基础知识

操作文档 HTML 的 JS 处理方式为 DOM 即 Document Object Model 文档对象模型。如果对 HTML 很了解使用 DOM 并不复杂。
浏览器在加载页面是会生成 DOM 对象，以供我们使用 JS 控制页面元素。

#文档渲染

浏览器会将 HTML 文本内容进行渲染，并生成相应的 JS 对象，同时会对不符规则的标签进行处理。

- 浏览器会将标签规范后渲染页面
- 目的一让页面可以正确呈现
- 目的二可以生成统一的 JS 可操作对象

#标签修复

在 html 中只有内容`houdunren.com`而没有任何标签时，通过浏览器的`检查>元素`标签查看会自动修复成以下格式的内容

下面 H1 标签结束错误并且属性也没有引号，浏览器在渲染中会进行修复

```
<body>
  <h1 id=houdunren>后盾人<h1>
</body>

```

处理后的结果

```
<html>
 <head></head>
 <body>
    <h1 id="houdunren">后盾人</h1>
```

```
  </body>
</html>

```

#表格处理

表格 tabel 中不允许有内容，浏览器在渲染过程中会进行处理

```
<table>
  houdunren.com
  <tr>
    <td>houdunwang.com</td>
  </tr>
</table>

```

渲染后会添加 tbody 标签并将 table 中的字符移出

```
houdunren.com
<table>
  <tbody>
        <tr>
      <td>houdunwang.com</td>
    </tr>
  </tbody>
</table>

```

#标签移动

所有内容要写在 BODY 标签中，下面的 SCRIPT 标签写在了 BODY 后面，浏览器渲染后也会进行处理

```
<body></body>
<script>
  console.dir('houdunren.com')
</script>

```

渲染后处理的结果

```
<body>
  <script>
    console.dir('houdunren.com')
  </script>
</body>

```

#操作时机

需要保证浏览器已经渲染了内容才可以读取的节点对象，下例将无法读取到节点对象

```
<script>
  const node = document.getElementById('houdunwang')
  console.log(node) //null
</script>
<h1 id="houdunwang">houdunren.com</h1>

```

不过我们可以将脚本通过事件放在页面渲染完执行

```
<script>
  window.onload = () => {
    const node = document.getElementById('houdunwang')
    console.log(node)
  }
</script>
<h1 id="houdunwang">houdunren.com</h1>

```

或使用定时器将脚本设置为异步执行

```
<script>
  setTimeout(() => {
    const node = document.getElementById('houdunwang')
    console.log(node)
  })
```

```
</script>
<h1 id="houdunwang">houdunren.com</h1>

```

也可以放在文档加载后的事件处理函数中

```
<script>
  window.onload = function () {
    let hd = document.getElementById('hd')
    console.log(hd)
  }
</script>
<div id="hd">houdunren</div>

```

或将脚本设置在外部文件并使用 defer 属性加载，defer 即会等到 DOM 解析后迟延执行

```
<script defer="defer" src="3.js"></script>
<div id="houdunwang"></div>

```

#节点对象

JS 中操作 DOM 的内容称为节点对象（node），即然是对象就包括操作 NODE 的属性和方法

- 包括 12 种类型的节点对象
- 常用了节点为 document、标签元素节点、文本节点、注释节点
- 节点均继承自 Node 类型，所以拥有相同的属性或方法
- document 是 DOM 操作的起始节点

```
<body id="houdunwang">
  <!-- 后盾人 -->
</body>
<script>
        // document节点 noteType为9
  console.log(document.nodeType)

  // 第一个子节点为<!DOCTYPE html>，且nodetype为10
  console.log(document.childNodes.item(0).nodeType)
```

```
10
11    // body 是标签节点 nodeType为1
12    console.log(document.body.nodeType)
13
14    // body的属性节点 nodeType 为2
15    console.log(document.body.attributes[0].nodeType)
16
17          // body的第一个节点为文本节点，nodeType为3
18    console.log(document.body.childNodes.item(0).nodeType)
19
20    // body的第二个节点为注释，nodeType类型为8
21    console.log(document.body.childNodes[1].nodeType)
22  </script>
23
```

#原型链

在浏览器渲染过程中会将文档内容生成为不同的对象，我们来对下例中的 h1 标签进行讨论，其他节点情况相似

- 不同类型节点由专有的构造函数创建对象
- 使用 console.dir 可以打印出 DOM 节点对象结构
- 节点也是对象所以也具有 JS 对象的特征

```
1  <h1 id="houdunwang">houdunren.com</h1>
2  <script>
3    function prototype(el) {
4      console.dir(el.__proto__)
5      el.__proto__ ? prototype(el.__proto__) : ''
6    }
7    const node = document.getElementById('houdunwang')
8    prototype(node)
9  </script>
10
```

最终得到的节点的原型链为

原型	说明
Object	根对象，提供 hasOwnProperty 等基本对象操作支持

EventTarget	提供 addEventListener、removeEventListener 等事件支持方法
Node	提供 firstChild、parentNode 等节点操作方法
Element	提供 getElementsByTagName、querySelector 等方法
HTMLElement	所有元素的基础类，提供 childNodes、nodeType、nodeName、className、nodeName 等方法
HTMLHeadingElement	Head 标题元素类

我们将上面的方法优化一下，实现提取节点原型链的数组

```
<h2 id="h2 value">houdunren.com</h2>
<input type="text" id="inputId" value="后盾人" />
<script>
    function prototype(el) {
        const prototypes = []
        prototypes.push(el.__proto__)
        prototypes.push(...(el.__proto__ ? prototype(el.__proto__) : []))
        return prototypes
    }
    const h2 = document.querySelector('h2')
    const input = document.querySelector('input')

    console.log(prototype(input))
</script>

```

下面为标题元素增加两个原型方法，改变颜色与隐藏元素

```
<h2 onclick="this.color('red')">houdunren.com</h2>
<script>
  const h2 = document.querySelector('h2')
  HTMLHeadingElement.prototype = Object.assign(HTMLHeadingElement.prototype, {
    color(color) {
      this.style.color = color
    },
    hide() {
      this.style.display = 'none'
```

```
    },
  })
</script>

```

#对象特征

即然 DOM 与我们其他 JS 创建的对象特征相仿，所以也可以为 DOM 对象添加属性或方法。
对于系统应用的属性，应该明确含义不应该随意使用，比如 ID 是用于标识元素唯一属性，不能用于其他目地

- 后面会讲到其他解决方案，来自定义属性，ID 属性可以直接修改但是不建议这么做

```
let hd = document.getElementById('hd')
hd.id = 'houdunren.com'
console.log(hd)

```

title 用于显示提示文档也不应该用于其他目地

```
<div id="hd">houdunren.com</div>
<script>
  let hd = document.getElementById('hd')
  hd.title = 'houdunren.com'
  console.log(hd)
</script>

```

下面是为对象合并属性的示例

```
<div id="hd">houdunren.com</div>
<script>
  let hd = document.getElementById('hd')

  Object.assign(hd, {
    //设置标签内容
    innerHTML: '向军大叔',
    color: 'red',
    change() {
```

```
      this.innerHTML = '后盾人'
      this.style.color = this.color
    },
    onclick() {
      this.change()
    },
  })
</script>

```

使用对象特性更改样式属性

```
<div id="hd">houdunren.com</div>
<script>
  let hd = document.getElementById('hd')
  Object.assign(hd.style, {
    color: 'white',
    backgroundColor: 'red',
  })
</script>

```

#常用节点

JS 提供了访问常用节点的 api

方法	说明
document	document 是 DOM 操作的起始节点
document.documentElement	文档节点即 html 标签节点
document.body	body 标签节点
document.head	head 标签节点
document.links	超链接集合
document.anchors	所有锚点集合
document.forms	form 表单集合
document.images	图片集合

#DOCUMENT

document 是 window 对象的属性，是由 HTMLDocument 类实现的实例。

- document 包含 DocumentType（唯一）或 html 元素（唯一）或 comment 等元素

原型链中也包含 Node，所以可以使用有关节点操作的方法如 nodeType/NodeName 等

```
console.dir(document.nodeType)
console.dir(document.nodeName)

```

> 有关使用 Document 操作 cookie 与本地储存将会在相应章节中介绍

使用 title 获取和设置文档标题

```
//获取文档标题
console.log(document.title)

//设置文档标签
document.title = '后盾人-houdunren.com'

```

获取当前 URL

```
console.log(document.URL)

```

获取域名

```
document.domain

```

获取来源地址

```
console.log(document.referrer)

```

系统针对特定标签提供了快速选择的方式

#ID

下面是直接使用 ID 获取元素（这是非标准操作，对浏览器有挑剔）

```
<div id="app">后盾人</div>
<script>
  // 直接通过 ID 获取元素（非标准操作）
  console.dir(app)
</script>

```

#links

下面展示的是获取所有 a 标签

```
<div name="app">
  <a href="">houdunren.com</a>
  <a href="">houdunwang.com</a>
</div>
<script>
  const nodes = document.links
  console.dir(nodes)
</script>

```

#anchors

下例是获取锚点集合后能过 锚点 name 属性获取元素

```
<div>
  <a href="" name="n1">houdunren.com</a>
  <a href="" name="n2">houdunwang.com</a>
</div>
<script>
  // 通过锚点获取元素
  console.dir(document.anchors.n2)
</script>

```

#images

下面是获取所有图片节点

```
<img src="" alt="" />
<img src="" alt="" />
<img src="" alt="" />
<script>
  // 获取所有图片节点
  console.dir(document.images)
</script>

```

#节点属性

不同类型的节点拥有不同属性，下面是节点属性的说明与示例

#nodeType

nodeType 指以数值返回节点类型

nodeType	说明
1	元素节点
2	属性节点
3	文本节点
8	注释节点
9	document 对象

下面是节点 nodeType 的示例

```
<div id="app">
  <div class="houdunren" data="hd">houdunren.com</div>
  <div class="houdunwang">houdunwang.com</div>
  <div class="xiangjun"><!-- 向军大叔 --></div>
</div>
<script>
  const node = document.querySelector(`#app`)
  console.log(node.nodeType) //1
  console.log(node.firstChild.nodeType) //3
```

```
  console.log(node.attributes.id.nodeType) //2

  const xj = document.querySelector('.xiangjun')
  console.log(xj.childNodes[0].nodeType) //8
</script>

```

下面是根据指定的 nodeType 递归获取节点元素的示例

- 可获取文本、注释、标签等节点元素

```
<!-- 后盾人 -->
后盾人 houdunren.com
<div id="app">
  <ul>
    <li>
      <span></span>
      <span>
        <!-- 向军 -->
      </span>
    </li>
    <li><span></span><span></span></li>
    <li><span></span><span></span></li>
  </ul>
</div>

<script>
  function all(el, nodeType = 1) {
    const nodes = []

    Array.from(el.childNodes).map(node => {
      if (node.nodeType == nodeType) nodes.push(node)

      if (node.nodeType == 1) nodes.push(...all(node, nodeType))
    })
    return nodes
  }
  console.log(all(document.body))
</script>

```

#Prototype

当然也可以使用对象的原型进行检测

- section 、main、aslide 标签的原型对象为 HTMLElement
- 其他非系统标签的原型对象为 HTMLUnknownElement

```
let h1 = document.querySelector('h1')
let p = document.querySelector('p')
console.log(h1 instanceof HTMLHeadingElement) //true
console.log(p instanceof HTMLHeadingElement) //false
console.log(p instanceof Element) //true

```

下例是通过构建函数获取节点的示例

```
<!-- 后盾人 -->
后盾人 houdunren.com
<div id="app">
  <ul>
    <li>
      <span></span>
      <span>
        <!-- 向军 -->
      </span>
    </li>
    <li><span></span><span></span></li>
    <li><span></span><span></span></li>
  </ul>
</div>

<script>
  function all(el, prototype) {
    const nodes = []

    Array.from(el.childNodes).map(node => {
      if (node instanceof prototype) nodes.push(node)

      if (node.nodeType == 1) nodes.push(...all(node, prototype))
    })
```

```
    return nodes
  }

  console.log(all(document.body, HTMLSpanElement))
</script>

```

#nodeName

nodeName 指定节点的名称

- 获取值为大写形式

nodeType	nodeName
1	元素名称如 DIV
2	属性名称
3	#text
8	#comment

下面来操作 nodeName

```
<div id="app">
  <div class="houdunren" data="hd">houdunren.com</div>
  <div class="houdunwang">houdunwang.com</div>
  <div class="xiangjun"><!-- 向军大叔 --></div>
  <span> 后盾人</span>
</div>
<script>
  const div = document.querySelector(`#app`)
  const span = document.querySelector('span')

  // 标签节点为大写的标签名DIV
  console.log(div.nodeName)
  console.log(span.nodeName)

  // 文本节点为 #text
  console.log(div.firstChild.nodeName)

  //属性节点为属性名
```

```
  console.log(div.attributes.id.nodeName)

  // 注释节点为#comment
  const xj = document.querySelector('.xiangjun')
  console.log(xj.childNodes[0].nodeName)
</script>

```

#tagName

nodeName 可以获取不限于元素的节点名，tagName 仅能用于获取标签节点的名称

- tagName 存在于 Element 类的原型中
- 文本、注释节点值为 undefined
- 获取的值为大写的标签名

```
<div id="app">
  <div class="houdunren" data="hd">houdunren.com</div>
  <div class="houdunwang">houdunwang.com</div>
  <div class="xiangjun"><!-- 向军大叔 --></div>
  <span> 后盾人</span>
</div>
<script>
  const div = document.querySelector(`#app`)
  const span = document.querySelector('span')

  // 标签节点为大写的标签名 如DIV、SPAN
  console.log(div.tagName)
  console.log(span.tagName)

  // 文本节点为undefined
  console.log(div.firstChild.tagName)

  //属性节点为undefined
  console.log(div.attributes.id.tagName)

  // 注释节点为 undefined
  const xj = document.querySelector('.xiangjun')
  console.log(xj.childNodes[0].tagName)
</script>
```

```
25
```

#nodeValue

使用 nodeValue 或 data 函数获取节点值，也可以使用节点的 data 属性获取节点内容

nodeType	nodeValue
1	null
2	属性值
3	文本内容
8	注释内容

下面来看 nodeValue 的示例

```
<div id="app">
  <div class="houdunren">houdunren.com</div>
  <div class="houdunwang">houdunwang.com</div>
  <div class="xiangjun"><!-- 向军大叔 --></div>
</div>
<script>
  const node = document.querySelector(`#app`)
  //标签的 nodeValue 值为 null
  console.log(node.nodeValue)

  //属性的 nodeVale 值为属性值
  console.log(node.attributes.id.nodeValue)

  //文本的 nodeValue 值为文本内容
  const houdunwang = document.querySelector('.houdunwang')
  console.log(houdunwang.firstChild.nodeValue)

  //注释的 nodeValue 值为注释内容
  const xj = document.querySelector('.xiangjun')
  console.log(xj.childNodes[0].nodeValue)
</script>

```

使用 data 属性可以获取文本与注释内容

```
<div id="app">
  houdunren.com
  <!-- 后盾人 注释内容-->
</div>

<script>
  const app = document.querySelector('#app')
  console.log(app.childNodes[0].data)
  console.log(app.childNodes[1].data)
</script>

```

#树状节点

下面获取标签树状结构即多级标签结构，来加深一下节点的使用

```
<div id="app">
  <ul>
    <li><span></span><span></span></li>
    <li><span></span><span></span></li>
    <li><span></span><span></span></li>
  </ul>
</div>

<script>
function tree(el) {
  return Array.from(el.childNodes)
    .filter(node =>node.tagName)
    .map(node => ({
      name: node.nodeName,
      children: tree(node),
    }))
}
console.log(tree(document.getElementById('app')))

```

上例结果如下

```
Array(2)
0: {name: 'HEAD', children: Array(4)}
1: {name: 'BODY', children: Array(2)}
```

#节点集合

Nodelist 与 HTMLCollection 都是包含多个节点标签的集合，大部分功能也是相同的。

- getElementsBy...等方法返回的是 HTMLCollection
- querySelectorAll 返回的是 NodeList
- NodeList 节点列表是动态的，即内容添加后会动态更新

```
<div></div>
<div></div>
<script>
  //结果为NodeList
  console.log(document.querySelectorAll('div'))

  //结果为HTMLCollection
  console.log(document.getElementsByTagName('div'))
</script>
```

#length

Nodelist 与 HTMLCollection 包含 length 属性，记录了节点元素的数量

```
<div name="app">
  <div id="houdunren">houdunren.com</div>
  <div name="houdunwang">houdunwang.com</div>
</div>
<script>
  const nodes = document.getElementsByTagName('div')
  for (let i = 0; i < nodes.length; i++) {
    console.log(nodes[i])
  }
```

```
</script>

```

#item

Nodelist 与 HTMLCollection 提供了 item()方法来根据索引获取元素

```
<div name="app">
  <div id="houdunren">houdunren.com</div>
  <div name="houdunwang">houdunwang.com</div>
</div>

<script>
  const nodes = document.getElementsByTagName('div')
  console.dir(nodes.item(0))
</script>

```

使用数组索引获取更方便

```
<div name="app">
  <div id="houdunren">houdunren.com</div>
  <div name="houdunwang">houdunwang.com</div>
</div>

<script>
  const nodes = document.getElementsByTagName('div')
  console.dir(nodes[0])
</script>

```

#namedItem

HTMLCollection 具有 namedItem 方法可以按 name 或 id 属性来获取元素

```
<div name="app">
  <div id="houdunren">houdunren.com</div>
  <div name="houdunwang">houdunwang.com</div>
```

```
</div>

<script>
  const nodes = document.getElementsByTagName('div')
  console.dir(nodes.namedItem('houdunwang'))
   console.dir(nodes.namedItem('houdunren'))
</script>

```

也可以使用数组或属性方式获取

```
<div name="app">
  <div id="houdunren">houdunren.com</div>
  <div name="houdunwang">houdunwang.com</div>
</div>

<script>
  const nodes = document.getElementsByTagName('div')
  console.dir(nodes['houdunwang']);
  console.dir(nodes.houdunren)
</script>

```

数字索引时使用 item 方法，字符串索引时使用 namedItem 或 items 方法

```
<h1 id="hd">houdunren.com</h1>
<h1 name="xj">向军大叔</h1>
<script>
  let items = document.getElementsByTagName('h1')
  console.log(items[0])
  console.log(items['xj'])
</script>

```

#动态与静态

通过 getElementsByTagname 等 getElementsBy... 函数获取的 Nodelist 与 HTMLCollection 集合是动态的，即有元素添加或移动操作将实时反映最新状态。

- 使用 getElement...返回的都是动态的集合

- 使用 querySelectorAll 返回的是静态集合

#动态特性

下例中通过按钮动态添加元素后，获取的元素集合是动态的，而不是上次获取的固定快照。

```
<h1>houdunren.com</h1>
<h1>houdunwang.com</h1>
<button id="add">添加元素</button>

<script>
  let elements = document.getElementsByTagName('h1')
  console.log(elements)
  let button = document.querySelector('#add')
  button.addEventListener('click', () => {
    document.querySelector('body').insertAdjacentHTML('beforeend', '<h1>向军大叔</h1>')
    console.log(elements)
  })
</script>

```

document.querySelectorAll 获取的集合是静态的

```
<h1>houdunren.com</h1>
<h1>houdunwang.com</h1>
<button id="add">添加元素</button>

<script>
  let elements = document.querySelectorAll('h1')
  console.log(elements.length)
  let button = document.querySelector('#add')
  button.addEventListener('click', () => {
    document.querySelector('body').insertAdjacentHTML('beforeend', '<h1>向军大叔</h1>')
    console.log(elements.length)
  })
</script>

```

#使用静态

如果需要保存静态集合，则需要对集合进行复制

```
<div id="houdunren">houdunren.com</div>
<div name="houdunwang">houdunwang.com</div>
<script>
  const nodes = document.getElementsByTagName('div')
  const clone = Array.prototype.slice.call(nodes)
  console.log(nodes.length);//2
  document.body.appendChild(document.createElement('div'))
  console.log(nodes.length);//3
  console.log(clone.length);//2
</script>

```

#遍历节点

#forOf

Nodelist 与 HTMLCollection 是类数组的可迭代对象所以可以使用 for...of 进行遍历

```
<div id="houdunren">houdunren.com</div>
<div name="houdunwang">houdunwang.com</div>
<script>
  const nodes = document.getElementsByTagName('div')
  for (const item of nodes) {
    console.log(item)
  }
</script>

```

#forEach

Nodelist 节点列表也可以使用 forEach 来进行遍历，但 HTMLCollection 则不可以

```
<div id="houdunren">houdunren.com</div>
<div name="houdunwang">houdunwang.com</div>
```

```
<script>
  const nodes = document.querySelectorAll('div')
  nodes.forEach((node, key) => {
    console.log(node)
  })
</script>

```

#call/apply

节点集合对象原型中不存在 map 方法，但可以借用 Array 的原型 map 方法实现遍历

```
<div id="houdunren">houdunren.com</div>
<div name="houdunwang">houdunwang.com</div>

<script>
  const nodes = document.querySelectorAll('div')
  Array.prototype.map.call(nodes, (node, index) => {
    console.log(node, index)
  })
</script>

```

当然也可以使用以下方式操作

```
;[].filter.call(nodes, node => {
        console.log(node)
})

```

#Array.from

Array.from 用于将类数组转为组件，并提供第二个迭代函数。所以可以借用 Array.from 实现遍历

```
<div id="houdunren">houdunren.com</div>
<div name="houdunwang">houdunwang.com</div>

<script>
```

```
  const nodes = document.getElementsByTagName('div')
  Array.from(nodes, (node, index) => {
    console.log(node, index)
  })
</script>

```

#展开语法

下面使用点语法转换节点为数组

```
<h1>houdunren.com</h1>
<h1>houdunwang.com</h1>
<script>
  let elements = document.getElementsByTagName('h1')
  console.log(elements)
  ;[...elements].map((item) => {
    item.addEventListener('click', function () {
      this.style.textTransform = 'uppercase'
    })
  })
</script>

```

#节点关系

节点是父子级嵌套与前后兄弟关系，使用 DOM 提供的 API 可以获取这种关系的元素。

- 文本和注释也是节点，所以也在匹配结果中

#基础知识

节点是根据 HTML 内容产生的，所以也存在父子、兄弟、祖先、后代等节点关系，下例中的代码就会产生这种多重关系

- h1 与 ul 是兄弟关系
- span 与 li 是父子关系
- ul 与 span 是后代关系
- span 与 ul 是祖先关系

```
<h1>后盾人</h1>
```

```
<ul>
  <li>
    <span>houdunren</span>
    <strong>houdunwang</strong>
  </li>
</ul>

```

下面是通过节点关系获取相应元素的方法

节点属性	说明
childNodes	获取所有子节点
parentNode	获取父节点
firstChild	第一个子节点
lastChild	最后一个子节点
nextSibling	下一个兄弟节点
previousSibling	上一个兄弟节点

子节点集合与首、尾节点获取

- 文本也是 node 所以也会在匹配当中

```
<div id="app">
  <div class="houdunren" data="hd">houdunren.com</div>
  <div class="houdunwang">houdunwang.com</div>
  <div class="xiangjun">向军大叔</div>
</div>
<script>
  const node = document.querySelector(`#app`)
  console.log(node.childNodes) //所有子节点
  console.log(node.firstChild) //第一个子节点是文本节点
  console.log(node.lastChild) //最后一个子节点也是文本节点
</script>

```

下面通过示例操作节点关联

- 文本也是 node 所以也会在匹配当中

```
<div id="app">
  <div class="houdunren" data="hd">houdunren.com</div>
  <div class="houdunwang">houdunwang.com</div>
  <div class="xiangjun">向军大叔</div>
</div>
<script>
  const node = app.querySelector(`.houdunwang`)
  console.log(node.parentNode) //div#app
  console.log(node.childNodes) //文本节点
  console.log(node.nextSibling) //下一个兄弟节点是文本节点
  console.log(node.previousSibling) //上一个节点也是文本节点
</script>

```

document 是顶级节点 html 标签的父节点是 document

```
<script>
  console.log(document.documentElement.parentNode === document)
</script>

```

#父节点集合

下例是查找元素的所有父节点

```
<div id="houdunren">houdunren.com</div>

<script>
  function parentNodes(node) {
    let nodes = []
    while ((node = node.parentNode)) nodes.push(node)
    return nodes
  }
  const el = document.getElementById('houdunren')
  const nodes = parentNodes(el)
  console.log(nodes)
</script>
```

```
13
```

使用递归获取所有父级节点

```
<div>
  <ul>
    <li><span></span></li>
  </ul>
</div>
<script>
  const span = document.querySelector('span')

  function parentNodes(node) {
    const nodes = new Array(node.parentNode)
    if (node.parentNode) nodes.push(...parentNodes(node.parentNode))
    return nodes
  }

  const nodes = parentNodes(document.querySelector('span'))
  console.log(nodes)
</script>

```

#后代节点集合

获取所有的后代元素 SPAN 的内容

```
<div id="app">
  <span>houdunren.com</span>
  <h2>
    <span>houdunwang.com</span>
  </h2>
</div>

<script>
  function getChildNodeByName(el, name) {
    const items = []
    Array.from(el.children).forEach(node => {
      if (node.tagName == name.toUpperCase()) items.push(node)
```

```
      items.push(...getChildNodeByName(node, name))
    })

    return items
  }
  const nodes = getChildNodeByName(document, 'span')
  console.log(nodes)
</script>

```

#标签关系

使用 childNodes 等获取的节点包括文本与注释，但这不是我们常用的，为此系统也提供了只操作元素的关系方法。

#基础知识

下面是处理标签关系的常用 API

节点属性	说明
parentElement	获取父元素
children	获取所有子元素
childElementCount	子标签元素的数量
firstElementChild	第一个子标签
lastElementChild	最后一个子标签
previousElementSibling	上一个兄弟标签
nextElementSibling	下一个兄弟标签
contains	返回布尔值，判断传入的节点是否为该节点的后代节点

以下实例展示怎样通过元素关系获取元素

```
<div id="app">
  <div class="houdunren" data="hd">houdunren.com</div>
  <div class="houdunwang">houdunwang.com</div>
  <div class="xiangjun"><!-- 向军大叔 --></div>
</div>
```

```
<script>
  const app = document.querySelector(`#app`)
  console.log(app.children) //所有子元素
  console.log(app.firstElementChild) //第一个子元素 div.houdunren
  console.log(app.lastElementChild) //最后一个子元素 div.xiangjun

  const houdunwang = document.querySelector('.houdunwang')
  console.log(houdunwang.parentElement) //父元素 div#app

  console.log(houdunwang.previousElementSibling) //上一个兄弟元素 div.houdunren
  console.log(houdunwang.nextElementSibling) //下一个兄弟元素 div.xiangjun
</script>
```

html 标签的父节点是 document，但父标签节点不存在

```
<script>
  console.log(document.documentElement.parentNode === document) //true
  console.log(document.documentElement.parentElement) //null
</script>
```

#按类名获取标签

下例是按 className 来获取标签

```
<div>
  <ul>
    <li class="hd item">houdunren.com</li>
    <li class="item">后盾人</li>
    <li class="hd">向军</li>
  </ul>
</div>
<script>
  function getTagByClassName(className, tag = document) {
    const items = []
    Array.from(tag.children).map(el => {
      if (el.className.includes(className)) items.push(el)
```

```
      items.push(...getTagByClassName(className, el))
    })
    return items
  }

  console.log(getTagByClassName('hd'))
</script>

```

#标签获取

系统提供了丰富的选择节点（NODE）的操作方法，下面我们来一一说明。

#getElementById

使用 ID 选择是非常方便的选择具有 ID 值的节点元素，但注意 ID 应该是唯一的

> 只能通过 document 对象上使用

```
<div id="houdunren">houdunren.com</div>
<script>
  const node = document.getElementById('houdunren')
  console.dir(node)
</script>

```

getElementById 只能通过 document 访问，不能通过元素读取拥有 ID 的子元素，下面的操作将产生错误

```
<div id="app">
  houdunren.com
  <div id="houdunwang">houdunwang.com</div>
</div>
<script>
  const app = document.getElementById('app')
  const node = app.getElementById('houdunwang') //app.getElementById is not a function
  console.log(node)
</script>

```

下面自定义函数来支持批量按 ID 选择元素

```
<div id="houdunren">houdunren.com</div>
<div id="app"></div>
<script>
  function getByElementIds(ids) {
    return ids.map((id) => document.getElementById(id))
  }
  let nodes = getByElementIds(['houdunren', 'app'])
  console.dir(nodes)
</script>

```

拥有 ID 的元素可做为 WINDOW 的属性进行访问

```
<div id="app">
  houdunren.com
</div>
<script>
  console.log(app.innerHTML)
</script>

```

如果声明了变量这种访问方式将无效，所以并不建议使用这种方式访问对象

```
<div id="app">
  houdunren.com
</div>
<script>
  let app = 'houdunwang'
  console.log(app.innerHTML)
</script>

```

#getElementsByName

使用 getElementByName 获取设置了 name 属性的元素，虽然在 DIV 等元素上同样有效，但一般用来对表单元素进行操作时使用。

- 返回 NodeList 节点列表对象
- NodeList 顺序为元素在文档中的顺序
- 需要在 document 对象上使用

```
<div name="houdunren">houdunren.com</div>
<input type="text" name="username" />

<script>
  const div = document.getElementsByName('houdunren')
  console.dir(div)
  const input = document.getElementsByName('username')
  console.dir(input)
</script>

```

#getElementsByTagName

使用 getElementsByTagName 用于按标签名获取元素

- 返回 HTMLCollection 节点列表对象
- 是不区分大小的获取

```
<div name="houdunren">houdunren.com</div>
<div id="app"></div>
<script>
  const divs = document.getElementsByTagName('div')
  console.dir(divs)
</script>

```

通配符

可以使用通配符 * 获取所有元素

```
<div name="houdunren">houdunren.com</div>
<div id="app"></div>

<script>
  const nodes = document.getElementsByTagName('*')
  console.dir(nodes)
```

```
</script>

```

某个元素也可以使用通配置符 * 获取后代元素，下面获取 id 为 houdunren 的所有后代元素

```
<div id="houdunren">
  <span>houdunren.com</span>
  <span>houdunwang.com</span>
</div>

<script>
  const nodes = document.getElementsByTagName('*').namedItem('houdunren').getElementsByTagName('*')
  console.dir(nodes)
</script>

```

#getElementsByClassName

getElementsByClassName 用于按 class 样式属性值获取元素集合

- 设置多个值时顺序无关，指包含这些 class 属性的元素

```
<div class="houdunren houdunwang xiangjun">houdunren.com</div>
<div class="houdunwang">houdunwang.com</div>

<script>
  const nodes = document.getElementsByClassName('houdunwang')
  console.log(nodes.length) //2

  //查找同时具有 houdunwang 与 houdunren 两个class属性的元素
  const tags = document.body.getElementsByClassName('houdunwang houdunren ')
  console.log(tags.length) //1
</script>

```

下面我们来自己开发一个与 getElementsByClassName 相同的功能函数

```
<div class="houdunren houdunwang xiangjun">houdunren.com</div>
```

```
<div class="houdunwang">houdunwang.com</div>
<script>
  function getByClassName(names) {
    //将用户参数转为数组，并过滤掉空值
    const classNames = names.split(/\s+/).filter(t => t)

    return Array.from(document.getElementsByTagName('*')).filter(tag => {
      // 提取标签的所有 class 值为数组
      return classNames.every(className => {
        const names = tag.className
          .toUpperCase()
          .split(/\s+/)
          .filter(t => t)

        //检索标签是否存在class
        return names.some(name => name == className.toUpperCase())
      })
    })
  }

  console.log(getByClassName('houdunwang houdunren '))
</script>

```

#样式选择器

在 CSS 中可以通过样式选择器修饰元素样式，在 DOM 操作中也可以使用这种方式查找元素。使用过 jQuery 库的朋友，应该对这种选择方式印象深刻。

使用 getElementsByTagName 等方式选择元素不够灵活，建议使用下面的样式选择器操作，更加方便灵活

#querySelectorAll

使用 querySelectorAll 根据 CSS 选择器获取 Nodelist 节点列表

- 获取的 NodeList 节点列表是静态的，添加或删除元素后不变

获取所有 div 元素

```
<div class="xiangjun">向军大叔</div>
<div id="app">
```

```
  <div class="houdunren houdunwang">houdunren.com</div>
  <div class="houdunwang">houdunwang.com</div>
</div>

<script>
  const app = document.getElementById('app')
  const nodes = app.querySelectorAll('div')
  console.log(nodes.length) //2
</script>
```

获取 id 为 app 元素的，class 为 houdunren 的后代元素

```
<div class="xiangjun">向军大叔</div>
<div id="app">
  <div class="houdunren houdunwang">houdunren.com</div>
  <div class="houdunwang">houdunwang.com</div>
</div>
<script>
  const nodes = document.querySelectorAll('#app .houdunren')
  console.log(nodes.length) //2
</script>
```

根据元素属性值获取元素集合

```
<div id="app">
  <div class="houdunren houdunwang" data="hd">houdunren.com</div>
  <div class="houdunwang">houdunwang.com</div>
</div>
<script>
  const nodes = document.querySelectorAll(`#app .houdunren[data='hd']`)
  console.log(nodes.length) //2
</script>
```

再来看一些通过样式选择器查找元素

```
<div id="app">
```

```
  <div class="houdunren">houdunren.com</div>
  <div class="houdunwang">houdunwang.com</div>
  <span>后盾人</span>
</div>

<script>
  //查找紧临兄弟元素
  console.log(document.querySelectorAll('.houdunren+div.houdunwang'))

  //查找最后一个 div 子元素
  console.log(document.querySelector('#app div:last-of-type'))

  //查找第二个 div 元素
  console.log(document.querySelector('#app div:nth-of-type(2)').innerHTML)
</script>

```

#querySelector

querySelector 使用 CSS 选择器获取一个元素，下面是根据属性获取单个元素

```
<div id="app">
  <div class="houdunren houdunwang" data="hd">houdunren.com</div>
  <div class="houdunwang">houdunwang.com</div>
</div>
<script>
  const node = app.querySelector(`#app .houdunren[data='hd']`)
  console.log(node)
</script>

```

#matches

用于检测元素是否是指定的样式选择器匹配，下面过滤掉所有 name 属性的 LI 元素

```
<div id="app">
  <li>houdunren</li>
  <li>向军大叔</li>
  <li name="houdunwang">houdunwang.com</li>
```

```
</div>
<script>
  const nodes = [...document.querySelectorAll('li')].filter(node => {
    return !node.matches(`[name]`)
  })
  console.log(nodes)
</script>

```

#closest

查找最近的符合选择器的祖先元素（包括自身），下例查找父级拥有 `.comment`类的元素

```
<div class="comment">
  <ul class="comment">
    <li>houdunren.com</li>
  </ul>
</div>

<script>
  const li = document.getElementsByTagName('li')[0]
  const node = li.closest(`.comment`)
  //结果为 ul.comment
  console.log(node)
</script>

```

#标准属性

元素的标准属性具有相对应的 DOM 对象属性

- 操作属性区分大小写
- 多个单词属性命名规则为第一个单词小写，其他单词大写
- 属性值是多类型并不全是字符串，也可能是对象等
- 事件处理程序属性值为函数
- style 属性为 CSSStyleDeclaration 对象
- DOM 对象不同生成的属性也不同

#属性别名

有些属性名与 JS 关键词冲突，系统已经起了别名

属性	别名
class	className
for	htmlFor

#操作属性

元素的标准属性可以直接进行操作，下面是直接设置元素的 className

```
<div id="app">
  <div class="houdunren" data="hd">houdunren.com</div>
  <div class="houdunwang">houdunwang.com</div>
</div>
<script>
  const app = document.querySelector(`#app`)
  app.className = 'houdunren houdunwang'
</script>

```

下面设置图像元素的标准属性

```
<img src="" alt="" />
<script>
  let img = document.images[0]
  img.src = '
https://www.houdurnen.com/avatar.jpg
  img.src = '
  img.alt = '后盾人'
</script>

```

使用 hidden 隐藏元素

```
<div id="app">houdunren.com</div>
<script>
  const app = document.querySelector('#app')
  app.addEventListener('click', function () {
```

```
    this.hidden = true
  })
</script>
```

#多类型

大部分属性值是都是字符串，但并不是全部，下例中需要转换为数值后进行数据运算

```
<input type="number" name="age" value="88" />

<script>
  let input = document.getElementsByName('age').item(0)
  input.value = parseInt(input.value) + 100
</script>
```

下面表单 checked 属性值为 Boolean 类型

```
<label for="hot"> <input id="hot" type="checkbox" name="hot" />热门 </label>
<script>
  const node = document.querySelector(`[name='hot']`)
  node.addEventListener('change', function () {
    console.log(this.checked)
  })
</script>
```

属性值并都与 HTML 定义的值一样，下面返回的 href 属性值是完整链接

```
<a href="#houdunren" id="home">后盾人</a>
<script>
  const node = document.querySelector(`#home`)
  console.log(node.href)
</script>
```

#元素特征

对于标准的属性可以使用 DOM 属性的方式进行操作，但对于标签的非标准的定制属性则不可以。但 JS 提供了方法来控制标准或非标准的属性

可以理解为元素的属性分两个地方保存，DOM 属性中记录标准属性，特征中记录标准和定制属性

- 使用特征操作时属性名称不区分大小写
- 特征值都为字符串类型

方法	说明
getAttribute	获取属性
setAttribute	设置属性
removeAttribute	删除属性
hasAttribute	属性检测

特征是可迭代对象，下面使用 for...of 来进行遍历操作

```
<div id="app" content="后盾人" color="red">houdunwang.com</div>
<script>
  const app = document.querySelector('#app')
  for (const { name, value } of app.attributes) {
    console.log(name, value)
  }
</script>

```

属性值都为字符串，所以数值类型需要进行转换

```
<input type="number" name="age" value="88" />
<script>
  let input = document.getElementsByName('age').item(0)
  let value = input.getAttribute('value') * 1 + 100
  input.setAttribute('value', value)
</script>

```

使用 removeAttribute 删除元素的 class 属性，并通过 hasAttribute 进行检测删除结果

```
<div class="houdunwang">houdunwang.com</div>
<script>
```

```
  let houdunwang = document.querySelector('.houdunwang')
  houdunwang.removeAttribute('class')
  console.log(houdunwang.hasAttribute('class')) //false
</script>

```

特征值与 HTML 定义是一致的，这和属性值是不同的

```
<a href="#houdunren" id="home">后盾人</a>
<script>
  const node = document.querySelector(`#home`)

  //
http://127.0.0.1:5500/test.html#houdunren
  //
  console.log(node.href)

  // #houdunren
  console.log(node.getAttribute('href'))
</script>

```

#attributes

元素提供了 attributes 属性可以只读的获取元素的属性

```
<div class="houdunwang" data-content="后盾人">houdunwang.com</div>
<script>
  let houdunwang = document.querySelector('.houdunwang')
  console.dir(houdunwang.attributes['class'].nodeValue) //houdunwang
  console.dir(houdunwang.attributes['data-content'].nodeValue) //后盾人
</script>

```

#自定义特征

虽然可以随意定义特征并使用 getAttribute 等方法管理，但很容易造成与标签的现在或未来属性重名。建议使用以 data-为前缀的自定义特征处理，针对这种定义方式 JS 也提供了接口方便操作。

- 元素中以 data-为前缀的属性会添加到属性集中

- 使用元素的 dataset 可获取属性集中的属性
- 改变 dataset 的值也会影响到元素上

下面演示使用属性集设置 DIV 标签内容

```
<div class="houdunwang" data-content="后盾人" data-color="red">houdunwang.com</div>

<script>
  let houdunwang = document.querySelector('.houdunwang')
  let content = houdunwang.dataset.content
  console.log(content) //后盾人
  houdunwang.innerHTML = `<span style="color:${houdunwang.dataset.color}">${content}</span>`
</script>

```

多个单词的特征使用驼峰命名方式读取

```
<div class="houdunwang" data-title-color="red">houdunwang.com</div>
<script>
  let houdunwang = document.querySelector('.houdunwang')
  houdunwang.innerHTML = `
    <span style="color:${houdunwang.dataset.titleColor}">${houdunwang.innerHTML}</span>
  `
</script>

```

改变 dataset 值也会影响到页面元素上

```
<div class="houdunwang" data-title-color="red">houdunwang.com</div>
<script>
  let houdunwang = document.querySelector('.houdunwang')
  houdunwang.addEventListener('click', function () {
    this.dataset.titleColor = ['red', 'green', 'blue'][Math.floor(Math.random() * 3)]
    this.style.color = this.dataset.titleColor
  })
</script>
```

```
9
```

#属性同步

特征和属性是记录元素属性的两个不同场所，大部分更改会进行同步操作。

下面使用属性更改了 className，会自动同步到了特征集中，反之亦然

```
<div id="app" class="red">houdunren.com</div>
<script>
  const app = document.querySelector('#app')
  app.className = 'houdunwang'
  console.log(app.getAttribute('class')) //houdunwang
  app.setAttribute('class', 'blue')
  console.log(app.className) //blue
</script>

```

下面对 input 值使用属性设置，但并没有同步到特征

```
<input type="text" name="package" value="houdunren.com" />
<script>
  const package = document.querySelector(`[name='package']`)
  package.value = 'houdunwang.com'
  console.log(package.getAttribute('value'))//houdunren.com
</script>

```

但改变 input 的特征 value 会同步到 DOM 对象属性

```
<input type="text" name="package" value="houdunren.com" />
<script>
  const package = document.querySelector(`[name='package']`)
  package.setAttribute('value', 'houdunwang.com')
  console.log(package.value) //houdunwang.com
</script>

```

#创建节点

创建节点的就是构建出 DOM 对象，然后根据需要添加到其他节点中

#append

append 也是用于添加元素，同时他也可以直接添加文本等内容。

```
<script>
    document.body.append((document.createElement('div').innerText = '向军'))
    document.body.append('houdunren.com')
</script>

```

#createTextNode

创建文本对象并添加到元素中

```
<div id="app"></div>
<script>
  let app = document.querySelector('#app')
  let text = document.createTextNode('houdunren')
  app.append(text)
</script>

```

#createElement

使用 createElement 方法可以标签节点对象，创建 span 标签新节点并添加到 div#app

```
<div id="app"></div>
<script>
  let app = document.querySelector('#app')
  let span = document.createElement('span')
  span.innerHTML = 'houdunren'
  app.append(span)
</script>

```

使用 PROMISE 结合节点操作来加载外部 JAVASCRIPT 文件

```
function js(file) {
  return new Promise((resolve, reject) => {
    let js = document.createElement('script')
    js.type = 'text/javascript'
    js.src = file
    js.onload = resolve
    js.onerror = reject
    document.head.appendChild(js)
  })
}

js('11.js')
  .then(() => console.log('加载成功'))
  .catch((error) => console.log(`${error.target.src} 加载失败`))

```

使用同样的逻辑来实现加载 CSS 文件

```
function css(file) {
  return new Promise((resolve, reject) => {
    let css = document.createElement('link')
    css.rel = 'stylesheet'
    css.href = file
    css.onload = resolve
    css.onerror = reject
    document.head.appendChild(css)
  })
}
css('1.css').then(() => {
  console.log('加载成功')
})

```

#cloneNode&importNode

使用 cloneNode 和 document.importNode 用于复制节点对象操作

- cloneNode 是节点方法
- cloneNode 参数为 true 时递归复制子节点即深拷贝
- importNode 是 documet 对象方法

复制 div#app 节点并添加到 body 元素中

```
<div id="app">houdunren</div>
<script>
  let app = document.querySelector('#app')
  let newApp = app.cloneNode(true)
  document.body.appendChild(newApp)
</script>

```

document.importNode 方法是部分 IE 浏览器不支持的，也是复制节点对象的方法

- 第一个参数为节点对象
- 第二个参数为 true 时递归复制

```
<div id="app">houdunren</div>
<script>
  let app = document.querySelector('#app')
  let newApp = document.importNode(app, true)
  document.body.appendChild(newApp)
</script>

```

#节点内容

#innerHTML

inneHTML 用于向标签中添加 html 内容，同时触发浏览器的解析器重绘 DOM。

下例使用 innerHTML 获取和设置 div 内容

- innerHTML 中只解析 HTML 标签语法，所以其中的 script 不会做为 JS 处理

```
<div id="app">
  <div class="houdunren" data="hd">houdunren.com</div>
  <div class="houdunwang">houdunwang.com</div>
</div>
<script>
  let app = document.querySelector('#app')
  console.log(app.innerHTML)

```

```
  app.innerHTML = '<h1>后盾人</h1>'
</script>
```

重绘节点

使用 innertHTML 操作会重绘元素，下面在点击第二次就没有效果了

- 因为对#app 内容进行了重绘，即删除原内容然后设置新内容
- 重绘后产生的 button 对象没有事件
- 重绘后又产生了新 img 对象，所以在控制台中可看到新图片在加载

```
<div id="app">
  <button>houdunren.com</button>
  <img src="1.jpg" alt="" />
</div>
<script>
  const app = document.querySelector('#app')
  app.querySelector('button').addEventListener('click', function () {
    alert(this.innerHTML)
    this.parentElement.innerHTML += '<hr/>向军大叔'
  })
</script>
```

#outerHTML

outerHTML 与 innerHTML 的区别是包含父标签

- outerHTML 不会删除原来的旧元素
- 只是用新内容替换替换旧内容，旧内容（元素）依然存在

下面将 div#app 替换为新内容

```
<div id="app">
  <div class="houdunren" data="hd">houdunren.com</div>
  <div class="houdunwang">houdunwang.com</div>
</div>
<script>
  let app = document.querySelector('#app')
  console.log(app.outerHTML)

  app.outerHTML = '<h1>后盾人</h1>'
```

```
</script>

```

使用 innerHTML 内容是被删除然后使用新内容

```
<div id="app">
  houdunren.com
</div>
<script>
  const app = document.querySelector('#app')
  console.log(app)
  app.innerHTML = 'houdunwang.com'
  console.log(app)
</script>

```

而使用 outerHTML 是保留旧内容，页面中使用新内容

```
<div id="app">
  houdunren.com
</div>
<script>
  const app = document.querySelector('#app')
  console.log(app)
  app.outerHTML = 'houdunwang.com'
  console.log(app)
</script>

```

#textContent 与 innerText

textContent 与 innerText 是访问或添加文本内容到元素中

- textContentb 部分 IE 浏览器版本不支持
- innerText 部分 FireFox 浏览器版本不支持
- 获取时忽略所有标签,只获取文本内容
- 设置时将内容中的标签当文本对待不进行标签解析

获取时忽略内容中的所有标签

```
<div id="app">
  <h1>houdunren.com</h1>
</div>
<script>
  let app = document.querySelector('#app')
  console.log(app.textContent)
</script>

```

设置时将标签当文本对待，即转为 HTML 实体内容

```
<div id="app">
  <div class="houdunren" data="hd">houdunren.com</div>
  <div class="houdunwang">houdunwang.com</div>
</div>
<script>
  let app = document.querySelector('#app')
  app.textContent="<h1>后盾人</h1>"
</script>

```

#outerText

与 innerText 差别是会影响所操作的标签

```
<h1>houdunren.com</h1>
<script>
  let h1 = document.querySelector('h1')
  h1.outerText = '后盾人'
</script>

```

#insertAdjacentText

将文本插入到元素指定位置，不会对文本中的标签进行解析，包括以下位置

选项	说明

beforebegin	元素本身前面
afterend	元素本身后面
afterbegin	元素内部前面
beforeend	元素内部后面

添加文本内容到 div#app 前面

```
<div id="app">
  <div class="houdunren" data="hd">houdunren.com</div>
  <div class="houdunwang">houdunwang.com</div>
</div>
<script>
  let app = document.querySelector('#app')
  let span = document.createElement('span')
  app.insertAdjacentText('beforebegin', '<h1>后盾人</h1>')
</script>

```

#节点管理

现在我们来讨论下节点元素的管理，包括添加、删除、替换等操作

#推荐方法

方法	说明
append	节点尾部添加新节点或字符串
prepend	节点开始添加新节点或字符串
before	节点前面添加新节点或字符串
after	节点后面添加新节点或字符串
replaceWith	将节点替换为新节点或字符串

在标签内容后面添加新内容

```
<div id="app">
  houdunren.com
```

```
</div>
<script>
  let app = document.querySelector('#app')
  app.append('-houdunwang.com')
</script>

```

同时添加多个内容，包括字符串与元素标签

```
<div id="app">
  houdunren.com
</div>
<script>
  let app = document.querySelector('#app')
  let h1 = document.createElement('h1')
  h1.append('后盾人')
  app.append('@', h1)
</script>

```

将标签替换为新内容

```
<div id="app">
  houdunren.com
</div>
<script>
  let app = document.querySelector('#app')
  let h1 = document.createElement('h1')
  h1.append('houdunwang.com')
  app.replaceWith(h1)
</script>

```

添加新元素 h1 到目标元素 div#app 里面

```
<div id="app"></div>
<script>
  let app = document.querySelector('#app')
  let h1 = document.createElement('h1')
```

```
  h1.innerHTML = 'houdunren'
  app.append(h1)
</script>

```

将 h2 移动到 h1 之前

```
<h1>houdunren.com@h1</h1>
<h2>houdunwang@h2</h2>
<script>
  let h1 = document.querySelector('h1')
  let h2 = document.querySelector('h2')
  h1.before(h2)
</script>

```

使用 remove 方法可以删除节点

```
<div id="app">
  houdunren.com
</div>
<script>
  let app = document.querySelector('#app')
  app.remove()
</script>

```

#insertAdjacentHTML

将 html 文本插入到元素指定位置，浏览器会对文本进行标签解析，包括以下位置

选项	说明
beforebegin	元素本身前面
afterend	元素本身后面
afterbegin	元素内部前面
beforeend	元素内部后面

在 div#app 前添加 HTML 文本

```
<div id="app">
  <div class="houdunren" data="hd">houdunren.com</div>
  <div class="houdunwang">houdunwang.com</div>
</div>
<script>
  let app = document.querySelector('#app')
  let span = document.createElement('span')
  app.insertAdjacentHTML('beforebegin', '<h1>后盾人</h1>')
</script>

```

#insertAdjacentElement

insertAdjacentElement() 方法将指定元素插入到元素的指定位置，包括以下位置

- 第一个参数是位置
- 第二个参数为新元素节点

选项	说明
beforebegin	元素本身前面
afterend	元素本身后面
afterbegin	元素内部前面
beforeend	元素内部后面

在 div#app 前插入 span 标签

```
<div id="app">
  <div class="houdunren" data="hd">houdunren.com</div>
  <div class="houdunwang">houdunwang.com</div>
</div>
<script>
  let app = document.querySelector('#app')
  let span = document.createElement('span')
  span.innerHTML = '后盾人'
  app.insertAdjacentElement('beforebegin', span)
</script>

```

#古老方法

下面列表过去使用的操作节点的方法，现在不建议使用了。但在阅读老代码时可来此查看语法

方法	说明
appendChild	添加节点
insertBefore	用于插入元素到另一个元素的前面
removeChild	删除节点
replaceChild	进行节点的替换操作

#DocumentFragment

当对节点进行添加、删除等操作时，都会引起页面回流来重新渲染页面,即重新渲染颜色，尺寸，大小、位置等等。所以会带来对性能的影响。

解决以上问题可以使用以下几种方式

1. 可以将 DOM 写成 html 字符串，然后使用 innerHTML 添加到页面中，但这种操作会比较麻烦，且不方便使用节点操作的相关方法。

2. 使用 createDocumentFragment 来管理节点时，此时节点都在内存中，而不是 DOM 树中。对节点的操作不会引发页面回流,带来比较好的性能体验。

DocumentFragment 特点

- createDocumentFragment 父节点为 null
- 继承自 node 所以可以使用 NODE 的属性和方法
- createDocumentFragment 创建的是文档碎片，节点类型 nodeType 为 11。因为不在 DOM 树中所以只能通过 JS 进行操作
- 添加 createDocumentFragment 添加到 DOM 后,就不可以再操作 createDocumentFragment 元素了,这与 DOM 操作是不同的
- 将文档 DOM 添加到 createDocumentFragment 时,会移除文档中的 DOM 元素
- createDocumentFragment 创建的节点添加到其他节点上时，会将子节点一并添加
- createDocumentFragment 是虚拟节点对象，不直接操作 DOM 所以性能更好
- 在排序/移动等大量 DOM 操作时建议使用 createDocumentFragment

#表单控制

表单是高频操作的元素，下面来掌握表单项的 DOM 操作

#表单查找

JS 为表单的操作提供了单独的集合控制

- 使用 document.forms 获取表单集合
- 使用 form 的 name 属性获取指定 form 元素
- 根据表单项的 name 属性使用 form.elements.title 获取表单项，
- 也可以直接写成 form.name 形式，不需要 form.elements.title
- 针对 radio/checkbox 获取的表单项是一个集合

```
<form action="" name="hd">
  <input type="text" name="title" />
</form>
<script>
  const form = document.forms.hd
  console.log(form.elements.title)
</script>

```

通过表单项可以反向查找 FORM

```
<form action="" name="hd">
  <input type="text" name="title" />
</form>
<script>
  const form = document.forms.hd
  console.log(form.title.form === form) //true
</script>

```

#样式管理

通过 DOM 修改样式可以通过更改元素的 class 属性或通过 style 对象设置行样式来完成。

- 建议使用 class 控制样式，将任务交给 CSS 处理，更简单高效

#批量设置

使用 JS 的 className 可以批量设置样式

```
<div id="app" class="d-flex container">后盾人</div>
<script>
```

```
  let app = document.getElementById('app')
  app.className = 'houdunwang'
</script>
```

也可以通过特征的方式来更改

```
<div id="app" class="d-flex container">后盾人</div>
<script>
  let app = document.getElementById('app')
  app.setAttribute('class', 'houdunwang')
</script>
```

#classList

如果对类单独进行控制使用 classList 属性操作

方法	说明
node.classList.add	添加类名
node.classList.remove	删除类名
node.classList.toggle	切换类名
node.classList.contains	类名检测

在元素的原有 class 上添加新 class

```
<div id="app" class="d-flex container">后盾人</div>
<script>
  let app = document.getElementById('app')
  app.classList.add('houdunwang')
</script>
```

使用 classList 也可以移除 class 列表中的部分 class

```
<div id="app" class="d-flex container">后盾人</div>
<script>
```

```
  let app = document.getElementById('app')
  app.classList.remove('container')
</script>

```

使用 toggle 切换类，即类已经存在时删除，不存在时添加

```
<div id="app" class="d-flex container">后盾人</div>
<script>
  let app = document.getElementById('app')
  app.addEventListener('click', function () {
    this.classList.toggle('houdunwang')
  })
</script>

```

使用 contains 检查 class 是否存在

```
<div id="app" class="d-flex container">后盾人</div>
<script>
  let app = document.getElementById('app')
  console.log(app.classList.contains('container')) //true
  console.log(app.classList.contains('houdunwang')) //false
</script>

```

#设置行样式

使用 style 对象可以对样式属性单独设置，使用 cssText 可以批量设置行样式

样式属性设置

使用节点的 style 对象来设置行样式

- 多个单词的属性使用驼峰进行命名

```
<div id="app" class="d-flex container">后盾人</div>
<script>
  let app = document.getElementById('app')
  app.style.backgroundColor = 'red'
  app.style.color = 'yellow'
</script>
```

```
7
```

批量设置行样式

使用 cssText 属性可以批量设置行样式，属性名和写 CSS 一样不需要考虑驼峰命名

```
<div id="app" class="d-flex container">后盾人</div>
<script>
  let app = document.getElementById('app')
  app.style.cssText = `background-color:red;color:yellow`
</script>

```

也可以通过 setAttribute 改变 style 特征来批量设置样式

```
<div id="app" class="d-flex container">后盾人</div>
<script>
  let app = document.getElementById('app')
  app.setAttribute('style', `background-color:red;color:yellow;`)
</script>

```

#获取样式

可以通过 style 对象，window.window.getComputedStyle 对象获取样式属性，下面进行说明

style

可以使用 DOM 对象的 style 属性读取行样式

- style 对象不能获取行样式外定义的样式

```
<style>
  div {
    color: yellow;
  }
</style>
<div id="app" style="background-color: red; margin: 20px;">后盾人</div>
<script>
  let app = document.getElementById('app')
  console.log(app.style.backgroundColor)
  console.log(app.style.margin)
```

```
  console.log(app.style.marginTop)
  console.log(app.style.color)

```

getComputedStyle

使用 window.getComputedStyle 可获取所有应用在元素上的样式属性

- 函数第一个参数为元素
- 第二个参数为伪类
- 这是计算后的样式属性，所以取得的单位和定义时的可能会有不同

```
<style>
  div {
    font-size: 35px;
    color: yellow;
  }
</style>
<div id="app" style="background-color: red; margin: 20px;">后盾人</div>
<script>
  let app = document.getElementById('app')
  let fontSize = window.getComputedStyle(app).fontSize
  console.log(fontSize.slice(0, -2))
  console.log(parseInt(fontSize))
</script>
```