Web APIs

目标：

- [ ] 能够使用removeChild()方法删除节点
- [ ] 能够完成动态生成表格案例
- [ ] 能够使用传统方式和监听方式给元素注册事件能够说出事件流执行的三个阶段
- [ ] 能够在事件处理函数中获取事件对象
- [ ] 能够使用事件对象取消默认行为
- [ ] 能够使用事件对象阻止事件冒泡
- [ ] 能够使用事件对象获取鼠标的位置
- [ ] 能够完成跟随鼠标的天使案例

1.1. 节点操作

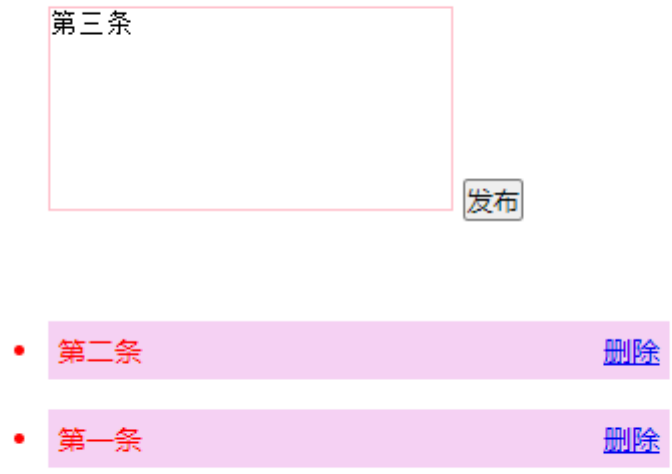
1.1.1 删除节点

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

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

```
    <button>删除</button>
    <ul>
        <li>熊大</li>
        <li>熊二</li>
        <li>光头强</li>
    </ul>
    <script>
        // 1.获取元素
        var ul = document.querySelector('ul');
        var btn = document.querySelector('button');
        // 2. 删除元素  node.removeChild(child)
        // ul.removeChild(ul.children[0]);
        // 3. 点击按钮依次删除里面的孩子
        btn.onclick = function() {
            if (ul.children.length == 0) {
                this.disabled = true;
            } else {
                ul.removeChild(ul.children[0]);
            }
        }
    </script>
```

1.1.2 案例：删除留言



分析：
① 当我们把文本域里面的值赋值给li 的时候，多添加一个删除的链接
② 需要把所有的链接获取过来，当我们点击当前的链接的时候，删除当前链接所在的li
③ 阻止链接跳转需要添加 javascript:void(0); 或者 javascript:;

```
    <textarea name="" id=""></textarea>
    <button>发布</button>
    <ul>

    </ul>
    <script>
        // 1. 获取元素
        var btn = document.querySelector('button');
        var text = document.querySelector('textarea');
        var ul = document.querySelector('ul');
        // 2. 注册事件
        btn.onclick = function() {
            if (text.value == '') {
                alert('您没有输入内容');
                return false;
            } else {
                // console.log(text.value);
                // (1) 创建元素
                var li = document.createElement('li');
                // 先有li 才能赋值
                li.innerHTML = text.value + "<a href='javascript:;'>删除</a>";
                // (2) 添加元素
                // ul.appendChild(li);
                ul.insertBefore(li, ul.children[0]);
                // (3) 删除元素 删除的是当前链接的li  它的父亲
                var as = document.querySelectorAll('a');
```

```
                for (var i = 0; i < as.length; i++) {
                    as[i].onclick = function() {
                        // 删除的是 li 当前a所在的li  this.parentNode;
                        ul.removeChild(this.parentNode);
                    }
                }
            }
        }
    </script>
```

1.1.3 复制（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

```
    <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>
```

注意：

1. 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。
2. 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。

1.1.4 案例：动态生成表格

案例：动态生成表格

姓名	科目	成绩	操作
魏璎珞	JavaScript	100	删除
弘历	JavaScript	90	删除
傅恒	JavaScript	99	删除
明玉	JavaScript	89	删除

分析：
① 因为里面的学生数据都是动态的，我们需要js 动态生成。 这里我们模拟数据，自己定义好数据。 数据我们采取对象形式存储。
② 所有的数据都是放到tbody里面的行里面。
③ 因为行很多，我们需要循环创建多个行（对应多少人）
④ 每个行里面又有很多单元格（对应里面的数据），我们还继续使用循环创建多个单元格，并且把数据存入里面（双重for循环）
⑤ 最后一列单元格是删除，需要单独创建单元格。
⑥ 最后添加删除操作，单击删除，可以删除当前行。

```
    <script>
        // 1.先去准备好学生的数据
        var datas = [{
            name: '魏璎珞',
            subject: 'JavaScript',
            score: 100
        }, {
            name: '弘历',
            subject: 'JavaScript',
            score: 98
        }, {
            name: '傅恒',
            subject: 'JavaScript',
            score: 99
        }, {
            name: '明玉',
            subject: 'JavaScript',
            score: 88
        }, {
            name: '大猪蹄子',
            subject: 'JavaScript',
            score: 0
        }];
        // 2. 往tbody 里面创建行： 有几个人（通过数组的长度）我们就创建几行
        var tbody = document.querySelector('tbody');
        // 遍历数组
        for (var i = 0; i < datas.length; i++) {
            // 1. 创建 tr行
            var tr = document.createElement('tr');
            tbody.appendChild(tr);
            // 2. 行里面创建单元格td 单元格的数量取决于每个对象里面的属性个数
            // 使用for in遍历学生对象
            for (var k in datas[i]) {
                // 创建单元格
                var td = document.createElement('td');
                // 把对象里面的属性值 datas[i][k] 给 td
                td.innerHTML = datas[i][k];
                tr.appendChild(td);
            }
            // 3. 创建有删除2个字的单元格
            var td = document.createElement('td');
            td.innerHTML = '<a href="javascript:;">删除 </a>';
            tr.appendChild(td);

        }
        // 4. 删除操作 开始
```

```
        var as = document.querySelectorAll('a');
        for (var i = 0; i < as.length; i++) {
            as[i].onclick = function() {
                // 点击a 删除 当前a 所在的行(链接的爸爸的爸爸)  node.removeChild(child)

                tbody.removeChild(this.parentNode.parentNode)
            }
        }
    </script>
```

1.1.5 创建元素的三种方式

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

区别

1. 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
2. innerHTML 是将内容写入某个 DOM 节点，不会导致页面全部重绘
3. 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
4. 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 creatElement 高

```
    <script>
        // 三种创建元素方式区别
        // 1. document.write() 创建元素  如果页面文档流加载完毕，再调用这句话会导致页面重
绘
         var btn = document.querySelector('button');
         btn.onclick = function() {
             document.write('<div>123</div>');
         }

        // 2. innerHTML 创建元素
        var inner = document.querySelector('.inner');
         for (var i = 0; i <= 100; i++) {
             inner.innerHTML += '<a href="#">百度</a>'
         }
        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');
        for (var i = 0; i <= 100; i++) {
            var a = document.createElement('a');
            create.appendChild(a);
        }
    </script>
```

1.1.6 innerTHML和createElement效率对比

innerHTML字符串拼接方式（效率低）

```
<script>
    function fn() {
        var d1 = +new Date();
        var str = '';
        for (var i = 0; i < 1000; i++) {
            document.body.innerHTML += '<div style="width:100px; height:2px;
border:1px solid blue;"></div>';
        }
        var d2 = +new Date();
        console.log(d2 - d1);
    }
    fn();
</script>
```

createElement方式（效率一般）

```
<script>
    function fn() {
        var d1 = +new Date();

        for (var i = 0; i < 1000; i++) {
            var div = document.createElement('div');
            div.style.width = '100px';
            div.style.height = '2px';
            div.style.border = '1px solid red';
            document.body.appendChild(div);
        }
        var d2 = +new Date();
        console.log(d2 - d1);
    }
    fn();
</script>
```

innerHTML数组方式（效率高）

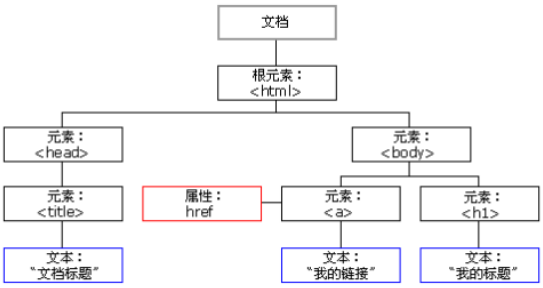
```
<script>
    function fn() {
        var d1 = +new Date();
        var array = [];
        for (var i = 0; i < 1000; i++) {
            array.push('<div style="width:100px; height:2px; border:1px solid
blue;"></div>');
        }
        document.body.innerHTML = array.join('');
        var d2 = +new Date();
        console.log(d2 - d1);
    }
    fn();
</script>
```

1.2. DOM的核心总结

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。

1. 对于JavaScript，为了能够使JavaScript操作HTML， JavaScript就有了一套自己的dom编程接口。
2. 对于HTML， dom使得html形成一棵dom树. 包含 文档、元素、节点



我们获取过来的DOM元素是一个对象（object），所以称为 文档对象模型

关于dom操作，我们主要针对于元素的操作。主要有创建、增、删、改、查、属性操作、事件操作。

1.2.1. 创建

1. document.write
2. innerHTML
3. createElement

1.2.2. 增加

1. appendChild
2. insertBefore

1.2.3. 删

1. removeChild

1.2.4. 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等
4. 修改元素样式： style、 className

1.2.5. 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 不太推荐
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

1.2.6. 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

1.2.7. 事件操作（重点）

给元素注册事件， 采取 事件源.事件类型 = 事件处理程序

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

1.3. 事件高级

1.3.1. 注册事件（2种方式）

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**

传统方式

- 利用 on 开头的事件 onclick
- `<button onclick="alert('hi~')"></button>`
- `btn.onclick = function() {}`
- 特点： 注册事件的唯一性
- 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

方法监听注册方式

- w3c 标准 推荐方式
- `addEventListener()` 它是一个方法
- IE9 之前的 IE 不支持此方法，可使用 attachEvent() 代替
- 特点：同一个元素同一个事件可以注册多个监听器
- 按注册顺序依次执行

1.3.2 事件监听

addEventListener()事件监听（IE9以后支持）

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

`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。

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

attacheEvent()事件监听（IE678支持）

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

`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。

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

注意： IE8 及早期版本支持

```
<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. 事件侦听注册事件 addEventListener
   // (1) 里面的事件类型是字符串 必定加引号 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>
```

事件监听兼容性解决方案

封装一个函数，函数中判断浏览器的类型：

```
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;
}
```

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

1.3.3. 删除事件（解绑事件）

传统注册方式

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

方法监听注册方式

```
1. eventTarget.removeEventListener(type, listener[, useCapture]);
2. eventTarget.detachEvent(eventNameWithOn, callback);
```

```
    <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. detachEvent
        divs[2].attachEvent('onclick', fn1);

        function fn1() {
            alert(33);
            divs[2].detachEvent('onclick', fn1);
        }
    </script>
```

删除事件兼容性解决方案

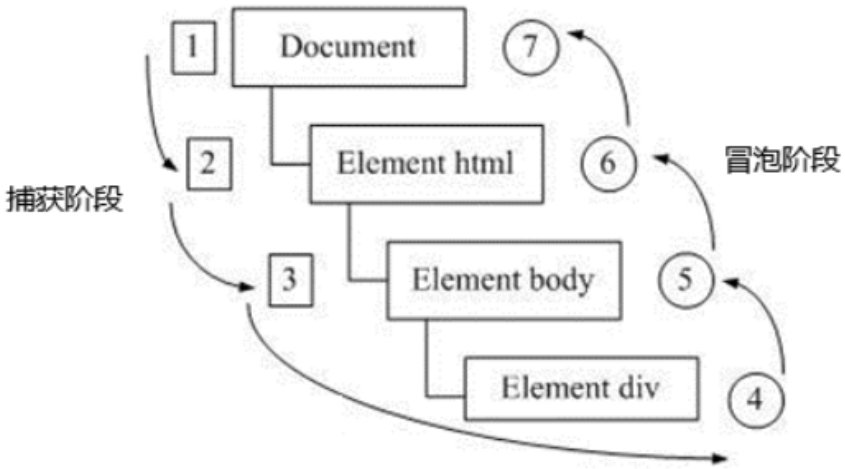
```
function removeEventListener(element, eventName, fn) {
    // 判断当前浏览器是否支持 removeEventListener 方法
    if (element.removeEventListener) {
        element.removeEventListener(eventName, fn); // 第三个参数 默认是false
    } else if (element.detachEvent) {
        element.detachEvent('on' + eventName, fn);
 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

1.3.4. DOM事件流

> html中的标签都是相互嵌套的，我们可以将元素想象成一个盒子装一个盒子，document是最外面的大盒子。
> 当你单击一个div时，同时你也单击了div的父元素，甚至整个页面。
>
> 那么是先执行父元素的单击事件，还是先执行div的单击事件 ？？？

- **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：我们给页面中的一个div注册了单击事件，当你单击了div时，也就单击了body，单击了html，单击了document。



DOM事件流

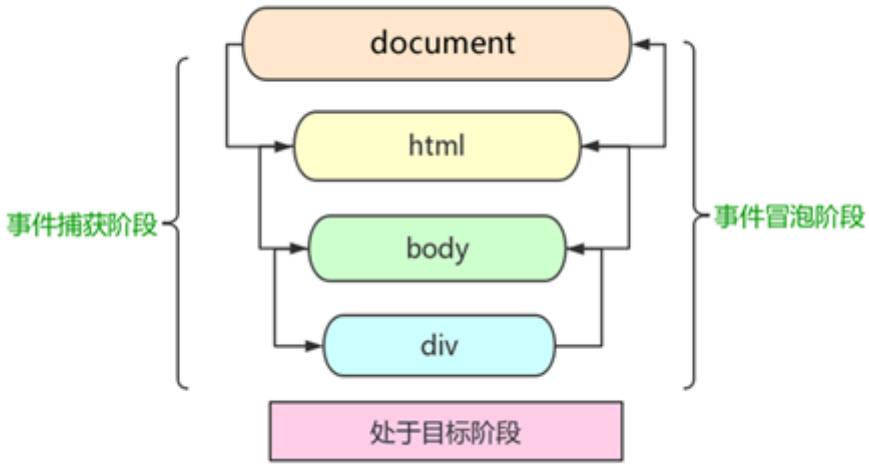
- 事件冒泡： IE 最早提出，事件开始时由最具体的元素接收，然后逐级向上传播到到 DOM 最顶层节点的过程。
- 事件捕获： 网景最早提出，由 DOM 最顶层节点开始，然后逐级向下传播到到最具体的元素接收的过程。

> IE 提出从目标元素开始，然后一层一层向外接收事件并响应，也就是冒泡型事件流。
> Netscape（网景公司）提出从最外层开始，然后一层一层向内接收事件并响应，也就是捕获型事件流。
>
> w3c 采用折中的方式，制定了统一的标准 ---- 先捕获再冒泡。
> 现代浏览器都遵循了此标准，所以当事件发生时，会经历3个阶段。

DOM 事件流会经历3个阶段：

1. 捕获阶段
2. 当前目标阶段
3. 冒泡阶段

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。



注意

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 只能得到冒泡阶段。
3. 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。
4. 实际开发中我们很少使用事件捕获，我们更关注事件冒泡。
5. 有些事件是没有冒泡的，比如 onblur、 onfocus、 onmouseenter、 onmouseleave。
6. 事件冒泡有时候会带来麻烦，有时候又会帮助很巧妙的做某些事件，我们后面讲解。

事件冒泡

```
    <div class="father">
        <div class="son">son盒子</div>
    </div>
    <script>
        // onclick 和 attachEvent（ie） 在冒泡阶段触发
        // 冒泡阶段 如果addEventListener 第三个参数是 false 或者 省略
        // son -> father ->body -> html -> document
        var son = document.querySelector('.son');
        // 给son注册单击事件
        son.addEventListener('click', function() {
            alert('son');
        }, false);
```

```
        // 给father注册单击事件
        var father = document.querySelector('.father');
        father.addEventListener('click', function() {
            alert('father');
        }, false);
        // 给document注册单击事件，省略第3个参数
        document.addEventListener('click', function() {
            alert('document');
        })
    </script>
```

事件捕获

```
    <div class="father">
        <div class="son">son盒子</div>
    </div>
    <script>
        // 如果addEventListener() 第三个参数是 true 那么在捕获阶段触发
        // document -> html -> body -> father -> son
         var son = document.querySelector('.son');
        // 给son注册单击事件，第3个参数为true
         son.addEventListener('click', function() {
             alert('son');
         }, true);
         var father = document.querySelector('.father');
        // 给father注册单击事件，第3个参数为true
         father.addEventListener('click', function() {
             alert('father');
         }, true);
        // 给document注册单击事件，第3个参数为true
        document.addEventListener('click', function() {
            alert('document');
        }, true)
    </script>
```

1.3.5. 事件对象

什么是事件对象

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。

比如：

1. 谁绑定了这个事件。
2. 鼠标触发事件的话，会得到鼠标的相关信息，如鼠标位置。
3. 键盘触发事件的话，会得到键盘的相关信息，如按了哪个键。

事件对象的使用

事件触发发生时就会产生事件对象，并且系统会以实参的形式传给事件处理函数。

所以，在事件处理函数中声明1个形参用来接收事件对象。

```
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
})

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） 。
```

事件对象的兼容性处理

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;
```

> 只要“||”前面为false, 不管“||”后面是true 还是 false，都返回 “||” 后面的值。
> 只要“||”前面为true, 不管“||”后面是true 还是 false，都返回 “||” 前面的值。

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

事件对象的属性和方法

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 阻止默认事件（默认行为） 非标准 ie6-8使用 比如不让链接跳转
e.preventDefault()	该方法 阻止默认事件（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（绑定这个事件处理函数的元素）。
- e.target 是事件触发的元素。

> 常情况下terget 和 this是一致的，
> 但有一种情况不同，那就是在事件冒泡时（父子元素有相同事件，单击子元素，父元素的事件处理函数也会被触发执行），
> 这时候this指向的是父元素，因为它是绑定事件的元素对象，
> 而target指向的是子元素，因为他是触发事件的那个具体元素对象。

```
<div>123</div>
<script>
    var div = document.querySelector('div');
    div.addEventListener('click', function(e) {
        // e.target 和 this指向的都是div
        console.log(e.target);
        console.log(this);

    });
</script>
```

事件冒泡下的e.target和this

```
<ul>
    <li>abc</li>
    <li>abc</li>
    <li>abc</li>
</ul>
<script>
    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); // ul

          // e.target 触发了事件的对象 我们点击的是li e.target 指向的就是li
          console.log(e.target); // li
    });
</script>
```

1.3.6 阻止默认行为

> html中一些标签有默认行为，例如a标签被单击后，默认会进行页面跳转。

```
<a href="http://www.baidu.com">百度</a>
<script>
    // 2. 阻止默认行为 让链接不跳转
    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 = false;
            // 我们可以利用return false 也能阻止默认行为 没有兼容性问题
            return false;
        }
    </script>
```

1.3.7 阻止事件冒泡

事件冒泡本身的特性，会带来的坏处，也会带来的好处。

- 标准写法：利用事件对象里面的 stopPropagation()方法

```
e.stopPropagation()
```

- 非标准写法： IE 6-8 利用事件对象 cancelBubble 属性

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

```
    <div class="father">
        <div class="son">son儿子</div>
    </div>
    <script>
        var son = document.querySelector('.son');
        // 给son注册单击事件
        son.addEventListener('click', function(e) {
            alert('son');
            e.stopPropagation(); // stop 停止  Propagation 传播
            window.event.cancelBubble = true; // 非标准 cancel 取消 bubble 泡泡
        }, false);

        var father = document.querySelector('.father');
        // 给father注册单击事件
        father.addEventListener('click', function() {
            alert('father');
        }, false);
        // 给document注册单击事件
        document.addEventListener('click', function() {
            alert('document');
        })
    </script>
```

阻止事件冒泡的兼容性处理

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

1.3.8 事件委托

事件冒泡本身的特性，会带来的坏处，也会带来的好处。

什么是事件委托

> 把事情委托给别人，代为处理。

事件委托也称为事件代理，在 jQuery 里面称为事件委派。

> 说白了就是，不给子元素注册事件，给父元素注册事件，把处理代码在父元素的事件中执行。

生活中的代理：

咱们班有100个学生， 快递员有100个快递， 如果一个个的送花费时间较长。同时每个学生领取的时候，也需
要排队领取，也花费时间较长，何如?

解决方案： 快递员把100个快递， **委托**给班主任，班主任把这些快递放到办公室，同学们下课自行领取即可。

优势： 快递员省事，委托给班主任就可以走了。 同学们领取也方便，因为相信班主任。

js事件中的代理：

```
<ul>
    <li>知否知否，应该有弹框在手</li>
    <li>知否知否，应该有弹框在手</li>
    <li>知否知否，应该有弹框在手</li>
    <li>知否知否，应该有弹框在手</li>
    <li>知否知否，应该有弹框在手</li>
</ul>
```

点击每个 li 都会弹出对话框，以前需要给每个 li 注册事件，是非常辛苦的， 而且访问 DOM 的次数越多，这就
会延长整个页面的交互就绪时间。

事件委托的原理

给父元素注册事件，利用事件冒泡，当子元素的事件触发，会冒泡到父元素，然后去控制相应的子元素。

事件委托的作用

- 我们只操作了一次 DOM ， 提高了程序的性能。
- 动态新创建的子元素，也拥有事件。

```
    <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) {
            // e.target 这个可以得到我们点击的对象
            e.target.style.backgroundColor = 'pink';
        })
    </script>
```

1.4. 常用鼠标事件

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

1.4.1 案例：禁止选中文字和禁止右键菜单

禁止鼠标右键菜单

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

```
document.addEventListener('contextmenu', function(e) {
    e.preventDefault();
})
```

禁止鼠标选中（ selectstart 开始选中)

```
document.addEventListener('selectstart', function(e) {
    e.preventDefault();
})
```

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

1.4.2 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	返回鼠标相对于文档页面的 X 坐标 IE9+ 支持
e.pageY	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

1.4.3 获取鼠标在页面的坐标

```
    <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>
```

1.4.4 案例：跟随鼠标的天使

案例： 跟随鼠标的天使

这个天使图片一直跟随鼠标移动

案例分析
① 鼠标不断的移动，使用鼠标移动事件： mousemove
② 在页面中移动，给document注册事件
③ 图片要移动距离，而且不占位置，我们使用绝对定位即可
④ 核心原理： 每次鼠标移动，我们都会获得最新的鼠标坐标， 把这个x和y坐标做为图片的top和left 值就可以移动图片

```
<img src="images/angel.gif" alt="">
<script>
    var pic = document.querySelector('img');
    document.addEventListener('mousemove', function(e) {
        // 1. mousemove只要我们鼠标移动1px 就会触发这个事件
        // 2.核心原理： 每次鼠标移动，我们都会获得最新的鼠标坐标，
        // 把这个x和y坐标做为图片的top和left 值就可以移动图片
        var x = e.pageX;
        var y = e.pageY;
        console.log('x坐标是' + x, 'y坐标是' + y);
        //3 . 千万不要忘记给left 和top 添加px 单位
        pic.style.left = x - 50 + 'px';
        pic.style.top = y - 40 + 'px';
    });
</script>
```