DOM（上）

- DOM（Document Object Model）： 文档对象模型
- 其实就是操作 html 中的标签的一些能力
- 我们可以操作哪些内容
  - 获取一个元素
  - 移除一个元素
  - 创建一个元素
  - 向页面里面添加一个元素
  - 给元素绑定一些事件
  - 获取元素的属性
  - 给元素添加一些 css 样式
  - …
- DOM 的核心对象就是 docuemnt 对象
- document 对象是浏览器内置的一个对象，里面存储着专门用来操作元素的各种方法
- DOM： 页面中的标签，我们通过 js 获取到以后，就把这个对象叫做 DOM 对象

获取一个元素

- 通过 js 代码来获取页面中的标签
- 获取到以后我们就可以操作这些标签了

getElementById

- `getElementById` 是通过标签的 id 名称来获取标签的
- 因为在一个页面中 id 是唯一的，所以获取到的就是一个元素

```
<body>
 <div id="box"></div>
 <script>
 var box = document.getElementById('box')
 console.log(box) // <div></div>
 </script>
</body>
```

  - 获取到的就是页面中的那个 id 为 box 的 div 标签

getElementsByClassName

- `getElementsByClassName` 是用过标签的 class 名称来获取标签的

- 因为页面中可能有多个元素的 class 名称一样，所以获取到的是一组元素
- 哪怕你获取的 class 只有一个，那也是获取一组元素，只不过这一组中只有一个 DOM 元素而已

```
<body>
 <div calss="box"></div>
 <script>
 var box = document.getElementsByClassName('box')
 console.log(box) // [<div></div>]
 console.log(box[0]) // <div></div>
 </script>
</body>
```

  - 获取到的是一组元素，是一个长得和数组一样的数据结构，但是不是数组，是伪数组
  - 这个一组数据也是按照索引排列的，所以我们想要准确的拿到这个 div，需要用索引来获取

getElementsByTagName

- `getElementsByTagName` 是用过标签的 标签 名称来获取标签的
- 因为页面中可能有多个元素的 标签 名称一样，所以获取到的是一组元素
- 哪怕真的只有一个这个标签名，那么也是获取一组元素，只不过这一组中只有一个 DOM 元素而已

```
<body>
 <div></div>
 <script>
 var box = document.getElementsByTagName('div')
 console.log(box) // [<div></div>]
 console.log(box[0]) // <div></div>
 </script>
</body>
```

  - 和 `getElementsByClassName` 一样，获取到的是一个长得很像数组的元素
  - 必须要用索引才能得到准确的 DOM 元素

querySelector

- `querySelector` 是按照选择器的方式来获取元素
- 也就是说，按照我们写 css 的时候的选择器来获取
- 这个方法只能获取到一个元素，并且是页面中第一个满足条件的元素

```
console.log(document.querySelector('div')) // 获取页面中的第一个 div 元素
console.log(document.querySelector('.box')) // 获取页面中第一个有 box 类名的元素
```

```
console.log(document.querySelector('.box')) // 获取页面中第一个有 box 类名的元素
console.log(document.querySelector('#box')) // 获取页面中第一个 id 名为 box 的元素
```

querySelectorAll

- `querySelectorAll` 是按照选择器的方式来获取元素
- 这个方法能获取到所有满足条件的元素，以一个伪数组的形式返回

```
console.log(document.querySelectorAll('div')) // 获取页面中的所有的 div 元素
console.log(docuemnt.querySelectorAll('.box')) // 获取页面中所有有 box 类名的元素
```

  - 获取到的是一组数据，也是需要用索引来获取到准确的每一个 DOM 元素

操作属性

- 通过我们各种获取元素的方式获取到页面中的标签以后
- 我们可以直接操作 DOM 元素的属性，就能直接把效果展示在页面上

innerHTML

- 获取元素内部的 HTML 结构

```
<body>
 <div>
 <p>
 <span>hello</span>
 </p>
 </div>

 <script>
 var div = document.querySelector('div')
 console.log(div.innerHTML)
 /*

 <p>
 <span>hello</span>
 </p>

 */
 </script>
</body>
```

- 设置元素的内容

```
<body>
 <div></div>

 <script>
 var div = document.querySelector('div')
 div.innerHTML = '<p>hello</p>'
 </script>
</body>
```

- 设置完以后，页面中的 div 元素里面就会嵌套一个 p 元素

innerText

- 获取元素内部的文本（只能获取到文本内容，获取不到 html 标签）

```
<body>
 <div>
 <p>
 <span>hello</span>
 </p>
 </div>

 <script>
 var div = document.querySelector('div')
 console.log(div.innerText) // hello
 </script>
</body>
```

- 可以设置元素内部的文本

```
<body>
 <div></div>

 <script>
 var div = document.querySelector('div')
 div.innerText = '<p>hello</p>'
 </script>
</body>
```

- 设置完毕以后，会把 `<p>hello</p>` 当作一个文本出现在 div 元素里面，而不会把 p 解析成标签

getAttribute

- 获取元素的某个属性（包括自定义属性）

```
<body>
 <div a="100" class="box"></div>

 <script>
 var div = document.querySelector('div')
 console.log(div.getAttribute('a')) // 100
 console.log(div.getAttribute('class')) // box
 </script>
</body>
```

setAttribute

- 给元素设置一个属性（包括自定义属性）

```
<body>
 <div></div>

 <script>
 var div = document.querySelector('div')
 div.setAttribute('a', 100)
 div.setAttribute('class', 'box')
 console.log(div) // <div a="100" class="box"></div>
 </script>
</body>
```

removeAttribute

- 直接移除元素的某个属性

```
<body>
 <div a="100" class="box"></div>

 <script>
 var div = document.querySelector('div')
 div.removeAttribute('class')
 console.log(div) // <div a="100"></div>
 </script>
</body>
```

style

- 专门用来给元素添加 css 样式的
- 添加的都是行内样式

```
<body>
 <div></div>

 <script>
 var div = document.querySelector('div')
 div.style.width = "100px"
 div.style.height = "100px"
 div.style.backgroundColor = "pink"
 console.log(div)
 // <div style="width: 100px; height: 100px; background-color: pink;"></div>
 </script>
</body>
```

  - 页面中的 div 就会变成一个宽高都是100，背景颜色是粉色

className

- 专门用来操作元素的 类名的

```
<body>
 <div class="box"></div>

 <script>
 var div = document.querySelector('div')
 console.log(div.className) // box
 </script>
</body>
```

- 也可以设置元素的类名，不过是全覆盖式的操作

```
<body>
 <div class="box"></div>

 <script>
 var div = document.querySelector('div')
 div.className = 'test'
 console.log(div) // <div class="test"></div>
 </script>
</body>
```

- 在设置的时候，不管之前有没有类名，都会全部被设置的值覆盖

DOM（下）

- DOM 就是我们 html 结构中一个一个的节点构成的
- 不光我们的标签是一个节点，我们写的文本内容也是一个节点，注释，包括空格都是节点

DOM节点

- DOM 的节点我们一般分为常用的三大类 **元素节点 / 文本节点 / 属性节点**
- 什么是分类，比如我们在获取元素的时候，通过各种方法获取到的我们叫做元素节点（标签节点）
- 比如我们标签里面写的文字，那么就是文本节点
- 写在每一个标签上的属性，就是属性节点

元素节点

- 我们通过 `getElementBy...` 获取到的都是元素节点

属性节点

- 我们通过 `getAttribute` 获取的就是元素的属性节点

文本节点

- 我们通过 `innerText` 获取到的就是元素的文本节点

获取节点

- `childNodes` ：获取某一个节点下 **所有的子一级节点**

```
<body>
 <div>
 <p>hello</p>
 </div>

 <script>
 // 这个 oDiv 获取的是页面中的 div 元素，就是一个元素节点
 var oDiv = document.querySelector('div')

```

```
    console.log(oDiv.childNodes)
    /*
    NodeList(3) [text, p, text]
    0: text
    1: p
    2: text
    length: 3
    __proto__: NodeList
    */
    </script>
</body>
```

- 我们会发现，拿到以后是一个伪数组，里面有三个节点
- 一个 text：从
  一直到
  中间有一个换行和一堆空格，这个是第一个节点，是一个文本节点
- 一个 p：这个 p 标签就是第二个节点，这个是一个元素节点
- 一个 text：从 \

  一直到 \ 中间有一个换行和一堆空格，这个是第三个节点，是一个文本节点
-
- 这个时候就能看到我们有不同的节点类型了
- `children` ：获取某一节点下所有的子一级 **元素节点**

```
<body>
    <div>
    <p>hello</p>
    </div>

    <script>
    // 这个 oDiv 获取的是页面中的 div 元素，就是一个元素节点
    var oDiv = document.querySelector('div')

    console.log(oDiv.children)
    /*
    HTMLCollection [p]
    0: p
    length: 1
    __proto__: HTMLCollection
```

```
__proto__: HTMLCollection
*/
</script>
</body>
```

- 我们发现只有一个节点了，因为 `children` 只要元素节点
- div 下面又只有一个元素节点，就是 p
- 所以就只有一个，虽然只有一个，但是也是一个 **伪数组**

- `firstChild` ：获取某一节点下子一级的 **第一个节点**

```
<body>
<div>
<p>hello</p>
</div>

<script>
// 这个 oDiv 获取的是页面中的 div 元素，就是一个元素节点
var oDiv = document.querySelector('div')

console.log(oDiv.firstChild) // #text
</script>
</body>
```

- 这个是只获取一个节点，不再是伪数组了
- 获取的是第一个
- 第一个就是 `<div>` 一直到 `<p>` 的那个换行和空格，是个文本节点

- `lastChild` ：获取某一节点下子一级的 **最后一个节点**

```
<body>
<div>
<p>hello</p>
</div>

<script>
// 这个 oDiv 获取的是页面中的 div 元素，就是一个元素节点
var oDiv = document.querySelector('div')

console.log(oDiv.lastChild) // #text
</script>
</body>
```

- 只获取一个节点，不再是伪数组
- 获取的是最后一个
- 最后一个就是 \n

- 最后一个就是 \

  一直到 \ 之间的换行和空格，是个文本节点

- `firstElementChild` ：获取某一节点下子一级 **第一个元素节点**

```
<body>
 <div>
 <p>hello</p>
 </div>

 <script>
 // 这个 oDiv 获取的是页面中的 div 元素，就是一个元素节点
 var oDiv = document.querySelector('div')

 console.log(oDiv.firstElementChild) // <p>hello</p>
 </script>
</body>
```

  - 只获取一个节点，不在是伪数组
  - 获取的是第一个 **元素节点**
  - 第一个元素节点就是 p 标签，是一个元素节点

- `lastElementChild` ：获取某一节点下子一级 **最后一个元素节点**

```
<body>
 <div>
 <p>hello</p>
 <p>world</p>
 </div>

 <script>
 // 这个 oDiv 获取的是页面中的 div 元素，就是一个元素节点
 var oDiv = document.querySelector('div')

 console.log(oDiv.lastElementChild) // <p>world</p>
 </script>
</body>
```

  - 只获取一个节点，不在是伪数组
  - 获取的是最后一个 **元素节点**
  - 最后一个元素节点是 `<p>world</p>` ，是一个元素节点

- `nextSibling` ：获取某一个节点的 **下一个兄弟节点**

```
<body>
 <ul>
 <li id="a">hello</li>
 <li id="b">world</li>
 <li id="c">!!!</li>
 </ul>

 <script>
 // 这个 oLi 获取的是页面中的 li 元素，就是一个元素节点
 var oLi = document.querySelector('#b')

 console.log(oLi.nextSibling) // #text
 </script>
</body>
```

- 只获取一个节点，不在是伪数组
- 获取的是 `id="b"` 这个 li 的下一个兄弟节点
- 因为 `id="b"` 的下一个节点，是两个 li 标签之间的换行和空格，所以是一个文本节点

- `previousSibling` ：获取某一个节点的 **上一个兄弟节点**

```
<body>
 <ul>
 <li id="a">hello</li>
 <li id="b">world</li>
 <li id="c">!!!</li>
 </ul>

 <script>
 // 这个 oLi 获取的是页面中的 li 元素，就是一个元素节点
 var oLi = document.querySelector('#b')

 console.log(oLi.previousSibling) // #text
 </script>
</body>
```

- 只获取一个节点，不在是伪数组
- 获取的是 `id="b"` 这个 li 的上一个兄弟节点
- 因为 `id="b"` 的上一个节点，是两个 li 标签之间的换行和空格，所以是一个文本节点

- `nextElementSibling` ：获取某一个节点的 **下一个元素节点**

```
<body>
 <ul>
```

```
  <li id="a">hello</li>
  <li id="b">world</li>
  <li id="c">!!!</li>
  </ul>

  <script>
  // 这个 oLi 获取的是页面中的 li 元素，就是一个元素节点
  var oLi = document.querySelector('#b')

  console.log(oLi.nextElementSibling) // <li id="c">!!!</li>
  </script>
 </body>
```

- 只获取一个节点，不在是伪数组
- 获取的是 `id="b"` 这个 li 的下一个兄弟元素节点
- 因为 `id="b"` 的下一个兄弟元素节点就是 `id="c"` 的 li，是一个元素节点

- `previousElementSibling` ：获取某一个节点的 **上一个元素节点**

```
 <body>
  <ul>
  <li id="a">hello</li>
  <li id="b">world</li>
  <li id="c">!!!</li>
  </ul>

  <script>
  // 这个 oLi 获取的是页面中的 li 元素，就是一个元素节点
  var oLi = document.querySelector('#b')

  console.log(oLi.previousElementSibling) // <li id="a">hello</li>
  </script>
 </body>
```

- 只获取一个节点，不在是伪数组
- 获取的是 `id="b"` 这个 li 的上一个兄弟元素节点
- 因为 `id="b"` 的上一个兄弟元素节点就是 `id="a"` 的 li，是一个元素节点

- `parentNode` ：获取某一个节点的 **父节点**

```
 <body>
  <ul>
  <li id="a">hello</li>
  <li id="b">world</li>
```

```
  <li id="c">!!!</li>
  </ul>

  <script>
  // 这个 oLi 获取的是页面中的 li 元素，就是一个元素节点
  var oLi = document.querySelector('#b')

  console.log(oLi.parentNode) // <ul>...</ul>
  </script>
 </body>
```

- 只获取一个节点，不在是伪数组
- 获取的是当前这个 li 的父元素节点
- 因为这个 li 的父亲就是 ul，所以获取到的就是 ul，是一个元素节点

- `attributes` ：获取某一个 **元素节点** 的所有 **属性节点**

```
 <body>
  <ul>
  <li id="a" a="100" test="test">hello</li>
  </ul>

  <script>
  // 这个 oLi 获取的是页面中的 li 元素，就是一个元素节点
  var oLi = document.querySelector('#a')

  console.log(oLi.attributes)
  /*
  NamedNodeMap {0: id, 1: a, 2: test, id: id, a: a, test: test, length: 3}
  0: id
  1: a
  2: test
  length: 3
  a: a
  id: id
  test: test
  __proto__: NamedNodeMap

  */
  </script>
 </body>
```

- 获取的是一组数据，是该元素的所有属性，也是一个伪数组

- 这个 li 有三个属性，id / a / test 三个，所以就获取到了这三个

节点属性

- 我们已经知道节点会分成很多种，而且我们也能获取到各种不同的节点
- 接下来我们就来聊一些各种节点之间属性的区别
- 我们先准备一段代码

```
<body>
 <ul test="我是 ul 的一个属性">
 <li>hello</li>
 </ul>

 <script>
 // 先获取 ul
 var oUl = document.querySelector('ul')

 // 获取到 ul 下的第一个子元素节点，是一个元素节点
 var eleNode = oUl.firstElementChild

 // 获取到 ul 的属性节点组合，因为是个组合，我们要拿到节点的话要用索引
 var attrNode = oUl.attributes[0]

 // 获取到 ul 下的第一个子节点，是一个文本节点
 var textNode = oUl.firstChild
 </script>
</body>
```

nodeType

- `nodeType` ：获取节点的节点类型，用数字表示

```
console.log(eleNode.nodeType) // 1
console.log(attrNode.nodeType) // 2
console.log(textNode.nodeType) // 3
```

- `nodeType === 1` 就表示该节点是一个 **元素节点**
- `nodeType === 2` 就表示该节点是一个 **属性节点**
- `nodeType === 3` 就表示该节点是一个 **注释节点**

nodeName

- `nodeName` ：获取节点的节点名称

```
console.log(eleNode.nodeName) // LI
console.log(attrNode.nodeName) // test
console.log(textNode.nodeName) // #text
```

  - 元素节点的 `nodeName` 就是 **大写标签名**
  - 属性节点的 `nodeName` 就是 **属性名**
  - 文本节点的 `nodeName` 都是 **#text**

nodeValue

- `nodeValue` ： 获取节点的值

```
console.log(eleNode.nodeValue) // null
console.log(attrNode.nodeValue) // 我是 ul 的一个属性
console.log(textNode.nodeValue) // 换行 + 空格
```

  - 元素节点没有 `nodeValue`
  - 属性节点的 `nodeValue` 就是 **属性值**
  - 文本节点的 `nodeValue` 就是 **文本内容**

汇总

-	nodeType	nodeName	nodeValue
元素节点	1	大写标签名	null
属性节点	2	属性名	属性值
文本节点	3	#text	文本内容

操作 DOM 节点

- 我们所说的操作无非就是 **增删改查（CRUD）**
- 创建一个节点（因为向页面中增加之前，我们需要先创建一个节点出来）
- 向页面中增加一个节点
- 删除页面中的某一个节点
- 修改页面中的某一个节点
- 获取页面中的某一个节点

创建一个节点

- `createElement` ：用于创建一个元素节点

```
// 创建一个 div 元素节点
var oDiv = document.createElement('div')

console.log(oDiv) // <div></div>
```

  - 创建出来的就是一个可以使用的 div 元素

- `createTextNode` ：用于创建一个文本节点

```
// 创建一个文本节点
var oText = document.createTextNode('我是一个文本')

console.log(oText) // "我是一个文本"
```

向页面中加入一个节点

- `appendChild` ：是向一个元素节点的末尾追加一个节点
- 语法： `父节点.appendChild(要插入的子节点)`

```
// 创建一个 div 元素节点
var oDiv = document.createElement('div')
var oText = document.createTextNode('我是一个文本')

// 向 div 中追加一个文本节点
oDiv.appendChild(oText)

console.log(oDiv) // <div>我是一个文本</div>
```

- `insertBefore` ：向某一个节点前插入一个节点
- 语法： `父节点.insertBefore(要插入的节点，插入在哪一个节点的前面)`

```
<body>
 <div>
 <p>我是一个 p 标签</p>
 </div>

 <script>
 var oDiv = document.querySelector('div')
 var oP = oDiv.querySelector('p')

 // 创建一个元素节点
```

```
11.  var oSpan = document.createElement('span')
12.
13.  // 将这个元素节点添加到 div 下的 p 的前面
14.  oDiv.insertBefore(oSpan, oP)
15.
16.  console.log(oDiv)
17.  /*
18.  <div>
19.  <span></span>
20.  <p>我是一个 p 标签</p>
21.  </div>
22.  */
23.  </script>
24. </body>
```

删除页面中的某一个节点

- `removeChild` ：移除某一节点下的某一个节点
- 语法：`父节点.removeChild(要移除的字节点)`

```
1.  <body>
2.   <div>
3.   <p>我是一个 p 标签</p>
4.   </div>
5.
6.   <script>
7.   var oDiv = document.querySelector('div')
8.   var oP = oDiv.querySelector('p')
9.
10.  // 移除 div 下面的 p 标签
11.  oDiv.removeChild(oP)
12.
13.  console.log(oDiv) // <div></div>
14.  </script>
15. </body>
```

修改页面中的某一个节点

- `replaceChild` ：将页面中的某一个节点替换掉

- 语法：`父节点.replaceChild(新节点，旧节点)`

```
<body>
 <div>
 <p>我是一个 p 标签</p>
 </div>

 <script>
 var oDiv = document.querySelector('div')
 var oP = oDiv.querySelector('p')

 // 创建一个 span 节点
 var oSpan = document.createElement('span')
 // 向 span 元素中加点文字
 oSpan.innerHTML = '我是新创建的 span 标签'

 // 用创建的 span 标签替换原先 div 下的 p 标签
 oDiv.replaceChild(oSpan, oP)

 console.log(oDiv)
 /*
 <div>
 <span>我是新创建的 span 标签</span>
 </div>
 */
 </script>
</body>
```

获取元素的非行间样式

- 我们在操作 DOM 的时候，很重要的一点就是要操作元素的 css 样式
- 那么在操作 css 样式的时候，我们避免不了就要获取元素的样式
- 之前我们说过可以用 `元素.style.xxx` 来获取
- 但是这个方法只能获取到元素 **行间样式**，也就是写在行内的样式

```
<style>
 div {
 width: 100px;
 }
```

```
</style>
<body>
 <div style="height: 100px;">
 <p>我是一个 p 标签</p>
 </div>

 <script>
 var oDiv = document.querySelector('div')
 console.log(oDiv.style.height) // 100px
 console.log(oDIv.style.width) // ''
 </script>
</body>
```

- 不管是外链式还是内嵌式，我们都获取不到该元素的样式
- 这里我们就要使用方法来获取了 **getComputedStyle** 和 **currentStyle**
- 这两个方法的作用是一样的，只不过一个在 **非 IE** 浏览器，一个在 **IE** 浏览器

getComputedStyle（非IE使用）

- 语法：`window.getComputedStyle(元素, null).要获取的属性`

```
<style>
 div {
 width: 100px;
 }
</style>
<body>
 <div style="height: 100px;">
 <p>我是一个 p 标签</p>
 </div>

 <script>
 var oDiv = document.querySelector('div')
 console.log(window.getComputedStyle(oDiv).width) // 100px
 console.log(window.getComputedStyle(oDiv).height) // 100px
 </script>
</body>
```

  - 这个方法获取行间样式和非行间样式都可以

currentStyle（IE使用）

currentStyle（IE使用）

- 语法：`元素.currentStyle.要获取的属性`

```
<style>
 div {
 width: 100px;
 }
</style>
<body>
 <div style="height: 100px;">
 <p>我是一个 p 标签</p>
 </div>

 <script>
 var oDiv = document.querySelector('div')
 console.log(oDiv.currentStyle.width) // 100px
 console.log(oDiv.currentStyle.height) // 100px
 </script>
</body>
```

获取元素的偏移量

- 就是元素在页面上的什么位置
- 我们有几个属性来获取，offsetLeft 和 offsetTop 和 offsetWidth 和 offsetHeight

offsetLeft 和 offsetTop

- 获取的是元左边的偏移量和上边的偏移量
- 分成两个情况来看
- 没有定位的情况下
  - 获取元素边框外侧到页面内侧的距离
- 有定位的情况下
  - 获取元素边框外侧到定位父级边框内侧的距离（其实就是我们写的 left 和 top 值）

offsetWidth 和 offsetHeight

- 获取元素 `内容宽高 + padding宽高 + border宽高` 的和

强化练习2

1. 根据浏览器的滚动条，让顶部通览出现和消失
2. 实现点击按钮回到顶部功能
3. 在页面上根据数据，动态创建表格

```
var arr = [
 {
 name: 'Jack',
 age: 18,
 gender: '男'
 },{
 name: 'Rose',
 age: 20,
 gender: '女'
 },{
 name: 'Top',
 age: 22,
 gender: '男'
 }
]
```

- 创建一个表格，展示上面数组中的信息

强化练习3

4. 根据数据动态创建一个表格
5. 动态创建的表格有删除功能，可以删除表格中的某一项