Document 节点

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，或 getRootNode() 方法。
- document 对象继承了 EventTarget 接口和 Node 接口，并且混入（mixin）了 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。

```
document === window.document  //true
```

1. 属性

1.1. 快捷方式属性

以下属性是指文档内部的某个节点的快捷方式。

1.1.1 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。

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

1.1.2. 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，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。

```
document.firstChild === document.doctype  //true
typeof document.doctype  // "object"
document.doctype instanceof DocumentType  // true
document.doctype.name  // "html"
```

以上代码说明：

- document 的第一个子节点是 document.doctype。
- document.doctype 是一种对象。
- document.doctype 是 DocumentType 的实例。
- 文档类型是 HTML。

1.1.3. 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。

```
document.documentElement === document.lastChild  // true
document.documentElement === document.firstElementChild  // true
document.documentElement instanceof HTMLHtmlElement  // true
HTMLHtmlElement.__proto__ === HTMLElement  // true
```

以上代码说明：

- document 的最后一个子节点（第二个）是 HTML 节点。
- document 的第一个元素子节点是 HTML 节点。
- document.documentElement 是 HTMLHtmlElement 的实例。
- HTMLHtmlElement 的原型对象是 HTMLHtmlElement。

1.1.4. 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。

1.1.5. document.scrollingElement

```
document.scrollingElement.scrollTop = 0  // 网页将滚动到顶部
```

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>）。IE 浏览器下没有返回。

1.1.6. 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。

1.1.7. 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`。

```
document.fullscreenElement  // null
```

1.2 节点集合属性

- `document.forms` 返回所有 `<form>` 表单节点
- `document.images` 返回所有 `<img>` 图片节点
- `document.links` 返回所有 `<a>` 和 `<area>` 节点
- `document.scripts` 返回所有 `<script>` 节点
- `document.embeds` 返回所有`<embed>` 节点。
- `document.plugins` 返回所有 `<embed>`节点。
- `document.styleSheets` 属性返回文档内嵌或引入的样式表集合

```
document.embeds === document.plugins // true
```

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

```
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
```

1.3. 文档静态信息属性

1.3.1. document.documentURI 和 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。

如果文档的锚点（#anchor）变化，这两个属性都会跟着变化。

```
document.URL === document.documentURI
```

1.3.2. 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`。

```
documentURL  // http://news.baidu.com/internet
document.domain // "news.baidu.com"
```

`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`。

```
document.URL
// "http://news.baidu.com/internet"

document.domain
// "news.baidu.com"

document.domain = "baidu.com"
// "baidu.com"

document.domain
// "baidu.com"
```

1.3.3. 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。

```
window.location === document.location  // true
window.location === location  // true
document.location === location  // true
```

1.3.4. 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。

```
document.title  // Document
document.title = "DOM-Document"; // "DOM-Document"
document.title  // "DOM-Document"
```

该标签页的标题会显示修改后的标题。

1.3.5. document.characterSet

`characterSet` 属性返回当前文档的编码，比如： `UTF-8`。

```
document.characterSet  // "UTF-8"
```

1.3.6. 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。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。

1.3.7 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。

```
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <base href="https://www.baidu.com" />
    <title>Document</title>
  </head>
  <body>
    <a href="https://www.baidu.com" title="baidu" name="baidu">百度</a>
  </body>
</html>
```

对以上 HTML 文档返回文档处理模式，将返回 BackCompat。它没有设置文档类型。

```
document.compatMode  // "BackCompat"
document.doctype  // null
```

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <base href="https://www.baidu.com" />
    <title>Document</title>
  </head>
```

```
  <body>
    <a href="https://www.baidu.com" title="baidu" name="baidu">百度</a>
  </body>
</html>
```

对以上 HTML 文档返回文档处理模式，将返回 `CSS1Compat`。它的文档类型是 `<!DOCTYPE html>`。

```
document.compatMode  // "CSS1Compat"
document.doctype  // <!DOCTYPE html>
```

文档类型可小写 `<!doctype html>`

1.4. 文档状态属性

1.4.1. 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。

1.4.2. 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。

1.4.3. 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:

> - 浏览器开始解析 HTML 文档，`document.readyState` 属性等于 `loading`。
> - 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`。
> - HTML 文档解析完成，`document.readyState` 属性变成 `interactive`。

> - 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。

```
// 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。

1.5. document.cookie

`document.cookie` 属性用来操作浏览器 Cookie。

1.6. 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。

```
// <iframe id="editor" src="about:blank"></iframe>
var editor = document.getElementById('editor');
editor.contentDocument.designMode = 'on';
```

1.7 document.currentScript

`document.currentScript` 属性只用在 `<script>` 元素的内嵌脚本或加载的外部脚本之中，返回当前脚本所在的那个 `DOM` 节点，即 `<script>` 元素的 `DOM` 节点。

```
<script id="foo">
  console.log( document.currentScript === document.getElementById('foo')); // true
</script>
```

1.8. 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。

```
let doc = document.implementation.createHTMLDocument('Title');
let 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`。***此方法需要谨慎使用***

2. 方法

2.1. document.open() 和 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。

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

`document.open()`将会清除当前文档所有内容，应该谨慎使用

2.2. document.write() 和 document.writeln()

`document.write` 方法用于向当前文档写入内容，它会先调用open方法，擦除当前文档所有内容，然后再写入。

`document.write()`将会清除当前文档所有内容，应该谨慎使用

2.3. document.querySelector() 和 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`。

```
let el1 = document.querySelector('.className');
let el1 = document.querySelector('#idName');
let 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。

```
document.querySelectorAll("div") instanceof NodeList // true
document.querySelector("div") instanceof HTMLDivElement  // true
```

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。

```
let 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。

```
// 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（包括子级元素、所有后代元素）。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。

2.4. 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。

HTML 标签名是大小写不敏感的，因此 `getElementsByTagName()` 方法的参数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。

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。

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

2.5. 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。

参数可以是多个class，它们之间使用空格分隔。

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

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。

2.6. 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。

```
// <img src="http://ip:port/logo.png" alt="logo" id="logo" name="dom" />
// <a href="https://www.baidu.com" title="baidu" name="dom">百度</a>

let dom = document.getElementsByName("dom");
console.log( dom);

// NodeList(2) [img#logo, a]
```

2.7. 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`。

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

`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。

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

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。

2.8. document.elementFromPoint() 和 document.elementsFromPoint()

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

```
let element = document.elementFromPoint(50, 50);
// <code>document.getElementById()</code>
```

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。

```
let elements = document.elementsFromPoint(50, 50);
// (9) [code, p, article, ..., section]
```

2.9. document.createElement()

`document.createElement()` 方法用来创建元素节点，并返回该节点。

```
let 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。

```
document.createElement("<div>")
// Uncaught DOMException: Failed to execute 'createElement' on 'Document': The tag
name provided ('<div>') is not a valid name.
// 未捕获的 DOM 异常，未能执行在 `document` 上 `createElement`，提供的 `div` 标签名不
是一个有效的名称。
```

2.10. 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。

```
let content = document.createTextNode("hello world");
content instanceof Text  // true
```

2.11. document.createAttribute()

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

```
let attr = document.createAttribute("data-attr");

attr instanceof Attr  // true
```

```
let node = document.getElementById('div1');

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

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

2.12. document.createComment()

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

```
let 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。

```
let footer = document.getElementsByClassName("footer")[0];
let comment = document.createComment("comment");
footer.after(comment);
```

上面代码在 `footer` 元素后面插入了一段注释：

2.13. document.createDocumentFragment()

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

```
let 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。

```
let docFrag = document.createDocumentFragment();

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

let 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。

2.14. 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。

2.15. 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。

```
// 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);
```

2.16. 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。

```
let focused = document.hasFocus();
```

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。