

Element 节点

`Element` 节点对象对应网页的 HTML 元素。每一个 HTML 元素，在 DOM 树上都会转化成一个 `Element` 节点对象（以下简称元素节点）。

元素节点的 `nodeType` 属性都是 1。

```
let p = document.querySelector('p');
p.nodeName // "P"
p.nodeType // 1
```

`Element` 对象继承了 `Node` 接口，因此 `Node` 的属性和方法在 `Element` 对象都存在。

此外，不同的 HTML 元素对应的元素节点是不一样的，浏览器使用不同的构造函数，生成不同的元素节点，比如 `<a>` 元素的构造函数是 `HTMLAnchorElement()`，`<button>` 是 `HTMLButtonElement()`。因此，元素节点不是一种对象，而是许多种对象，这些对象除了继承 `Element` 对象的属性和方法，还有各自独有的属性和方法。

1. 实例属性

1.1. 元素特征相关属性

1.1.1. `Element.id`

`Element.id` 属性返回元素的 `id` 属性，该属性可读写。

```
// 
let logo = document.getElementById("logo");
logo.id // "logo"
```

`id` 属性的值是大小写敏感，即浏览器能正确识别 `<p id="foo">` 和 `<p id="F00">` 这两个元素的 `id` 属性，但是最好不要这样命名，统一使用小写。

1.1.2. `Element.tagName`

`Element.tagName` 属性返回指定元素的大写标签名，与 `nodeName` 属性的值相等。

```
// 
let logo = document.getElementById("logo");
console.log(logo.tagName); // IMG
console.log(logo.nodeName); // IMG
```

1.1.3. `Element.accesskey`

`Element.accesskey` 属性用于读写分配给当前元素的快捷键。

```
// <a href="https://www.baidu.com" accesskey="h" id="dom" title="baidu"
name="dom">百度</a>
let anchor = document.getElementById("dom");
anchor.accesskey // "h"
```

上面代码给 `<a>` 标签定义了快捷键 `h`，当使用快捷键时，可代替点击跳转到 <https://www.baidu.com> 网页。在 window 系统中是使用 `alt + h` 快捷键。

1.1.4. Element.draggable

`Element.draggable` 属性返回一个布尔值，表示当前元素是否可拖动。该属性可读写。

1.1.5. Element.lang

`Element.lang` 属性返回当前元素的语言设置。该属性可读写。

```
// <html lang="zh-CN">
document.documentElement.lang // "zh-CN"

// <html lang="en">
document.documentElement.lang // "en"
```

1.1.6. Element.tabIndex

`Element.tabIndex` 属性返回一个整数，表示当前元素在 Tab 键遍历时的顺序。该属性可读写。

`tabIndex` 属性值如果是负值（通常是-1），则 Tab 键不会遍历到该元素。如果是正整数，则按照顺序，从小到大遍历。如果两个元素的 `tabIndex` 属性的正整数值相同，则按照出现的顺序遍历。遍历完所有 `tabIndex` 为正整数的元素以后，再遍历所有 `tabIndex` 等于0、或者属性值是非法值、或者没有 `tabIndex` 属性的元素，顺序为它们在网页中出现的顺序。

1.1.7. Element.title

`Element.title` 属性用来读写当前元素的 HTML 属性 `title`。该属性通常用来指定，鼠标悬浮时弹出的文字提示框。

```
let footer = document.getElementsByClassName("footer")[0];
footer.title // ""

footer.title = "footer"
```

上面代码获取 `<footer>` 元素的 `title` 属性时，是空，表示没有 `title` 属性或者 `title` 属性为空。然后给它设置了 `title` 属性，当鼠标移动到该元素上悬浮时将显示 `footer` 提示。

```
<!DOCTYPE html>
<html lang="zh-CN" prefix="og: http://ogp.me/ns#">
  <head>...</head>
  <body>
    <nav class="navbar is-light" role="navigation" id="navbar" aria-label="main navigation">...</nav>
    <section class="section main article" style="min-height: 717px;">...
  </section> == $0
    <footer class="footer is-size-5-widescreen" title="footer">...
  </footer>
```

1.2. 元素状态相关属性

1.2.1. Element.hidden

`Element.hidden` 属性返回一个布尔值，表示当前元素的 `hidden` 属性，用来控制当前元素是否可见。该属性可读写。

```
let btn = document.getElementById('btn');
let myDiv = document.getElementById('myDiv');

btn.addEventListener('click', function () {
  myDiv.hidden = !myDiv.hidden;
}, false);
```

注意，该属性与 CSS 设置是互相独立的。CSS 对这个元素可见性的设置，`Element.hidden` 并不能反映出来。也就是说，**这个属性并不能用来判断当前元素的实际可见性。**

CSS 的设置高于 `Element.hidden`。如果 CSS 指定了该元素不可见 (`display: none`) 或可见 (`display: hidden`)，那么 `Element.hidden` 并不能改变该元素实际的可见性。换言之，**这个属性只在 CSS 没有明确设定当前元素的可见性时才有效。**

1.2.2. Element.contentEditable 和 Element.isContentEditable

HTML 元素可以设置 `contentEditable` 属性，使得元素的内容可以编辑。

```
<div contenteditable>123</div>
```

上面代码中，`<div>` 元素有 `contenteditable` 属性，因此用户可以在网页上编辑这个区块的内容。

`Element.contentEditable` 属性返回一个字符串，表示是否设置了 `contenteditable` 属性，有三种可能的值。该属性可写。

- `"true"`：元素内容可编辑
- `"false"`：元素内容不可编辑
- `"inherit"`：元素是否可编辑，继承了父元素的设置

`Element.isContentEditable` 属性返回一个布尔值，同样表示是否设置了 `contenteditable` 属性。该属性只读。

```
<article contenteditable="true">
  <p>
    4月24日，由四川省成都市人民政府主办，成都市体育局、都江堰市人民政府承办的“2021成都双
    遗马拉松”在都江堰鸣枪起跑。
  </p>
  <p>
    据悉，赛事自2015年首届举办以来，吸引了广大运动员和民众参与。比赛设置了全马（42.195公
    里）、半马（21.0975公里）、欢乐跑（5公里）三个项目，今年参赛运动员达2万余人，其中参加全马
    的有6000人。
  </p>
</article>
```

上面代码设置了元素的 `contenteditable` 属性，该元素可编辑。

4月24日，由四川省成都市人民政府主办，成都市体育局、都江堰市人民政府承办的“2021成都双遗马拉松”在都江堰鸣枪起跑。

据悉，赛事自2015年首届举办以来，吸引了广大运动员和民众参与。比赛设置了全马（42.195公里）、半马（21.0975公里）、欢乐跑（5公里）三个项目，今年参赛运动员达2万余人，其中参加全马的有6000人。

点击 `article` 元素区域，可以编辑该元素的内容。

```
let article = document.getElementsByTagName("article")[0];
article.contentEditable // "true" 获取 contentEditable 值，是 "true"

article.contentEditable = "false" // "false"
article.isContentEditable // false
```

上面代码：

1. 先获取元素的 `contentEditable` 属性，这时候返回 `"true"`，这时候元素可编辑
2. 将元素的编辑属性设置为 `"false"`，这时候元素不可编辑。
3. 获取元素的 `isContentEditable` 属性，返回 `false`。

1.2.3. Element.attributes

`Element.attributes` 属性返回该元素的所有属性节点。

```
// html
// <nav class="navbar" role="navigation" id="navbar" aria-label="main avigation">
// </nav>

let navbarLikeArray = document.getElementById("navbar");
```

```
navbarLikeArray.attributes
// NamedNodeMap {0: class, 1: role, 2: id, 3: aria-label, length: 4}
```

`Element.attribute` 属性返回类数组对象，所谓类数组对象是指本身属于对象，但包含 `length` 属性以及 `0, 1` 这样的自然数属性。

与它相似的还有 `Element.getAttributeNames()` 方法，这个方法返回包含全部属性名的数组。

```
navbarLikeArray.getAttributeNames()
// (4) ["class", "role", "id", "aria-label"]
```

1.2.4. Element.className 和 Element.classList

`className` 属性用来读写当前元素节点的 `class` 属性。它的值是一个字符串，每个 `class` 之间用空格分割。

`classList` 属性返回一个类似数组的对象，当前元素节点的每个 `class` 就是这个对象的一个成员。

```
// HTML 代码 <div class="one two three" id="myDiv"></div>
var div = document.getElementById('myDiv');

div.className
// "one two three"

div.classList
// {
//   0: "one"
//   1: "two"
//   2: "three"
//   length: 3
// }
```

上面代码中，`className` 属性返回一个空格分隔的字符串，而 `classList` 属性指向一个类似数组的对象，该对象的 `length` 属性（只读）返回当前元素的 `class` 数量。

`classList` 对象有下列方法。

- `add()`：增加一个 `class`。
- `remove()`：移除一个 `class`。
- `contains()`：检查当前元素是否包含某个 `class`。
- `toggle()`：将某个 `class` 移入或移出当前元素。
- `item()`：返回指定索引位置的 `class`。
- `toString()`：将 `class` 的列表转为字符串。

```
var div = document.getElementById('myDiv');

div.classList.add('myCssClass');
```

```
div.classList.add('foo', 'bar');
div.classList.remove('myCssClass');
div.classList.toggle('myCssClass'); // 如果 myCssClass 不存在就加入, 否则移除
div.classList.contains('myCssClass'); // 返回 true 或者 false
div.classList.item(0); // 返回第一个 Class
div.classList.toString();
```

下面比较一下, `className` 和 `classList` 在添加和删除某个 `class` 时的写法。

```
var foo = document.getElementById('foo');

// 添加class
foo.className += 'bold';
foo.classList.add('bold');

// 删除class
foo.classList.remove('bold');
foo.className = foo.className.replace(/^bold$/, '');
```

`toggle()` 方法可以接受一个布尔值, 作为第二个参数。如果为 `true`, 则添加该属性; 如果为 `false`, 则去除该属性。

```
el.classList.toggle('abc', boolValue);

// 等同于
if (boolValue) {
  el.classList.add('abc');
} else {
  el.classList.remove('abc');
}
```

1.2.5. Element.dataset

网页元素可以自定义 `data-` 属性, 用来添加数据。

```
<div data-label="lableName"></div>
```

上面代码中, `<div>` 元素有一个自定义的 `data-label` 属性, 用来为该元素添加一个标签。

`Element.dataset` 属性返回一个对象, 可以从这个对象读写 `data-` 属性。

```
// <article
//   id="foo"
//   data-columns="3"
//   data-index-number="12314"
```

```
// data-parent="cars">
// ...
// </article>
var article = document.getElementById('foo');
article.dataset.columns // "3"
article.dataset.indexNumber // "12314"
article.dataset.parent // "cars"
```

dataset 上面的各个属性返回都是字符串。

HTML 代码中, **data-** 属性的属性名, 只能包含英文字母、数字、连词线 (-)、点 (.)、冒号 (:) 和下划线 (_)。它们转成 JavaScript 对应的 **dataset** 属性名, 规则如下。

- 开头的 **data-** 会省略。
- 如果连词线后面跟了一个英文字母, 那么连词线会取消, 该字母变成大写。
- 其他字符不变。

因此, **data-abc-def** 对应 **dataset.abcDef**, **data-abc-1** 对应 **dataset["abc-1"]**。

除了使用 **dataset** 读写 **data-** 属性, 也可以使用 **Element.getAttribute()** 和 **Element.setAttribute()**, 通过完整的属性名读写这些属性。

```
<article data-from="baiduNews">
  <p>百年前, 贯通燕山山脉通往北京西北方向的京张铁路是中国人自主设计、施工修建的第一条干线铁路。</p>
  <p>京张铁路的建成不仅是中国铁路的骄傲, 更极大地振奋了中国人民的民族自信。</p>
</article>
```

```
let article2 = document.getElementsByTagName("article")[0];

article2.dataset.from
// "baiduNews"
article2.getAttribute("data-from")
// "baiduNews"
```

1.2.6. Element.innerHTML

Element.innerHTML 属性返回一个字符串, 等同于该元素包含的所有 HTML 代码。该属性可读写, 常用来设置某个节点的内容。它能改写所有元素节点的内容, 包括 **<HTML>** 和 **<body>** 元素。

如果将 **innerHTML** 属性设为空, 等于删除所有它包含的所有节点。

```
el.innerHTML = '';
```

上面代码等于将 **el** 节点变成了一个空节点, **el** 原来包含的节点被全部删除。

注意，读取属性值的时候，如果文本节点包含 &、小于号 (<) 和大于号 (>)，`innerHTML` 属性会将它们转为实体形式 `&`、`<`、`>`。如果想得到原文，建议使用 `element.textContent` 属性。

```
// HTML代码如下 <p id="para"> 5 > 3 </p>
document.getElementById('para').innerHTML
// 5 &gt; 3
```

写入的时候，如果插入的文本包含 HTML 标签，会被解析成为节点对象插入 DOM。注意，如果文本之中含有 `<script>` 标签，虽然可以生成 `script` 节点，但是插入的代码不会执行。

```
var name = "<script>alert('script')</script>";
el.innerHTML = name;
```

上面代码将脚本插入内容，脚本并不会执行。但是，`innerHTML` 还是有安全风险的。

```
var name = "<img src=x onerror=alert(1)>";
el.innerHTML = name;
```

上面代码中，`alert` 方法是会执行的。会弹出一个警告，且必须手动确定关闭。因此为了安全考虑，如果插入的是文本，最好用 `textContent` 属性代替 `innerHTML`。

1.2.7. Element.outerHTML

`Element.outerHTML` 属性返回一个字符串，表示当前元素节点的所有 HTML 代码，**包括该元素本身和所有子元素**。

```
// HTML 代码如下
// <div id="d"><p>Hello</p></div>
var d = document.getElementById('d');
d.outerHTML
// '<div id="d"><p>Hello</p></div>'
```

`outerHTML` 属性是可读写的，对它进行赋值，等于替换掉当前元素。

```
// HTML 代码如下
// <div id="container"><div id="d">Hello</div></div>
var container = document.getElementById('container');
var d = document.getElementById('d');
container.firstChild.nodeName // "DIV"
d.nodeName // "DIV"

d.outerHTML = '<p>Hello</p>';
```



```
container.firstChild.nodeName // "P"
d.nodeName // "DIV"
```

上面代码中，变量 `d` 代表子节点，它的 `outerHTML` 属性重新赋值以后，内层的 `div` 元素就不存在了，被 `p` 元素替换了。但是，变量 `d` 依然指向原来的 `div` 元素，这表示 **被替换的 `DIV` 元素还存在于内存中**。

注意，如果一个节点没有父节点，设置 `outerHTML` 属性会报错。

```
var div = document.createElement('div');
div.outerHTML = '<p>test</p>';
// DOMException: This element has no parent node.
```

上面代码中，`div` 元素没有父节点，设置 `outerHTML` 属性会报错。

1.2.8. Element.clientHeight 和 Element.clientWidth

`Element.clientHeight` 属性返回一个整数值，表示元素节点的 CSS 高度（单位像素），只对块级元素生效，**对于行内元素返回 0**。如果块级元素没有设置 CSS 高度，则返回实际高度。

除了元素本身的高度，它还包括 `padding` 部分，但是不包括 `border`、`margin`。如果有水平滚动条，还要减去水平滚动条的高度。注意，这个值始终是整数，如果是小数会被四舍五入。

`Element.clientWidth` 属性返回元素节点的 CSS 宽度，同样只对块级元素有效，也是只包括元素本身的宽度和 `padding`，如果有垂直滚动条，还要减去垂直滚动条的宽度。

`document.documentElement` 的 `clientHeight` 属性，返回当前视口的高度（即浏览器窗口的高度），等同于 `window.innerHeight` 属性减去水平滚动条的高度（如果有的话）。`document.body` 的高度则是网页的实际高度（全部内容所占的高度）。一般来说，`document.body.clientHeight` 大于 `document.documentElement.clientHeight`。

```
// 视口高度
document.documentElement.clientHeight

// 网页总高度
document.body.clientHeight
```

1.2.9. Element.clientLeft 和 Element.clientTop

`Element.clientLeft` 属性等于元素节点左边框（`left border`）的宽度（单位像素），不包括左侧的 `padding` 和 `margin`。如果没有设置左边框，或者是行内元素（`display: inline`），该属性返回 0。该属性总是返回整数值，如果是小数，会四舍五入。

`Element.clientTop` 属性等于网页元素顶部边框的宽度（单位像素），其他特点都与 `clientLeft` 相同。

```
<style>
  article {
```

```

    border: 5px solid #ccc;
    width: 400px;
    height: 140px;
    padding: 10px;
    margin: 10px;
    box-sizing: content-box;
  }
</style>

<article data-from="baiduNews">
  <p>
    百年前，贯通燕山山脉通往北京西北方向的京张铁路是中国人自主设计、施工修建的第一条干线
    铁路。
  </p>
  <p>
    京张铁路的建成不仅是中国铁路的骄傲，更极大地振奋了中国人民的民族自信。
  </p>
</article>

<script>
  let article = document.getElementsByTagName("article")[0];
  console.log(article.clientLeft); // 5
  console.log(article.clientTop); // 5
  console.log(article.clientWidth); // 420
  console.log(article.clientHeight); // 160
</script>

```

1.2.10. Element.scrollHeight 和 Element.scrollWidth

`Element.scrollHeight` 属性返回一个整数值（小数会四舍五入），表示当前元素的总高度（单位像素），包括溢出容器、当前不可见的部分。它包括 `padding`，但是不包括 `border`、`margin` 以及水平滚动条的高度（如果有水平滚动条的话），还包括伪元素（`::before` 或 `::after`）的高度。

`Element.scrollWidth` 属性表示当前元素的总宽度（单位像素），其他地方都与 `scrollHeight` 属性类似。这两个属性只读。

整张网页的总高度可以从 `document.documentElement` 或 `document.body` 上读取。

```

// 返回网页的总高度
document.documentElement.scrollHeight
document.body.scrollHeight

```

如果元素节点的内容出现溢出，即使溢出的内容是隐藏的，`scrollHeight` 属性仍然返回元素的总高度。

```

// HTML 代码如下
// <div id="myDiv" style="height: 200px; overflow: hidden;">...</div>
document.getElementById('myDiv').scrollHeight // 356

```

上面代码中，即使 `myDiv` 元素的 CSS 高度只有200像素，且溢出部分不可见，但是 `scrollHeight` 仍然会返回该元素的原始高度。

1.2.11. Element.scrollLeft 和 Element.scrollTop

`Element.scrollLeft` 属性表示当前元素的水平滚动条向右侧滚动的像素数量，`Element.scrollTop` 属性表示当前元素的垂直滚动条向下滚动的像素数量。对于那些没有滚动条的网页元素，这两个属性总是等于 0。

如果要查看整张网页的水平的和垂直的滚动距离，要从 `document.documentElement` 元素上读取。

```
document.documentElement.scrollLeft
document.documentElement.scrollTop
```

这两个属性都可读写，设置该属性的值，会导致浏览器将当前元素自动滚动到相应的位置。

1.2.12. Element.offsetParent

`Element.offsetParent` 属性返回最靠近当前元素的、并且 CSS 的 `position` 属性不等于 `static` 的上层元素。

```
<div style="position: absolute;">
  <p>
    <span>Hello</span>
  </p>
</div>
```

上面代码中，`span` 元素的 `offsetParent` 属性就是 `div` 元素。

该属性主要用于确定子元素位置偏移的计算基准，`Element.offsetTop` 和 `Element.offsetLeft` 就是 `offsetParent` 元素计算的。

如果该元素是不可见的（`display` 属性为 `none`），或者位置是固定的（`position` 属性为 `fixed`），则 `offsetParent` 属性返回 `null`。

```
<div style="position: absolute;">
  <p>
    <span style="display: none;">Hello</span>
  </p>
</div>
```

上面代码中，`span` 元素的 `offsetParent` 属性是 `null`。

如果某个元素的所有上层节点的 `position` 属性都是 `static`，则 `Element.offsetParent` 属性指向 `<body>` 元素。

1.2.13. Element.offsetHeight 和 Element.offsetWidth

`Element.offsetHeight` 属性返回一个整数，表示元素的 CSS 垂直高度（单位像素），包括元素本身的高度、padding 和 border，以及水平滚动条的高度（如果存在滚动条）。

`Element.offsetWidth` 属性表示元素的 CSS 水平宽度（单位像素），其他都与 `Element.offsetHeight` 一致。

这两个属性都是只读属性，**只比 `Element.clientHeight` 和 `Element.clientWidth` 多了边框的高度或宽度**。如果元素的 CSS 设为不可见（比如 `display: none;`），则返回 0。

1.2.14. `Element.offsetLeft` 和 `Element.offsetTop`

`Element.offsetLeft` 返回当前元素左上角相对于 `Element.offsetParent` 节点的水平位移，`Element.offsetTop` 返回垂直位移，单位为像素。通常，这两个值是指相对于父节点的位移。

```
<style>
  article {
    border: 5px solid #ccc;
    width: 300px;
    height: 100px;
    padding: 10px;
    margin: 0 10px 0 5px;
    box-sizing: content-box;
    overflow-y: hidden;
  }
</style>

<article data-from="baiduNews">
  <p>
    百年前，贯通燕山山脉通往北京西北方向的京张铁路是中国人自主设计、施工修建的第一条干线铁路。
  </p>
  <p>
    京张铁路的建成不仅是中国铁路的骄傲，更极大地振奋了中国人民的民族自信。
  </p>
</article>

<script>
  let article = document.getElementsByTagName("article")[0];
  console.log(article.clientLeft); // 5
  console.log(article.clientTop); // 5
  console.log(article.clientWidth); // 320
  console.log(article.clientHeight); // 120
  console.log(article.offsetParent); // <body>...</body>
  console.log(article.offsetLeft); // 13
  console.log(article.offsetTop); // 8
  console.log(article.offsetHeight); // 130
  console.log(article.offsetWidth); // 330
</script>
```

上面代码，`offsetLeft` 是 13，是因为 `<body>` 元素默认有 `margin: 8px`。

下面的代码可以算出元素左上角相对于整张网页的坐标。

```
function getElementPosition(e) {
    var x = 0;
    var y = 0;
    while (e !== null) {
        x += e.offsetLeft;
        y += e.offsetTop;
        e = e.offsetParent;
    }
    return {x: x, y: y};
}
```

1.2.15. Element.children 和 Element.childElementCount

`Element.children` 属性返回一个类似数组的对象（`HTMLCollection` 实例），包括当前元素节点的所有子元素。如果当前元素没有子元素，则返回的对象包含零个成员。

```
if (para.children.length) {
    var children = para.children;
    for (var i = 0; i < children.length; i++) {
        // ...
    }
}
```

上面代码遍历了 `para` 元素的所有子元素。

这个属性与 `Node.childNodes` 属性的区别是，它只包括元素类型的子节点，不包括其他类型的子节点。

`Element.childElementCount` 属性返回当前元素节点包含的子元素节点的个数，与 `Element.children.length` 的值相同。

1.2.16. Element.firstChild 和 Element.lastElementChild

`Element.firstChild` 属性返回当前元素的第一个元素子节点，`Element.lastElementChild` 返回最后一个元素子节点。

如果没有元素子节点，这两个属性返回 `null`。

1.2.17. Element.nextElementSibling 和 Element.previousElementSibling

`Element.nextElementSibling` 属性返回当前元素节点的后一个同级元素节点，如果没有则返回 `null`。

```
<div id="div-01">Here is div-01</div>
<!-- comment -->
<div id="div-02">Here is div-02</div>
<script>
```

```
var el = document.getElementById('div-01');
el.nextElementSibling
// <div id="div-02">Here is div-02</div>
el.nextSibling
// #text   textContent: " "   下一个节点是文本节点（空格字符），因为与下一行有换行空格
</script>
```

`Element.previousElementSibling` 属性返回当前元素节点的前一个同级元素节点，如果没有则返回 `null`。

2. 实例方法

2.1. 属性相关方法

元素节点提供六个方法，用来操作属性。

- `getAttribute()`：读取某个属性的值
- `getAttributeNames()`：返回当前元素的所有属性名
- `setAttribute()`：写入属性值
- `hasAttribute()`：某个属性是否存在
- `hasAttributes()`：当前元素是否有属性
- `removeAttribute()`：删除属性

将在《属性的操作》中讲述。

2.2. 选择元素方法

有很多和 `document` 节点中讲到的一样的方法。

- `Element.querySelector()` 接受 CSS 选择器作为参数，返回父元素的第一个匹配的子元素。
- `Element.querySelectorAll()` 接受 CSS 选择器作为参数，返回一个 `NodeList` 实例，包含所有匹配的子元素。
- `Element.getElementsByClassName()` 方法返回一个 `HTMLCollection` 实例，成员是当前元素节点的所有具有指定 `class` 的子元素节点。
- `Element.getElementsByTagName()` 方法返回一个 `HTMLCollection` 实例，成员是当前节点的所有匹配指定标签名的子元素节点。

2.3. Element.closest()

`Element.closest()` 方法接受一个 CSS 选择器作为参数，返回匹配该选择器的、最接近当前节点的一个祖先节点（包括当前节点本身）。如果没有任何节点匹配 CSS 选择器，则返回 `null`。

```
// HTML 代码如下
// <article>
//   <div id="div-01">Here is div-01
//     <div id="div-02">Here is div-02
//       <div id="div-03">Here is div-03</div>
//     </div>
//   </div>
// </article>
```

```
var div03 = document.getElementById('div-03');

// div-03 最近的祖先节点
div03.closest("#div-02") // div-02
div03.closest("div div") // div-03
div03.closest("article > div") //div-01
div03.closest(":not(div)") // article
```

上面代码中，由于 `closest` 方法将当前节点也考虑在内，所以第二个 `closest` 方法返回 `div-03`。

2.4. Element.matches()

`Element.matches()` 方法返回一个布尔值，表示当前元素是否匹配给定的 CSS 选择器。

```
if (el.matches('.someClass')) {
  console.log('Match!');
}
```

2.5. 事件相关方法

以下三个方法与 `Element` 节点的事件相关。这些方法都继承自 `EventTarget` 接口，详见相关章节。

- `Element.addEventListener()`：添加事件的回调函数
- `Element.removeEventListener()`：移除事件监听函数
- `Element.dispatchEvent()`：触发事件

```
element.addEventListener('click', listener, false);
element.removeEventListener('click', listener, false);

var event = new Event('click');
element.dispatchEvent(event);
```

2.6. Element.scrollToView()

`Element.scrollToView()` 方法滚动当前元素，进入浏览器的可见区域，类似于设置 `window.location.hash` 的效果。

```
el.scrollToView(); // 等同于el.scrollToView(true)
el.scrollToView(false);
```

该方法可以接受一个布尔值作为参数。如果为 `true`，表示元素的顶部与当前区域的可见部分的顶部对齐（前提是当前区域可滚动）；如果为 `false`，表示元素的底部与当前区域的可见部分的尾部对齐（前提是当前区域可滚动）。如果没有提供该参数，默认为 `true`。

2.7. Element.scrollToView()

`Element.scrollToView()` 方法滚动当前元素，进入浏览器的可见区域，类似于设置 `window.location.hash` 的效果。通常，当一个元素先不在可视区域内时，具有滚动到可视区域的动画效果。

```
el.scrollToView(); // 等同于el.scrollToView(true)
el.scrollToView(false);
```

该方法可以接受一个布尔值作为参数。如果为 `true`，表示元素的顶部与当前区域的可见部分的顶部对齐（前提是当前区域可滚动）；如果为 `false`，表示元素的底部与当前区域的可见部分的尾部对齐（前提是当前区域可滚动）。如果没有提供该参数，默认为 `true`。

2.8. Element.getBoundingClientRect()

`Element.getBoundingClientRect()` 方法返回一个对象，提供当前元素节点的大小、位置等信息，基本上就是 CSS 盒状模型的所有信息。

```
var rect = obj.getBoundingClientRect();
```

上面代码中，`getBoundingClientRect()` 方法返回的 `rect` 对象，具有以下属性（全部为只读）。

- `x`：元素左上角相对于视口的横坐标
- `y`：元素左上角相对于视口的纵坐标
- `height`：元素高度
- `width`：元素宽度
- `left`：元素左上角相对于视口的横坐标，与 `x` 属性相等
- `right`：元素右边界相对于视口的横坐标（等于 `x + width`）
- `top`：元素顶部相对于视口的纵坐标，与 `y` 属性相等
- `bottom`：元素底部相对于视口的纵坐标（等于 `y + height`）

由于元素相对于视口（`viewport`）的位置，会随着页面滚动变化，因此表示位置的四个属性值，都不是固定不变的。如果想得到绝对位置，可以将 `left` 属性加上 `window.scrollX`，`top` 属性加上 `window.scrollY`。

注意，`getBoundingClientRect()` 方法的所有属性，都把边框（`border` 属性）算作元素的一部分。也就是说，都是从边框外缘的各个点来计算。因此，**`width` 和 `height` 包括了元素本身 + `padding` + `border`**。

另外，上面的这些属性，都是继承自原型的属性，`Object.keys` 会返回一个空数组，这一点也需要注意。

```
var rect = document.body.getBoundingClientRect();
Object.keys(rect) // []
```

上面代码中，`rect` 对象没有自身属性，而 `Object.keys()` 方法只返回对象自身的属性，所以返回了一个空数组。

2.9. Element.getClientRects()

`Element.getClientRects()` 方法返回一个类似数组的对象，里面是当前元素在页面上形成的所有矩形（所以方法名中的 `Rect` 用的是复数）。每个矩形都有 `bottom`、`height`、`left`、`right`、`top` 和 `width` 六个属性，表示它们相对于视口的四个坐标，以及本身的高度和宽度。

对于盒状元素（比如 `<div>` 和 `<p>`），该方法返回的对象中只有该元素一个成员。对于行内元素（比如 ``、`<a>`、``），该方法返回的对象有多少个成员，取决于该元素在页面上占据多少行。这是它和 `Element.getBoundingClientRect()` 方法的主要区别，后者对于行内元素总是返回一个矩形。

```
<span id="inline">Hello World Hello World Hello World</span>
```

上面代码是一个行内元素 ``，如果它在页面上占据三行，`getClientRects()` 方法返回的对象就有三个成员，如果它在页面上占据一行，`getClientRects()` 方法返回的对象就只有一个成员。

```
var el = document.getElementById('inline');
el.getClientRects().length // 3
el.getClientRects()[0].left // 8
el.getClientRects()[0].right // 113.908203125
el.getClientRects()[0].bottom // 31.200000762939453
el.getClientRects()[0].height // 23.200000762939453
el.getClientRects()[0].width // 105.908203125
```

这个方法主要用于判断行内元素是否换行，以及行内元素的每一行的位置偏移。

注意，如果行内元素包括换行符，那么该方法会把换行符考虑在内。

```
<span id="inline">
  Hello World
  Hello World
  Hello World
</span>
```

上面代码中，`` 节点内部有三个换行符，即使 HTML 语言忽略换行符，将它们显示为一行，`getClientRects()` 方法依然会返回三个成员。如果行宽设置得特别窄，上面的 `` 元素显示为 6 行，那么就会返回六个成员。

2.10. Element.insertAdjacentElement()

`Element.insertAdjacentElement()` 方法在相对于当前元素的指定位置，插入一个新的节点。该方法返回被插入的节点，如果插入失败，返回 `null`。

```
element.insertAdjacentElement(position, element);
```

`Element.insertAdjacentElement()` 方法一共可以接受两个参数，第一个参数是一个字符串，表示插入的位置，第二个参数是要插入的节点。第一个参数只可以取如下的值。

- `beforebegin`: 当前元素之前
- `beforeend`: 当前元素内部的最后一个子节点后面
- `afterbegin`: 当前元素内部的第一个子节点前面
- `afterend`: 当前元素之后

注意, `beforebegin` 和 `afterend` 这两个值, 只在当前节点有父节点时才会生效。如果当前节点是由脚本创建的, 没有父节点, 那么插入会失败。

```
var p1 = document.createElement('p')
var p2 = document.createElement('p')
p1.insertAdjacentElement('afterend', p2) // null
```

上面代码中, `p1` 没有父节点, 所以插入 `p2` 到它后面就失败了。

如果插入的节点是一个文档里现有的节点, 它会从原有位置删除, 放置到新的位置。

2.11. `Element.insertAdjacentHTML()` 和 `Element.insertAdjacentText()`

`Element.insertAdjacentHTML()` 方法用于将一个 HTML 字符串, 解析生成 DOM 结构, 插入相对于当前节点的指定位置。

```
element.insertAdjacentHTML(position, text);
```

该方法接受两个参数, 第一个是一个表示指定位置的字符串, 第二个是待解析的 HTML 字符串。第一个参数只能设置下面四个值之一。

- `beforebegin`: 当前元素之前
- `beforeend`: 当前元素内部的最后一个子节点后面
- `afterbegin`: 当前元素内部的第一个子节点前面
- `afterend`: 当前元素之后

```
// HTML 代码: <div id="one">one</div>
var d1 = document.getElementById('one');
d1.insertAdjacentHTML('afterend', '<div id="two">two</div>');
// 执行后的 HTML 代码:
// <div id="one">one</div><div id="two">two</div>
```

该方法只是在现有的 DOM 结构里面插入节点, 这使得它的执行速度比 `innerHTML` 方法快得多。

注意, 该方法不会转义 HTML 字符串, 这导致它不能用来插入用户输入的内容, 否则会有安全风险。

`Element.insertAdjacentText()` 方法在相对于当前节点的指定位置, 插入一个文本节点, 用法与 `Element.insertAdjacentHTML` 方法完全一致。

```
// HTML 代码: <div id="one">one</div>
var d1 = document.getElementById('one');
d1.insertAdjacentText('afterend', 'two');
// 执行后的 HTML 代码:
// <div id="one">one</div>two
```

2.12. Element.remove()

`Element.remove()` 方法继承自 `ChildNode` 接口，用于将当前元素节点从它的父节点移除。

```
var el = document.getElementById('mydiv');
el.remove();
```

上面代码将 `el` 节点从 DOM 树里面移除。

2.13. Element.focus() 和 Element.blur()

`Element.focus()` 方法用于将当前页面的焦点，转移到指定元素上。

```
document.getElementById('my-span').focus();
```

该方法可以接受一个对象作为参数。参数对象的 `preventScroll` 属性是一个布尔值，指定是否将当前元素停留在原始位置，而不是滚动到可见区域。

```
function getFocus() {
  document.getElementById('btn').focus({preventScroll:false});
}
```

上面代码会让 `btn` 元素获得焦点，并滚动到可见区域。

最后，从 `document.activeElement` 属性可以得到当前获得焦点的元素。

`Element.blur` 方法用于将焦点从当前元素移除。

2.14. Element.click()

`Element.click()` 方法用于在当前元素上模拟一次鼠标点击，相当于触发了 `click` 事件。