Node 接口

所有 DOM 节点对象都继承了 Node 接口，拥有一些共同的属性和方法。这是 DOM 操作的基础。

属性

Node.prototype.nodeType

`nodeType`属性返回一个整数值，表示节点的类型。

```javascript
document.nodeType // 9
```

上面代码中，文档节点的类型值为9。

Node 对象定义了几个常量，对应这些类型值。

```javascript
document.nodeType === Node.DOCUMENT_NODE // true
```

上面代码中，文档节点的`nodeType`属性等于常量`Node.DOCUMENT_NODE`。

不同节点的`nodeType`属性值和对应的常量如下。

- 文档节点（document）：9，对应常量`Node.DOCUMENT_NODE`
- 元素节点（element）：1，对应常量`Node.ELEMENT_NODE`
- 属性节点（attr）：2，对应常量`Node.ATTRIBUTE_NODE`
- 文本节点（text）：3，对应常量`Node.TEXT_NODE`
- 文档片断节点（DocumentFragment）：11，对应常量`Node.DOCUMENT_FRAGMENT_NODE`
- 文档类型节点（DocumentType）：10，对应常量`Node.DOCUMENT_TYPE_NODE`
- 注释节点（Comment）：8，对应常量`Node.COMMENT_NODE`

确定节点类型时，使用`nodeType`属性是常用方法。

```javascript
var node = document.documentElement.firstChild;
if (node.nodeType === Node.ELEMENT_NODE) {
  console.log('该节点是元素节点');
}
```

Node.prototype.nodeName

`nodeName`属性返回节点的名称。

```javascript
// HTML 代码如下
// <div id="d1">hello world</div>
var div = document.getElementById('d1');
div.nodeName // "DIV"
```

上面代码中，元素节点`<div>`的`nodeName`属性就是大写的标签名`DIV`。

不同节点的`nodeName`属性值如下。

- 文档节点（document）：`#document`
- 元素节点（element）：大写的标签名
- 属性节点（attr）：属性的名称
- 文本节点（text）：`#text`
- 文档片断节点（DocumentFragment）：`#document-fragment`
- 文档类型节点（DocumentType）：文档的类型
- 注释节点（Comment）：`#comment`

Node.prototype.nodeValue

`nodeValue`属性返回一个字符串，表示当前节点本身的文本值，该属性可读写。

只有文本节点（text）、注释节点（comment）和属性节点（attr）有文本值，因此这三类节点的`nodeValue`可以返回结果，其他类型的节点一律返回`null`。同样的，也只有这三类节点可以设置`nodeValue`属性的值，其他类型的节点设置无效。

```javascript
// HTML 代码如下
// <div id="d1">hello world</div>
var div = document.getElementById('d1');
div.nodeValue // null
div.firstChild.nodeValue // "hello world"
```

上面代码中，`div`是元素节点，`nodeValue`属性返回`null`。`div.firstChild`是文本节点，所以可以返回文本值。

Node.prototype.textContent

`textContent`属性返回当前节点和它的所有后代节点的文本内容。

```javascript
// HTML 代码为
```

```
// <div id="divA">This is <span>some</span> text</div>

document.getElementById('divA').textContent
// This is some text
```

`textContent`属性自动忽略当前节点内部的 HTML 标签，返回所有文本内容。

该属性是可读写的，设置该属性的值，会用一个新的文本节点，替换所有原来的子节点。它还有一个好处，就是自动对 HTML 标签转义。这很适合用于用户提供的内容。

```javascript
document.getElementById('foo').textContent = '<p>GoodBye!</p>';
```

上面代码在插入文本时，会将`<p>`标签解释为文本，而不会当作标签处理。

对于文本节点（text）、注释节点（comment）和属性节点（attr），`textContent`属性的值与`nodeValue`属性相同。对于其他类型的节点，该属性会将每个子节点（不包括注释节点）的内容连接在一起返回。如果一个节点没有子节点，则返回空字符串。

文档节点（document）和文档类型节点（doctype）的`textContent`属性为`null`。如果要读取整个文档的内容，可以使用`document.documentElement.textContent`。

Node.prototype.baseURI

`baseURI`属性返回一个字符串，表示当前网页的绝对路径。浏览器根据这个属性，计算网页上的相对路径的 URL。该属性为只读。

```javascript
// 当前网页的网址为
// http://www.example.com/index.html
document.baseURI
// "http://www.example.com/index.html"
```

如果无法读到网页的 URL，`baseURI`属性返回`null`。

该属性的值一般由当前网址的 URL（即`window.location`属性）决定，但是可以使用 HTML 的`<base>`标签，改变该属性的值。

```html
<base href="http://www.example.com/page.html">
```

设置了以后，`baseURI`属性就返回`<base>`标签设置的值。

Node.prototype.ownerDocument

`Node.ownerDocument`属性返回当前节点所在的顶层文档对象，即`document`对象。

```javascript
var d = p.ownerDocument;
d === document // true
```

`document`对象本身的`ownerDocument`属性，返回`null`。

Node.prototype.nextSibling

`Node.nextSibling`属性返回紧跟在当前节点后面的第一个同级节点。如果当前节点后面没有同级节点，则返回`null`。

```javascript
// HTML 代码如下
// <div id="d1">hello</div><div id="d2">world</div>
var d1 = document.getElementById('d1');
var d2 = document.getElementById('d2');

d1.nextSibling === d2 // true
```

上面代码中，`d1.nextSibling`就是紧跟在`d1`后面的同级节点`d2`。

注意，该属性还包括文本节点和注释节点（`<!-- comment -->`）。因此如果当前节点后面有空格，该属性会返回一个文本节点，内容为空格。

`nextSibling`属性可以用来遍历所有子节点。

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

while (el !== null) {
  console.log(el.nodeName);
  el = el.nextSibling;
}
```

上面代码遍历`div1`节点的所有子节点。

Node.prototype.previousSibling

`previousSibling`属性返回当前节点前面的、距离最近的一个同级节点。如果当前节点前面没有同级节点，则返回`null`。

```javascript
// HTML 代码如下
```

```
// <div id="d1">hello</div><div id="d2">world</div>
var d1 = document.getElementById('d1');
var d2 = document.getElementById('d2');

d2.previousSibling === d1 // true
```

上面代码中，`d2.previousSibling`就是`d2`前面的同级节点`d1`。

注意，该属性还包括文本节点和注释节点。因此如果当前节点前面有空格，该属性会返回一个文本节点，内容为空格。

Node.prototype.parentNode

`parentNode`属性返回当前节点的父节点。对于一个节点来说，它的父节点只可能是三种类型：元素节点（element）、文档节点（document）和文档片段节点（documentfragment）。

```javascript
if (node.parentNode) {
  node.parentNode.removeChild(node);
}
```

上面代码中，通过`node.parentNode`属性将`node`节点从文档里面移除。

文档节点（document）和文档片段节点（documentfragment）的父节点都是`null`。另外，对于那些生成后还没插入 DOM 树的节点，父节点也是`null`。

Node.prototype.parentElement

`parentElement`属性返回当前节点的父元素节点。如果当前节点没有父节点，或者父节点类型不是元素节点，则返回`null`。

```javascript
if (node.parentElement) {
  node.parentElement.style.color = 'red';
}
```

上面代码中，父元素节点的样式设定了红色。

由于父节点只可能是三种类型：元素节点、文档节点（document）和文档片段节点（documentfragment）。`parentElement`属性相当于把后两种父节点都排除了。

Node.prototype.firstChild，Node.prototype.lastChild

`firstChild`属性返回当前节点的第一个子节点，如果当前节点没有子节点，则返回`null`。

```javascript
// HTML 代码如下
// <p id="p1"><span>First span</span></p>
var p1 = document.getElementById('p1');
p1.firstChild.nodeName // "SPAN"
```

上面代码中，`p`元素的第一个子节点是`span`元素。

注意，`firstChild`返回的除了元素节点，还可能是文本节点或注释节点。

```javascript
// HTML 代码如下
// <p id="p1">
//   <span>First span</span>
//  </p>
var p1 = document.getElementById('p1');
p1.firstChild.nodeName // "#text"
```

上面代码中，`p`元素与`span`元素之间有空白字符，这导致`firstChild`返回的是文本节点。

`lastChild`属性返回当前节点的最后一个子节点，如果当前节点没有子节点，则返回`null`。用法与`firstChild`属性相同。

Node.prototype.childNodes

`childNodes`属性返回一个类似数组的对象（`NodeList`集合），成员包括当前节点的所有子节点。

```javascript
var children = document.querySelector('ul').childNodes;
```

上面代码中，`children`就是`ul`元素的所有子节点。

使用该属性，可以遍历某个节点的所有子节点。

```javascript
var div = document.getElementById('div1');
var children = div.childNodes;

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

文档节点（document）就有两个子节点：文档类型节点（docType）和 HTML 根元素节点。

```javascript
var children = document.childNodes;
for (var i = 0; i < children.length; i++) {
  console.log(children[i].nodeType);
}
// 10
// 1
```

上面代码中，文档节点的第一个子节点的类型是10（即文档类型节点），第二个子节点的类型是1（即元素节点）。

注意，除了元素节点，`childNodes`属性的返回值还包括文本节点和注释节点。如果当前节点不包括任何子节点，则返回一个空的`NodeList`集合。由于`NodeList`对象是一个动态集合，一旦子节点发生变化，立刻会反映在返回结果之中。

Node.prototype.isConnected

`isConnected`属性返回一个布尔值，表示当前节点是否在文档之中。

```javascript
var test = document.createElement('p');
test.isConnected // false

document.body.appendChild(test);
test.isConnected // true
```

上面代码中，`test`节点是脚本生成的节点，没有插入文档之前，`isConnected`属性返回`false`，插入之后返回`true`。

方法

Node.prototype.appendChild()

`appendChild()`方法接受一个节点对象作为参数，将其作为最后一个子节点，插入当前节点。该方法的返回值就是插入文档的子节点。

```javascript
var p = document.createElement('p');
document.body.appendChild(p);
```

上面代码新建一个`<p>`节点，将其插入`document.body`的尾部。

如果参数节点是 DOM 已经存在的节点，`appendChild()`方法会将其从原来的位置，移动到新位置。

```javascript
```

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

上面代码中，插入的是一个已经存在的节点`myDiv`，结果就是该节点会从原来的位置，移动到`document.body`的尾部。

如果`appendChild()`方法的参数是`DocumentFragment`节点，那么插入的是`DocumentFragment`的所有子节点，而不是`DocumentFragment`节点本身。返回值是一个空的`DocumentFragment`节点。

Node.prototype.hasChildNodes()

`hasChildNodes`方法返回一个布尔值，表示当前节点是否有子节点。

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

if (foo.hasChildNodes()) {
  foo.removeChild(foo.childNodes[0]);
}
```

上面代码表示，如果`foo`节点有子节点，就移除第一个子节点。

注意，子节点包括所有类型的节点，并不仅仅是元素节点。哪怕节点只包含一个空格，`hasChildNodes`方法也会返回`true`。

判断一个节点有没有子节点，有许多种方法，下面是其中的三种。

- `node.hasChildNodes()`
- `node.firstChild !== null`
- `node.childNodes && node.childNodes.length > 0`

`hasChildNodes`方法结合`firstChild`属性和`nextSibling`属性，可以遍历当前节点的所有后代节点。

```javascript
function DOMComb(parent, callback) {
  if (parent.hasChildNodes()) {
    for (var node = parent.firstChild; node; node = node.nextSibling) {
      DOMComb(node, callback);
    }
  }
  callback(parent);
}

// 用法
DOMComb(document.body, console.log)
```

```

上面代码中，`DOMComb`函数的第一个参数是某个指定的节点，第二个参数是回调函数。这个回调函数会依次作用于指定节点，以及指定节点的所有后代节点。

### Node.prototype.cloneNode()

`cloneNode`方法用于克隆一个节点。它接受一个布尔值作为参数，表示是否同时克隆子节点。它的返回值是一个克隆出来的新节点。

```javascript
var cloneUL = document.querySelector('ul').cloneNode(true);
```

该方法有一些使用注意点。

（1）克隆一个节点，会拷贝该节点的所有属性，但是会丧失`addEventListener`方法和`on-`属性（即`node.onclick = fn`），添加在这个节点上的事件回调函数。

（2）该方法返回的节点不在文档之中，即没有任何父节点，必须使用诸如`Node.appendChild`这样的方法添加到文档之中。

（3）克隆一个节点之后，DOM 有可能出现两个有相同`id`属性（即`id="xxx"`）的网页元素，这时应该修改其中一个元素的`id`属性。如果原节点有`name`属性，可能也需要修改。

Node.prototype.insertBefore()

`insertBefore`方法用于将某个节点插入父节点内部的指定位置。

```javascript
var insertedNode = parentNode.insertBefore(newNode, referenceNode);
```

`insertBefore`方法接受两个参数，第一个参数是所要插入的节点`newNode`，第二个参数是父节点`parentNode`内部的一个子节点`referenceNode`。`newNode`将插在`referenceNode`这个子节点的前面。返回值是插入的新节点`newNode`。

```javascript
var p = document.createElement('p');
document.body.insertBefore(p, document.body.firstChild);
```

上面代码中，新建一个`<p>`节点，插在`document.body.firstChild`的前面，也就是成为`document.body`的第一个子节点。

如果`insertBefore`方法的第二个参数为`null`，则新节点将插在当前节点内部的最后位置，即变成最后一个子节点。
```

```javascript
var p = document.createElement('p');
document.body.insertBefore(p, null);
```

上面代码中，`p`将成为`document.body`的最后一个子节点。这也说明`insertBefore`的第二个参数不能省略。

注意，如果所要插入的节点是当前 DOM 现有的节点，则该节点将从原有的位置移除，插入新的位置。

由于不存在`insertAfter`方法，如果新节点要插在父节点的某个子节点后面，可以用`insertBefore`方法结合`nextSibling`属性模拟。

```javascript
parent.insertBefore(s1, s2.nextSibling);
```

上面代码中，`parent`是父节点，`s1`是一个全新的节点，`s2`是可以将`s1`节点，插在`s2`节点的后面。如果`s2`是当前节点的最后一个子节点，则`s2.nextSibling`返回`null`，这时`s1`节点会插在当前节点的最后，变成当前节点的最后一个子节点，等于紧跟在`s2`的后面。

如果要插入的节点是`DocumentFragment`类型，那么插入的将是`DocumentFragment`的所有子节点，而不是`DocumentFragment`节点本身。返回值将是一个空的`DocumentFragment`节点。

Node.prototype.removeChild()

`removeChild`方法接受一个子节点作为参数，用于从当前节点移除该子节点。返回值是移除的子节点。

```javascript
var divA = document.getElementById('A');
divA.parentNode.removeChild(divA);
```

上面代码移除了`divA`节点。注意，这个方法是在`divA`的父节点上调用的，不是在`divA`上调用的。

下面是如何移除当前节点的所有子节点。

```javascript
var element = document.getElementById('top');
while (element.firstChild) {
  element.removeChild(element.firstChild);
}
```

被移除的节点依然存在于内存之中，但不再是 DOM 的一部分。所以，一个节点移除以后，依然可以使用它，比如插入到另一个节点下面。

如果参数节点不是当前节点的子节点，`removeChild`方法将报错。

Node.prototype.replaceChild()

`replaceChild`方法用于将一个新的节点，替换当前节点的某一个子节点。

```javascript
var replacedNode = parentNode.replaceChild(newChild, oldChild);
```

上面代码中，`replaceChild`方法接受两个参数，第一个参数`newChild`是用来替换的新节点，第二个参数`oldChild`是将要替换走的子节点。返回值是替换走的那个节点`oldChild`。

```javascript
var divA = document.getElementById('divA');
var newSpan = document.createElement('span');
newSpan.textContent = 'Hello World!';
divA.parentNode.replaceChild(newSpan, divA);
```

上面代码是如何将指定节点`divA`替换走。

Node.prototype.contains()

`contains`方法返回一个布尔值，表示参数节点是否满足以下三个条件之一。

- 参数节点为当前节点。
- 参数节点为当前节点的子节点。
- 参数节点为当前节点的后代节点。

```javascript
document.body.contains(node)
```

上面代码检查参数节点`node`，是否包含在当前文档之中。

注意，当前节点传入`contains`方法，返回`true`。

```javascript
nodeA.contains(nodeA) // true
```

Node.prototype.compareDocumentPosition()

`compareDocumentPosition`方法的用法，与`contains`方法完全一致，返回一个六个比特位的二进制值，表示参数节点与当前节点的关系。

二进制值	十进制值	含义
000000	0	两个节点相同
000001	1	两个节点不在同一个文档（即有一个节点不在当前文档）
000010	2	参数节点在当前节点的前面
000100	4	参数节点在当前节点的后面
001000	8	参数节点包含当前节点
010000	16	当前节点包含参数节点
100000	32	浏览器内部使用

```javascript
// HTML 代码如下
// <div id="mydiv">
//   <form><input id="test" /></form>
// </div>

var div = document.getElementById('mydiv');
var input = document.getElementById('test');

div.compareDocumentPosition(input) // 20
input.compareDocumentPosition(div) // 10
```

上面代码中，节点`div`包含节点`input`（二进制`010000`），而且节点`input`在节点`div`的后面（二进制`000100`），所以第一个`compareDocumentPosition`方法返回`20`（二进制`010100`，即`010000 + 000100`），第二个`compareDocumentPosition`方法返回`10`（二进制`001010`）。

由于`compareDocumentPosition`返回值的含义，定义在每一个比特位上，所以如果要检查某一种特定的含义，就需要使用比特位运算符。

```javascript
var head = document.head;
var body = document.body;
if (head.compareDocumentPosition(body) & 4) {
  console.log('文档结构正确');
} else {
  console.log('<body> 不能在 <head> 前面');
}
```

上面代码中，`compareDocumentPosition`的返回值与`4`（又称掩码）进行与运算（`&`），得到一个布尔值，表示`<head>`是否在`<body>`前面。

Node.prototype.isEqualNode()，Node.prototype.isSameNode()

`isEqualNode`方法返回一个布尔值，用于检查两个节点是否相等。所谓相等的节点，指的是两个节点的类型相同、属性相同、子节点相同。

```javascript
var p1 = document.createElement('p');
var p2 = document.createElement('p');

p1.isEqualNode(p2) // true
```

`isSameNode`方法返回一个布尔值，表示两个节点是否为同一个节点。

```javascript
var p1 = document.createElement('p');
var p2 = document.createElement('p');

p1.isSameNode(p2) // false
p1.isSameNode(p1) // true
```

Node.prototype.normalize()

`normalize`方法用于清理当前节点内部的所有文本节点（text）。它会去除空的文本节点，并且将毗邻的文本节点合并成一个，也就是说不存在空的文本节点，以及毗邻的文本节点。

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

wrapper.appendChild(document.createTextNode('Part 1 '));
wrapper.appendChild(document.createTextNode('Part 2 '));

wrapper.childNodes.length // 2
wrapper.normalize();
wrapper.childNodes.length // 1
```

上面代码使用`normalize`方法之前，`wrapper`节点有两个毗邻的文本子节点。使用`normalize`方法之后，两个文本子节点被合并成一个。

该方法是`Text.splitText`的逆方法，可以查看《Text 节点对象》一章，了解更多内容。

Node.prototype.getRootNode()

`getRootNode()`方法返回当前节点所在文档的根节点`document`，与`ownerDocument`属性的作用相同。

```javascript
document.body.firstChild.getRootNode() === document
```

```
// true
document.body.firstChild.getRootNode() === document.body.firstChild.ownerDocument
// true
```

该方法可用于`document`节点自身，这一点与`document.ownerDocument`不同。

```javascript
document.getRootNode() // document
document.ownerDocument // null
```