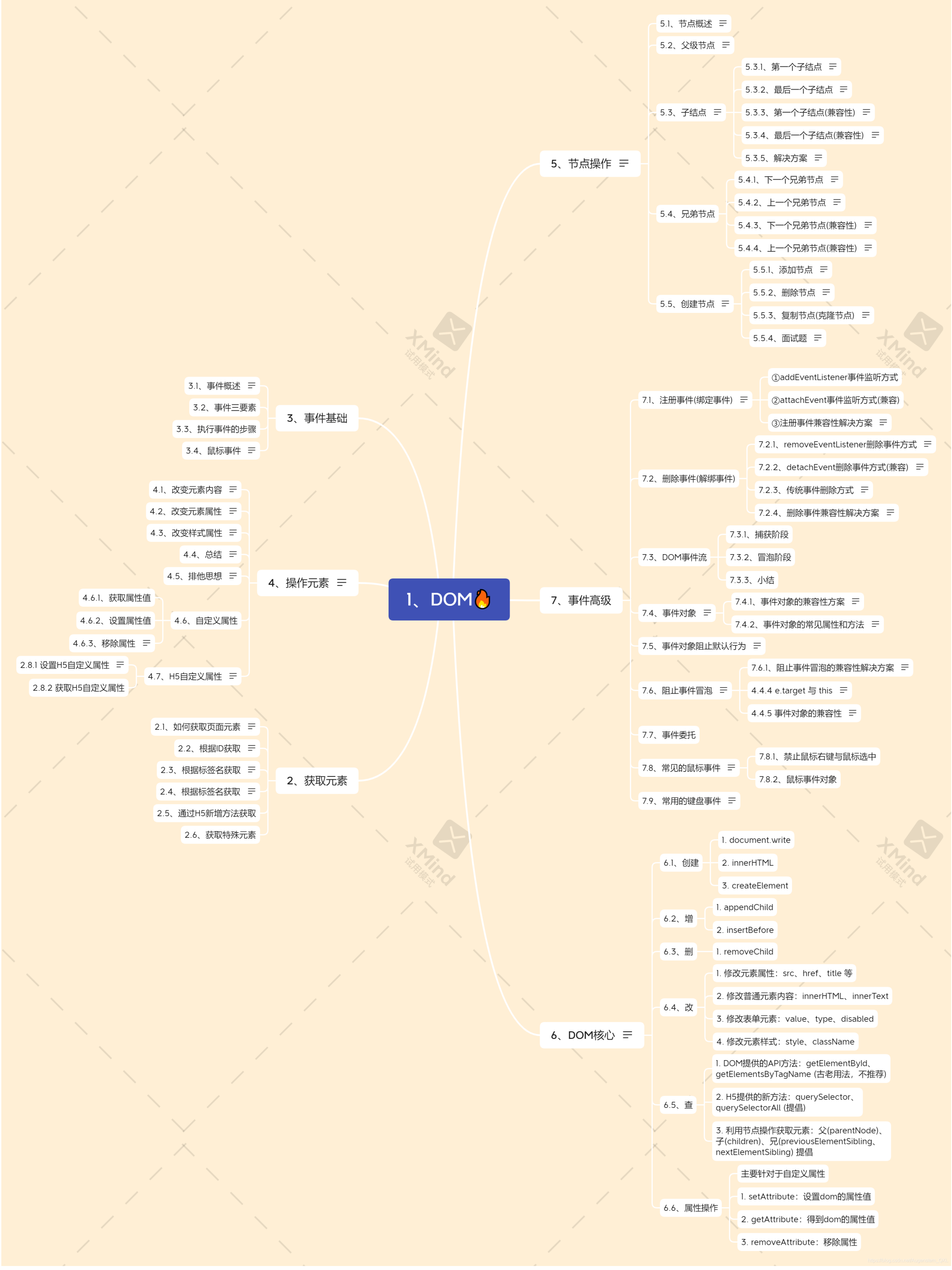
JavaScript进阶班之DOM技术(四)

✍JavaScript帝国之行🔥

内容	地址
JavaScript基础大总结(一) 🔥	https://blog.csdn.net/Augenstern_QXL/article/details/119249534
JavaScript基础之函数与作用域(二) 🔥	https://blog.csdn.net/Augenstern_QXL/article/details/119250991
JavaScript基础之对象与内置对象(三)🔥	https://blog.csdn.net/Augenstern_QXL/article/details/119250137
JavaScript进阶之DOM技术(四) 🔥	https://blog.csdn.net/Augenstern_QXL/article/details/115416921
JavaScript进阶之BOM技术(五) 🔥	https://blog.csdn.net/Augenstern_QXL/article/details/115406408
JavaScript提高之面向对象(六) 🔥	https://blog.csdn.net/Augenstern_QXL/article/details/115219073
JavaScript提高之ES6(七) 🔥	https://blog.csdn.net/Augenstern_QXL/article/details/115344398

✍目录总览

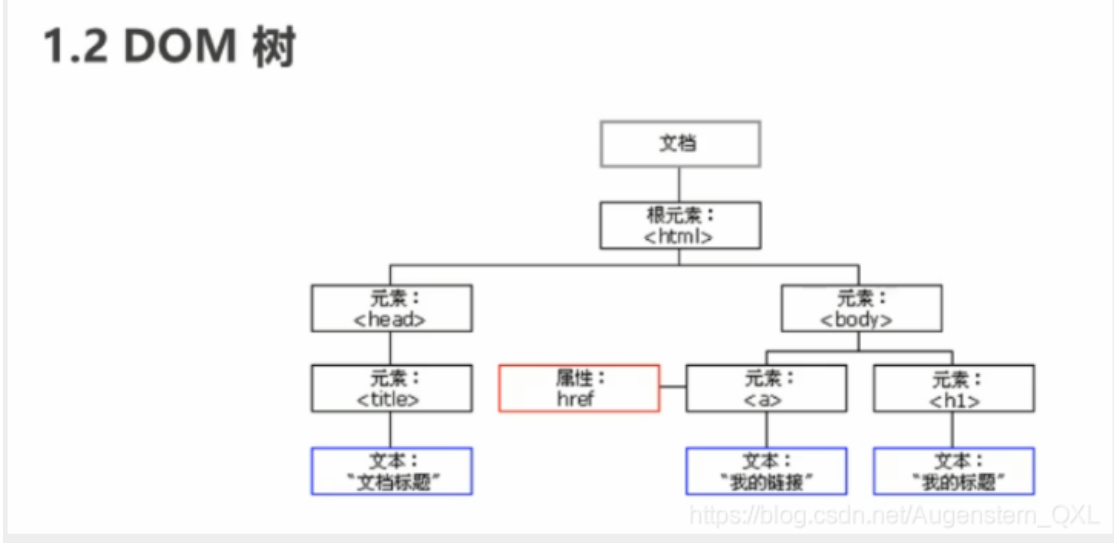


1、DOM简介

1.1、什么是DOM

文档对象模型（Document Object Model，简称 DOM），是 W3C 组织推荐的处理可扩展标记语言（HTML或者XML）的标准编程接口

W3C 已经定义了一系列的 DOM 接口，通过这些 DOM 接口可以改变网页的内容、结构和样式。



- 文档：一个页面就是一个文档，DOM中使用doucument来表示
- 元素：页面中的所有标签都是元素，DOM中使用 element 表示
- 节点：网页中的所有内容都是节点（标签，属性，文本，注释等），DOM中使用node表示

DOM 把以上内容都看做是**对象**

2、获取元素

2.1、如何获取页面元素

DOM在我们实际开发中主要用来操作元素。

我们如何来获取页面中的元素呢?

获取页面中的元素可以使用以下几种方式:

- 根据 ID 获取
- 根据标签名获取
- 通过 HTML5 新增的方法获取
- 特殊元素获取

2.2、根据ID获取

使用 `getElementByld()` 方法可以获取带ID的元素对象

```
doucument.getElementByld('id名')
```

使用 `console.dir()` 可以打印我们获取的元素对象，更好的查看对象里面的属性和方法。

示例

```
<div id="time">2019-9-9</div>
<script>
    // 1.因为我们文档页面从上往下加载，所以得先有标签，所以script写在标签下面
    // 2.get 获得 element 元素 by 通过 驼峰命名法
    // 3.参数 id是大小写敏感的字符串
    // 4.返回的是一个元素对象
    var timer = document.getElementById('time');
    console.log(timer);
    // 5. console.dir 打印我们的元素对象，更好的查看里面的属性和方法
    console.dir(timer);
</script>
```

2.3、根据标签名获取

根据**标签名**获取，使用 `getElementByTagName()` 方法可以返回带有指定标签名的**对象的集合**

```
doucument.getElementsByTagName('标签名');
```

- 因为得到的是一个对象的集合，所以我们想要操作里面的元素就需要遍历
- 得到元素对象是动态的
- 返回的是获取过来元素对象的集合，以伪数组的形式存储
- 如果获取不到元素，则返回为空的伪数组(因为获取不到对象)

```
<ul>
    <li>知否知否，应是等你好久</li>
    <li>知否知否，应是等你好久</li>
    <li>知否知否，应是等你好久</li>
    <li>知否知否，应是等你好久</li>
    <li>知否知否，应是等你好久</li>
</ul>
<script>
    // 1.返回的是获取过来元素对象的集合 以伪数组的形式存储
    var lis = document.getElementsByTagName('li');
    console.log(lis);
    console.log(lis[0]);
    // 2.依次打印,遍历
    for (var i = 0; i < lis.length; i++) {
```

```
        console.log(lis[i]);
    }
    // 3.如果页面中只有 1 个 li，返回的还是伪数组的形式
    // 4.如果页面中没有这个元素，返回的是空伪数组
</script>
```

2.4、根据标签名获取

还可以根据标签名获取某个元素（父元素）内部所有指定标签名的子元素,获取的时候不包括父元素自己

```
element.getElementsByTagName('标签名')

ol.getElementsByTagName('li');
```

注意：父元素必须是**单个对象(必须指明是哪一个元素对象)**，获取的时候不包括父元素自己

```
<script>
    //element.getElementsByTagName('标签名'); 父元素必须是指定的单个元素
    var ol = document.getElementById('ol');
    console.log(ol.getElementsByTagName('li'));
</script>
```

2.5、通过H5新增方法获取

①getElementsByClassName

根据类名返回元素对象合集

- `document.getElementsByClassName('类名')`

```
document.getElementsByClassName('类名');
```

②document.querySelector

根据指定选择器返回第一个元素对象

```
document.querySelector('选择器');

// 切记里面的选择器需要加符号
// 类选择器.box
// id选择器 #nav
var firstBox = document.querySelector('.box');
```

③document.querySelectorAll

根据指定选择器返回所有元素对象

```
document.querySelectorAll('选择器');
```

注意:

`querySelector` 和 `querySelectorAll` 里面的选择器需要加**符号**,比如: `document.querySelector('#nav');`

④例子

```
<script>
    // 1. getElementsByClassName 根据类名获得某些元素集合
    var boxs = document.getElementsByClassName('box');
    console.log(boxs);
    // 2. querySelector 返回指定选择器的第一个元素对象  切记 里面的选择器需要加符号 .box  #nav
    var firstBox = document.querySelector('.box');
    console.log(firstBox);
    var nav = document.querySelector('#nav');
    console.log(nav);
    var li = document.querySelector('li');
    console.log(li);
    // 3. querySelectorAll()返回指定选择器的所有元素对象集合
    var allBox = document.querySelectorAll('.box');
    console.log(allBox);
    var lis = document.querySelectorAll('li');
    console.log(lis);
</script>
```

2.6、获取特殊元素

①获取body元素

返回body元素对象

```
document.body;
```

②获取html元素

返回html元素对象

```
document.documentElement;
```

3、事件基础

3.1、事件概述

JavaScript 使我们有能力创建动态页面，而事件是可以被 JavaScript 侦测到的行为。

简单理解： 触发— 响应机制。

网页中的每个元素都可以产生某些可以触发 JavaScript 的事件，例如，我们可以在用户点击某按钮时产生一个事件，然后去执行某些操作。

3.2、事件三要素

1. 事件源(谁)
2. 事件类型(什么事件)
3. 事件处理程序(做啥)

```
<script>
    // 点击一个按钮，弹出对话框
    // 1. 事件是有三部分组成  事件源  事件类型  事件处理程序    我们也称为事件三要素
    //(1) 事件源 事件被触发的对象    谁   按钮
    var btn = document.getElementById('btn');
    //(2) 事件类型  如何触发 什么事件 比如鼠标点击(onclick) 还是鼠标经过 还是键盘按下
    //(3) 事件处理程序  通过一个函数赋值的方式 完成
    btn.onclick = function() {
        alert('点秋香');
    }
</script>
```

3.3、执行事件的步骤

1. 获取事件源
2. 注册事件(绑定事件)
3. 添加事件处理程序(采取函数赋值形式)

```
<script>
    // 执行事件步骤
    // 点击div 控制台输出 我被选中了
    // 1. 获取事件源
    var div = document.querySelector('div');
    // 2.绑定事件 注册事件
    // div.onclick
    // 3.添加事件处理程序
    div.onclick = function() {
        console.log('我被选中了');
    }
</script>
```

3.4、鼠标事件

鼠标事件	触发条件
onclick	鼠标点击左键触发
onmouseover	鼠标经过触发
onmouseout	鼠标离开触发
onfocus	获得鼠标焦点触发
onblur	失去鼠标焦点触发
onmousemove	鼠标移动触发
onmouseup	鼠标弹起触发
onmousedown	鼠标按下触发

4、操作元素

JavaScript 的 DOM 操作可以改变网页内容、结构和样式，我们可以利用 DOM 操作元素来改变元素里面的内容 、属性等。注意以下都是属性

4.1、改变元素内容

从起始位置到终止位置的内容，但它去除html标签，同时空格和换行也会去掉。

```
element.innerText
```

起始位置到终止位置的全部内容，包括HTML标签，同时保留空格和换行

```
element.innerHTML
```

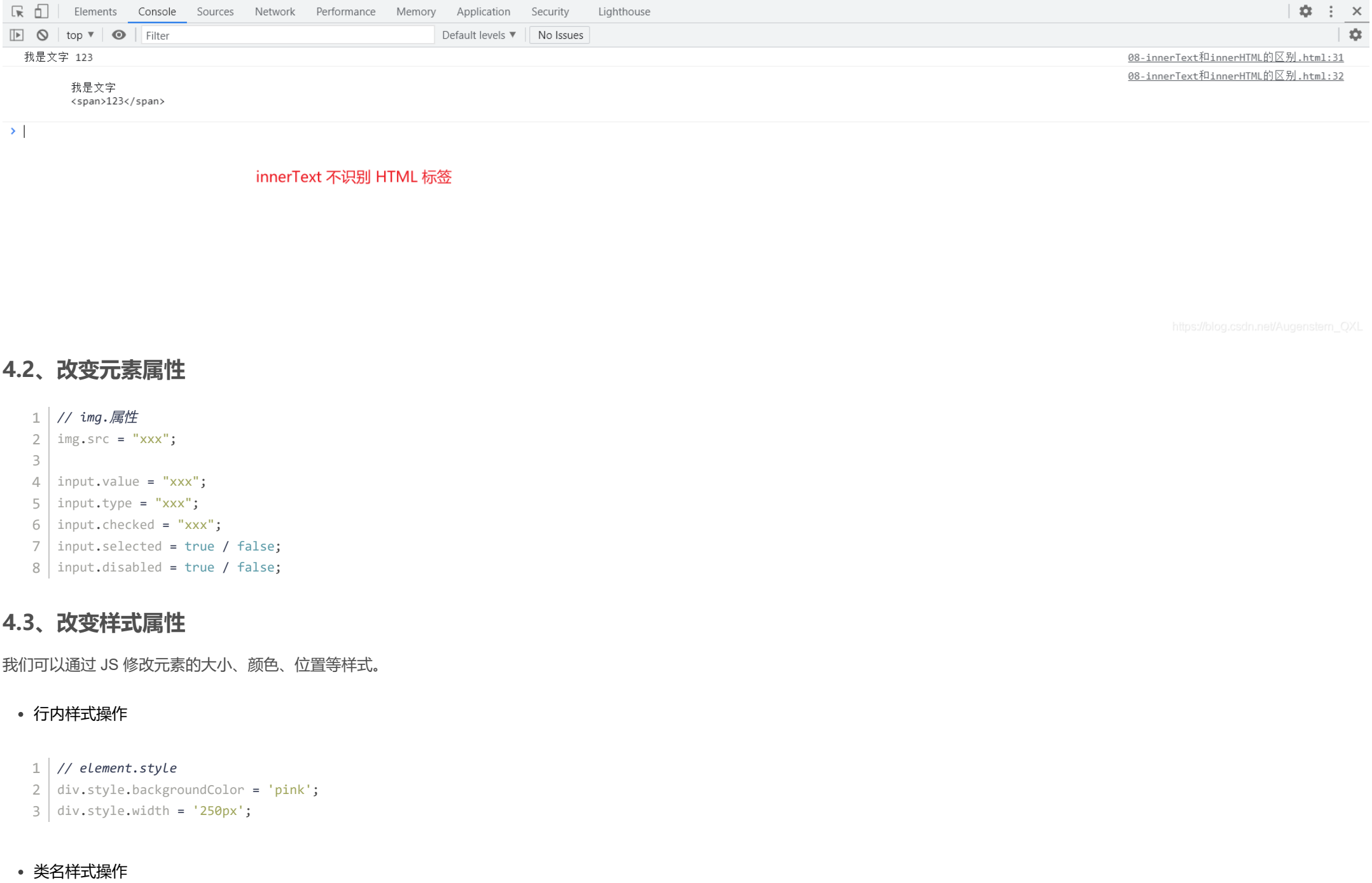
```
<body>
    <div></div>
    <p>
        我是文字
        <span>123</span>
    </p>

```

```
    <script>
        // innerText 和 innerHTML的区别
        // 1. innerText 不识别html标签,去除空格和换行
        var div = document.querySelector('div');
        div.innerText = '<strong>今天是：</strong> 2019';
        // 2. innerHTML 识别html标签 保留空格和换行的
        div.innerHTML = '<strong>今天是：</strong> 2019';
        // 这两个属性是可读写的  可以获取元素里面的内容
        var p = document.querySelector('p');
        console.log(p.innerText);
        console.log(p.innerHTML);
    </script>
</body>
```

今天是： 2019 ——→ innerText 不识别 HTML 标签

今天是： 2019 ——→ innerHTML 识别 HTML 标签

我是文字 123



4.2、改变元素属性

```
// img.属性
img.src = "xxx";

input.value = "xxx";
input.type = "xxx";
input.checked = "xxx";
input.selected = true / false;
input.disabled = true / false;
```

4.3、改变样式属性

我们可以通过 JS 修改元素的大小、颜色、位置等样式。

- 行内样式操作

```
// element.style
div.style.backgroundColor = 'pink';
div.style.width = '250px';
```

- 类名样式操作

```
// element.className
```

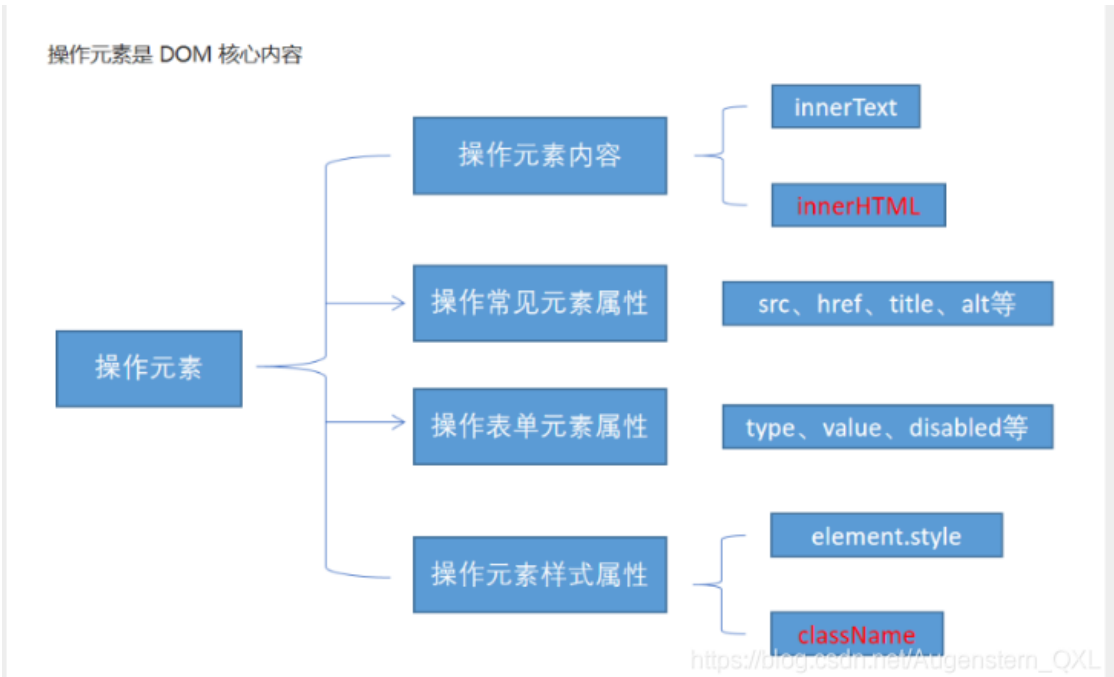
注意：

1. JS里面的样式采取驼峰命名法，比如 fontSize ， backgroundColor
2. JS 修改 style 样式操作 ，产生的是行内样式，CSS权重比较高
3. 如果样式修改较多，可以采取操作类名方式更改元素样式
4. class 因为是个保留字，因此使用 `className` 来操作元素类名属性
5. className 会直接更改元素的类名，会覆盖原先的类名

```
<body>
    <div class="first">文本</div>
    <script>
        // 1. 使用 element.style 获得修改元素样式  如果样式比较少 或者 功能简单的情况下使用
        var test = document.querySelector('div');
        test.onclick = function() {
            // this.style.backgroundColor = 'purple';
            // this.style.color = '#fff';
            // this.style.fontSize = '25px';
            // this.style.marginTop = '100px';
            // 让我们当前元素的类名改为了 change

            // 2. 我们可以通过 修改元素的className更改元素的样式 适合于样式较多或者功能复杂的情况
            // 3. 如果想要保留原先的类名，我们可以这么做 多类名选择器
            // this.className = 'change';
            this.className = 'first change';
        }
```

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

4.4、总结



4.5、排他思想

如果有同一组元素，我们相要某一个元素实现某种样式，需要用到循环的排他思想算法：

1. 所有元素全部清除样式（干掉其他人）
2. 给当前元素设置样式 （留下我自己）
3. 注意顺序不能颠倒，首先干掉其他人，再设置自己

```
<body>
    <button>按钮1</button>
    <button>按钮2</button>
    <button>按钮3</button>
    <button>按钮4</button>
    <button>按钮5</button>
    <script>
        // 1. 获取所有按钮元素
        var btns = document.getElementsByTagName('button');
        // btns得到的是伪数组  里面的每一个元素 btns[i]
        for (var i = 0; i < btns.length; i++) {
            btns[i].onclick = function() {
                // (1) 我们先把所有的按钮背景颜色去掉  干掉所有人
                for (var i = 0; i < btns.length; i++) {
                    btns[i].style.backgroundColor = '';
                }
                // (2) 然后才让当前的元素背景颜色为pink 留下我自己
                this.style.backgroundColor = 'pink';

            }
        }
        //2. 首先先排除其他人，然后才设置自己的样式 这种排除其他人的思想我们成为排他思想
    </script>
</body>
```

按钮1 按钮2 按钮3 按钮4 按钮5

4.6、自定义属性

4.6.1、获取属性值

- 获取内置属性值(元素本身自带的属性)

```
element.属性;
```

- 获取自定义的属性

```
element.getAttribute('属性');
```

4.6.2、设置属性值

- 设置内置属性值

```
element.属性 = '值';
```

- 主要设置自定义的属性

```
element.setAttribute('属性','值');
```

4.6.3、移除属性

```
element.removeAttribute('属性');
```

```
<body>
    <div id="demo" index="1" class="nav"></div>
    <script>
        var div = document.querySelector('div');
        // 1. 获取元素的属性值
        // (1) element.属性
        console.log(div.id);
        //(2) element.getAttribute('属性')  get得到获取 attribute 属性的意思 我们程序员自己添加的属性我们称为自定义属性 index
        console.log(div.getAttribute('id'));
        console.log(div.getAttribute('index'));
        // 2. 设置元素属性值
        // (1) element.属性= '值'
        div.id = 'test';
        div.className = 'navs';
        // (2) element.setAttribute('属性', '值');  主要针对于自定义属性
        div.setAttribute('index', 2);
        div.setAttribute('class', 'footer'); // class 特殊  这里面写的就是class 不是className
        // 3 移除属性 removeAttribute(属性)
        div.removeAttribute('index');
    </script>
</body>
```

4.7、H5自定义属性

自定义属性目的：

- 保存并保存数据，有些数据可以保存到页面中而不用保存到数据库中
- 有些自定义属性很容易引起歧义，不容易判断到底是内置属性还是自定义的，所以H5有了规定

4.7.1 设置H5自定义属性

H5规定自定义属性 `data-` 开头作为属性名并赋值

```
<div data-index = "1"></>
// 或者使用JavaScript设置
div.setAttribute('data-index',1);
```

4.7.2 获取H5自定义属性

- 兼容性获取 `element.getAttribute('data-index')`
- H5新增的： `element.dataset.index` 或 `element.dataset['index']` IE11才开始支持

```
<body>
    <div getTime="20" data-index="2" data-list-name="andy"></div>
    <script>
        var div = document.querySelector('div');
        console.log(div.getAttribute('getTime'));
        div.setAttribute('data-time', 20);
        console.log(div.getAttribute('data-index'));
        console.log(div.getAttribute('data-list-name'));
        // h5新增的获取自定义属性的方法 它只能获取data-开头的
        // dataset 是一个集合里面存放了所有以data开头的自定义属性只能获取data_开头的
        console.log(div.dataset);
        console.log(div.dataset.index);
        console.log(div.dataset['index']);
        // 如果自定义属性里面有多个-链接的单词，我们获取的时候采取 驼峰命名法
        console.log(div.dataset.listName);
        console.log(div.dataset['listName']);
    </script>
</body>
```

5、节点操作

获取元素通常使用两种方式：

1.利用DOM提供的方法获取元素	2.利用节点层级关系获取元素
document.getElementById()	利用父子兄节点关系获取元素
document.getElementsByTagName()	逻辑性强，但是兼容性较差
document.querySelector 等	
逻辑性不强，繁琐	

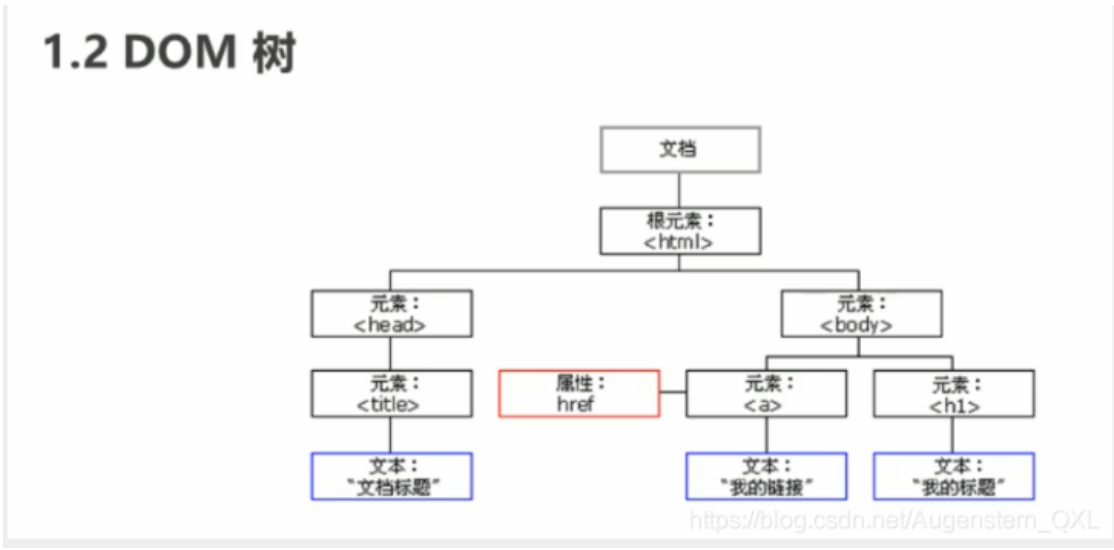
这两种方式都可以获取元素节点，我们后面都会使用，但是节点操作更简单

一般的，节点至少拥有三个基本属性

5.1、节点概述

网页中的所有内容都是节点（标签、属性、文本、注释等），在DOM 中，节点使用 node 来表示。

HTML DOM 树中的所有节点均可通过 JavaScript 进行访问，所有 HTML 元素（节点）均可被修改，也可以创建或删除。



一般的，节点至少拥有nodeType（节点类型）、nodeName（节点名称）和nodeValue（节点值）这三个基本属性。

- 元素节点：nodeType 为1
- 属性节点：nodeType 为2
- 文本节点：nodeType 为3(文本节点包括文字、空格、换行等)

我们在实际开发中，节点操作主要操作的是**元素节点**

利用 DOM 树可以把节点划分为不同的层级关系，常见的是**父子兄层级关系。**

5.2、父级节点

```
node.parentNode
```

- `parentNode` 属性可以返回某节点的父结点，注意是最近的一个父结点
- 如果指定的节点没有父结点则返回null

```
<body>
    <!-- 节点的优点 -->
    <div>我是div</div>
    <span>我是span</span>
    <ul>
        <li>我是li</li>
        <li>我是li</li>
        <li>我是li</li>
        <li>我是li</li>
    </ul>
    <div class="demo">
        <div class="box">
            <span class="erweima">×</span>
        </div>
    </div>

    <script>
        // 1. 父节点 parentNode
        var erweima = document.querySelector('.erweima');
        // var box = document.querySelector('.box');
        // 得到的是离元素最近的父级节点(亲爸爸) 如果找不到父节点就返回为 null
        console.log(erweima.parentNode);
    </script>
</body>
```

5.3、子结点

```
parentNode.childNodes(标准)
```

- `parentNode.childNodes` 返回包含指定节点的子节点的集合，该集合为即时更新的集合
- 返回值包含了所有的子结点，包括元素节点，文本节点等
- 如果只想要获得里面的元素节点，则需要专门处理。所以我们一般不提倡使用 `childNodes`

```
parentNode.children(非标准)
```

- `parentNode.children` 是一个只读属性，返回所有的子元素节点
- 它只返回子元素节点，其余节点不返回 （**这个是我们重点掌握的**）
- 虽然 children 是一个非标准，但是得到了各个浏览器的支持，因此我们可以放心使用

```
<body>
    <ul>
        <li>我是li</li>
        <li>我是li</li>
        <li>我是li</li>
```

```
        <li>我是li</li>
    </ul>
    <ol>
        <li>我是li</li>
        <li>我是li</li>
        <li>我是li</li>
        <li>我是li</li>
    </ol>
    <script>
        // DOM 提供的方法（API）获取
        var ul = document.querySelector('ul');
        var lis = ul.querySelectorAll('li');
        // 1. 子节点  childNodes 所有的子节点 包含 元素节点 文本节点等等
        console.log(ul.childNodes);
        console.log(ul.childNodes[0].nodeType);
        console.log(ul.childNodes[1].nodeType);
        // 2. children 获取所有的子元素节点 也是我们实际开发常用的
        console.log(ul.children);
    </script>
</body>
```

5.3.1、第一个子结点

```
parentNode.firstChild
```

- `firstChild` 返回第一个子节点，找不到则返回null
- 同样，也是包含所有的节点

5.3.2、最后一个子结点

```
parentNode.lastChild
```

- `lastChild` 返回最后一个子节点，找不到则返回null
- 同样，也是包含所有的节点

```
<body>
    <ol>
        <li>我是li1</li>
        <li>我是li2</li>
        <li>我是li3</li>
        <li>我是li4</li>
        <li>我是li5</li>
    </ol>
    <script>
        var ol = document.querySelector('ol');
        // 1. firstChild 第一个子节点 不管是文本节点还是元素节点
        console.log(ol.firstChild);
        console.log(ol.lastChild);
        // 2. firstElementChild 返回第一个子元素节点 ie9才支持
        console.log(ol.firstElementChild);
        console.log(ol.lastElementChild);
        // 3. 实际开发的写法  既没有兼容性问题又返回第一个子元素
        console.log(ol.children[0]);            //第一个子元素节点
        console.log(ol.children[ol.children.length - 1]);//最后一个子元素节点
    </script>
</body>
```

---

1. 我是li1
2. 我是li2
3. 我是li3
4. 我是li4
5. 我是li5

5.3.3、第一个子结点(兼容性)

```
parentNode.firstElementChild
```

- `firstElementChild` 返回第一个子节点，找不到则返回null

- 有兼容性问题，IE9以上才支持

5.3.4、最后一个子结点(兼容性)

```
parentNode.lastElementChild
```

- `lastElementChild` 返回最后一个子节点，找不到则返回null
- 有兼容性问题，IE9以上才支持

5.3.5、解决方案

实际开发中，firstChild 和 lastChild 包含其他节点，操作不方便，而 firstElementChild 和 lastElementChild 又有兼容性问题，那么我们如何获取第一个子元素节点或最后一个子元素节点呢?

> 解决方案

- 如果想要第一个子元素节点，可以使用 `parentNode.chilren[0]`
- 如果想要最后一个子元素节点，可以使用

```
// 数组元素个数减1 就是最后一个元素的索引号
parentNode.chilren[parentNode.chilren.length - 1]
```

- 示例：

```
<body>
    <ol>
        <li>我是li1</li>
        <li>我是li2</li>
        <li>我是li3</li>
        <li>我是li4</li>
    </ol>
    <script>
        var ol = document.querySelector('ol');
        // 1.firstChild 获取第一个子结点的，包含文本结点和元素结点
        console.log(ol.firstChild);
        // 返回的是文本结点 #text(第一个换行结点)

        console.log(ol.lastChild);
        // 返回的是文本结点 #text(最后一个换行结点)
        // 2. firstElementChild 返回第一个子元素结点
        console.log(ol.firstElementChild);
        // <li>我是li1</li>

        // 第2个方法有兼容性问题，需要IE9以上才支持
        // 3.实际开发中，既没有兼容性问题，又返回第一个子元素
        console.log(ol.children[0]);
        // <li>我是li1</li>
        console.log(ol.children[3]);
        // <li>我是li4</li>
        // 当里面li个数不唯一时候，需要取到最后一个结点时这么写
        console.log(ol.children[ol.children.length - 1]);
    </script>
</body>
```

5.4、兄弟节点

5.4.1、下一个兄弟节点

```
node.nextSibling
```

- `nextSibling` 返回当前元素的下一个兄弟元素节点，找不到则返回null
- 同样，也是包含所有的节点

5.4.2、上一个兄弟节点

```
node.previousSibling
```

- `previousSibling` 返回当前元素上一个兄弟元素节点，找不到则返回null
- 同样，也是包含所有的节点

5.4.3、下一个兄弟节点(兼容性)

```
node.nextElementSibling
```

- `nextElementSibling` 返回当前元素下一个兄弟元素节点，找不到则返回null
- 有兼容性问题，IE9才支持

5.4.4、上一个兄弟节点(兼容性)

```
node.previousElementSibling
```

- `previousElementSibling` 返回当前元素上一个兄弟元素节点，找不到则返回null
- 有兼容性问题，IE9才支持

示例

```
<body>
    <div>我是div</div>
    <span>我是span</span>
    <script>
        var div = document.querySelector('div');
        // 1.nextSibling 下一个兄弟节点 包含元素节点或者 文本节点等等
        console.log(div.nextSibling);       // #text
        console.log(div.previousSibling);   // #text
        // 2. nextElementSibling 得到下一个兄弟元素节点
        console.log(div.nextElementSibling);    //<span>我是span</span>
        console.log(div.previousElementSibling);//null
    </script>
</body>
```

如何解决兼容性问题？

答：自己封装一个兼容性的函数

```
function getNextElementSibling(element) {
    var el = element;
    while(el = el.nextSibling) {
        if(el.nodeType === 1){
            return el;
        }
    }
    return null;
}
```

5.5、创建节点

```
document.createElement('tagName');
```

- `document.createElement()` 方法创建由 tagName 指定的HTML 元素
- 因为这些元素原先不存在，是根据我们的需求动态生成的，所以我们也称为**动态创建元素节点**

5.5.1、添加节点

```
node.appendChild(child)
```

- `node.appendChild()` 方法将一个节点添加到指定父节点的子节点列表**末尾。**类似于 CSS 里面的 after 伪元素。

```
node.insertBefore(child,指定元素)
```
插入指定元素的前面

- `node.insertBefore()` 方法将一个节点添加到父节点的指定子节点**前面。**类似于 CSS 里面的 before 伪元素。

示例

```
<body>
    <ul>
        <li>123</li>
    </ul>
    <script>
        // 1. 创建节点元素节点
        var li = document.createElement('li');
        // 2. 添加节点 node.appendChild(child)  node 父级  child 是子级 后面追加元素  类似于数组中的push
        // 先获取父亲ul
        var ul = document.querySelector('ul');
        ul.appendChild(li);
        // 3. 添加节点 node.insertBefore(child, 指定元素);
        var lili = document.createElement('li');
        ul.insertBefore(lili, ul.children[0]);
        // 4. 我们想要页面添加一个新的元素分两步: 1. 创建元素 2. 添加元素
    </script>
</body>
```

5.5.2、删除节点

```
node.removeChild(child)
```

- `node.removeChild()` 方法从 DOM 中删除一个子节点，返回删除的节点

5.5.3、复制节点(克隆节点)

```
node.cloneNode()
```

- `node.cloneNode()` 方法返回调用该方法的节点的一个副本。 也称为克隆节点/拷贝节点
- 如果括号参数为空或者为 false ， 则是浅拷贝，即只克隆复制节点本身，不克隆里面的子节点

- 如果括号参数为 true ， 则是深度拷贝，会复制节点本身以及里面所有的子节点

示例

```
<body>
    <ul>
        <li>1111</li>
        <li>2</li>
        <li>3</li>
    </ul>
    <script>
        var ul = document.querySelector('ul');
        // 1. node.cloneNode(); 括号为空或者里面是false 浅拷贝 只复制标签不复制里面的内容
        // 2. node.cloneNode(true); 括号为true 深拷贝 复制标签复制里面的内容
        var lili = ul.children[0].cloneNode(true);
        ul.appendChild(lili);
    </script>
</body>
```

5.5.4、面试题

三种动态创建元素的区别 关键点：页面是否重绘

- doucument.write()
- element.innerHTML
- document.createElement()

区别:

- `document.write()` 是直接将内容写入页面的内容流，但是文档流执行完毕，则它会导致页面全部重绘会覆盖原先的内容
- `innerHTML` 是将内容写入某个 DOM 节点，不会导致页面全部重绘
- `innerHTML` 创建多个元素效率更高（不要拼接字符串，采取数组形式拼接），结构稍微复杂

```
<body>
    <div class="innner"></div>
    <div class="create"></div>
    <script>
        // 2. innerHTML 创建元素
        var inner = document.querySelector('.inner');
        // 2.1 innerHTML 用拼接字符串方法
        for (var i = 0; i <= 100; i++) {
            inner.innerHTML += '<a href="#">百度</a>';
        }
        // 2.2 innerHTML 用数组形式拼接
        var arr = [];
        for (var i = 0; i <= 100; i++) {
            arr.push('<a href="#">百度</a>');
        }
        inner.innerHTML = arr.join('');

        // 3.document.createElement() 创建元素
        var create = document.querySelector('.create');
        var a = document.createElement('a');
        create.appendChild(a);
    </script>
</body>
```

- `createElement()` 创建多个元素效率稍低一点点，但是结构更清晰

> 总结：不同浏览器下， innerHTML 效率要比 createElement 高

6、DOM核心 获取过来的DOM元素是一个对象（object），所以称为文档对象模型

对于DOM操作，我们主要针对子元素的操作，主要有

- 创建
- 增
- 删
- 改

- 查
- 属性操作
- 时间操作

6.1、创建

1. document.write
2. innerHTML
3. createElement

6.2、增

1. appendChild 在后面添加
2. insertBefore在前面添加

6.3、删

1. removeChild

6.4、改

- 主要修改dom的元素属性，dom元素的内容、属性、表单的值等

1. 修改元素属性：src、href、title 等
2. 修改普通元素内容：innerHTML、innerText
3. 修改表单元素：value、type、disabled
4. 修改元素样式：style、className

6.5、查

- 主要获取查询dom的元素

1. **DOM提供的API方法**：getElementById、getElementsByTagName (古老用法，不推荐)
2. **H5提供的新方法**：querySelector、querySelectorAll (提倡)
3. 利用节点操作获取元素：父(parentNode)、子(children)、兄(previousElementSibling、nextElementSibling) 提倡

6.6、属性操作

- 主要针对于自定义属性

1. setAttribute：设置dom的属性值
2. getAttribute：得到dom的属性值
3. removeAttribute：移除属性

7、事件高级

7.1、注册事件(绑定事件)

给元素添加事件，称为**注册事件**或者**绑定事件**。

注册事件有两种方式：**传统方式和方法监听注册方式**

传统注册方式	方法监听注册方式
利用 on 开头的事件 onclick	w3c 标准推荐方式
`<button onclick = "alert("hi")"></button>`	addEventListener() 它是一个方法
btn.onclick = function() {}	IE9 之前的 IE 不支持此方法，可使用 attachEvent() 代替
特点：注册事件的**唯一性**	特点：同一个元素同一个事件可以注册多个监听器
同一个元素同一个事件只能设置一个处理函数，最后注册的处理函数将会覆盖前面注册的处理函数	按注册顺序依次执行

①addEventListener事件监听方式

- `eventTarget.addEventListener()` 方法将指定的监听器注册到 eventTarget（目标对象）上
- 当该对象触发指定的事件时，就会执行事件处理函数

```
eventTarget.addEventListener(type,listener[,useCapture])
```

该方法接收三个参数：

- `type` :事件类型字符串，比如click,mouseover,**注意这里不要带on**
- `listener` ： 事件处理函数，事件发生时，会调用该监听函数
- `useCapture` ： 可选参数，是一个布尔值，默认是 false。学完 DOM 事件流后，我们再进一步学习

```
<body>
    <button>传统注册事件</button>
    <button>方法监听注册事件</button>
    <button>ie9 attachEvent</button>
    <script>
        var btns = document.querySelectorAll('button');
        // 1. 传统方式注册事件
        btns[0].onclick = function() {
            alert('hi');
        }
        btns[0].onclick = function() {
                alert('hao a u');
            }
            // 2. 事件监听注册事件 addEventListener
            // (1) 里面的事件类型是字符串 所以加引号 而且不带on
            // (2) 同一个元素 同一个事件可以添加多个侦听器（事件处理程序）
        btns[1].addEventListener('click', function() {
            alert(22);
        })
        btns[1].addEventListener('click', function() {
                alert(33);
            })
            // 3. attachEvent ie9以前的版本支持
        btns[2].attachEvent('onclick', function() {
            alert(11);
        })
    </script>
</body>
```

②attachEvent事件监听方式(兼容)

- `eventTarget.attachEvent()` 方法将指定的监听器注册到 eventTarget（目标对象） 上
- 当该对象触发指定的事件时，指定的回调函数就会被执行

```
eventTarget.attachEvent(eventNameWithOn,callback)
```

该方法接收两个参数：

- `eventNameWithOn`： 事件类型字符串，比如 onclick 、onmouseover ，这里要带 on
- `callback`： 事件处理函数，当目标触发事件时回调函数被调用
- ie9以前的版本支持

③注册事件兼容性解决方案

兼容性处理的原则：首先照顾大多数浏览器，再处理特殊浏览器

```
function addEventListener(element, eventName, fn) {
    // 判断当前浏览器是否支持 addEventListener 方法
    if (element.addEventListener) {
      element.addEventListener(eventName, fn);  // 第三个参数 默认是false
    } else if (element.attachEvent) {
      element.attachEvent('on' + eventName, fn);
    } else {
      // 相当于 element.onclick = fn;
      element['on' + eventName] = fn;
}

```

7.2、删除事件(解绑事件)

7.2.1、removeEventListener删除事件方式

```
eventTarget.removeEventListener(type,listener[,useCapture]);
```

该方法接收三个参数：

- `type` :事件类型字符串，比如click,mouseover,注意这里不要带on
- `listener`： 事件处理函数，事件发生时，会调用该监听函数
- `useCapture`： 可选参数，是一个布尔值，默认是 false。学完 DOM 事件流后，我们再进一步学习

7.2.2、detachEvent删除事件方式(兼容)

```
eventTarget.detachEvent(eventNameWithOn,callback);
```

该方法接收两个参数：

- `eventNameWithOn`： 事件类型字符串，比如 onclick 、onmouseover ，这里要带 on
- `callback`： 事件处理函数，当目标触发事件时回调函数被调用
- ie9以前的版本支持

7.2.3、传统事件删除方式

```
eventTarget.onclick = null;
```

事件删除示例：

```
<body>
    <div>1</div>
    <div>2</div>
    <div>3</div>
    <script>
        var divs = document.querySelectorAll('div');
        divs[0].onclick = function() {
            alert(11);
            // 1. 传统方式删除事件
            divs[0].onclick = null;
        }
        // 2.removeEventListener 删除事件
        divs[1].addEventListener('click',fn);   //里面的fn不需要调用加小括号

        function fn(){
            alert(22);
            divs[1].removeEventListener('click',fn);
        }
        // 3.IE9 中的删除事件方式
        divs[2].attachEvent('onclick',fn1);
        function fn1() {
            alert(33);
            divs[2].detachEvent('onclick',fn1);
        }
    </script>

</body>
```

7.2.4、删除事件兼容性解决方案

```
function removeEventListener(element, eventName, fn) {
    // 判断当前浏览器是否支持 removeEventListener 方法
    if (element.removeEventListener) {
      element.removeEventListener(eventName, fn);  // 第三个参数 默认是false
    } else if (element.detachEvent) {
      element.detachEvent('on' + eventName, fn);
    } else {
      element['on' + eventName] = null;
}

```

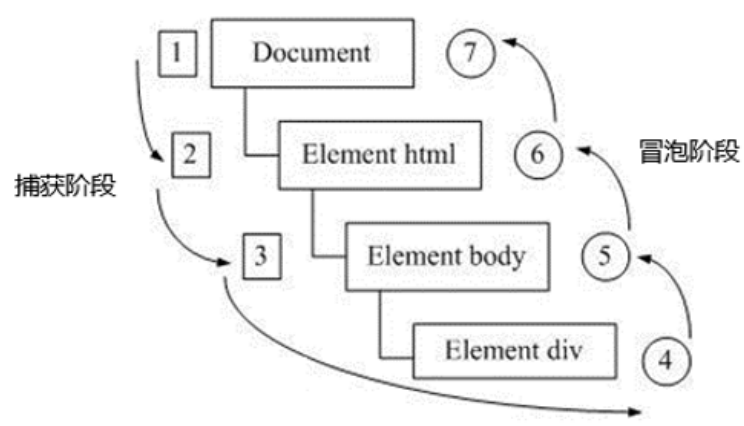
7.3、DOM事件流

- 事件流描述的是从页面中接收事件的顺序
- 事件发生时会在元素节点之间按照特定的顺序传播，这个传播过程即DOM事件流

比如我们给一个div 注册了点击事件：

DOM 事件流分为3个阶段：

1. 捕获阶段
2. 当前目标阶段
3. 冒泡阶段

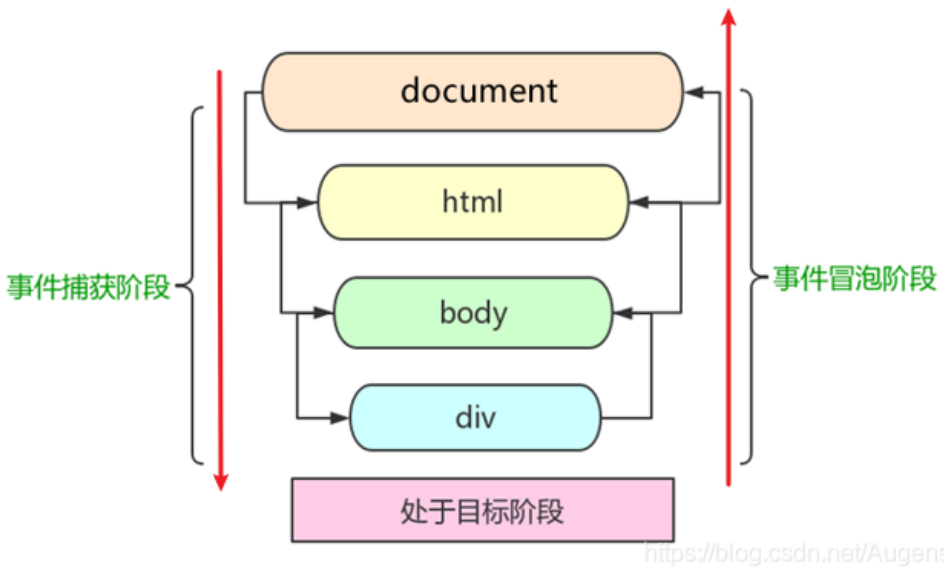


DOM事件流

- **事件冒泡**： IE 最早提出，事件开始时由最具体的元素接收，然后逐级向上传播到到 DOM 最顶层节点的过程。
- **事件捕获**： 网景最早提出，由 DOM 最顶层节点开始，然后逐级向下传播到到最具体的元素接收的过程。

加深理解：

我们向水里面扔一块石头，首先它会有一个下降的过程，这个过程就可以理解为从最顶层向事件发生的最具体元素（目标点）的捕获过程；之后会产生泡泡，会在最低点（ 最具体元素）之后漂浮到水面上，这个过程相当于事件冒泡。



7.3.1、捕获阶段

- document -> html -> body -> father -> son

两个盒子嵌套，一个父盒子一个子盒子，我们的需求是当点击父盒子时弹出 father ， 当点击子盒子时弹出 son

```
<body>
    <div class="father">
        <div class="son">son盒子</div>
    </div>
    <script>
        // dom 事件流 三个阶段
        // 1. JS 代码中只能执行捕获或者冒泡其中的一个阶段。
        // 2. onclick 和 attachEvent (ie)  只能得到冒泡阶段。
        // 3. 捕获阶段 如果addEventListener 第三个参数是 true 那么则处于捕获阶段  document -> html -> body -> father -> son
        var son = document.querySelector('.son');
        son.addEventListener('click', function() {
             alert('son');
        }, true);
        var father = document.querySelector('.father');
        father.addEventListener('click', function() {
            alert('father');
        }, true);
    </script>
</body>
```

从小到大是冒泡，从大到小是捕获

但是因为DOM流的影响，我们点击子盒子，会先弹出 father，之后再弹出 son

这是因为捕获阶段由 DOM 最顶层节点开始，然后逐级向下传播到到最具体的元素接收

- document -> html -> body -> father -> son
- 先看 document 的事件，没有；再看 html 的事件，没有；再看 body 的事件，没有；再看 father 的事件，有就先执行；再看 son 的事件，再执行。

7.3.2、冒泡阶段

- son -> father ->body -> html -> document

```
<body>
    <div class="father">
        <div class="son">son盒子</div>
    </div>
    <script>
        // 4. 冒泡阶段 如果addEventListener 第三个参数是 false 或者 省略 那么则处于冒泡阶段  son -> father ->body -> html -> document
        var son = document.querySelector('.son');
        son.addEventListener('click', function() {
            alert('son');
        }, false);
        var father = document.querySelector('.father');
        father.addEventListener('click', function() {
            alert('father');
        }, false);
        document.addEventListener('click', function() {
            alert('document');
        })
    </script>
</body>
```

我们点击子盒子，会弹出 son、father、document

这是因为冒泡阶段开始时由最具体的元素接收，然后逐级向上传播到到 DOM 最顶层节点

- son -> father ->body -> html -> document

7.3.3、小结

- JS 代码中只能执行捕获或者冒泡其中的一个阶段
- `onclick` 和 `attachEvent` 只能得到冒泡阶段

- addEventListener(type,listener[,useCapture]) 第三个参数如果是 true，表示在事件捕获阶段调用事件处理程序；如果是 false (不写默认就是false),表示在事件冒泡阶段调用事件处理程序
- 实际开发中我们很少使用事件捕获，我们更关注事件冒泡。
- 有些事件是没有冒泡的，比如 onblur、onfocus、onmouseenter、onmouseleave

7.4、事件对象 e或evt

```
eventTarget.onclick = function(event) {
   // 这个 event 就是事件对象，我们还喜欢的写成 e 或者 evt
}
eventTarget.addEventListener('click', function(event) {
   // 这个 event 就是事件对象，我们还喜欢的写成 e 或者 evt
})
```

- 官方解释：event 对象代表事件的状态，比如键盘按键的状态、鼠标的位置、鼠标按钮的状态
- 简单理解：
  - 事件发生后，跟事件相关的一系列信息数据的集合都放到这个对象里面
  - 这个对象就是事件对象 event，它有很多属性和方法，比如“
    - 谁绑定了这个事件
    - 鼠标触发事件的话，会得到鼠标的相关信息，如鼠标位置
    - 键盘触发事件的话，会得到键盘的相关信息，如按了哪个键
- 这个 event 是个形参，系统帮我们设定为事件对象，不需要传递实参过去
- 当我们注册事件时， event 对象就会被系统自动创建，并依次传递给事件监听器（事件处理函数）

```
<body>
    <div>123</div>
    <script>
        // 事件对象
        var div = document.querySelector('div');
        div.onclick = function(e) {
                // console.log(e);
                // console.log(window.event);
                // e = e || window.event;
                console.log(e);


            }
        // 1. event 就是一个事件对象 写到我们侦听函数的 小括号里面 当形参来看
        // 2. 事件对象只有有了事件才会存在，它是系统给我们自动创建的，不需要我们传递参数
        // 3. 事件对象 是 我们事件的一系列相关数据的集合 跟事件相关的 比如鼠标点击里面就包含了鼠标的相关信息，鼠标坐标啊，如果是键盘事件里面就包含的键盘事件的信息 比如 判断用户按下了那个键
        // 4. 这个事件对象我们可以自己命名 比如 event 、 evt、 e
        // 5. 事件对象也有兼容性问题 ie678 通过 window.event 兼容性的写法  e = e || window.event;
    </script>
</body>
```

7.4.1、事件对象的兼容性方案

事件对象本身的获取存在兼容问题：

1. 标准浏览器中是浏览器给方法传递的参数，只需要定义形参 e 就可以获取到。
2. 在 IE6~8 中，浏览器不会给方法传递参数，如果需要的话，需要到 window.event 中获取查找

解决：

```
e = e || window.event;
```

7.4.2、事件对象的常见属性和方法

事件对象属性方法	说明
e.target	返回触发事件的对象 标准
e.srcElement	返回触发事件的对象 非标准 ie6-8使用
e.type	返回事件的类型 比如 click mouseover 不带on
e.cancelBubble	该属性阻止冒泡，非标准，ie6-8使用
e.returnValue	该属性阻止默认行为 非标准，ie6-8使用
e.preventDefault()	该方法阻止默认行为 标准 比如不让链接跳转
e.stopPropagation()	阻止冒泡 标准

e.target 和 this 的区别：

- this 是事件绑定的元素， 这个函数的调用者（绑定这个事件的元素）
- e.target 是事件触发的元素。

7.5、事件对象阻止默认行为

```
<body>
    <div>123</div>
```

```
    <a href="http://www.baidu.com">百度</a>
    <form action="http://www.baidu.com">
        <input type="submit" value="提交" name="sub">
    </form>
    <script>
        // 常见事件对象的属性和方法
        // 1. 返回事件类型
        var div = document.querySelector('div');
        div.addEventListener('click', fn);
        div.addEventListener('mouseover', fn);
        div.addEventListener('mouseout', fn);

        function fn(e) {
            console.log(e.type);

        }
        // 2. 阻止默认行为（事件） 让链接不跳转 或者让提交按钮不提交
        var a = document.querySelector('a');
        a.addEventListener('click', function(e) {
                e.preventDefault(); //  dom 标准写法
            })
            // 3. 传统的注册方式
        a.onclick = function(e) {
            // 普通浏览器 e.preventDefault();  方法
            // e.preventDefault();
            // 低版本浏览器 ie678  returnValue  属性
            // e.returnValue;
            // 我们可以利用return false 也能阻止默认行为 没有兼容性问题 特点:  return 后面的代码不执行了,  而且只限于传统的注册方式
            return false;
            alert(11);
        }
    </script>
</body>
```

7.6、阻止事件冒泡 这是面试常见问题！！！！

事件冒泡：开始时由最具体的元素接收，然后逐级向上传播到到 DOM 最顶层节点

事件冒泡本身的特性，会带来的坏处，也会带来的好处，需要我们灵活掌握。

- 标准写法

```
e.stopPropagation();
```

- 非标准写法： IE6-8 利用对象事件 cancelBubble属性

```
e.cancelBubble = true;
```

```
<body>
    <div class="father">
        <div class="son">son儿子</div>
    </div>
    <script>
        // 常见事件对象的属性和方法
        // 阻止冒泡  dom 推荐的标准 stopPropagation()
        var son = document.querySelector('.son');
        son.addEventListener('click', function(e) {
            alert('son');
            e.stopPropagation(); // stop 停止  Propagation 传播
            e.cancelBubble = true; // 非标准 cancel 取消 bubble 泡泡
        }, false);

        var father = document.querySelector('.father');
        father.addEventListener('click', function() {
            alert('father');
        }, false);
        document.addEventListener('click', function() {
            alert('document');
        })
    </script>
</body>
```

7.6.1、阻止事件冒泡的兼容性解决方案

```
if(e && e.stopPropagation){
      e.stopPropagation();
  }else{
      window.event.cancelBubble = true;
  }
```

4.4.4 e.target 与 this

> e.target 与 this 的区别

- `this` 是事件绑定的元素，这个函数的调用者(绑定这个事件的元素)
- `e.target` 是事件触发的元素

```html
<body>
    <div>123</div>
    <ul>
        <li>abc</li>
        <li>abc</li>
        <li>abc</li>
    </ul>
    <script>
        // 常见事件对象的属性和方法
        // 1. e.target 返回的是触发事件的对象（元素）  this 返回的是绑定事件的对象（元素）
        // 区别 ： e.target 点击了那个元素，就返回那个元素 this 那个元素绑定了这个点击事件，那么就返回谁
        var div = document.querySelector('div');
        div.addEventListener('click', function(e) {
            console.log(e.target);
            console.log(this);

        })
        var ul = document.querySelector('ul');
        ul.addEventListener('click', function(e) {
                // 我们给ul 绑定了事件  那么this 就指向ul
                console.log(this);
                console.log(e.currentTarget);

                // e.target 指向我们点击的那个对象 谁触发了这个事件 我们点击的是li e.target 指向的就是li
                console.log(e.target);

            })
            // 了解兼容性
            // div.onclick = function(e) {
            //     e = e || window.event;
            //     var target = e.target || e.srcElement;
            //     console.log(target);

        // }
        // 2. 了解 跟 this 有个非常相似的属性 currentTarget  ie678不认识
    </script>
</body>
```

4.4.5 事件对象的兼容性

事件对象本身的获取存在兼容问题：

- 标准浏览器中浏览器是给方法传递的参数，只需定义形参e就可以获取到
- 在 IE6 -> 8 中，浏览器不会给方法传递参数，如果需要的话，需要到 `window.event` 中获取查找

> 解决方案

- `e = e || window.event`

```html
<body>
    <div>123</div>
    <script>
        // 事件对象
        var div = document.querySelector('div');
        div.onclick = function(e) {
                // e = e || window.event;
                console.log(e);
                // 事件对象也有兼容性问题 ie678 通过 window.event 兼容性的写法  e = e || window.event;

            }
</body>
```

7.7、事件委托

- 事件委托也称为事件代理，在 jQuery 里面称为事件委派
- 事件委托的原理面试重点！！！必须叙述出来
  - **不是每个子节点单独设置事件监听器，而是事件监听器设置在其父节点上，然后利用冒泡原理影响设置每个子节点**

就是把事件委托给了父节点，比如这个例子，我想要一点到li就冒出一句话，
但是li太多个了，一个个添加监听事件很麻烦，于是就给ul添加点击事件，利用冒泡影响

```html
<body>
    <ul>
        <li>知否知否，点我应有弹框在手！</li>
        <li>知否知否，点我应有弹框在手！</li>
        <li>知否知否，点我应有弹框在手！</li>
        <li>知否知否，点我应有弹框在手！</li>
        <li>知否知否，点我应有弹框在手！</li>
    </ul>
    <script>
        // 事件委托的核心原理：给父节点添加侦听器， 利用事件冒泡影响每一个子节点
        var ul = document.querySelector('ul');
        ul.addEventListener('click', function(e) {
            // alert('知否知否，点我应有弹框在手！');
            // e.target 这个可以得到我们点击的对象
            e.target.style.backgroundColor = 'pink';
            // 点了谁，就让谁的style里面的backgroundColor颜色变为pink
        })
    </script>
</body>
```

疑问：冒泡是从小到大，也就是从li到ul，
但是都没有给li添加任何事件，为什么点击了li会有反应呢？？？
答：虽然是从ul开始的事件冒泡,但是最内层元素也会执行，
顺序是死的，必须从最内层向外执行

以上案例：给 ul 注册点击事件，然后利用事件对象的 target 来找到当前点击的 li，因为点击 li，事件会冒泡到 ul 上， ul 有注册事件，就会触发事件监听器。

7.8、常见的鼠标事件

鼠标事件	触发条件
onclick	鼠标点击左键触发
onmouseover	鼠标经过触发
onmouseout	鼠标离开触发
onfocus	获得鼠标焦点触发
onblur	失去鼠标焦点触发
onmousemove	鼠标移动触发
onmouseup	鼠标弹起触发
onmousedown	鼠标按下触发

7.8.1、禁止鼠标右键与鼠标选中

- `contextmenu` 主要控制应该何时显示上下文菜单，主要用于程序员取消默认的上下文菜单
- `selectstart` 禁止鼠标选中

```
<body>
    <h1>我是一段不愿意分享的文字</h1>
    <script>
        // 1. contextmenu 我们可以禁用右键菜单
        document.addEventListener('contextmenu', function(e) {
                e.preventDefault(); // 阻止默认行为
            })
            // 2. 禁止选中文字 selectstart
        document.addEventListener('selectstart', function(e) {
            e.preventDefault();

        })
    </script>
</body>
```

7.8.2、鼠标事件对象

- **event**对象代表事件的状态，跟事件相关的一系列信息的集合
- 现阶段我们主要是用鼠标事件对象 **MouseEvent** 和键盘事件对象 **KeyboardEvent**。

鼠标事件对象	说明
e.clientX	返回鼠标相对于浏览器窗口**可视区**的X坐标
e.clientY	返回鼠标相对于浏览器窗口**可视区**的Y坐标
e.pageX（重点）	返回鼠标相对于文档页面的X坐标 IE9+ 支持
e.pageY（重点）	返回鼠标相对于文档页面的Y坐标 IE9+ 支持
e.screenX	返回鼠标相对于电脑屏幕的X坐标
e.screenY	返回鼠标相对于电脑屏幕的Y坐标

```
<body>
    <script>
        // 鼠标事件对象 MouseEvent
        document.addEventListener('click', function(e) {
            // 1. client 鼠标在可视区的x和y坐标
            console.log(e.clientX);
            console.log(e.clientY);
            console.log('----------------------');

            // 2. page 鼠标在页面文档的x和y坐标
            console.log(e.pageX);
            console.log(e.pageY);
            console.log('----------------------');

            // 3. screen 鼠标在电脑屏幕的x和y坐标
            console.log(e.screenX);
            console.log(e.screenY);

        })
    </script>
</body>
```

7.9、常用的键盘事件

键盘事件	触发条件
onkeyup	某个键盘按键被松开时触发
onkeydown	某个键盘按键被按下时触发

键盘事件	触发条件
onkeypress	某个键盘按键被按下时触发，但是它不识别功能键，比如 ctrl shift 箭头等

- **如果使用addEventListener 不需要加 on**
- `onkeypress` 和前面2个的区别是，它不识别功能键，比如左右箭头，shift 等
- 三个事件的执行顺序是： keydown – keypress — keyup

```
<body>
    <script>
        // 常用的键盘事件
        //1. keyup 按键弹起的时候触发
        // document.onkeyup = function() {
        //         console.log('我弹起了');

        //     }
        document.addEventListener('keyup', function() {
            console.log('我弹起了');
        })

        //3. keypress 按键按下的时候触发  不能识别功能键 比如 ctrl shift 左右箭头啊
        document.addEventListener('keypress', function() {
                console.log('我按下了press');
            })
            //2. keydown 按键按下的时候触发  能识别功能键 比如 ctrl shift 左右箭头啊
        document.addEventListener('keydown', function() {
                console.log('我按下了down');
            })
            // 4. 三个事件的执行顺序  keydown -- keypress -- keyup
    </script>
</body>
```

7.9.1、键盘对象属性

键盘事件对象 属性	说明
keyCode	返回该键值的ASCII值

- `onkeydown` 和 `onkeyup` 不区分字母大小写， `onkeypress` 区分字母大小写。
- 在我们实际开发中，我们更多的使用keydown和keyup， 它能识别所有的键（包括功能键）
- `Keypress` 不识别功能键，但是 `keyCode` 属性能区分大小写，返回不同的ASCII值

```
<body>
    <script>
        // 键盘事件对象中的keyCode属性可以得到相应键的ASCII码值
        // 1. 我们的keyup 和keydown事件不区分字母大小写  a 和 A 得到的都是65
        // 2. 我们的keypress 事件 区分字母大小写  a  97 和 A 得到的是65
        document.addEventListener('keyup', function(e) {
            console.log('up:' + e.keyCode);
            // 我们可以利用keycode返回的ASCII码值来判断用户按下了那个键
            if (e.keyCode === 65) {
                alert('您按下的a键');
            } else {
                alert('您没有按下a键')
            }

        })
        document.addEventListener('keypress', function(e) {
            console.log('press:' + e.keyCode);
        })
    </script>
</body>
```