的是同一种节点。如果参数里面包含尖括号（即`<`和`>`）会报错。

```javascript
document.createElement('<div>');
// DOMException: The tag name provided ('<div>') is not a valid name
```

注意，`document.createElement`的参数可以是自定义的标签名。

```javascript
document.createElement('foo');
```

document.createTextNode()

`document.createTextNode`方法用来生成文本节点（`Text`实例），并返回该节点。它的参数是文本节点的内容。

```javascript
var newDiv = document.createElement('div');
var newContent = document.createTextNode('Hello');
newDiv.appendChild(newContent);
```

上面代码新建一个`div`节点和一个文本节点，然后将文本节点插入`div`节点。

这个方法可以确保返回的节点，被浏览器当作文本渲染，而不是当作 HTML 代码渲染。因此，可以用来展示用户的输入，避免 XSS 攻击。

```javascript
var div = document.createElement('div');
div.appendChild(document.createTextNode('<span>Foo & bar</span>'));
```

```
console.log(div.innerHTML)
// &lt;span&gt;Foo &amp; bar&lt;/span&gt;
```

上面代码中，`createTextNode`方法对大于号和小于号进行转义，从而保证即使用户输入的内容包含恶意代码，也能正确显示。

需要注意的是，该方法不对单引号和双引号转义，所以不能用来对 HTML 属性赋值。

```html
function escapeHtml(str) {
  var div = document.createElement('div');
  div.appendChild(document.createTextNode(str));
  return div.innerHTML;
};

var userWebsite = '" onmouseover="alert(\'derp\')" "';
var profileLink = '<a href="' + escapeHtml(userWebsite) + '">Bob</a>';
var div = document.getElementById('target');
div.innerHTML = profileLink;
// <a href="" onmouseover="alert('derp')" "">Bob</a>
```

上面代码中，由于`createTextNode`方法不转义双引号，导致`onmouseover`方法被注入了代码。

document.createAttribute()

`document.createAttribute`方法生成一个新的属性节点（`Attr`实例），并返回它。

```javascript
var attribute = document.createAttribute(name);
```

`document.createAttribute`方法的参数`name`，是属性的名称。

```javascript
var node = document.getElementById('div1');

var a = document.createAttribute('my_attrib');
a.value = 'newVal';

node.setAttributeNode(a);
// 或者
node.setAttribute('my_attrib', 'newVal');
```

上面代码为`div1`节点，插入一个值为`newVal`的`my_attrib`属性。

document.createComment()

`document.createComment`方法生成一个新的注释节点，并返回该节点。

```javascript
var CommentNode = document.createComment(data);
```

`document.createComment`方法的参数是一个字符串，会成为注释节点的内容。

document.createDocumentFragment()

`document.createDocumentFragment`方法生成一个空的文档片段对象（`DocumentFragment`实例）。

```javascript
var docFragment = document.createDocumentFragment();
```

`DocumentFragment`是一个存在于内存的 DOM 片段，不属于当前文档，常常用来生成一段较复杂的 DOM 结构，然后再插入当前文档。这样做的好处在于，因为`DocumentFragment`不属于当前文档，对它的任何改动，都不会引发网页的重新渲染，比直接修改当前文档的 DOM 有更好的性能表现。

```javascript
var docfrag = document.createDocumentFragment();

[1, 2, 3, 4].forEach(function (e) {
  var li = document.createElement('li');
  li.textContent = e;
  docfrag.appendChild(li);
});

var element  = document.getElementById('ul');
element.appendChild(docfrag);
```

上面代码中，文档片断`docfrag`包含四个`<li>`节点，这些子节点被一次性插入了当前文档。

document.createEvent()

`document.createEvent`方法生成一个事件对象（`Event`实例），该对象可以被`element.dispatchEvent`方法使用，触发指定事件。

```javascript
var event = document.createEvent(type);
```

`document.createEvent`方法的参数是事件类型，比如`UIEvents`、`MouseEvents`、`MutationEvents`、`HTMLEvents`。

```javascript
var event = document.createEvent('Event');
event.initEvent('build', true, true);
document.addEventListener('build', function (e) {
  console.log(e.type); // "build"
}, false);
document.dispatchEvent(event);
```

上面代码新建了一个名为`build`的事件实例，然后触发该事件。

document.addEventListener()，document.removeEventListener()，document.dispatchEvent()

这三个方法用于处理`document`节点的事件。它们都继承自`EventTarget`接口，详细介绍参见《EventTarget 接口》一章。

```javascript
// 添加事件监听函数
document.addEventListener('click', listener, false);

// 移除事件监听函数
document.removeEventListener('click', listener, false);

// 触发事件
var event = new Event('click');
document.dispatchEvent(event);
```

document.hasFocus()

`document.hasFocus`方法返回一个布尔值，表示当前文档之中是否有元素被激活或获得焦点。

```javascript
var focused = document.hasFocus();
```

注意，有焦点的文档必定被激活（active），反之不成立，激活的文档未必有焦点。比如，用户点击按钮，从当前窗口跳出一个新窗口，该新窗口就是激活的，但是不拥有焦点。

document.adoptNode()，document.importNode()

`document.adoptNode`方法将某个节点及其子节点，从原来所在的文档或`DocumentFragment`里面移除，归属当前`document`对象，返回插入后的新节点。插入的节点对象的`ownerDocument`属性，会变成当前的`document`对象，而`parentNode`属性是`null`。

```javascript
var node = document.adoptNode(externalNode);
document.appendChild(node);
```

```

注意，`document.adoptNode`方法只是改变了节点的归属，并没有将这个节点插入新的文档树。所以，还要再用`appendChild`方法或`insertBefore`方法，将新节点插入当前文档树。

`document.importNode`方法则是从原来所在的文档或`DocumentFragment`里面，拷贝某个节点及其子节点，让它们归属当前`document`对象。拷贝的节点对象的`ownerDocument`属性，会变成当前的`document`对象，而`parentNode`属性是`null`。

```javascript
var node = document.importNode(externalNode, deep);
```

`document.importNode`方法的第一个参数是外部节点，第二个参数是一个布尔值，表示对外部节点是深拷贝还是浅拷贝，默认是浅拷贝（false）。虽然第二个参数是可选的，但是建议总是保留这个参数，并设为`true`。

注意，`document.importNode`方法只是拷贝外部节点，这时该节点的父节点是`null`。下一步还必须将这个节点插入当前文档树。

```javascript
var iframe = document.getElementsByTagName('iframe')[0];
var oldNode = iframe.contentWindow.document.getElementById('myNode');
var newNode = document.importNode(oldNode, true);
document.getElementById("container").appendChild(newNode);
```

上面代码从`iframe`窗口，拷贝一个指定节点`myNode`，插入当前文档。

document.createNodeIterator()

`document.createNodeIterator`方法返回一个子节点遍历器。

```javascript
var nodeIterator = document.createNodeIterator(
  document.body,
  NodeFilter.SHOW_ELEMENT
);
```

上面代码返回`<body>`元素子节点的遍历器。

`document.createNodeIterator`方法第一个参数为所要遍历的根节点，第二个参数为所要遍历的节点类型，这里指定为元素节点（`NodeFilter.SHOW_ELEMENT`）。几种主要的节点类型写法如下。

- 所有节点：NodeFilter.SHOW_ALL
```

- 元素节点：NodeFilter.SHOW_ELEMENT
- 文本节点：NodeFilter.SHOW_TEXT
- 评论节点：NodeFilter.SHOW_COMMENT

`document.createNodeIterator`方法返回一个“遍历器”对象（`NodeFilter`实例）。该实例的`nextNode()`方法和`previousNode()`方法，可以用来遍历所有子节点。

```javascript
var nodeIterator = document.createNodeIterator(document.body);
var pars = [];
var currentNode;

while (currentNode = nodeIterator.nextNode()) {
  pars.push(currentNode);
}
```

上面代码中，使用遍历器的`nextNode`方法，将根节点的所有子节点，依次读入一个数组。`nextNode`方法先返回遍历器的内部指针所在的节点，然后会将指针移向下一个节点。所有成员遍历完成后，返回`null`。`previousNode`方法则是先将指针移向上一个节点，然后返回该节点。

```javascript
var nodeIterator = document.createNodeIterator(
  document.body,
  NodeFilter.SHOW_ELEMENT
);

var currentNode = nodeIterator.nextNode();
var previousNode = nodeIterator.previousNode();

currentNode === previousNode // true
```

上面代码中，`currentNode`和`previousNode`都指向同一个的节点。

注意，遍历器返回的第一个节点，总是根节点。

```javascript
pars[0] === document.body // true
```

document.createTreeWalker()

`document.createTreeWalker`方法返回一个 DOM 的子树遍历器。它与`document.createNodeIterator`方法基本是类似的，区别在于它返回的是`TreeWalker`实例，后者返回的是`NodeIterator`实例。另外，它的第一个节点不是根节点。

`document.createTreeWalker`方法的第一个参数是所要遍历的根节点，第二个参数指定所要遍历的节点类型（与`document.createNodeIterator`方法的第二个参数相同）。

```javascript
var treeWalker = document.createTreeWalker(
  document.body,
  NodeFilter.SHOW_ELEMENT
);

var nodeList = [];

while(treeWalker.nextNode()) {
  nodeList.push(treeWalker.currentNode);
}
```

上面代码遍历`<body>`节点下属的所有元素节点，将它们插入`nodeList`数组。

document.execCommand()，document.queryCommandSupported()，document.queryCommandEnabled()

（1）document.execCommand()

如果`document.designMode`属性设为`on`，那么整个文档用户可编辑；如果元素的`contenteditable`属性设为`true`，那么该元素可编辑。这两种情况下，可以使用`document.execCommand()`方法，改变内容的样式，比如`document.execCommand('bold')`会使得字体加粗。

```javascript
document.execCommand(command, showDefaultUI, input)
```

该方法接受三个参数。

- `command`：字符串，表示所要实施的样式。
- `showDefaultUI`：布尔值，表示是否要使用默认的用户界面，建议总是设为`false`。
- `input`：字符串，表示该样式的辅助内容，比如生成超级链接时，这个参数就是所要链接的网址。如果第二个参数设为`true`，那么浏览器会弹出提示框，要求用户在提示框输入该参数。但是，不是所有浏览器都支持这样做，为了兼容性，还是需要自己部署获取这个参数的方式。

```javascript
var url = window.prompt('请输入网址');

if (url) {
  document.execCommand('createlink', false, url);
}
```

上面代码中，先提示用户输入所要链接的网址，然后手动生成超级链接。注意，第二个参数是`false`，表示此时不需要自动弹出提示框。

`document.execCommand()`的返回值是一个布尔值。如果为`false`，表示这个方法无法生效。

这个方法大部分情况下，只对选中的内容生效。如果有多个内容可编辑区域，那么只对当前焦点所在的元素生效。

`document.execCommand()`方法可以执行的样式改变有很多种，下面是其中的一些：bold、insertLineBreak、selectAll、createLink、insertOrderedList、subscript、delete、insertUnorderedList、superscript、formatBlock、insertParagraph、undo、forwardDelete、insertText、unlink、insertImage、italic、unselect、insertHTML、redo。这些值都可以用作第一个参数，它们的含义不难从字面上看出来。

（2）document.queryCommandSupported()

`document.queryCommandSupported()`方法返回一个布尔值，表示浏览器是否支持`document.execCommand()`的某个命令。

```javascript
if (document.queryCommandSupported('SelectAll')) {
  console.log('浏览器支持选中可编辑区域的所有内容');
}
```

（3）document.queryCommandEnabled()

`document.queryCommandEnabled()`方法返回一个布尔值，表示当前是否可用`document.execCommand()`的某个命令。比如，`bold`（加粗）命令只有存在文本选中时才可用，如果没有选中文本，就不可用。

```javascript
// HTML 代码为
// <input type="button" value="Copy" onclick="doCopy()">

function doCopy(){
  // 浏览器是否支持 copy 命令（选中内容复制到剪贴板）
  if (document.queryCommandSupported('copy')) {
    copyText('你好');
  }else{
    console.log('浏览器不支持');
  }
}

function copyText(text) {
  var input = document.createElement('textarea');
```

```
  document.body.appendChild(input);
  input.value = text;
  input.focus();
  input.select();

  // 当前是否有选中文字
  if (document.queryCommandEnabled('copy')) {
    var success = document.execCommand('copy');
    input.remove();
    console.log('Copy Ok');
  } else {
    console.log('queryCommandEnabled is false');
  }
}
```

上面代码中，先判断浏览器是否支持`copy`命令（允许可编辑区域的选中内容，复制到剪贴板），如果支持，就新建一个临时文本框，里面写入内容“你好”，并将其选中。然后，判断是否选中成功，如果成功，就将“你好”复制到剪贴板，再删除那个临时文本框。

document.getSelection()

这个方法指向`window.getSelection()`，参见`window`对象一节的介绍。