1.1. 节点操作

1.1.1 删除节点

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

> 删除节点

```
    <button>删除</button>
    <ul>
        <li>深圳</li>
        <li>广州</li>
        <li>北京</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>
```

> 删除留言案例

```
<style>
        * {
            margin: 0;
            padding: 0;
        }
        body {
            padding: 100px;
        }
        textarea {
            width: 200px;
            height: 100px;
            border: 1px solid pink;
            outline: none;
            resize: none;
        }
        ul {
            margin-top: 50px;
        }
```

```
        li {
            width: 300px;
            padding: 5px;
            background-color: rgb(245, 209, 243);
            color: red;
            font-size: 14px;
            margin: 15px 0;
        }
        li a {
            float: right;
        }
    </style>
</head>
<body>
    <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() {
                        // node.removeChild(child); 删除的是 li 当前a所在的li  this.parentNode;
                        ul.removeChild(this.parentNode);
                    }
                }
            }
        }
    </script>
</body>
```

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

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。

```
<ul>
    <li>深圳</li>
    <li>广州</li>
    <li>北京</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>
```

动态生成表格

```
<script>
    // 1.数据
    var datas = [
        {name: '郑小红',subject: 'JavaScript',score: 100},
        {name: '李四光',subject: 'vue',score: 98},
        {name: '张忠城',subject: 'es6',score: 99},
        {name: '王五',subject: 'react',score: 88},
        {name: '周小慧',subject: '小程序',score: 0}
    ];
    // 2. 往tbody 里面创建行：
    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datas对象
        for (var k in datas[i]) {
            // 创建td
            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  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绘
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

> 三种创建元素方式区别

```
    <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>
```

> 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
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、getElementByTagName(不推荐)
2.H5提供的新方法：querySelector、querySelectAll
3.利用节点操作获取元素：parentNode、children、previosSlibling、nextSibling、previosElementSibling、
nextElementSigling
```

1.2.6. 属性操作

```
主要针对定义属性
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

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 监听注册方式

传统注册方式

- 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

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

00attacheEvent()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

> 注册事件两种方式

```
<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;
}
```

1.3.3. 删除事件（解绑事件）

1. 传统注册方式

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

2. 方法监听注册方式

① `eventTarget.removeEventListener(type, listener[, useCapture]);`

② `eventTarget.detachEvent(eventNameWithOn, callback);`

> 删除事件

```
    <div>深圳</div>
    <div>广州</div>
    <div>北京</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;
    }
}
```

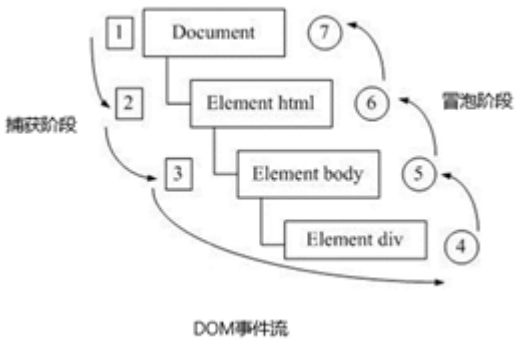
1.3.4. DOM事件流

> 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

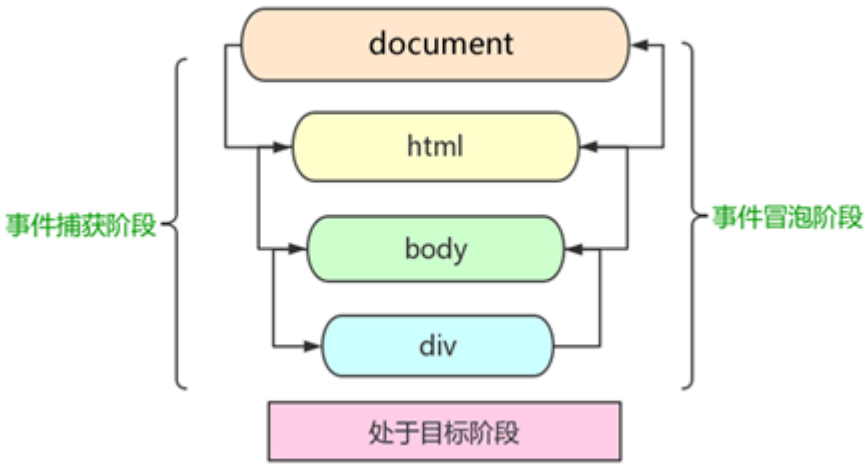
- 事件冒泡： IE 最早提出，事件开始时由最具体的元素接收，然后逐级向上传播到到 DOM 最顶层节点的过程。
- 事件捕获： 网景最早提出，由 DOM 最顶层节点开始，然后逐级向下传播到到最具体的元素接收的过程。

> 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。
> 最终，w3c 采用折中的方式，平息了战火，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。



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。

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。

> **DOM 事件流三个阶段**

```
<style>
    .box {
```

```
            overflow: hidden;
            width: 300px;
            height: 300px;
            margin: 100px auto;
            background-color: pink;
            text-align: center;
        }
        .one {
            width: 200px;
            height: 200px;
            margin: 50px;
            background-color: purple;
            line-height: 200px;
            color: #fff;
        }
    </style>
</head>
<body>
    <div class="box">
        <div class="one">one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 one = document.querySelector('.one');
        // one.addEventListener('click', function() {
        //     alert('one');
        // }, true);
        // var box = document.querySelector('.box');
        // box.addEventListener('click', function() {
        //     alert('box');
        // }, true);
        // 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 one = document.querySelector('.one');
        one.addEventListener('click', function() {
            alert('one');
        }, false);
        var box = document.querySelector('.box');
        box.addEventListener('click', function() {
            alert('father');
        }, false);
        document.addEventListener('click', function() {
            alert('document');
        })
    </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
})
eventTarget.addEventListener('click', fn)
function(event) {
  // 这个 event 就是事件对象，我们还喜欢的写成 e 或者 evt
}
```

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。
```

> **事件对象**

```
<style>
        div {
            width: 100px;
            height: 100px;
            background-color: pink;
        }
    </style>
</head>
<body>
```

```
    <div>小米</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);
            }
            // div.addEventListener('click', function(e) {
            //         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;
```

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。

> **事件对象e.target**

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

```
            height: 100px;
            background-color: pink;
        }
    </style>
</head>
<body>
    <div>小米</div>
    <ul>
        <li>京东</li>
        <li>淘宝</li>
        <li>唯品会</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
```

```
   <div>小米</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。

> **事件对象阻止默认行为**

```
    <div>搜狐</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>
```

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;
```

```
    <style>
        .box {
            overflow: hidden;
            width: 300px;
            height: 300px;
            margin: 100px auto;
            background-color: pink;
            text-align: center;
        }
        .one {
            width: 200px;
            height: 200px;
            margin: 50px;
            background-color: purple;
            line-height: 200px;
            color: #fff;
        }
    </style>
</head>
<body>
    <div class="box">
        <div class="one">你好</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 one = document.querySelector('.one');
        one.addEventListener('click', function(e) {
            alert('one');
            e.stopPropagation(); // stop 停止  Propagation 传播
            e.cancelBubble = true; // 非标准 cancel 取消 bubble 泡泡
        }, false);
        var box = document.querySelector('.box');
        box.addEventListener('click', function() {
            alert('father');
        }, false);

        document.addEventListener('click', function() {
```

```
            alert('document');
        })
    </script>
</body>
```

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。

> 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。

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>学习是一件很快乐的事情</li>
        <li>学习是一件很快乐的事情</li>
        <li>学习是一件很快乐的事情</li>
        <li>学习是一件很快乐的事情</li>
        <li>学习是一件很快乐的事情</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

1.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();
})
```

2.禁止鼠标选中（selectstart 开始选中）

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

> 常见鼠标事件

```
<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 事件对象是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>
```

> 跟随鼠标的天使

```
<style>
        img {
            position: absolute;
            top: 2px;
        }
    </style>
</head>
<body>
    <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
            // console.log(1);
            // 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>
</body>
```

1.1. 常用的键盘事件

1.1.1 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
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

注意：

1. 如果使用addEventListener 不需要加on

2. 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。

3. 三个事件的执行顺序是： keydown -- keypress --- keyup

> 常用的键盘事件

```
<script>
        // 常用的键盘事件
        //1. keyup 按键弹起的时候触发
        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>
```

1.1.2 键盘事件对象

键盘事件对象 属性	说明
keyCode	返回该键的ASCII 值

注意：

1) onkeydown 和 onkeyup 不区分字母大小写，onkeypress 区分字母大小写。

2）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）

3）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

> keyCode属性

```
    <script>
        // 键盘事件对象中的keyCode属性可以得到相应键的ASCII码值
        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(e);
            console.log('press:' + e.keyCode);
        })
    </script>
```

1.1.3 案例：模拟京东按键输入内容

当我们按下 s 键， 光标就定位到搜索框（文本框获得焦点）。

案例分析

① 核心思路： 检测用户是否按下了s 键，如果按下s 键，就把光标定位到搜索框里面
② 使用键盘事件对象里面的keyCode 判断用户按下的是否是s键
③ 搜索框获得焦点： 使用 js 里面的 focus() 方法

> 注意：触发获得焦点事件，可以使用 元素对象.focus()

```
    <input type="text">
    <script>
        // 获取输入框
        var search = document.querySelector('input');
        // 给document注册keyup事件
        document.addEventListener('keyup', function(e) {
            // 判断keyCode的值
            if (e.keyCode === 83) {
                // 触发输入框的获得焦点事件
                search.focus();
            }
        })
    </script>
```

1.1.4 案例：模拟京东快递单号查询

要求：当我们在文本框中输入内容时，文本框上面自动显示大字号的内容。

案例分析

① 快递单号输入内容时， 上面的大号字体盒子（con）显示(这里面的文字

② 同时把快递单号里面的值（value）获取过来赋值给 con盒子（innerText）做为内容

③ 如果快递单号里面内容为空，则隐藏大号字体盒子(con)盒子

④ 注意： keydown 和 keypress 在文本框里面的特点： 他们两个事件触发的时候，文字还没有落入文本框中。

⑤ keyup事件触发的时候， 文字已经落入文本框里面了

⑥ 当我们失去焦点，就隐藏这个con盒子

⑦ 当我们获得焦点，并且文本框内容不为空，就显示这个con盒子

```
<style>
    * {
        margin: 0;
        padding: 0;
    }
    .search {
        position: relative;
        width: 178px;
        margin: 100px;
    }
    .con {
        display: none;
        position: absolute;
        top: -40px;
        width: 171px;
        border: 1px solid rgba(0, 0, 0, .2);
        box-shadow: 0 2px 4px rgba(0, 0, 0, .2);
        padding: 5px 0;
        font-size: 18px;
        line-height: 20px;
        color: #333;
    }
```

```
        .con::before {
            content: '';
            width: 0;
            height: 0;
            position: absolute;
            top: 28px;
            left: 18px;
            border: 8px solid #000;
            border-style: solid dashed dashed;
            border-color: #fff transparent transparent;
        }
    </style>
</head>
<body>
    <div class="search">
        <div class="con">123</div>
        <input type="text" placeholder="请输入您的快递单号" class="jd">
    </div>
    <script>
        // 快递单号输入内容时， 上面的大号字体盒子（con）显示(这里面的字号更大）
        // 表单检测用户输入： 给表单添加键盘事件
        // 同时把快递单号里面的值（value）获取过来赋值给 con盒子（innerText）做为内容
        // 如果快递单号里面内容为空，则隐藏大号字体盒子(con)盒子
        var con = document.querySelector('.con');
        var jd_input = document.querySelector('.jd');
        jd_input.addEventListener('keyup', function() {
                // console.log('输入内容啦');
                if (this.value == '') {
                    con.style.display = 'none';
                } else {
                    con.style.display = 'block';
                    con.innerText = this.value;
                }
            })
            // 当我们失去焦点，就隐藏这个con盒子
        jd_input.addEventListener('blur', function() {
                con.style.display = 'none';
            })
            // 当我们获得焦点，就显示这个con盒子
        jd_input.addEventListener('focus', function() {
            if (this.value !== '') {
                con.style.display = 'block';
            }
        })
    </script>
```

1.2. BOM

1.2.1. 什么是BOM

BOM（Browser Object Model）即浏览器对象模型，它提供了独立于内容而与浏览器窗口进行交互的对象，其核心对象是 window。

BOM 由一系列相关的对象构成，并且每个对象都提供了很多方法与属性。

BOM 缺乏标准，JavaScript 语法的标准化组织是 ECMA，DOM 的标准化组织是 W3C，BOM 最初是Netscape 浏览器标准的一部分。

DOM

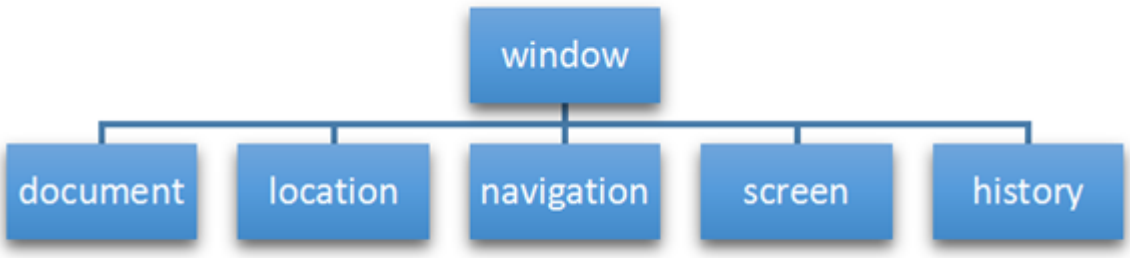
- 文档对象模型
- DOM 就是把「文档」当做一个「对象」来看待
- DOM 的顶级对象是 document
- DOM 主要学习的是操作页面元素
- DOM 是 W3C 标准规范

BOM

- 浏览器对象模型
- 把「浏览器」当做一个「对象」来看待
- BOM 的顶级对象是 window
- BOM 学习的是浏览器窗口交互的一些对象
- BOM 是浏览器厂商在各自浏览器上定义的，兼容性较差

1.2.2. BOM的构成

BOM 比 DOM 更大，它包含 DOM。



1.2.3. 顶级对象window

window 对象是浏览器的顶级对象，它具有双重角色。

1. 它是 JS 访问浏览器窗口的一个接口。
2. 它是一个全局对象。定义在全局作用域中的变量、函数都会变成 window 对象的属性和方法。

在调用的时候可以省略 window，前面学习的对话框都属于 window 对象方法，如 alert()、prompt() 等。

注意：window下的一个特殊属性 window.name

> BOM顶级对象window

```
<script>
        // window.document.querySelector()
        var num = 10;
        console.log(num);

        console.log(window.num);
```

```
        function fn() {
            console.log(11);
        }
        fn();
        window.fn();
        // alert(11);
        // window.alert(11)
        console.dir(window);
        // var name = 10;
        console.log(window.name);
    </script>
```

1.2.4. window对象的常见事件

页面（窗口）加载事件（2种）

第1种

```
window.onload = function(){}
或者
window.addEventListener("load",function(){});
```

window.onload 是窗口 (页面）加载事件，**当文档内容完全加载完成**会触发该事件(包括图像、脚本文件、CSS 文件等), 就调用的处理函数。

注意：

1. 有了 window.onload 就可以把 JS 代码写到页面元素的上方，因为 onload 是等页面内容全部加载完毕，再去执行处理函数。

2. window.onload 传统注册事件方式 只能写一次，如果有多个，会以最后一个 window.onload 为准。

3. 如果使用 `addEventListener` 则没有限制

第2种

```
document.addEventListener('DOMContentLoaded',function(){})
```

DOMContentLoaded 事件触发时，仅当DOM加载完成，不包括样式表，图片，flash等等。

IE9以上才支持！！！

如果页面的图片很多的话, 从用户访问到onload触发可能需要较长的时间, 交互效果就不能实现，必然影响用户的体验，此时用 DOMContentLoaded 事件比较合适。

```
    <script>
        // window.onload = function() {
        //     var btn = document.querySelector('button');
        //     btn.addEventListener('click', function() {
        //         alert('点击我');
        //     })
        // }
        // window.onload = function() {
        //     alert(22);
        // }
        window.addEventListener('load', function() {
            var btn = document.querySelector('button');
            btn.addEventListener('click', function() {
                alert('点击我');
            })
        })
        window.addEventListener('load', function() {
            alert(22);
        })
        document.addEventListener('DOMContentLoaded', function() {
                alert(33);
        })
            // load 等页面内容全部加载完毕，包含页面dom元素 图片 flash  css 等等
            // DOMContentLoaded 是DOM 加载完毕，不包含图片 falsh css 等就可以执行 加载速度比 load更快一些
    </script>
</head>
<body>
    <button>点击</button>
</body>
```

调整窗口大小事件

```
window.onresize = function(){}

window.addEventListener("resize",function(){});
```

window.onresize 是调整窗口大小加载事件, 当触发时就调用的处理函数。

注意：

1. 只要窗口大小发生像素变化，就会触发这个事件。
2. 我们经常利用这个事件完成响应式布局。 window.innerWidth 当前屏幕的宽度

> 调整窗口大小事件

```
    <script>
        // 注册页面加载事件
        window.addEventListener('load', function() {

            var div = document.querySelector('div');
```

```
            // 注册调整窗口大小事件
            window.addEventListener('resize', function() {
                // window.innerWidth 获取窗口大小
                console.log('变化了');
                if (window.innerWidth <= 800) {
                    div.style.display = 'none';
                } else {
                    div.style.display = 'block';
                }
            })
        })
    </script>
    <div></div>
```

1.2.5. 定时器（两种）

window 对象给我们提供了 2 个非常好用的方法-定时器。

- setTimeout()
- setInterval()

setTimeout() 炸弹定时器

开启定时器

```
window.setTimeout(调用函数, [延迟的毫秒数]);
```

setTimeout() 这个调用函数我们也称为回调函数 callback

注意：

1. window 可以省略。
2. 这个调用函数可以直接写函数，或者写函数名或者采取字符串 '函数名()'三种形式。第三种不推荐
3. 延迟的毫秒数省略默认是 0，如果写，必须是毫秒。
4. 因为定时器可能有很多，所以我们经常给定时器赋值一个标识符。

> 普通函数是按照代码顺序直接调用。
>
> 简单理解： 回调，就是回头调用的意思。上一件事干完，再回头再调用这个函数。
> 例如：定时器中的调用函数，事件处理函数，也是回调函数。
>
> 以前我们讲的 `element.onclick = function(){}` 或者 `element.addEventListener("click", fn);` 里面的 函数也是回调函数。

```
<script>
      // 1. setTimeout
      // 语法规范：  window.setTimeout(调用函数, 延时时间);
      // 1. 这个window在调用的时候可以省略
      // 2. 这个延时时间单位是毫秒 但是可以省略，如果省略默认的是0
      // 3. 这个调用函数可以直接写函数 还可以写 函数名 还有一个写法 '函数名()'
      // 4. 页面中可能有很多的定时器，我们经常给定时器加标识符 （名字）
      // setTimeout(function() {
      //     console.log('时间到了');
      // }, 2000);
      function callback() {
          console.log('爆炸了');
      }
      var timer1 = setTimeout(callback, 3000);
      var timer2 = setTimeout(callback, 5000);
      // setTimeout('callback()', 3000); // 我们不提倡这个写法
  </script>
```

```
    <script>
        // 回调函数是一个匿名函数
         setTimeout(function() {
             console.log('时间到了');

         }, 2000);
        function callback() {
            console.log('爆炸了');
        }
        // 回调函数是一个有名函数
        var timer1 = setTimeout(callback, 3000);
        var timer2 = setTimeout(callback, 5000);
    </script>
```

> **5秒之后自定关闭的广告**

```
<body>
    <img src="images/ad.jpg" alt="" class="ad">
    <script>
        // 获取要操作的元素
        var ad = document.querySelector('.ad');
        // 开启定时器
        setTimeout(function() {
            ad.style.display = 'none';
        }, 5000);
    </script>
</body>
```

停止定时器

```
window.clearTimeout(timeoutID)
```

clearTimeout()方法取消了先前通过调用 setTimeout() 建立的定时器。

注意：

1. window 可以省略。

2. 里面的参数就是定时器的标识符。

> **点击停止定时器**

```
<button>点击停止定时器</button>
    <script>
        var btn = document.querySelector('button');
        var timer = setTimeout(function() {
            console.log('爆炸了');
        }, 5000);
        btn.addEventListener('click', function() {
            clearTimeout(timer);
        })
    </script>
```

setInterval() 闹钟定时器

开启定时器

```
window.setInterval(回调函数, [间隔的毫秒数]);
```

setInterval() 方法重复调用一个函数，每隔这个时间，就去调用一次回调函数。

注意：

1. window 可以省略。

2. 这个调用函数可以直接写函数，或者写函数名或者采取字符串 '函数名()' 三种形式。

3. 间隔的毫秒数省略默认是 0，如果写，必须是毫秒，表示每隔多少毫秒就自动调用这个函数。

4.因为定时器可能有很多，所以我们经常给定时器赋值一个标识符。

5. 第一次执行也是间隔毫秒数之后执行，之后每隔毫秒数就执行一次。

```
<script>
        // 1. setInterval
        // 语法规范：  window.setInterval(调用函数, 延时时间);
        setInterval(function() {
            console.log('继续输出')
        }, 1000);
        // 2. setTimeout  延时时间到了，就去调用这个回调函数，只调用一次 就结束了这个定时器
        // 3. setInterval  每隔这个延时时间，就去调用这个回调函数，会调用很多次，重复调用这个函数
    </script>
```

```
    <style>
        div {
            margin: 200px;
        }
        span {
            display: inline-block;
            width: 40px;
            height: 40px;
            background-color: #333;
            font-size: 20px;
            color: #fff;
            text-align: center;
            line-height: 40px;
        }
    </style>
</head>
<body>
    <div>
        <span class="hour">1</span>
        <span class="minute">2</span>
        <span class="second">3</span>
    </div>
    <script>
        // 1. 获取元素
        var hour = document.querySelector('.hour'); // 小时的黑色盒子
        var minute = document.querySelector('.minute'); // 分钟的黑色盒子
        var second = document.querySelector('.second'); // 秒数的黑色盒子
        var inputTime = +new Date('2020-12-1 18:00:00'); // 返回的是用户输入时间总的毫秒数
        countDown(); // 我们先调用一次这个函数，防止第一次刷新页面有空白
        // 2. 开启定时器
        setInterval(countDown, 1000);
        function countDown() {
            var nowTime = +new Date(); // 返回的是当前时间总的毫秒数
            var times = (inputTime - nowTime) / 1000; // times是剩余时间总的秒数
            var h = parseInt(times / 60 / 60 % 24); //时
            h = h < 10 ? '0' + h : h;
            hour.innerHTML = h; // 把剩余的小时给 小时黑色盒子
            var m = parseInt(times / 60 % 60); // 分
            m = m < 10 ? '0' + m : m;

            minute.innerHTML = m;
```

```
            var s = parseInt(times % 60); // 当前的秒
            s = s < 10 ? '0' + s : s;
            second.innerHTML = s;
        }
    </script>
</body>
```

停止定时器

```
window.clearInterval(intervalID);
```

clearInterval()方法取消了先前通过调用 setInterval()建立的定时器。

注意:

1. window 可以省略。

2. 里面的参数就是定时器的标识符。

```
<button class="begin">开启定时器</button>
    <button class="stop">停止定时器</button>
    <script>
        var begin = document.querySelector('.begin');
        var stop = document.querySelector('.stop');
        var timer = null; // 全局变量  null是一个空对象
        begin.addEventListener('click', function() {
            timer = setInterval(function() {
                console.log('ni hao ma');
            }, 1000);
        })
        stop.addEventListener('click', function() {
            clearInterval(timer);
        })
    </script>
```

发送短信倒计时

```
    手机号码： <input type="number"> <button>发送</button>
    <script>
        var btn = document.querySelector('button');
        // 全局变量，定义剩下的秒数
        var time = 3;
        // 注册单击事件
        btn.addEventListener('click', function() {
            // 禁用按钮
            btn.disabled = true;
            // 开启定时器
            var timer = setInterval(function() {

                // 判断剩余秒数
```

```
                if (time == 0) {
                    // 清除定时器和复原按钮
                    clearInterval(timer);
                    btn.disabled = false;
                    btn.innerHTML = '发送';
                } else {
                    btn.innerHTML = '还剩下' + time + '秒';
                    time--;
                }
            }, 1000);
        });
    </script>
```

1.2.6. this指向问题

this的指向在函数定义的时候是确定不了的，只有函数执行的时候才能确定this到底指向谁，一般情况下this的最终指向的是那个调用它的对象。

现阶段，我们先了解一下几个this指向

1. 全局作用域或者普通函数中this指向全局对象window（注意定时器里面的this指向window）
2. 方法调用中谁调用this指向谁
3. 构造函数中this指向构造函数的实例

```
    <button>点击</button>
    <script>
        // this 指向问题 一般情况下this的最终指向的是那个调用它的对象
        // 1. 全局作用域或者普通函数中this指向全局对象window（ 注意定时器里面的this指向window）
        console.log(this);
        function fn() {
            console.log(this);
        }
        window.fn();
        window.setTimeout(function() {
            console.log(this);
        }, 1000);
        // 2. 方法调用中谁调用this指向谁
        var o = {
            sayHi: function() {
                console.log(this); // this指向的是 o 这个对象
            }
        }
        o.sayHi();
        var btn = document.querySelector('button');
        btn.addEventListener('click', function() {
                console.log(this); // 事件处理函数中的this指向的是btn这个按钮对象
            })
        // 3. 构造函数中this指向构造函数的实例
        function Fun() {
            console.log(this); // this 指向的是fun 实例对象
        }
        var fun = new Fun();
    </script>
```

> js 执行队列

```
<script>
    // 第一个问题
    // console.log(1);
    // setTimeout(function() {
    //     console.log(3);
    // }, 1000);
    // console.log(2);
    // 2. 第二个问题
    // console.log(1);
    // setTimeout(function() {
    //     console.log(3);
    // }, 0);
    // console.log(2);
    // 3. 第三个问题
    console.log(1);
    document.onclick = function() {
        console.log('click');
    }
    console.log(2);
    setTimeout(function() {
        console.log(3)
    }, 3000)
</script>
```

> 5秒钟之后跳转页面

```
<button>点击</button>
 <div></div>
 <script>
    var btn = document.querySelector('button');
    var div = document.querySelector('div');
    btn.addEventListener('click', function() {
        // console.log(location.href);
        location.href = 'http://www.itcast.cn';
    })
    var timer = 5;
    setInterval(function() {
        if (timer == 0) {
            location.href = 'http://www.itcast.cn';
        } else {
            div.innerHTML = '您将在' + timer + '秒钟之后跳转到首页';
            timer--;
        }
    }, 1000);
</script>
```

1.2.7. location对象

什么是 location 对象

window 对象给我们提供了一个 location 属性用于获取或设置窗体的 URL，并且可以用于解析 URL 。 因为这个属性返回的是一个对象，所以我们将这个属性也称为 location 对象。

URL

统一资源定位符 (Uniform Resource Locator, URL) 是互联网上标准资源的地址。互联网上的每个文件都有一个唯一的 URL，它包含的信息指出文件的位置以及浏览器应该怎么处理它。

URL 的一般语法格式为：

```
protocol://host[:port]/path/[?query]#fragment

http://www.itcast.cn/index.html?name=andy&age=18#link
```

组成	说明
protocol	通信协议 常用的http,ftp,maito等
host	主机（域名） www.itheima.com
port	端口号 可选，省略时使用方案的默认端口 如http的默认端口为80
path	路径 由 零或多个'/'符号隔开的字符串，一般用来表示主机上的一个目录或文件地址
query	参数 以键值对的形式,通过 & 符号分隔开来
fragment	片段 #后面内容 常见于链接 锚点

location 对象的属性

location对象属性	返回值
location.href	获取或者设置 整个URL
location. host	返回主机（域名） www.itheima.com
location.port	返回端口号 如果未写返回 空字符串
location.pathname	返回路径
location. search	返回参数
location. hash	返回片段 #后面内容 常见于链接 锚点

重点记住： href 和 search

案例：5分钟自动跳转页面

案例分析

① 利用定时器做倒计时效果

② 时间到了，就跳转页面。 使用 location.href

```
    <button>点击</button>
    <div></div>
    <script>
        var btn = document.querySelector('button');
        var div = document.querySelector('div');
        btn.addEventListener('click', function() {
            // console.log(location.href);
            location.href = 'http://www.itcast.cn';
        })
        var timer = 5;
        setInterval(function() {
            if (timer == 0) {
                location.href = 'http://www.itcast.cn';
            } else {
                div.innerHTML = '您将在' + timer + '秒钟之后跳转到首页';
                timer--;
            }
        }, 1000);
    </script>
```

获取URL参数

> login.html

```
<form action="index.html">
        用户名： <input type="text" name="uname">
        <input type="submit" value="登录">
    </form>
```

> 获取URL参数

```
    <div></div>
    <script>
        console.log(location.search); // ?uname=andy
        // 1.先去掉？  substr('起始的位置'，截取几个字符);
        var params = location.search.substr(1); // uname=andy
        console.log(params);
        // 2. 利用=把字符串分割为数组 split('=');
        var arr = params.split('=');
        console.log(arr); // ["uname", "ANDY"]
        var div = document.querySelector('div');
        // 3.把数据写入div中
        div.innerHTML = arr[1] + '欢迎您';
    </script>
```

location对象的常见方法

location对象方法	返回值
location.assign()	跟 href 一样，可以跳转页面（也称为重定向页面）
location.replace()	替换当前页面，因为不记录历史，所以不能后退页面
location.reload()	重新加载页面，相当于刷新按钮或者 f5 如果参数为true 强制刷新 ctrl+f5

```
    <button>点击</button>
    <script>
        var btn = document.querySelector('button');
        btn.addEventListener('click', function() {
            // 记录浏览历史，所以可以实现后退功能
            // location.assign('http://www.itcast.cn');
            // 不记录浏览历史，所以不可以实现后退功能
            // location.replace('http://www.itcast.cn');
            location.reload(true);
        })
    </script>
```

1.2.8. navigator对象

navigator 对象包含有关浏览器的信息，它有很多属性，我们最常用的是 userAgent，该属性可以返回由客户机发送服务器的 user-agent 头部的值。

下面前端代码可以判断用户那个终端打开页面，实现跳转

```
if((navigator.userAgent.match(/(phone|pad|pod|iPhone|iPod|ios|iPad|Android|Mobile|BlackBerry|IEMobile|MQQBrowser|JUC|Fennec|wOSBrowser|BrowserNG|WebOS|Symbian|Windows Phone)/i))) {
    window.location.href = "";     //手机
 } else {
    window.location.href = "";     //电脑
 }
```

1.2.9 history对象

window对象给我们提供了一个 history对象，与浏览器历史记录进行交互。该对象包含用户（在浏览器窗口中）访问过的URL。

history对象方法	作用
back()	可以后退功能
forward()	前进功能
go(参数)	前进后退功能 参数如果是 1 前进1个页面 如果是-1 后退1个页面

history对象一般在实际开发中比较少用，但是会在一些 OA 办公系统中见到。

1.3. JS执行机制

以下代码执行的结果是什么？

```
console.log(1);
setTimeout(function () {
    console.log(3);
}, 1000);
console.log(2);
```

以下代码执行的结果是什么？

```
console.log(1);
setTimeout(function () {
    console.log(3);
}, 0);
console.log(2);
```

1.3.1 JS 是单线程

JavaScript 语言的一大特点就是单线程，也就是说，同一个时间只能做一件事。这是因为 Javascript 这门脚本语言诞生的使命所致——JavaScript 是为处理页面中用户的交互，以及操作 DOM 而诞生的。比如我们对某个 DOM 元素进行添加和删除操作，不能同时进行。 应该先进行添加，之后再删除。

> 单线程就意味着，所有任务需要排队，前一个任务结束，才会执行后一个任务。如果前一个任务耗时很长，后一个任务就不得不一直等着。
>
> 这样所导致的问题是： 如果 JS 执行的时间过长，这样就会造成页面的渲染不连贯，导致页面渲染加载阻塞的感觉。

1.3.2 同步任务和异步任务

单线程导致的问题就是后面的任务等待前面任务完成，如果前面任务很耗时（比如读取网络数据），后面任务不得不一直等待！！

为了解决这个问题，利用多核 CPU 的计算能力，HTML5 提出 Web Worker 标准，允许 JavaScript 脚本创建多个线程，但是子线程完全受主线程控制。于是，JS 中出现了**同步任务**和**异步任务**。

同步

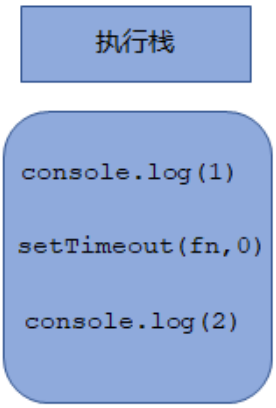
前一个任务结束后再执行后一个任务，程序的执行顺序与任务的排列顺序是一致的、同步的。比如做饭的同步做法：我们要烧水煮饭，等水开了（10分钟之后），再去切菜，炒菜。

异步

你在做一件事情时，因为这件事情会花费很长时间，在做这件事的同时，你还可以去处理其他事情。比如做饭的异步做法，我们在烧水的同时，利用这10分钟，去切菜，炒菜。

他们的本质区别： 这条流水线上各个流程的执行顺序不同。

> JS中所有任务可以分成两种，一种是同步任务（synchronous），另一种是异步任务（asynchronous）。
>
> 同步任务指的是：
> 在主线程上排队执行的任务，只有前一个任务执行完毕，才能执行后一个任务；
> 异步任务指的是：
> 不进入主线程、而进入"任务队列"的任务，当主线程中的任务运行完了，才会从"任务队列"取出异步任务放入主线程执行。



同步任务

同步任务都在主线程上执行，形成一个**执行栈。**

异步任务

JS 的异步是通过回调函数实现的。

一般而言，异步任务有以下三种类型:

1、普通事件，如 click、resize 等

2、资源加载，如 load、error 等

3、定时器，包括 setInterval、setTimeout 等

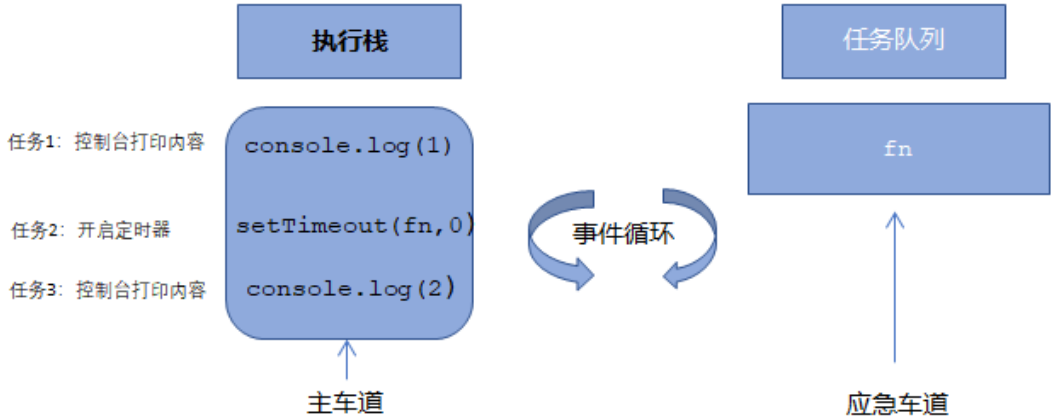
异步任务相关回调函数添加到**任务队列**中（任务队列也称为消息队列）。

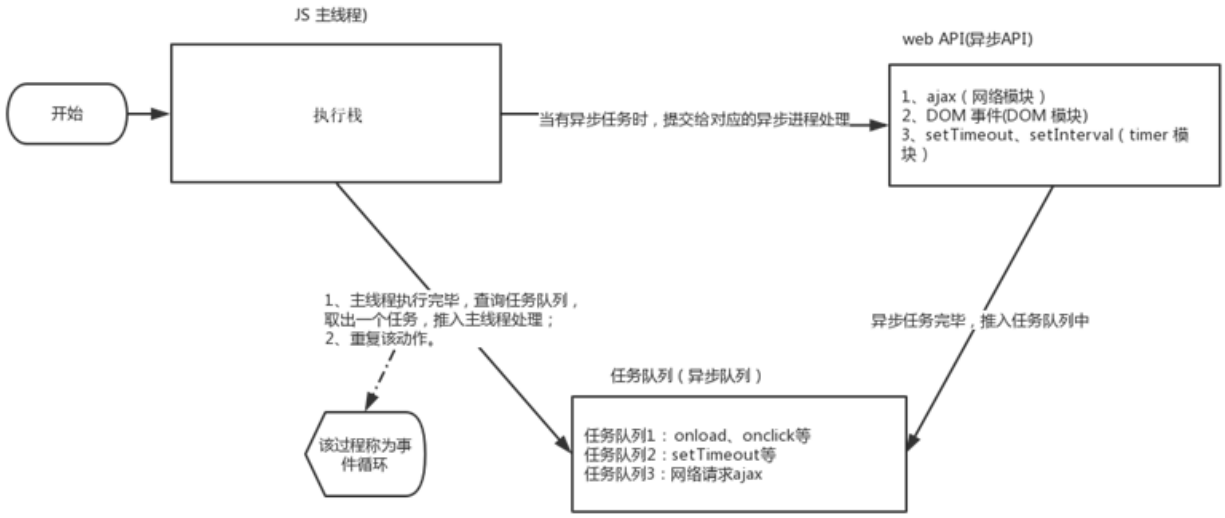
1.3.3 JS执行机制（事件循环）

1. 先执行执行栈中的同步任务。

2. 异步任务（回调函数）放入任务队列中。

3. 一旦执行栈中的所有同步任务执行完毕，系统就会按次序读取任务队列中的异步任务，于是被读取的异步任务结束等待状态，进入执行栈，开始执行。





由于主线程不断的重复获得任务、执行任务、再获取任务、再执行，所以这种机制被称为事件循环（event loop）。

1.3.4 代码思考题

```
console.log(1);
document.onclick = function() {
  console.log('click');
}

setTimeout(function() {
  console.log(3)
}, 3000)
console.log(2);
```