#DOMday01:

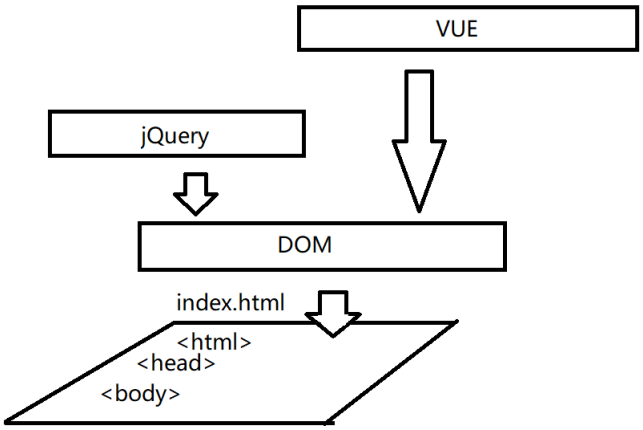
1. 什么是 DOM
2. DOM 树
3. 查找元素
4. 事件绑定基础
5. 示例: 购物车

张景元

一. 什么是 DOM: Document Object Model

文档 对象 模型

1. 什么是: 一套专门操作网页内容的对象、函数和属性的整体!
2. 为什么: ECMAScript 仅规定了 js 语言的核心语法（怎么写对，怎么写不对）。但是，ES 标准没有规定如何使用 js 操作网页内容!
3. 何时: 将来只要操作网页内容，都必须使用 DOM 提供的对象、函数和属性。
4. 问题: DOM 提供的对象、函数和属性比较繁琐，不太好用!
5. 解决: 后人在 DOM 提供的对象、函数和属性基础上，又封装了简化的 jQuery 和 Vue。



6. 问题: 因为 ECMAScript 标准不包括 DOM 的对象和函数。所以早期 DOM 没有标准，每种浏览器操作网页的方法各不相同！——严重的兼容性问题

7. 解决: W3C 组织作为第三方机构又制定了一个 DOM 标准, 规定了所有浏览器操作网页内容的标准方式——今后使用 DOM 标准操作网页，几乎所有浏览器 100%兼容。

8. 学了 DOM 可以做哪些事儿？5 件事: 增删改查+事件绑定

二. DOM 树:

1. 什么是: 一个 html 网页中所有元素、内容和属性, 在内存中都是被保存在一棵树形结构上!

2. 为什么: 因为 HTML 网页中的元素和内容都有明显的复杂的上下级包含关系，而树形结构是最直观的保存和反应上下级包含关系的结构!

3. 何时：当浏览器窗口加载了一个 HTML 网页文件后，浏览器都会扫描 HTML 文件的内容，并在内存中创建 DOM 树以及 DOM 树上的节点对象来保存 HTML 中每项内容。

4. 节点：树形结构或网状结构中多条线交汇的点，就称为节点！

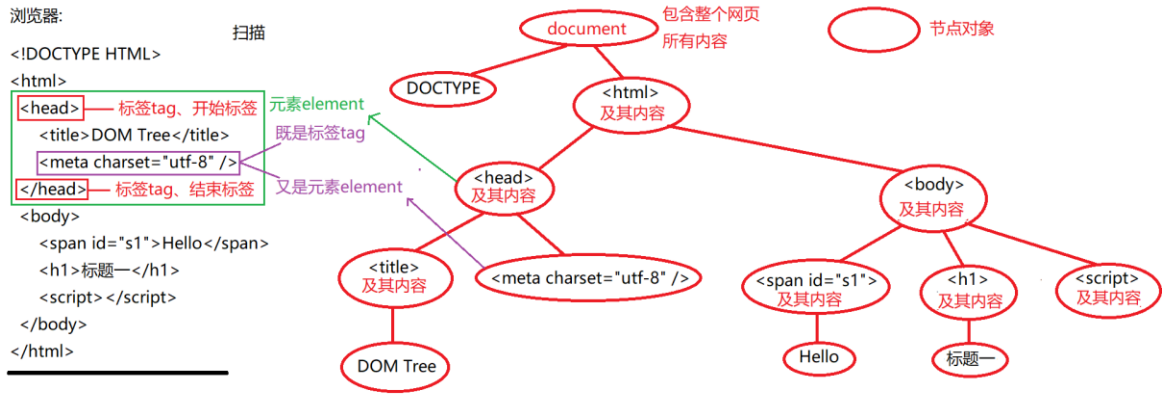
5. 如何：

(1). 先在内存中创建一个唯一的树根节点对象：document

(2). 浏览器扫描 HTML 文件内容

(3). 每扫描到一项内容(元素、文本、属性)，浏览器就会在树根节点 document 下对应位置，创建一个节点对象，保存当前扫描到的一项内容

(4). 当扫描整个 HTML 文件结束，内存中就形成了一棵属性结构



三. 查找元素: 4 种:

1. 不需要查找就可直接获得的元素: 4 种

(1). document 对象: document —— 代表整个网页所有内容

(2). <html> 元素对象: document.documentElement

(3). <head>元素对象: document.head

(4). <body>元素对象: document.body

2. 按节点间关系查找: 2 大类关系, 6 个属性:

(1). 其实有两种 DOM 树: 节点树和元素树

a. 节点树: 包含所有节点对象(元素、文本等)的完整树结构

b. 元素树: 仅包含元素节点, 不包含文本等其他类型节点的简化版树结构

(2). 节点树上的关系:

a. 父子关系: 4 个属性:

1). 获得一个节点对象的父节点: 节点对象.parentNode

父 节点

2).获得一个节点对象下的所有直接子节点对象: 节点对象.childNodes

孩子节点们

3).获得一个节点对象下的第一个直接子节点:
节点对象.firstChild

第一个孩子

4).获得一个节点对象下的最后一个直接子节点:
节点对象.lastChild

最后一个孩子

b. 兄弟关系: 2 个属性:

1). 获得当前节点对象相邻的前一个兄弟节点:
节点对象.previousSibling

前一个兄弟

2). 获得当前节点对象相邻的下一个兄弟节点:
节点对象.nextSibling

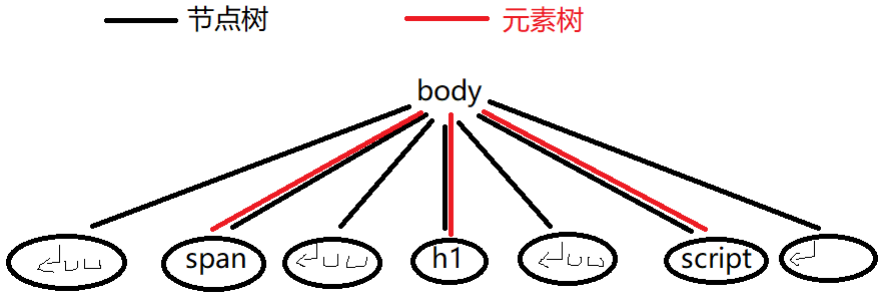
下一个兄弟

(3). 节点树的问题: 节点树认为连看不见的换行和空格，也是文本类型的节点对象。也会成为子节点和兄弟节点。——严重干扰我们的查找结果!

(4). 解决: 新 DOM 标准中在原有完整 DOM 树基础上规定了一棵新的 DOM 树——元素树。元素树上的关系仅指向元素类型的节点。不再指向其他类型的节点。——好处，查找结果不会受到看不见的空字符的

干扰!

(5). 强调: 元素树, 不是一棵新树。仅仅是原完整节点树中的部分元素节点的一个子集。



(6). 总结: 今后按节点间关系查找时, 都用元素树, 而不用节点树

a. 父子关系: 4 个属性:

1). 获得一个元素对象的父元素: 元素对象.parentElement

父 元素

2).获得一个元素对象下的所有直接子元素: 元素对象.children

孩子们

因为一个元素可能包含多个子元素, 所以 children 属性返回一个类数组对象, 其中包含找到的所有直接子元素对象。下标从 0 开始!

3).获得一个元素对象下的第一个直接子元素: 元素对象.firstElementChild

第一个 元素

孩子

4).获得一个元素对象下的最后一个直接子元素:

元素对象.lastElementChild

最后一个 元素 孩子

b. 兄弟关系: 2 个属性:

1). 获得当前元素对象相邻的前一个兄弟元素:

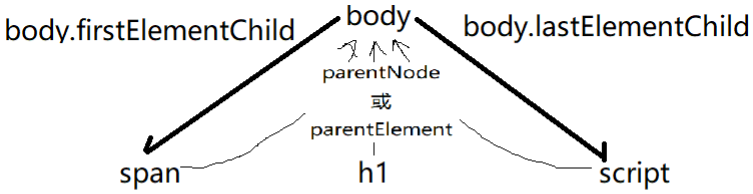
元素对象.previousElementSibling

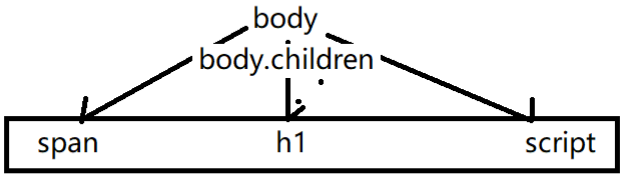
前一个 元素 兄弟

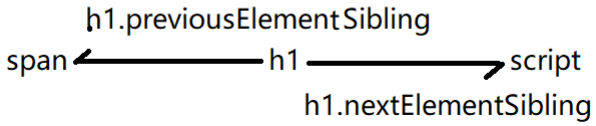
2). 获得当前元素对象相邻的下一个兄弟元素:

元素对象.nextElementSibling

下一个 元素 兄弟







(7). 示例: 使用节点间关系, 查找 body 下的元素

1_domTree.html

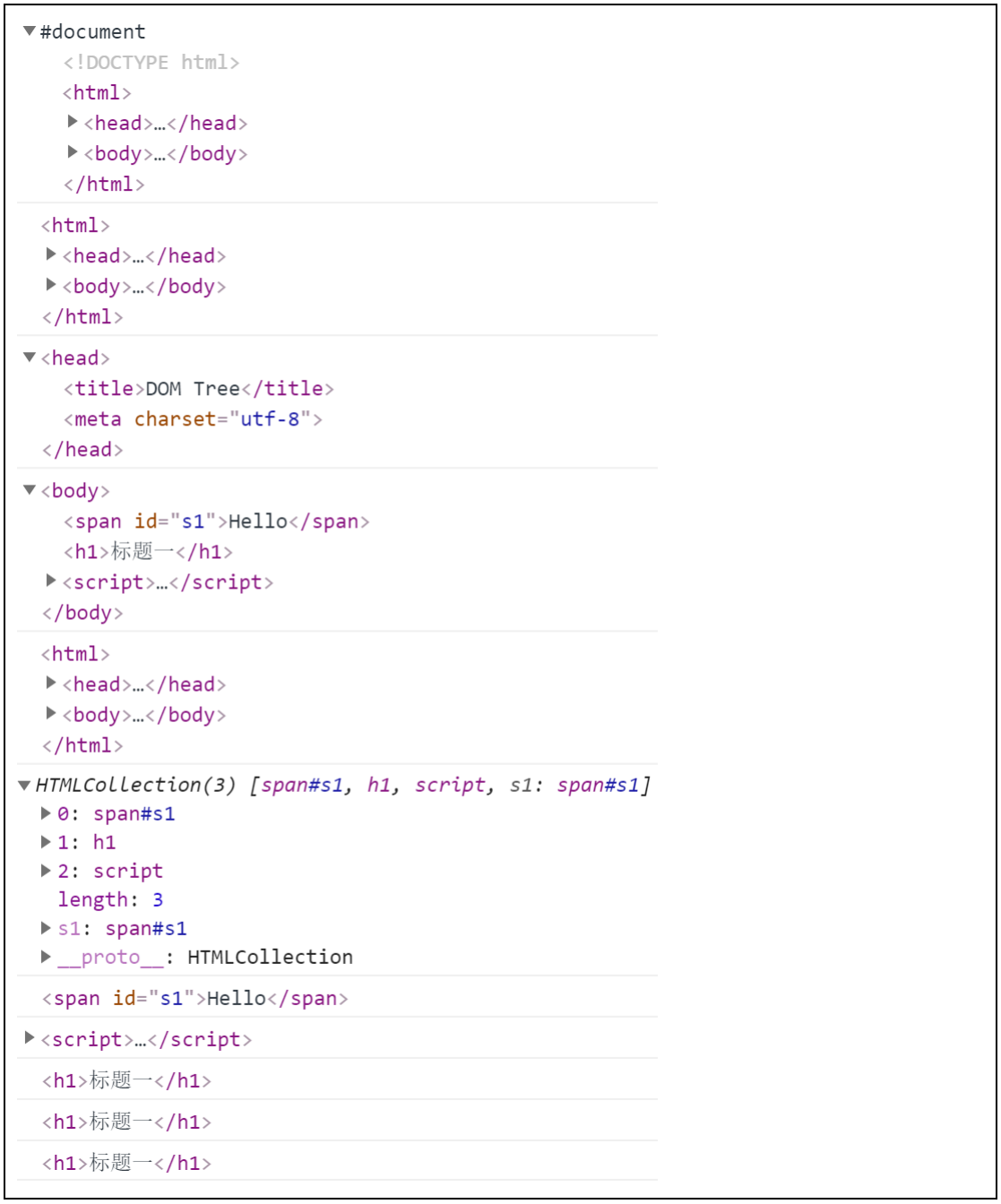
```
<!DOCTYPE HTML>
<html>

<head>
  <title>DOM Tree</title>
  <meta charset="utf-8" />
</head>

<body>
  <span id="s1">Hello</span>
  <h1>标题一</h1>
  <script>
    //想输出 document 对象
    console.log(document);
    //想输出 html 元素对象
    console.log(document.documentElement
);
    //想输出 head 元素对象
    console.log(document.head);
    //想输出 body 元素对象
    console.log(document.body);
```

```
    //本例暂时不用 live server 运行
    //因为 live server 会自动插入一个多余的 script，干扰我们的查找。
    //应该直接在硬盘上的文件夹中找到这个.html 文件，右键，打开方式，用 chrome 打开
    //想获得 body 的父节点：应该是<html>
    //既可以用 parentNode，又可以用 parentElement
    var html=document.body.parentNode;//首选
    //var html=document.body.parentElement;
    console.log(html);//正确
    //因为在网页中能当爹，包含其它子内容的，只可能是元素。<开始标签>子内容</结束标签>


    //想获得 body 下的所有直接子元素：应该是 3 个
    //错误做法：
    //var childNodes=document.body.childNodes;
```

```
    //console.log(childNodes);//7 个
    //正确做法:
    var children=document.body.children;
    console.log(children);//3 个
    //想获得 body 下的第一个直接子元素: 应该是 span
    //错误做法:
    //var span=document.body.firstChild;
    //正确做法:
    var span=document.body.firstElementChild;
    console.log(span);
    //想实现 body 中最后一个直接子元素: 应该是 script
    var script=document.body.lastElementChild;
    console.log(script);
    //想获得 body 中第二个孩子: 应该是 h1
    var h1=document.body.children[1];
    console.log(h1);
    //想获得 h1，还可以通过 span 的下一个兄弟元素获得
```

```
    var h1=span.nextElementSibling;
    console.log(h1);
    //想获得 h1，还可以通过 script 的前一个兄弟元素获得
    var h1=script.previousElementSibling;
    console.log(h1);
  </script>
</body>

</html>
```

运行结果:



(8). 何时: 今后如果已经获得一个 DOM 元素, 想找它周围附近的其它 DOM 元素时, 就用按节点间关系查找。

3. 按 HTML 特征查找: 4 个方法:

(1). 何时: 只要还没有获得任何 DOM 元素, 就要查

找元素，或要查找的元素离当前获得的元素很远，就选择按 HTML 特征查找。

(2). 4 大特征:

a. 按 id 名查找一个元素:

1). var 一个元素对象=document.getElementById("id 名")

在整个网页中获取 元素 以 id 为条件

2). 返回值:

i. 如果找到符合条件的一个元素，就返回一个元素对象

ii. 如果没找到符合条件的元素，就返回 null

3). 强调:

i. 必须用 document.作为开头

ii. 因为按 id 查找，只能找到一个元素，所以 Element 不带 s 结尾，是单数形式

iii. 如果网页中确实有多个相同的 id，则 getElementById()永远只能找第一个符合条件的。

b. 按标签名查找多个元素:

1). var 类数组对象 = 任意父元素.getElementsByTagName("标签名")

在指定父元素下获取多个元素以

标签名为查找条件

2). 返回值:

i. 如果找到符合条件的多个元素，则多个元素放在一个类数组对象中返回

ii. 如果找不到符合条件的元素，则返回空类数组对象: { length:0 }

3). 强调:

i. 不一定必须用 document 作为开头。可以用任意父元素作为开头！仅在当前父元素下查找符合条件的元素。——好处，限制查找范围，提高查找效率

ii. 因为可能返回多个符合要求的子元素，所以方法名中 Elements 是 s 结尾，复数形式。

iii. 不但查找直接子元素，而且在所有后代中查找符合要求的元素

iv. 问题: getElementsByTagName()注定会返回一个类数组对象。即使只找到一个元素，也会放在类数组对象中返回！但是，我们希望使用的绝不是类数组对象，我们通常希望使用的是这一个 DOM 元素对象

解决: 只要在查找结果基础上继续加[0]，从类数组对象中 0 位置，取出找到的唯一的一个 DOM 元素对象。

比如: body 中只有一个 span

错误:

var span=document.body.getElementsByTagName("span");

//返回的 span 不是 DOM 元素对象，而是一个类数组对象，其中 0 位置放着找到的唯一一个 DOM 元素对象 span： { 0:span 对象, length:0 }

正确:

var span=document.body.getElementsByTagName("span")[0];

//返回的 span 就是 HTML 中的 DOM 元素对象。[0]是从查找结果类数组对象中 0 位置取出唯一一个 DOM 元素对象的意思。

c. 按 class 名查找多个元素:

1). var 类数组对象 = 任意父元素.getElementsByClassName("class 名")

在任意父元素下获取多个元素以 className 为条件

2). 返回值:

i. 如果找到符合条件的多个元素，则多个元素放

在一个类数组对象中返回

ii. 如果找不到符合条件的元素，则返回空类数组对象：{ length:0 }

3). 强调:

i.可以用任意父元素作为开头！仅在当前父元素下查找符合条件的元素。

——好处，限制查找范围，提高查找效率

ii. 方法名中 Elements 是 s 结尾，复数形式。

iii.在所有后代中查找符合要求的元素

iv. 如果只找到一个元素，也必须加[0]，才能取出这唯一的 DOM 元素对象

v. 如果一个元素上同时被多个 class 修饰，那么只用其中一个 class 名就可找到该元素。

d. 按 name 属性查找多个表单元素:

1). var 类数组对象=document.getElementsByName("name 名")

在整个页面中获取多个元素以 name 名作为条件

2). 返回值:

i. 如果找到符合条件的多个元素，则多个元素放在一个类数组对象中返回

ii. 如果找不到符合条件的元素，则返回空类数组

对象: { length:0 }

3). 强调:

i.必须用 document.作为开头!不能以任意父元素开头!

ii. 方法名中 Elements 是 s 结尾，复数形式。

iii. 如果只找到一个元素，也必须加[0]，才能取出这唯一的 DOM 元素对象

(3). 示例: 使用按 HTML 特征查找四个函数查找想要的元素

2_iterator.html

```
<!DOCTYPE HTML>
<html>
  <head>
    <title>遍历节点树</title>
    <meta charset="utf-8"/>
  </head>
  <body>
    <span>Hello World !</span>
    <ul id="nav">
      <li class="parent">电影</li>
      <li class="parent">综艺
        <ul>
```

```
            <li class="child active">跑男
</li>
            <li class="child">爸爸</li>
            <li class="child">极限</li>
          </ul>
        </li>
        <li class="parent">剧集</li>
      </ul>
      用户
名: <input type="text" name="uname"><br/>
      性别:
      <label>
        <input type="radio" name="sex" val
ue="1">男
      </label>
      <label>
        <input type="radio" name="sex" val
ue="0">女
      </label>
      <script>
        //想查找 id 为 nav 的 ul
```

```
        var ul=document.getElementById("nav");
        console.log(ul);
        //强调：所有元素不查找，是不能直接使用!
        //要想使用某个元素，都必须先查找再使用
        //想查找 ul 下的所有 li
        var lis=ul.getElementsByTagName("li");
        console.log(lis);

        //想在 body 下找 span
        //只可能找到一个 span 时
        var span=
          //document.body.getElementsByTagName("span")
          //返回类数组对象，其中包含一个 span 元素对象

          document.body.getElementsByTagName("span")[0]
```

```
        //直接返回类数组对象中 0 位置的 span 元素对象。
        console.log(span);//

        //想查找 ul 下所有 class 为 parent 的 li
        var parents=ul.getElementsByClassName("parent");
        console.log(parents);
        //想查找 ul 下所有 class 为 child 的 li
        var children=ul.getElementsByClassName("child");
        console.log(children);
        //想查找 ul 下 class 为 active 的一个 li
        var liActive=ul.getElementsByClassName("active")[0];
        console.log(liActive);

        //想查找 name 为 sex 的两个元素
        var radios=document.getElementsByName("sex");
        console.log(radios);
        //想查找 name 为 uname 的一个文本框
```

```
        var txtUname=document.getElementsB
yName("uname")[0];
        console.log(txtUname);
    </script>
  </body>
</html>
```

运行结果:

```
▼<ul id="nav">                                                    2_iterator.html:31
    <li class="parent">电影</li>
  ▶<li class="parent">…</li>
    <li class="parent">剧集</li>
  </ul>

                                                                  2_iterator.html:36
▼HTMLCollection(6) [li.parent, li.parent, li.child.active, li.child, li.child,
li.parent]
  ▶0: li.parent
  ▶1: li.parent
  ▶2: li.child.active
  ▶3: li.child
  ▶4: li.child
  ▶5: li.parent
    length: 6
  ▶__proto__: HTMLCollection

  <span>Hello World !</span>                            2_iterator.html:46

▼HTMLCollection(3) [li.parent, li.parent, li.parent]              2_iterator.html:50
  ▶0: li.parent
  ▶1: li.parent
  ▶2: li.parent
    length: 3
  ▶__proto__: HTMLCollection

▼HTMLCollection(3) [li.child.active, li.child, li.child]          2_iterator.html:53
  ▶0: li.child.active
  ▶1: li.child
  ▶2: li.child
    length: 3
  ▶__proto__: HTMLCollection

  <li class="child active">跑男</li>                              2_iterator.html:56

▼NodeList(2) [input, input]                                       2_iterator.html:60
  ▶0: input
  ▶1: input
    length: 2
  ▶__proto__: NodeList

  <input type="text" name="uname">                                2_iterator.html:63
```

4. 按选择器查找: 2 个函数:

今后，只要元素藏的很深，查找条件很复杂时，都要选择按选择器查找

(1). 只查找一个符合条件的元素:

var 一个元素对象=任意父元素.querySelector("任意选择器");

(2). 查找多个符合条件的元素:

var 类数组对象=任意父元素.querySelectorAll("任意选择器");

四. 事件绑定基础:

1. 什么是事件: 浏览器自己触发的或用户手工触发的页面中元素内容和状态的改变

2. 什么是事件处理函数: 当事件发生时，希望浏览器自动调用的函数。

3. 什么是绑定事件: 提前在元素的事件属性上保存一个事件处理函数

结果: 等到事件发生时，自动执行该事件处理函数。

4. 为什么绑定事件: 网页中所有元素，默认即使发生了事件，也什么都不做。

5. 何时需要事件绑定: 如果希望一个元素触发事件时

可以自动执行一个操作，都要提前为元素绑定事件处理函数。

6. 如何绑定事件处理函数:

(1). 在 HTML 中绑定: —— 不好，几乎不用!

a. HTML 中: <元素 on 事件名="函数名()" >

b. js 中: function 函数名(){ ... }

c. 问题:

1). 不符合 js 程序与 HTML 内容分离的原则，不便于维护!

2). 开始标签中写死的事件属性，新生成的元素，无法自动获得。

d. 结论: 在 DOM 和 jQuery 中绝对不会使用 HTML 绑定事件处理函数

(2). 在 js 中用赋值方式绑定:

a. 查找到要绑定事件的元素对象

b. 每个元素对象上，都包含很多 on 开头的特殊事件属性

c. 当这个元素上发生事件时，浏览器会自动找到这个元素上对应的 onxxx 事件属性，并执行 onxxx 事件

属性上提前绑定的事件处理函数。

d. 所以，我们需要提前为这个元素对象的某个 onxxx 事件属性赋值一个事件处理函数备用!

元素对象.on 事件名=function(){

//当事件发生时，才执行的代码

}

e. 强调: 这里的 function, 仅赋值并保存在 on 事件名属性中，暂不执行!

f. 优点:

1). 可集中写在 js 中，便于维护

2). 是 js 语句，想什么时候执行，就什么时候执行! 想给谁绑定，就给谁绑定! ——灵活!

g. 示例: 只为第一个按钮绑定单击事件处理函数

3_shoppingcart.html

```
<!DOCTYPE HTML>
<html>
<head>
<title>使用 Selector API 实现购物车客户端计算</title>
<meta charset="utf-8" />
<style>
  table{width:600px; text-align:center;
```

```
    border-collapse:collapse;
  }
  td,th{border:1px solid black}
  td[colspan="3"]{text-align:right;}
</style>

</head>
<body>
  <table id="data">
    <thead>
      <tr>
        <th>商品名称</th>
        <th>单价</th>
        <th>数量</th>
        <th>小计</th>
      </tr>
    </thead>
    <tbody>
      <tr>
        <td>iPhone6</td>
        <td>¥4488.00</td>
        <td>
```

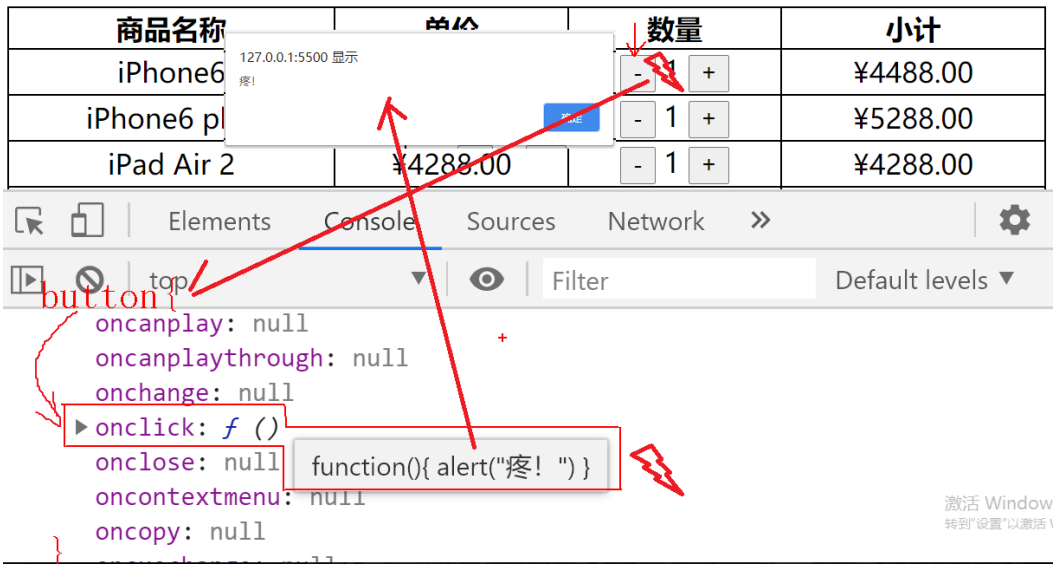
```
        <button>-</button>
        <span>1</span>
        <button>+</button>
      </td>
      <td>¥4488.00</td>
    </tr>
    <tr>
      <td>iPhone6 plus</td>
      <td>¥5288.00</td>
      <td>
        <button>-</button>
        <span>1</span>
        <button>+</button>
      </td>
      <td>¥5288.00</td>
    </tr>
    <tr>
      <td>iPad Air 2</td>
      <td>¥4288.00</td>
      <td>
        <button>-</button>
        <span>1</span>
```

```
            <button>+</button>
          </td>
          <td>¥4288.00</td>
        </tr>
      </tbody>
      <tfoot>
        <tr>
          <td colspan="3">Total: </td>
          <td>¥14064.00</td>
        </tr>
      </tfoot>
    </table>
    <script>
      //说人话
      //实现第一个功能: 点按钮改数量
      //DOM 4 步
      //1. 查找触发事件的元素
      //本例中:将来用户点 table 中每个 button 元素都会触发变化?
      //所以，应该查找 table 下所有 button 元素，2 步:
```

```
    //1.1 先查找 table 元素：查找 id 为 data 的 table 元素
    var table=document.getElementById("data");
    console.log(table);
    //1.2 再在 table 下查找所有的 button 元素
    var btns=table.getElementsByTagName("button");
    console.log(btns);
    //2. 绑定事件处理函数
    //实验：只给第一个按钮绑定单击事件，当单击第一个按钮时，它可以喊疼！
    var btn1=btns[0];
    //想看元素的属性，不能用 console.log()输出
    //console.log(btn1);
    //应该用 console.dir()输出，dir 专门输出内存中的对象存储结构
    console.dir(btn1);
    //为 btn1 的 onclick 事件属性提前赋值一个函数，会喊疼
```

```
    btn1.onclick=function(){ alert("疼!
") }


        //3. 查找要修改的元素
        //4. 修改元素
  </script>
</body>
</html>
```

运行结果: 当单击第一个按钮时



7. 示例: 为所有找到按钮都绑定单击事件处理函数:

3_shoppingcart.html

```
<!DOCTYPE HTML>
<html>
```

```
<head>
<title>使用 Selector API 实现购物车客户端计
算</title>
<meta charset="utf-8" />
<style>
  table{width:600px; text-align:center;
    border-collapse:collapse;
  }
  td,th{border:1px solid black}
  td[colspan="3"]{text-align:right;}
</style>

</head>
<body>
  <table id="data">
    <thead>
      <tr>
        <th>商品名称</th>
        <th>单价</th>
        <th>数量</th>
        <th>小计</th>
      </tr>
```

```
    </thead>
    <tbody>
      <tr>
        <td>iPhone6</td>
        <td>¥4488.00</td>
        <td>
          <button>-</button>
          <span>1</span>
          <button>+</button>
        </td>
        <td>¥4488.00</td>
      </tr>
      <tr>
        <td>iPhone6 plus</td>
        <td>¥5288.00</td>
        <td>
          <button>-</button>
          <span>1</span>
          <button>+</button>
        </td>
        <td>¥5288.00</td>
      </tr>
```

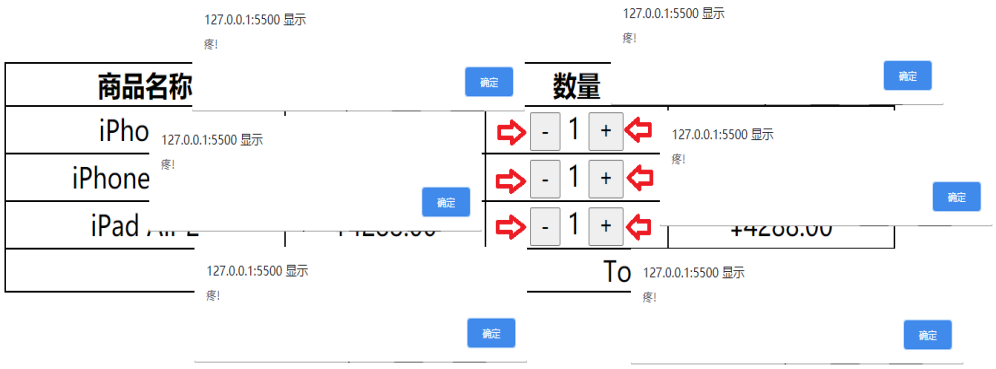
```
      <tr>
        <td>iPad Air 2</td>
        <td>¥4288.00</td>
        <td>
          <button>-</button>
          <span>1</span>
          <button>+</button>
        </td>
        <td>¥4288.00</td>
      </tr>
    </tbody>
    <tfoot>
      <tr>
        <td colspan="3">Total: </td>
        <td>¥14064.00</td>
      </tr>
    </tfoot>
  </table>
  <script>
    //说人话
    //实现第一个功能: 点按钮改数量
    //DOM 4 步
```

```
    //1. 查找触发事件的元素
    //本例中:将来用户点 table 中每个 button 元素都会触发变化?
    //所以，应该查找 table 下所有 button 元素，2 步:
    //1.1 先查找 table 元素: 查找 id 为 data 的 table 元素
    var table=document.getElementById("data");
    console.log(table);
    //1.2 再在 table 下查找所有的 button 元素
    var btns=table.getElementsByTagName("button");
    console.log(btns);
    //2. 绑定事件处理函数
    //实验: 只给第一个按钮绑定单击事件，当单击第一个按钮时，它可以喊疼!
    //var btn1=btns[0];
    //想看元素的属性，不能用 console.log()输出
    //console.log(btn1);
```

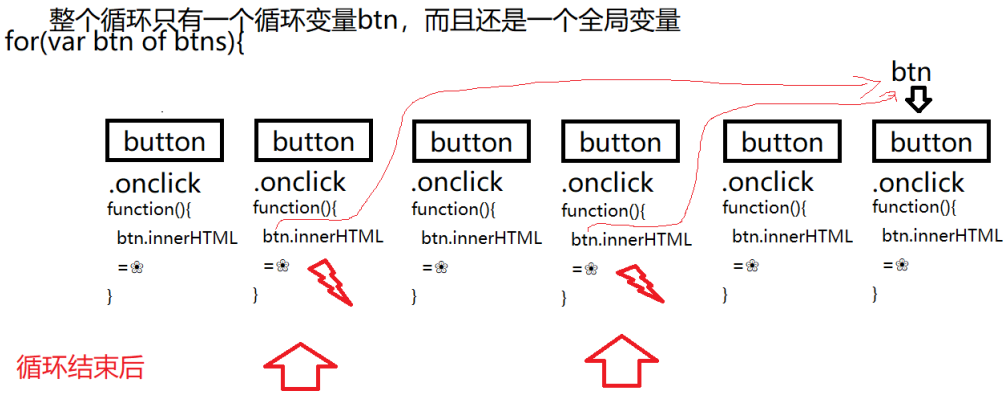
```
    //应该用 console.dir()输出，dir 专门输出内存中的对象存储结构
    //console.dir(btn1);
    //为 btn1 的 onclick 事件属性提前赋值一个函数，会喊疼
    //btn1.onclick=function(){ alert("疼！") }

    //本例中：所有 button 都可以点击。所以，要为每个 button 的 onclick 提前赋值一个事件处理函数！
    //遍历 btns 类数组对象中找到的所有按钮对象
    for(var btn of btns){
      //每遍历一个按钮对象，就为这个按钮对象提前保存一个事件处理函数
      btn.onclick=function(){
        alert("疼！");
      }
    }

      //3. 查找要修改的元素
```

```
        //4. 修改元素
    </script>
</body>
</html>
```

运行结果：



8. 问题：通过遍历为每个按钮都绑定一个单击事件处理函数，在事件处理函数中如果用循环变量代表“当前按钮”，执行后续操作，其实操作的都是最后一个按钮！

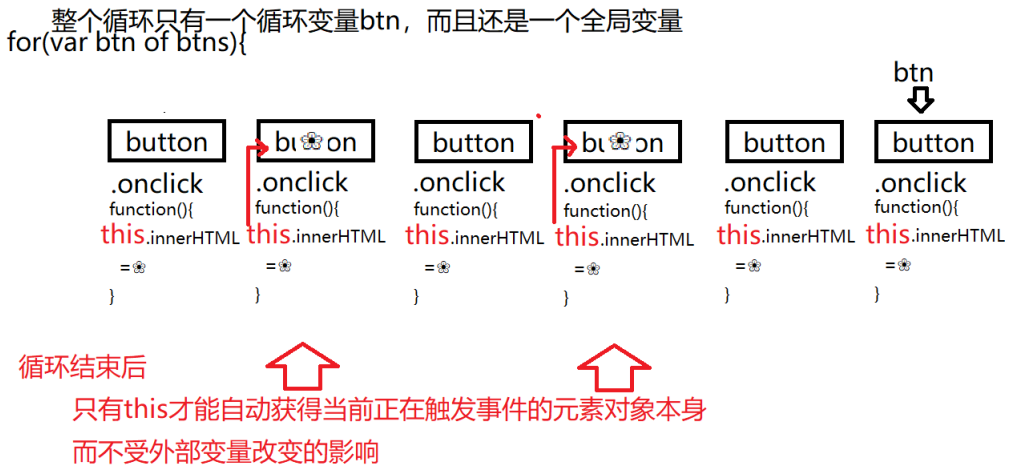


原因：整个循环中只有一个循环变量 btn，该循环变

量在循环结束后，会停在最后一个按钮元素上。所以，当用户点击按钮，执行事件处理函数时，永远找唯一的 btn 变量，而这个变量永远指最后一个按钮！

9. 解决：在事件处理函数中，如何获得当前正在触发事件的元素，而不受外部变量的影响！

事件处理函数中的 this->当前正在触发事件的元素



10. 示例：点那个按钮，让那个按钮变成小花：

3_shoppingcart.html

```
<!DOCTYPE HTML>
<html>
<head>
<title>使用 Selector API 实现购物车客户端计算</title>
<meta charset="utf-8" />
<style>
  table{width:600px; text-align:center;
```

```
    border-collapse:collapse;
  }
  td,th{border:1px solid black}
  td[colspan="3"]{text-align:right;}
</style>

</head>
<body>
  <table id="data">
    <thead>
      <tr>
        <th>商品名称</th>
        <th>单价</th>
        <th>数量</th>
        <th>小计</th>
      </tr>
    </thead>
    <tbody>
      <tr>
        <td>iPhone6</td>
        <td>¥4488.00</td>
        <td>
```

```
        <button>-</button>
        <span>1</span>
        <button>+</button>
      </td>
      <td>¥4488.00</td>
    </tr>
    <tr>
      <td>iPhone6 plus</td>
      <td>¥5288.00</td>
      <td>
        <button>-</button>
        <span>1</span>
        <button>+</button>
      </td>
      <td>¥5288.00</td>
    </tr>
    <tr>
      <td>iPad Air 2</td>
      <td>¥4288.00</td>
      <td>
        <button>-</button>
        <span>1</span>
```

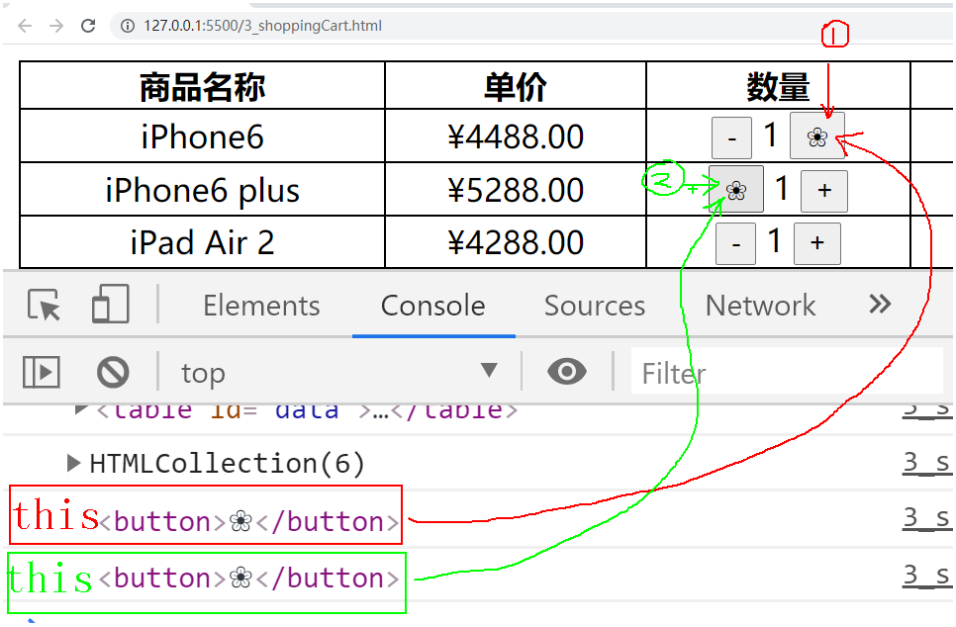
```
            <button>+</button>
          </td>
          <td>¥4288.00</td>
        </tr>
      </tbody>
      <tfoot>
        <tr>
          <td colspan="3">Total: </td>
          <td>¥14064.00</td>
        </tr>
      </tfoot>
    </table>
    <script>
      //说人话
      //实现第一个功能: 点按钮改数量
      //DOM 4 步
      //1. 查找触发事件的元素
      //本例中:将来用户点 table 中每个 button 元素都会触发变化?
      //所以，应该查找 table 下所有 button 元素，2 步:
```

```
    //1.1 先查找 table 元素：查找 id 为 data 的 table 元素
    var table=document.getElementById("data");
    console.log(table);
    //1.2 再在 table 下查找所有的 button 元素
    var btns=table.getElementsByTagName("button");
    console.log(btns);
    //2. 绑定事件处理函数
    //实验：只给第一个按钮绑定单击事件，当单击第一个按钮时，它可以喊疼！
    //var btn1=btns[0];
    //想看元素的属性，不能用 console.log()输出
    //console.log(btn1);
    //应该用 console.dir()输出，dir 专门输出内存中的对象存储结构
    //console.dir(btn1);
    //为 btn1 的 onclick 事件属性提前赋值一个函数，会喊疼
```

```
    //btn1.onclick=function(){ alert("
疼！") }

    //本例中：所有 button 都可以点击。所以，
要为每个 button 的 onclick 提前赋值一个事件处
理函数！
    //遍历 btns 类数组对象中找到的所有按钮对
象
    for(var btn of btns){
      //每遍历一个按钮对象，就为这个按钮对象
提前保存一个事件处理函数
      btn.onclick=function(){
        //alert("疼！");
        //新需求：点那个按钮，让那个按钮的
内容变成❀
        //错误的解决：btn 是唯一的全局变
量，也是循环变量。在触发事件前，btn 就移动到
最后一个按钮上了
        //btn.innerHTML="❀";
            //内容
        //正确的解决：
        console.log(this);
```

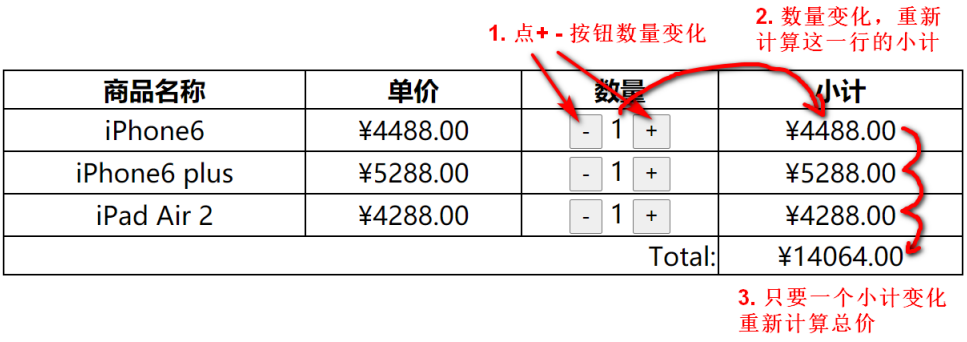
```
            this.innerHTML="❀";
        }
        //结果:
        //btns[0].onclick=function(){alert
("疼")}
        //btns[1].onclick=function(){alert
("疼")}
        //btns[2].onclick=function(){alert
("疼")}
        //btns[3].onclick=function(){alert
("疼")}
        //btns[4].onclick=function(){alert
("疼")}
        //btns[5].onclick=function(){alert
("疼")}
    }

        //3. 查找要修改的元素
        //4. 修改元素
  </script>
</body>
</html>
```

运行结果：



五. 示例: 购物车: 3 大功能:

1. 需求:



2. 需求一: 先实现点击按钮, 让按钮旁边的 span 数量+1 或-1

(1). 需求:

a. 每行的按钮都能点击

b. 如果点击+，则找到+前边的 span，将其内容中的数字+1

c. 如果点击-，则找到-后边的 span，将其内容中的数字-1，但是，购物车中商品数量不能<1.

(2). 今后几乎所有 DOM 操作的标准 4 步:

a. 先查找可能触发事件的元素

本例中: 查找 table 下所有 button 元素

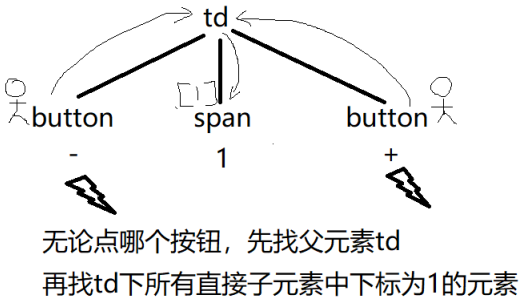
b. 再为元素绑定事件处理函数

本例中: 为每个 button 绑定 onclick 单击事件处理函数

c. 当事件发生时，查找要修改的元素

本例中: 先查找当前按钮的父元素 td，然后再找 td 下所有直接子元素中下标为 1 的 span 元素。

```
<td>
    <button>-</button>
    <span>1</span>
    <button>+</button>
</td>
```



d. 修改元素

本例中: 先获得 span 元素的内容，转为整数后，+1 或-1，再放回 span 的内容中

(3). 示例: 实现单击按钮，修改数量:

3_shoppingcart2.html

```
<!DOCTYPE HTML>
<html>
<head>
<title>使用 Selector API 实现购物车客户端计算</title>
<meta charset="utf-8" />
<style>
  table{width:600px; text-align:center;
    border-collapse:collapse;
  }
  td,th{border:1px solid black}
  td[colspan="3"]{text-align:right;}
</style>

</head>
<body>
  <table id="data">
    <thead>
      <tr>
        <th>商品名称</th>
        <th>单价</th>
        <th>数量</th>
```

```
        <th>小计</th>
      </tr>
    </thead>
    <tbody>
      <tr>
        <td>iPhone6</td>
        <td>¥4488.00</td>
        <td>
          <button>-</button>
          <span>1</span>
          <button>+</button>
        </td>
        <td>¥4488.00</td>
      </tr>
      <tr>
        <td>iPhone6 plus</td>
        <td>¥5288.00</td>
        <td>
          <button>-</button>
          <span>1</span>
          <button>+</button>
        </td>
```

```
        <td>¥5288.00</td>
      </tr>
      <tr>
        <td>iPad Air 2</td>
        <td>¥4288.00</td>
        <td>
          <button>-</button>
          <span>1</span>
          <button>+</button>
        </td>
        <td>¥4288.00</td>
      </tr>
    </tbody>
    <tfoot>
      <tr>
        <td colspan="3">Total: </td>
        <td>¥14064.00</td>
      </tr>
    </tfoot>
  </table>
  <script>
    //说人话
```

```
    //实现第一个功能：点按钮改数量
    //DOM 4步
    //1. 查找触发事件的元素
    //本例中:应该查找table下所有button元素，2步:
    //1.1 先查找table元素：查找id为data的table元素
    var table=document.getElementById("data");
    //1.2 再在table下查找所有的button元素
    var btns=table.getElementsByTagName("button");
    //2. 绑定事件处理函数
    //本例中：要为每个button的onclick提前赋值一个事件处理函数！
    //遍历btns类数组对象中找到的所有按钮对象
    for(var btn of btns){
      //每遍历一个按钮对象，就为这个按钮对象提前保存一个事件处理函数
      btn.onclick=function(){
```

```
        //this->才可获得当前正在单击的按钮
按钮对象
        //不要用外部变量 btn，因为 btn 会
变！不靠谱！
        //3. 查找要修改的元素
        //本例中：查找当前按钮的爹的所有直
接子元素中下标为 1 的 span 元素
        var span=this.parentNode.childre
n[1];
        //测试：让当前 span 的背景色变为红色
        //span.style.backgroundColor="re
d";
        //4. 修改元素
        //span:<span>1</span>
        //强调：一定不能直接对元素对象+1 或-
1，而应该对元素的内容+1 或-1
        //本例中：先获得 span 元素的内容，+1
或-1 后，再放回 span 中
        //4.1 先获得 span 的内容，保存在变量
n 中
        //坑：凡是从页面元素上获得的一切，
都是字符串类型！
```

```
        //解决：今后凡是从页面上获得的一切，在进行算数计算前，必须都要先转为number类型的数字。
        var n=parseInt(span.innerHTML);
        //4.2 如果当前按钮的内容是+，就n+1，否则如果当前按钮的内容是-，且n的值>1时，才允许n-1
        if(this.innerHTML=="+"){
          n++;
        }else if(n>1){//如果程序可以执行到这里，暗含着innerHTML肯定等于"-"，因为除了+和-，没有其它情况
          n--;
        }
        //坑：从页面上获取内容或值，拿到的都是副本！在程序中修改变量，原值或内容，不改变的！
        //解决：
        //4.3 必须将修改后的n变量的值，重新赋值回页面上span元素对象的内容中，页面上才能看到变化！
        span.innerHTML=n;
```

```
        }


    }
  </script>
</body>
</html>
```

运行结果：

商品名称	单价	数量	小计
iPhone6	¥4488.00	- 5 +	¥4488.00
iPhone6 plus	¥5288.00	- 7 +	¥5288.00
iPad Air 2	¥4288.00	- 2 +	¥4288.00

3. 需求 2：数量变化，重新计算本行的小计

(1). DOM 4 步：

a. 查找触发事件的元素：需求一已完成，和需求一共用同一个单击事件即可

b. 绑定事件处理函数：需求一已完成，和需求一共用同一个单击事件即可

c. 查找要修改的元素：

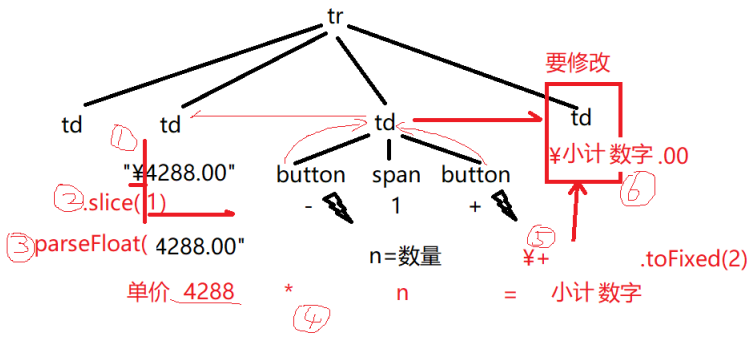
本例中：查找当前按钮的父元素 td 的下一个兄弟元素 td

d. 修改元素：

本例中：获得当前按钮的父元素 td 的前一个兄弟的

内容，去掉¥，转为浮点数，*数量 n。算出的小计，再拼上¥，并按两位小数四舍五入。最后放入小计 td 中.

```
<tr>
  <td>iPad Air 2</td>
  <td>¥4288.00</td>
  <td>
    <button>-</button>
    <span>1</span>
    <button>+</button>
  </td>
  <td>¥4288.00</td>
</tr>
```



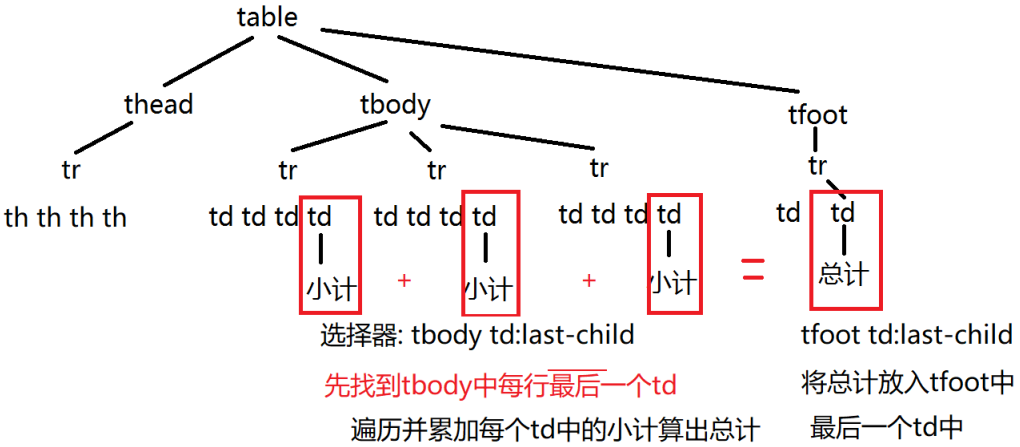
(2). 示例: 数量变化，修改本行小计

3_shoppingcart.html

```
```

4. 需求 3: 小计变化，重新计算总计

(1). 需求:



(2). 示例: 实现小计变化，自动修改总计:

3_shoppingcart2.html

```
<!DOCTYPE HTML>
<html>
```

```
<head>
<title>使用 Selector API 实现购物车客户端计
算</title>
<meta charset="utf-8" />
<style>
  table{width:600px; text-align:center;
    border-collapse:collapse;
  }
  td,th{border:1px solid black}
  td[colspan="3"]{text-align:right;}
  /*想让 tbody 中每行最后一个 td 背景变为粉色
*/
  tbody td:last-child{
    background-color:pink;
  }
  /*想让 tfoot 中最后一个 td 背景变为黄色*/
  /* tfoot td:last-child{
    background-color:yellow;
  } */
</style>

</head>
```

```
<body>
  <table id="data">
    <thead>
      <tr>
        <th>商品名称</th>
        <th>单价</th>
        <th>数量</th>
        <th>小计</th>
      </tr>
    </thead>
    <tbody>
      <tr>
        <td>iPhone6</td>
        <td>¥4488.00</td>
        <td>
          <button>-</button>
          <span>1</span>
          <button>+</button>
        </td>
        <td>¥4488.00</td>
      </tr>
      <tr>
```

```
        <td>iPhone6 plus</td>
        <td>¥5288.00</td>
        <td>
          <button>-</button>
          <span>1</span>
          <button>+</button>
        </td>
        <td>¥5288.00</td>
      </tr>
      <tr>
        <td>iPad Air 2</td>
        <td>¥4288.00</td>
        <td>
          <button>-</button>
          <span>1</span>
          <button>+</button>
        </td>
        <td>¥4288.00</td>
      </tr>
    </tbody>
    <tfoot>
      <tr>
```

```
        <td colspan="3">Total: </td>
        <td>¥14064.00</td>
      </tr>
    </tfoot>
  </table>
  <script>
    //说人话
    //实现第一个功能: 点按钮改数量
    //DOM 4 步
    //1. 查找触发事件的元素
    //本例中:应该查找 table 下所有 button 元素，2 步:
    //1.1 先查找 table 元素: 查找 id 为 data 的 table 元素
    var table=document.getElementById("data");
    //1.2 再在 table 下查找所有的 button 元素
    var btns=table.getElementsByTagName("button");
    //2. 绑定事件处理函数
```

```
    //本例中：要为每个 button 的 onclick 提前赋值一个事件处理函数！
    //遍历 btns 类数组对象中找到的所有按钮对象
    for(var btn of btns){
      //每遍历一个按钮对象，就为这个按钮对象提前保存一个事件处理函数
      btn.onclick=function(){
        //this->才可获得当前正在单击的按钮按钮对象
        //不要用外部变量 btn，因为 btn 会变！不靠谱！
        //3. 查找要修改的元素
        //本例中：查找当前按钮的爹的所有直接子元素中下标为 1 的 span 元素
        var span=this.parentNode.childrenn[1];
        //测试：让当前 span 的背景色变为红色
        //span.style.backgroundColor="red";
        //4. 修改元素
        //span:<span>1</span>
```

```
        //强调：一定不能直接对元素对象+1 或-1，而应该对元素的内容+1 或-1
        //本例中：先获得 span 元素的内容，+1 或-1 后，再放回 span 中
        //4.1 先获得 span 的内容，保存在变量 n 中
        //坑：凡是从页面元素上获得的一切，都是字符串类型！
        //解决：今后凡是从页面上获得的一切，在进行算数计算前，必须都要先转为 number 类型的数字。
        var n=parseInt(span.innerHTML);
        //4.2 如果当前按钮的内容是+，就 n+1，否则如果当前按钮的内容是-，且 n 的值>1 时，才允许 n-1
        if(this.innerHTML=="+"){
          n++;
        }else if(n>1){//如果程序可以执行到这里，暗含着 innerHTML 肯定等于"-"，因为除了+和-，没有其它情况
          n--;
        }
```

```
        //坑：从页面上获取内容或值，拿到的都是副本！在程序中修改变量，原值或内容，不改变的！
        //解决：
        //4.3 必须将修改后的 n 变量的值，重新赋值回页面上 span 元素对象的内容中，页面上才能看到变化！
        span.innerHTML=n;


        //实现需求二：数量修改，自动修改本行的小计
        //3. 查找要修改的元素：
        //本例中：当前按钮的父元素 td 的下一个兄弟元素 td
        var tdSub=this.parentNode.nextElementSibling;
        //4. 修改元素：
        //本例中：获得单价，*数量 n。算出的小计,最后放入小计 td 中.
        //4.1 获得单价：获得当前按钮的父元素 td 的前一个兄弟 td 的内容，去掉¥，转为浮点数
```

```
        var price=parseFloat(
          this.parentNode //按钮所在 td
              .previousElementSibling //
按钮所在 td 的前一个兄弟 td
              .innerHTML //前一个兄弟 td 的
内容: "¥单价.00"
              .slice(1) //"单价.00"
        )//单价
        //4.2 单价*数量=小计
        var sub=price*n;
        //4.3 将小计拼接¥，按两位小数四舍五
入，再放回小计 td 的内容中
        tdSub.innerHTML=`¥${sub.toFixed(
2)}`;


        //实现需求三: 总计
        //3. 查找要修改的元素:
        //本例中: 修改 tfoot 中最后一个 td
        //因为下边的选择器肯定只能找到一个
td，所以不加 all
        var lastTd=table.querySelector("
tfoot td:last-child");
```

```
        //测试:
        //lastTd.style.backgroundColor="
yellow";
        //4. 修改元素:
        //本例中: 获得 tbody 中每行最后一个
td 中的小计，累加后算出总计，放入 tfoot 中最
后一个 td 中
        //4.1 获得 tbody 中每行最后一个 td
        var tds=table.querySelectorAll("
tbody td:last-child");
        //先定义一个变量 total=0，用于累加
每行的小计
        var total=0;
        //4.2 遍历找到的每个 td
        for(var td of tds){
          //每遍历一个 td，就要获取 td 的内
容，去掉¥，转为浮点数，累加到变量 total 上
          total+=parseFloat(td.innerHTML
.slice(1))
        }
        //4.3 将算出总价拼上¥符号，按 2 位小
数四舍五入，最后放入 tfoot 中最后一个 td 中
```

```
            lastTd.innerHTML=`¥${total.toFix
ed(2)}`;
        }
    }
  </script>
</body>
</html>
```

运行结果:

商品名称	单价	数量	小计
iPhone6	¥4488.00	- 4 +	¥17952.00
iPhone6 plus	¥5288.00	- 6 +	¥31728.00
iPad Air 2	¥4288.00	- 1 +	¥4288.00
		Total:	¥53968.00

#DOMday02:

正课:

1. 修改:

一. 修改: 3 种东西

1. 内容: 3 种:

(1). 获取或修改元素开始标签到结束标签之间的原始 HTML 内容: 元素.innerHTML

(2). 获取或修改元素开始标签到结束标签之间的纯文本内容: 元素.textContent

textContent vs innerHTML

1. 获取内容时:

(1). innerHTML: 不做任何加工, 直接取出原始的 HTML 代码内容

(2). textContent: 将原始内容加工后, 只返回纯文本正文内容

a. 去掉所有内嵌的标签

b. 将特殊符号翻译为正文

2. 修改内容时:

(1). 交给 innerHTML 的内容, 先交给浏览器解析, 将解析后的内容显示给人看

(2). 交给 textContent 的内容, 不交给浏览器解析, 而是原样显示内容

(3). 获取或设置表单元素的值:

~~a. 因为大部分表单元素, 比如<input>是单标记, 所以不能用 innerHTML 或 textContent 获得内容~~

b. 正确: 元素.value

(4). 示例: 获取和修改各种元素的内容和值

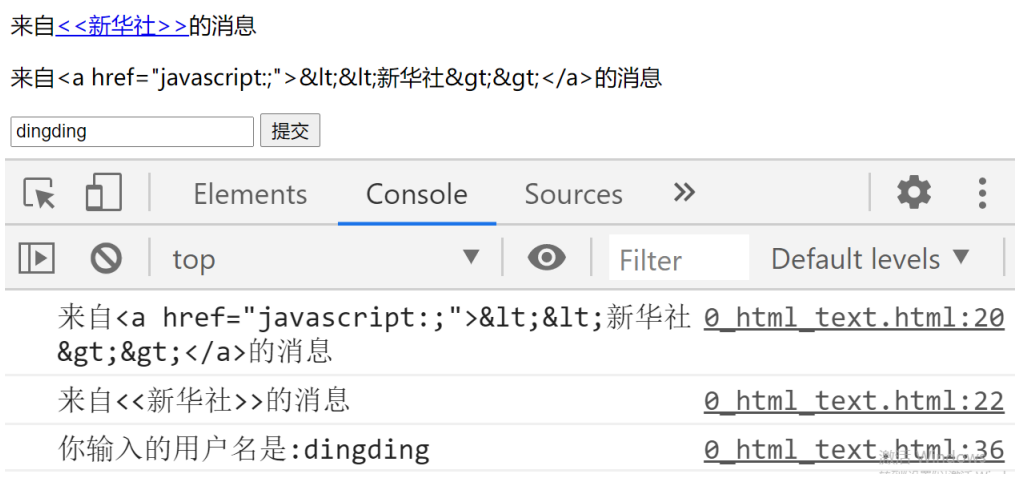
0_html_text.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>
  <p id="p1">来自<a href="javascript:;">&lt;&lt;新华社&gt;&gt;</a>的消息</p>
  <p id="p2"></p>
  <p id="p3"></p>
  <input id="uname" type="text"/>
  <button id="btn">提交</button>
  <script>
    var p1=document.getElementById("p1");
```

```
    var p2=document.getElementById("p2")
;
    var p3=document.getElementById("p3")
;
    //获取内容:
    //想获得 p1 中的原始的 HTML 内容
    console.log(p1.innerHTML);
    //想获得 p1 中的纯文本内容
    console.log(p1.textContent);
    //修改内容:
    //想修改 p2 的原始的 HTML 内容:
    p2.innerHTML=`来自
<a href="javascript:;">&lt;&lt;新华社
&gt;&gt;</a>的消息`;
    //想修改 p3 的纯文本内容:
    p3.textContent=`来自
<a href="javascript:;">&lt;&lt;新华社
&gt;&gt;</a>的消息`;

    //想点提交按钮时，获取文本框中用户输入的
内容
```

```
    var btn=document.getElementById("btn
");
    btn.onclick=function(){
      var uname=document.getElementById(
"uname");
      //错误:
      // console.log(`你输入的用户名
是:${uname.innerHTML}`);
      //正确:
      console.log(`你输入的用户名
是:${uname.value}`);
    }
  </script>
</body>
</html>
```

运行结果:



(5). 示例: 开关门效果:

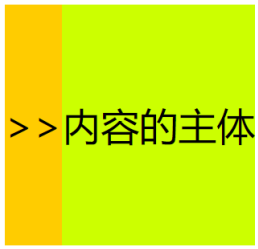
0_door.html

```
<!DOCTYPE HTML>
<html>
<head>
<title>读取并修改元素的内容</title>
<meta charset="utf-8" />
<style>
  div{float:left; height: 100px; line-
height: 100px; }
  #d1,#d3{ background-color: #ccff00; }
  #d2{ cursor: pointer; background-
color: #ffcc00; }
</style>
</head>
<body>
  <div id="d1">树形列表</div>
  <div id="d2">&lt;&lt;</div>
  <div id="d3">内容的主体</div>
  <script>
    //DOM 4 步
    //1. 查找触发事件的元素
```

```
    //本例中：用户点击 d2 触发变化
    var d2=document.getElementById("d2")
;
    //2. 绑定事件处理函数
    //本例中：用户单击 d2 触发变化
    d2.onclick=function(){
      //3. 查找要修改的元素
      //本例中：要修改 d1
      var d1=document.getElementById("d1
");
      //4. 修改元素
      //本例中：
      //如果 d1 是关着的(d1 的 style 中的
display 属性为 none)
      if(d1.style.display=="none"){
        //就打开 d1(清除 d1 的 style 中的
display 属性为"")
        d1.style.display="";
        //顺便将 d2 的内容改为<<,表示下次再
点就是关闭
        this.innerHTML="&lt;&lt;";
      }else{//否则如果 d1 是开着的，
```

```
            //就关上 d1(修改 d1 的 style 中的
display 属性为 none)
            d1.style.display="none";
            //顺便将 d2 的内容改为>>,表示下次再
点就是打开
            this.innerHTML="&gt;&gt;";
        }
    }
  </script>
</body>
</html>
```

运行结果:



2. 属性: 3 种:

(1). 字符串类型的 HTML 标准属性:

a. 什么是: HTML 标准中规定的大部分元素都有的值为字符串的 HTML 属性

b. 比如: title id name href src ...

c. 如何: 2 种:

1). 早期核心 DOM 四个函数:

i. 获取属性值: var 属性值=元素.getAttribute("属性名")

获取 属性

ii. 修改属性值: 元素.setAttribute("属性名","属性值")

设置 属性

iii. 判断是否包含某个属性: var bool= 元素.hasAttribute("属性名")

包含 属性

iv. 移除属性: 元素.removeAttribute("属性名")

移除 属性

v. 示例: 使用核心 DOM4 个函数操作元素属性

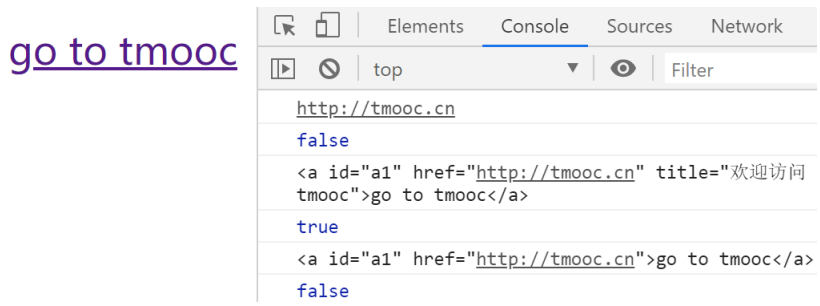
1_attribute.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
```

```
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>
  <a id="a1" href="http://tmooc.cn">go to tmooc</a>
  <script>
    var a1=document.getElementById("a1");
    //想获得 a1 的 href 属性值:
    var href=a1.getAttribute("href");
    console.log(href);
    //想判断 a1 是否包含 title 属性
    var bool=a1.hasAttribute("title");
    console.log(bool);//false
    //想为 a1 设置 title 属性
    a1.setAttribute("title","欢迎访问tmooc")
    console.log(a1.outerHTML);//输出自己的 HTML 元素代码
    //再判断 a1 是否包含 title 属性
```

```
    var bool=a1.hasAttribute("title");
    console.log(bool);//true
    //想移除 a1 的 title 属性
    a1.removeAttribute("title")
    console.log(a1.outerHTML);//输出自己的 HTML 元素代码
    //再判断 a1 是否包含 title 属性
    var bool=a1.hasAttribute("title");
    console.log(bool);//false
  </script>
</body>
</html>
```

运行结果:



2). 新版 HTML DOM 简化方式: 已经提前将元素所有可用的 HTML 标准属性都添加到内存中元素节点对象上! 只不过页面上暂时没有出现的属性, 值暂时为""而已。所以, 我们可以用"元素.属性名"来访问

HTML 标准属性

i. 获取属性值: 元素.属性名

ii. 修改属性值: 元素.属性名="属性值"

iii. 判断是否包含某个属性: 元素.属性名!==""

iv. 移除属性: 元素.属性名=""

v. 特例: class 属性:

html 中: <元素 class="class 名">

js 中: ~~元素对象.class="class 名"~~

ES6 中，class 是 js 的关键字，表示创建一种类型的意思!

DOM 中，元素对象.className

vi. 示例: 使用 HTML DOM 方式简化操作元素属性

1_property.html

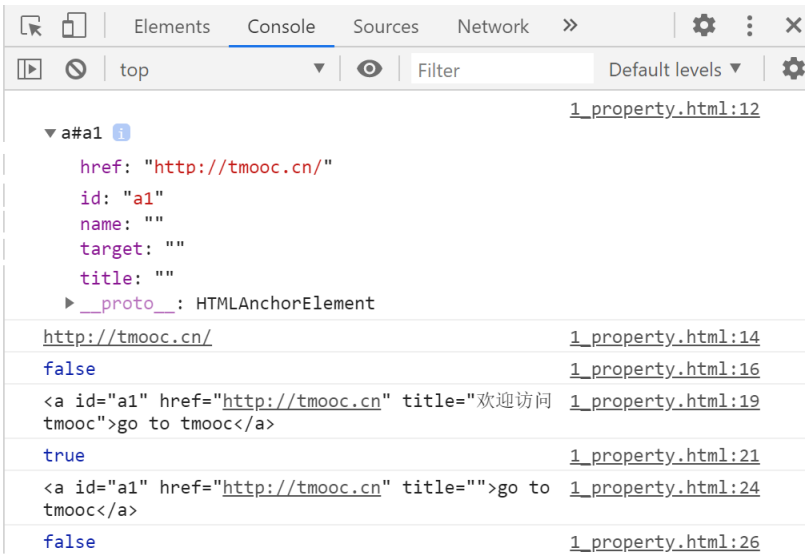
```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
```

```
<body>
  <a id="a1" href="http://tmooc.cn">go t
o tmooc</a>
  <script>
    var a1=document.getElementById("a1")
;
    console.dir(a1);
    //想获得 a1 的 href 属性值:
    console.log(a1.href);
    //想判断 a1 是否包含 title 属性
    console.log(a1.title!=="");//false
    //想为 a1 设置 title 属性
    a1.title="欢迎访问 tmooc";
    console.log(a1.outerHTML);
    //再判断 a1 是否包含 title 属性
    console.log(a1.title!=="");//true
    //想移除 a1 的 title 属性
    a1.title="";
    console.log(a1.outerHTML);
    //再判断 a1 是否包含 title 属性
    console.log(a1.title!=="");//false
  </script>
```

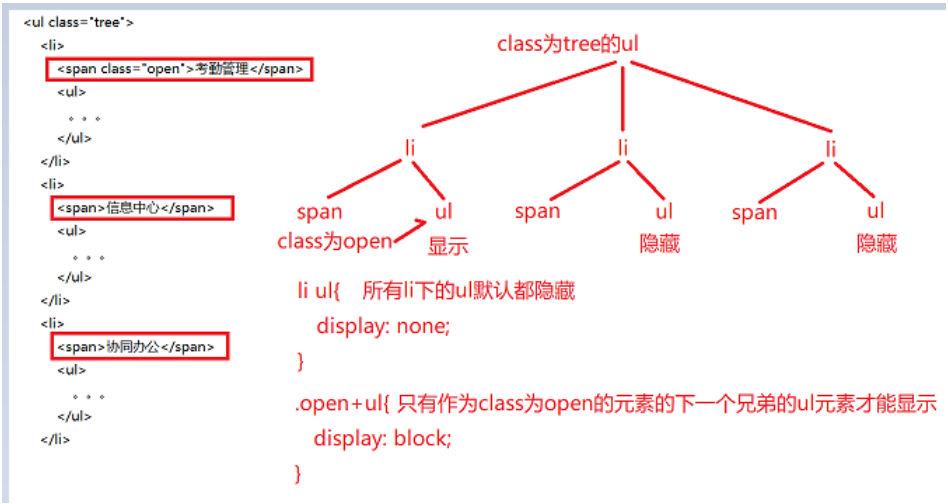
```
</body>
</html>
```

运行结果:

go to tmooc



d. 示例：下拉菜单，手风琴效果:



1_menu.html

```
<!DOCTYPE HTML>
<html>
<head>
```

```
<title>1. 实现伸缩二级菜单</title>
<meta charset="utf-8" />
<style>
  li{
    list-style:none;
  }
  li span{
    padding-left: 20px;
    cursor: pointer;
    background: url("../images/add.png")
 no-repeat center left;
  }
  li ul{
    display: none;
  }
  .open{
    background: url("../images/minus.png
") no-repeat center left;
  }
  .open+ul{
    display: block;
  }
```

```
</style>
</head>
<body>
  <ul class="tree">
    <li>
      <span class="open">考勤管理</span>
      <ul>
        <li>日常考勤</li>
        <li>请假申请</li>
        <li>加班/出差</li>
      </ul>
    </li>
    <li>
      <span>信息中心</span>
      <ul>
        <li>通知公告</li>
        <li>公司新闻</li>
        <li>规章制度</li>
      </ul>
    </li>
    <li>
      <span>协同办公</span>
```

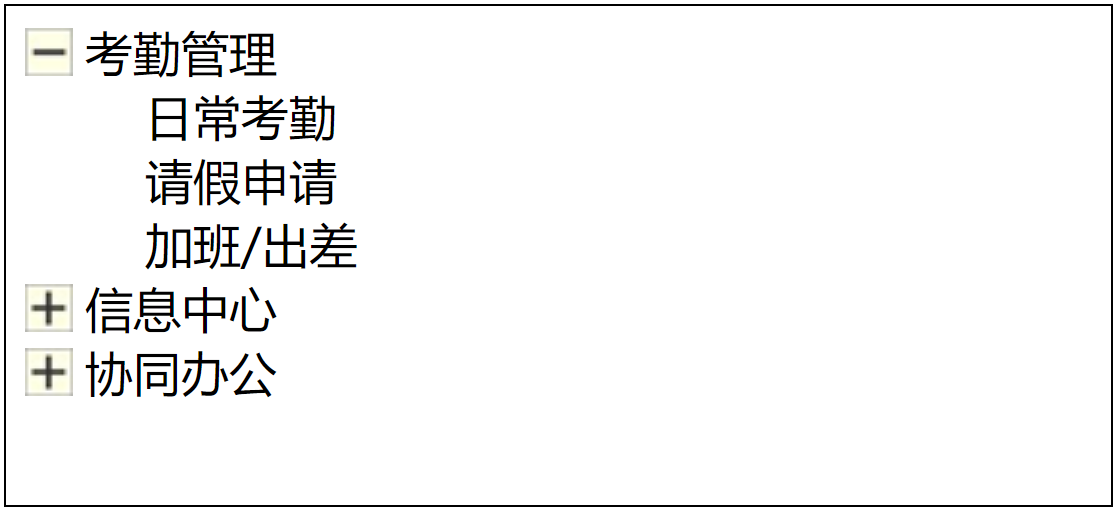
```
        <ul>
          <li>公文流转</li>
          <li>文件中心</li>
          <li>内部邮件</li>
          <li>即时通信</li>
          <li>短信提醒</li>
        </ul>
      </li>
    </ul>
    <script>
      //需求:
      //点击一级菜单:
      //  如果自己(当前 span)是开着的，只关闭自己（当前 span）即可，不用管别人
      //  否则如果自己(当前 span)是关着的，就需要做两件事
        //  先找到另一个开着的菜单(另一个 class 为 open 的 span),把它先关上
        //  再把自己(当前 span)打开
      //结果: 同一时刻最多只可能有一个菜单是开着的!
      //        也有可能所有菜单都关上
```

```
    //DOM 4步
    //1. 查找触发事件的元素
    //本例中：用户点class为tree的ul下的每个li下的span触发变化
    var spans=document.querySelectorAll("ul.tree>li>span");
    //2. 绑定事件处理函数
    //本例中：每个span都可以单击
    //遍历spans类数组对象中每个span元素
    for(var span of spans){
      //每遍历一个span元素，都要绑定单击事件
      span.onclick=function(){
        //测试：点哪个span，让哪个span的内容变成❀
        //this.innerHTML="❀";
        //3. 查找要修改的元素
        //本例中：注定要修改的就是当前span自己——this
        //所以不用找！
        //4. 修改元素：
```

```
        //如果当前 span 是开着的(如果当前
span 的 class 是 open，则当前 span 旁边的 ul
一定是开着的)
        if(this.className=="open"){
          //只关闭自己即可
          this.className=""; //清除当前
span 的 class open，当前 span 旁边的 ul，就默
认隐藏了！
        }else{//否则如果当前 span 是关着的
          //先尝试查找另一个开着的 span——注
定最多只能找到一个
          //查找 class 为 tree 下的 class 为
open 的 span
          var openSpan=document.querySel
ector(
            "ul.tree>li>span.open"
          );//如果没找到返回 null
          //如果找到另一个开着的 span
          if(openSpan!=null){
            //就把另一个开着的 span 先关上
```

```
                openSpan.className="";//清除另一个开着的 span 的 class open，则另一个开着的 span 旁边的 ul 默认隐藏
            }
            //无论前边是否找到另一个开着的 span。最后，只要自己是关着的，都要把自己打开！
            this.className="open";//为当前 span 自己加 class open，当前 span 旁边的 ul 就受牵连而显示！
        }
    }
}
//现在还不到你们原创的时候！
//现阶段！就是处在照猫画虎的阶段！
//你们要做的是反复练习人话！
  </script>
</body>
</html>
```

运行结果：



(2). bool 类型的 HTML 标准属性:

a. 什么是: HTML 标准中规定的, 但是不需要指定属性值, 只要放在元素上就起作用的属性!

b. 比如: disabled 禁用 checked 选中 ...

c. 如何: 只有一种访问方式:

~~1). 错误做法: 不能用核心 DOM4 个函数访问。因为核心 DOM4 个函数只支持字符串类型的属性值! 不支持 bool 类型的属性值!~~

2). 正确做法: 只能用 HTML DOM "元素.属性名", 且属性值都是 bool 类型的 true 或 false!

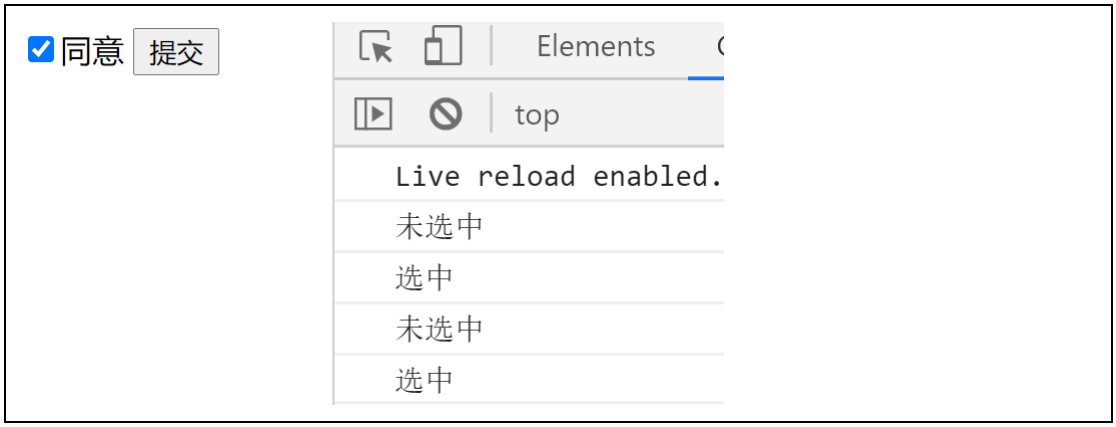
d. 示例: 使用 HTML DOM 访问元素的 bool 类型的 HTML 标准属性:

2_bool_property.html

```
<!DOCTYPE html>
<html lang="en">
```

```
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>
  <input id="chb" type="checkbox">同意
  <button id="btn1">提交</button>
  <script>
    var btn1=document.getElementById("btn1");
    btn1.onclick=function(){
      var chb=document.getElementById("chb");
      console.log(chb.checked==true?"选中":"未选中");
    }
  </script>
</body>
</html>
```

运行结果:



e. 补充: CSS3 中提供了一个选择器——: checked, 专门匹配选中的 checkbox 或 radio 元素

f. 示例: 全选和取消全选

2_selectAll.html

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8" />
<title>全选和取消全选</title>
</head>
<body>
  <h2>管理员列表</h2>
  <table border="1px" width="500px">
    <thead>
    <tr>
```

```
      <th><input type="checkbox"/>全选
</th>
      <th>管理员 ID</th>
      <th>姓名</th>
      <th>操作</th>
    </tr>
    </thead>
    <tbody>
      <tr>
        <td><input type="checkbox"/></td
>
        <td>1</td>
        <td>Tester</td>
        <td>修改 删除</td>
      </tr>
      <tr>
        <td><input type="checkbox"/></td
>
        <td>2</td>
        <td>Manager</td>
        <td>修改 删除</td>
      </tr>
```

```
      <tr>
        <td><input type="checkbox"/></td
>
        <td>3</td>
        <td>Analyst</td>
        <td>修改 删除</td>
      </tr>
      <tr>
        <td><input type="checkbox"/></td
>
        <td>4</td>
        <td>Admin</td>
        <td>修改 删除</td>
      </tr>
    </tbody>
  </table>
  <button>删除选定</button>
  <script>
    //需求:
    //1. 点表头中的 checkbox，可控制表体中
每行第一个 checkbox 的选中状态
```

```
    //2. 点表体中每行第一个 checkbox，有可能反过来影响表头中的 checkbox 的选中状态


    //需求一：点表头中的 checkbox，可控制表体中每行第一个 checkbox 的选中状态
    //DOM 4 步
    //1. 查找触发事件的元素
    //本例中：用户点表头中的 input，触发下边 input 的变化
    var chbAll=document.querySelector("thead>tr>th:first-child>input");
    console.log(chbAll);
    //2. 绑定事件处理函数
    //本例中：用户单击全选按钮，触发下边的变化
    chbAll.onclick=function(){
      //this->全选按钮 chbAll
      //3. 查找要修改的元素
      //本例中：点表头中的全选，要修改表体中每行第一个 td 中的 input
      var chbs=document.querySelectorAll("tbody>tr>td:first-child>input");
```

```
        //4. 修改元素
        //如果全选按钮是选中的
          //则下边每个按钮也都要选中
        //否则如果全选按钮时未选中的
          //则下边每个按钮也都要取消选中
        //结论: 其实就是让下边每个 chb 的
checked 属性值和全选按钮的 checked 属性值保
持一致!
        for(var chb of chbs){
          chb.checked=this.checked;
        }
      }

      //需求二: 点表体中每行第一个 checkbox,
有可能反过来影响表头中的 checkbox 的选中状态
      //DOM 4 步:
      //1. 查找触发事件的元素
      //本例中: 用户点的是表体中每行第一个 td
中的 checkbox
      var chbs=document.querySelectorAll("
tbody>tr>td:first-child>input");
      //2. 绑定事件处理函数
```

```
    //本例中：表体中每个 chb 都可以点击
    //遍历 chbs 中每个 chb
    for(var chb of chbs){
      //每遍历一个 chb 元素，为其就绑定单击事件
      chb.onclick=function(){
        //3. 查找要修改的元素
        //本例中：当单击下方每个 chb 时，都有可能修改表头中第一个 th 中的 input
        var chbAll=document.querySelector("thead>tr>th:first-child>input");
        //4. 修改元素
        //尝试在 tbody 中查找一个未选中的 checkbox
        var unchecked=document.querySelector(
          "tbody>tr>td:first-child>input:not(:checked)"
        );
        //如果找到了这样一个 checkbox，则 chbAll 不选中。
        if(unchecked!=null){
```

```
            chbAll.checked=false;
        }else{//否则如果没找到这样一个
checkbox，则 chbAll 选中
            chbAll.checked=true;
        }
      }
    }
  </script>
</body>
</html>
```

运行结果：

全选	管理员ID	姓名	操作
☐	1	Tester	修改 删除
☐	2	Manager	修改 删除
☐	3	Analyst	修改 删除
☐	4	Admin	修改 删除

全选	管理员ID	姓名	操作
☑	1	Tester	修改 删除
☐	2	Manager	修改 删除
☑	3	Analyst	修改 删除
☑	4	Admin	修改 删除

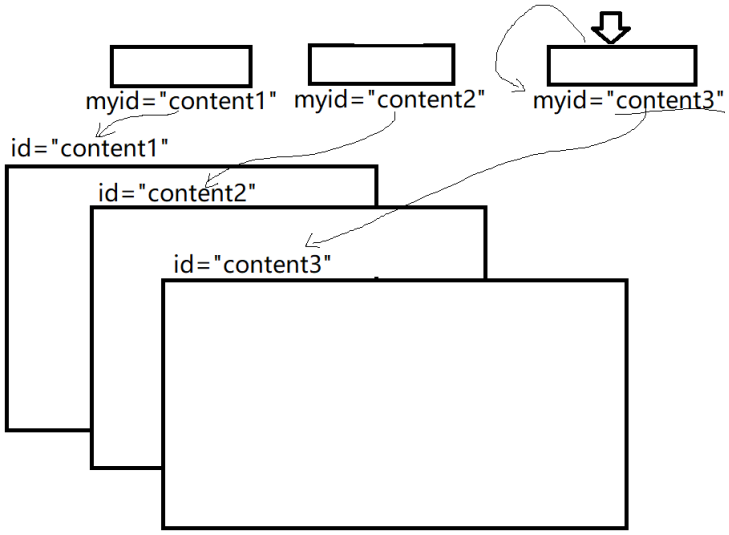
全选	管理员ID	姓名	操作
☑	1	Tester	修改 删除
☑	2	Manager	修改 删除
☑	3	Analyst	修改 删除
☑	4	Admin	修改 删除

(3). 自定义扩展属性:

a. 什么是: HTML 标准中没有定义的，程序员根据自身需要，自发添加的自定义属性。

b. 何时: 2 个场景

1). 在 HTML 元素上缓存一些业务相关的数据



2). 代替其他选择器，专门作为查找触发事件的元素的条件

i. 其他选择器的问题:

id 选择器: 一次只能选一个元素，不能选择多个元素

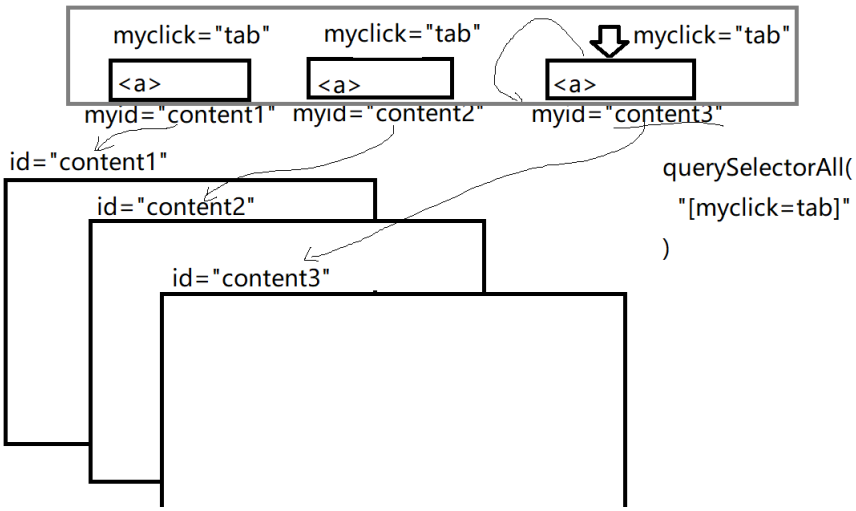
元素选择器: 实现同一种效果，不同的人使用的元素可能各不相同!

类选择器: 本职工作就不是为 js 服务的! 类选择器的本质工作是为了给元素添加样式! 所以，在类选

择器中如果出现与 css 无关的 class 名，很可能被轻易删掉。此时，如果你的 js 刚好用到这个被删除的类名，则 js 程序立刻出错!

ii. 以上三种选择器，尽量不要用于 DOM 中查找触发事件的元素!

iii. 今后在 DOM 中查找触发事件的元素最好都用自定义扩展属性



c. 如何使用自定义扩展属性: HTML5 标准

1). 在 HTML 中元素上添加自定义扩展属性:

<元素 data-自定义属性名="属性值">

2). 在 js 中操作元素的自定义扩展属性:

~~i. 错误: 不能用.直接访问自定义扩展属性~~

因为自定义扩展属性是后天添加的，不在 HTML 标准范围内。所以 HTML DOM 无法提前预知自定义属性，也就无法自动添加到内存中的元素对象上。

ii. 正确: 2 种:

①可以用旧式的核心 DOM4 个函数来操作——没有兼容性问题

获取: 元素对象.getAttribute("data-自定义属性名")

修改: 元素对象.setAttribute("data-自定义属性名","属性值")

强调: 如果用核心 DOM4 个函数, 自定义属性名前必须加 data-才行

②可以用 HTML5 提供的新的简写方式——兼容性问题

如果 html 中元素上定义自定义扩展属性时, 开头加了 data-前缀, 则可用:

元素对象.dataset.自定义属性名

强调:

- 使用 dataset 时, 不用加 data-前缀! dataset 会自动查找 data-前缀的属性
- 只有加了 data-前缀的自定义属性, 才能被 dataset 访问到!

d. 示例: 使用自定义扩展属性查找元素, 并操作元素的自定义扩展属性

3_data-.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>
  <a id="a1" data-click="btn" data-content="欢迎访问tmooc" href="javascript:;">go to tmooc</a>
  <script>
    //想查找带有 data-click 属性的，且属性值为 btn 的 a 元素
    var btn=document.querySelector("[data-click=btn]");
    console.log(btn.outerHTML);
    //想获得 data-content 属性的值
    //console.log(btn.getAttribute("data-content"));
```

```
    console.log(btn.dataset.content);
    //想修改 data-content 属性的值为
Welcome to tmooc
    //btn.setAttribute("data-
content","welcome to tmooc");
    btn.dataset.content="welcome to tmoo
c";
    console.log(btn);
  </script>
</body>
</html>
```

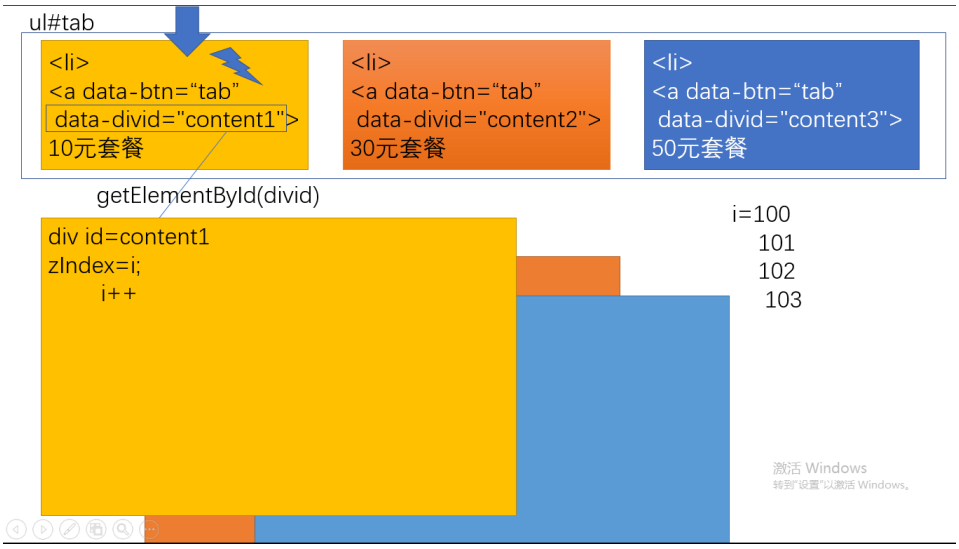
运行结果:

go to tmooc

欢迎访问 tmooc

go to tmooc

e. 示例: 标签页效果:



3_tabs.html

```
<!DOCTYPE HTML>
<html>
<head>
<title>读取并修改元素的属性</title>
<meta charset="utf-8" />
<style>
  *{
    margin:0;
    padding: 0;
  }
  #tab li{
    float: left; list-style: none;
  }
  #tab li a{
```

```
    display:inline-block;
    text-decoration:none;
    width: 80px; height: 40px;
    line-height: 40px;
    text-align: center;
    color:#000;
  }
  #container{
    position: relative;
  }
  #content1,#content2,#content3{
    width: 300px;
    height: 100px;
    padding:30px;
    position: absolute;
    top: 40px;
    left: 0;
  }
  #tab li:first-child,#content1{
    background-color: #ffcc00;
  }
  #tab li:first-child+li,#content2{
```

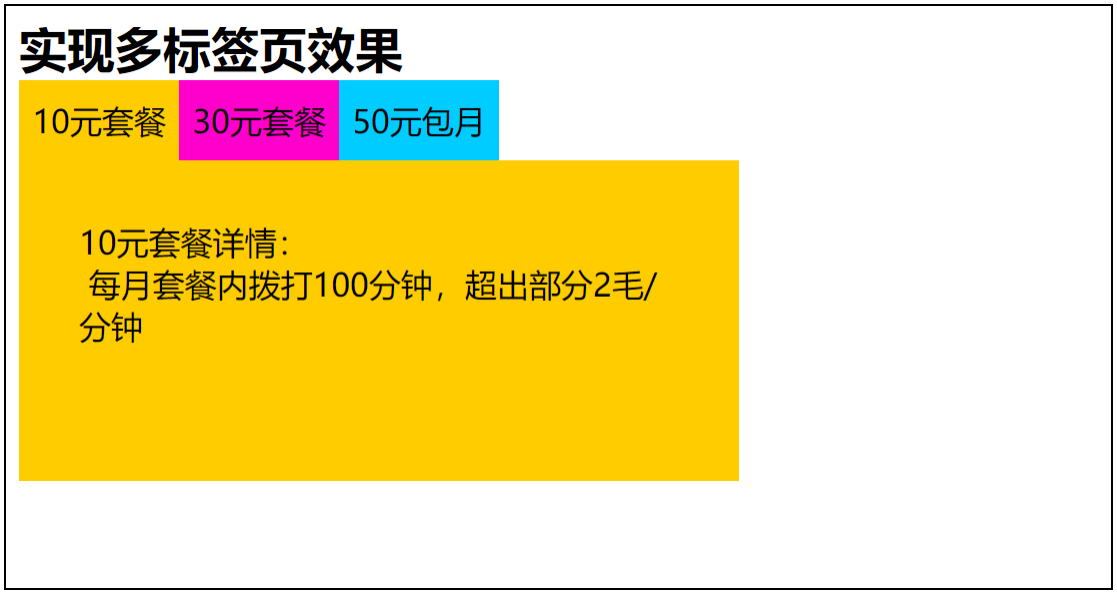
```
    background-color: #ff00cc;
  }
  #tab li:first-child+li+li,#content3{
    background-color: #00ccff;
  }
</style>

</head>
<body>
  <h2>实现多标签页效果</h2>
  <div class="tabs">
    <ul id="tab">
      <li><a href="#content1" data-
btn="tab" data-divid="content1">10 元套餐
</a></li>
      <li><a href="#content2" data-
btn="tab" data-divid="content2">30 元套餐
</a></li>
      <li><a href="#content3" data-
btn="tab" data-divid="content3">50 元包月
</a></li>
    </ul>
```

```
    <div id="container">
      <div id="content1">
        10 元套餐详情：<br /> 每月套餐内拨打 100 分钟，超出部分 2 毛/分钟
      </div>
      <div id="content2">
        30 元套餐详情：<br /> 每月套餐内拨打 300 分钟，超出部分 1.5 毛/分钟
      </div>
      <div id="content3">
        50 元包月详情：<br /> 每月无限量随心打
      </div>
    </div>
  </div>
  <script>
    //准备工作：
    //1. 为每个触发事件的按钮添加自定义属性，用于查找触发事件的元素
    //2. 为每个按钮添加自定义扩展属性，提前保存每个按钮对应的 div 的 id
    //DOM 4 步
```

```
    //1. 查找触发事件的元素
    //本例中：用户点 ul 下的包含 data-btn 属性，且属性值为 tab 的按钮元素触发变化
    var tabs=document.querySelectorAll(`[data-btn=tab]`);
    console.log(tabs);
    //先定义一个变量 i=100，记录目前最大的 zIndex 值
    var i=100;//每点一次标签页，都 i++。
    //结果，下次点标签页时的 i，一定比上次点标签页时 i 大 1
    //2. 绑定事件处理函数
    //本例中：每个标签都可以点
    //遍历 tabs 中每个标签按钮
    for(var tab of tabs){
      //每遍历一个标签按钮就要为其绑定单击事件
      tab.onclick=function(){
        //3. 查找要修改的元素
        //本例中：找到当前标签按钮对应的 div
        //2 步：
```

```
            //3.1 从当前按钮自己身上的 data-divid 属性中取出对一个 div 的 id 名
            var divid=this.getAttribute("data-divid");
                        //this.dataset.divid
            //3.2 用拿到的 divid，查找对应的 div 元素对象
            var div=document.getElementById(divid);
            //4. 修改元素
            //本例中：修改当前按钮对应的 div 的 zIndex 值最大！就可以让当前按钮对应 div 到最上方来，挡住其它 div
            div.style.zIndex=i;
            i++;
        }
    }
  </script>
</body>
</html>
```

运行结果：



3. 样式:

(1). 修改元素的内联样式:

元素的 style 属性中的 css 属性为属性值

a. html 中: <元素 style="css 属性:属性值; ..."></元素>

b. js 中: 元素对象.style.css 属性="属性值";

的 的

c. 坑: 有些 css 属性名中带-, -会和 js 中的减法的减号冲突!

d. 解决: 所有 css 属性名中带-的属性, 都要去横线变驼峰

比如: background-color backgroundColor

font-size　　fontSize

e. 强调: 大小，长短，位置有关的 css 属性值必须加 px 单位！不加单位不生效!

#DOMday03:

正课:

1. 修改: 修改样式
2. 添加/删除
3. HTML DOM 常用对象
4. BOM

一. 修改: 3 种

1. 内容: 3 种
2. 属性: 3 种
3. 样式:

(1).修改内联样式:

a. 元素.style.css 属性名="属性值"

//自动翻译

b. 相当于: html 中: <元素 style="...；css 属性:属性值"

c. 坑: .style 属性仅代表元素的内联样式，无法获取或修改样式表(内部和外部)中的样式。但是，实际开发中，考虑到可维护性，几乎不会用内联样式！几乎所有样式都是写在样式表中的。所以，用.style 属性，很可能只能获得内联样式，无法获得外部的大部分样式。

(2). 获取元素的完整样式:

a. 其实，每个元素都有一个计算后的样式（computedStyle），包含最终应用到这个元素上的所有 css 属性的集合!

b. 将来，只要获取样式，都应该获取计算后的样式，而不应该用 style 属性

c. 如何: 2 步:

1). 先获得这个元素所有计算后的样式的集合对象:

var style=getComputedStyle(元素对象);

2). 从 style 中获取某一个 css 属性继续使用:

var 属性值=style.css 属性名

d. 强调: 所有通过 getComputedStyle()获得的计算后的样式，都是只读的！禁止修改!

原因: 计算后的样式来源不确定，不确定同时有多少元素在使用这个 css 属性。一旦擅自修改，会牵一发而动全身!

(3). 总结:

a. 只要修改一个 css 属性，都用: 元素.style.css 属性="属性值" —— 只修改内联

b. 只要获取元素的 css 属性，都用: getComputedStyle(元素对象) —— 获得计算后的完整样式!

(4). 示例: 分别使用 style 和 getComputedStyle()获取和修改元素的 css 属性

0_style_getComputedStyle.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
  <style>
    h1{
      background-color:red;
    }
  </style>
</head>
```

```
<body>
  <h1 id="h1" style="color:yellow">欢迎访
问 tmooc</h1>
  <p>welcome to my first web site</p>
  <script>
    var h1=document.getElementById("h1")
;
    //想获得 h1 的字体颜色，背景颜色，字号
    //使用 style:
    // console.log(h1.style.color);//yel
low
    // console.log(h1.style.backgroundCo
lor);//空
    // console.log(h1.style.fontSize);//
空

    //获取计算后样式:
    var style=getComputedStyle(h1);
    console.log(style.color);//
    console.log(style.backgroundColor);/
/
    console.log(style.fontSize);//
```

```
    //想修改 h1 的字体为 64px
    //错误：修改计算后的样式
    //style.fontSize="64px";//报错：
Failed to set the 'font-
size' property on 'CSSStyleDeclaration':
 These styles are computed, and therefor
e the 'font-size' property is read-only.
    //正确：修改内联，仅影响当前元素自己，不
影响别人
    h1.style.fontSize="64px";
  </script>
</body>
</html>
```

运行结果:

rgb(255, 255, 0)

rgb(255, 0, 0)

32px

(5). 问题: .style 属性一句话只能修改一个 css 属性。但是实际开发中，一个效果的变化，可能同时要修改多个 css 属性。如果用.style，就会很繁琐。

.style.border="1px solid green"
.style.color="green"
.style.backgroundColor="lightGreen"
.style.backgroundImage="url(ok.png)"

.style.border="1px solid red "
.style.color=" red "
.style.backgroundColor="pink "
.style.backgroundImage="url(err.png)"

(6). 解决: 今后只要批量修改一个元素的多个 css 属性, 首选 class 修改

(7). 如何: 2 步

a. 先在 css 中提前定义好要用的各种备选 class 样式

b. 在 js 中根据程序需要, 动态修改元素的 class 为指定的某一种 class 样式

(8). 无论何时, 需要批量修改元素的样式, 只需要一句话就够了!

css中: 准备两身衣服

```
.success{
  border:1px solid green;
  color:green;
  background-color:lightGreen;
  background-image:url(ok.png)
}
```

```
.fail{
  border: 1px solid red;
  color: red;
  background-color:pink;
  background-image:url(err.png)
}
```

span.className="success"

span.className="fail"

(9). 示例: 带样式的表单验证:

css中:

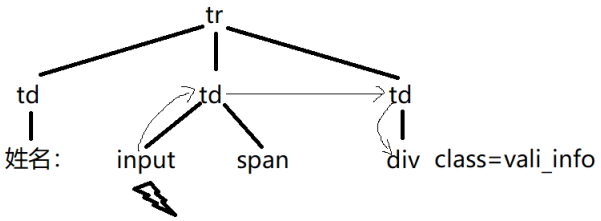
```
.vali_info{
    display:none
}
```

```
.vali_success{
}
```

```
.vali_fail{
}
```

xxxxxxx

```
<tr>
 <td>姓名：</td>
 <td>
  <input name="username"/>
  <span>*</span>
 </td>
 <td>
  <div class="vali_info">
     10个字符以内的字母、数字或下划线的组合
  </div>
 </td>
</tr>
```



0_valiWithCss.html

```
<!doctype html>
<html>
 <head>
  <meta charset="UTF-8">
  <title>实现带样式的表单验证</title>
  <style>
    table{width:700px}
    td:first-child{width:60px}
    td:nth-child(2){width:200px}
    td:first-child+td{width:200px}
    td span{color:red}
    /*文本框旁边 div 隐藏的根源*/
```

```
    .vali_info{
      display:none;
    }
    .txt_focus{
      border-top:2px solid black;
      border-left:2px solid black;
    }

    .vali_success,.vali_fail{
      background-repeat:no-repeat;
      background-position:left center;
      display:block;
    }
    /*文本框验证通过时 div 的样式*/
    .vali_success{
        background-
image:url("../images/ok.png");
      padding-left:20px;
      width:0px;height:20px;
      overflow:hidden;
    }
    /*文本框验证失败时 div 的样式*/
```

```
    .vali_fail{
        background-
image:url("../images/err.png");
        border:1px solid red;
        background-color:#ddd;
        color:Red;
        padding-left:30px;
    }


  </style>
 </head>
 <body>
  <form id="form1">
    <h2>增加管理员</h2>
    <table>
      <tr>
        <td>姓名：</td>
        <td>
          <input name="username"/>
          <span>*</span>
        </td>
```

```
        <td>
          <div class="vali_info base1 ba
se2">
            10 个字符以内的字母、数字或下划
线的组合
          </div>
        </td>
      </tr>
      <tr>
        <td>密码：</td>
        <td>
          <input type="password" name="p
wd"/>
          <span>*</span>
        </td>
        <td>
          <div class="vali_info">6 位数字
</div>
        </td>
      </tr>
      <tr>
        <td></td>
```

```
        <td colspan="2">
          <input type="submit" value="保
存"/>
          <input type="reset" value="重填
"/>
        </td>
      </tr>
      </table>
  </form>
<script>
//只实现姓名文本框的验证:
//2 个需求:
//1. 文本框获得焦点时，显示右边的提示
//2. 文本框失去焦点时，验证文本款中的内容
  //如果验证通过，就修改右边的提示为正确的样
式
  //如果验证不通过，就修改右边的提示为错误的
样式

//实现第一个需求: 文本框获得焦点时，显示右边
的提示
//DOM 4 步:
```

```
//1. 查找触发事件的元素
//本例中：name 属性=username 的文本框获得焦点触发变化
//元素缩写  业务单词
//    ↓      ↓
var txtName=
  document.getElementsByName("username")[0];//复习
console.log(txtName);
//2. 绑定事件处理函数
//本例中：文本框获得焦点时触发变化
txtName.onfocus=function(){
  //3. 查找要修改的元素
  //本例中：当前文本框获得焦点，却要修改当前文本框的父元素 td 的下一个兄弟元素 td 的第一个孩子元素 div 的样式
  var div=this.parentNode.nextElementSibling.children[0];
  //4. 修改元素
  //本例中：div 因为带有 vali_info class 才默认隐藏，所以，我们需要将 div 的 class 暂时清除，让 div 显示出来
```

```
  div.className="";
}

//实现第二个需求：当文本框失去焦点时，验证文本框的内容，修改旁边 div 的样式
//DOM 4 步
//1. 查找触发事件的元素
//本例中：还是姓名文本框失去焦点——上边已经找过了 txtName
//2. 绑定事件处理函数
//本例中：文本框失去焦点时，触发验证和修改
txtName.onblur=function(){
  //3. 查找要修改的元素
  //本例中：当前文本框失去焦点，却要修改当前文本框的父元素 td 的下一个兄弟元素 td 的第一个孩子元素 div 的样式
  var div=this.parentNode.nextElementSibling.children[0];
  //4. 修改元素
  //定义正则表达式
  var reg=/^\w{1,10}$/;
  //如果验证文本框的内容通过
```

```
    if(reg.test(this.value)==true){
        //就修改 div 为正确的样式
        div.className="vali_success";
    }else{//否则如果验证文本框的内容不通过
        //就修改 div 为错误的样式
        div.className="vali_fail";
    }
}
</script>
 </body>
</html>
```

运行结果：

二. 添加/删除

1. 添加一个新元素: 3 步

(1). 创建一个新的空元素对象:

var 新对象=document.createElement("标签名")

比如: 想创建一个 a 元素:

var a=document.createElement("a");

结果:<a>

(2). 为新元素添加必要属性:

比如: 想将 a 设置为跳转到 tmooc 的一个超链接

a.href="http://tmooc.cn";

a.innerHTML="go to tmooc";

结果:

 go to tmooc

(3). 将新元素添加到 DOM 树

a. 问题: 已经创建出来，且已经设置了关键属性的元素，竟然在网页上看不到!

b. 原因: 浏览器要显示网页内容，必须经过三个阶段!

1). 扫描 HTML 代码，生成 DOM 树

2). 根据 DOM 树中的元素结构，对网页进行排版

3). 根据排版的布局和位置绘制网页中的内容。

而新添加的元素，默认是不在 DOM 树中的！浏览器根本不知道页面上需要显示新元素！

c. 解决：今后，只要希望页面上添加一个新元素，都要将新元素先添加到 DOM 树的指定位置。然后浏览器才能自动侦测到 DOM 树的变化，重新排版和绘制网页！

d. 如何：把元素添加到 DOM 树，3 种：

1). 在一个指定父元素下所有直接子元素末尾追加新元素：

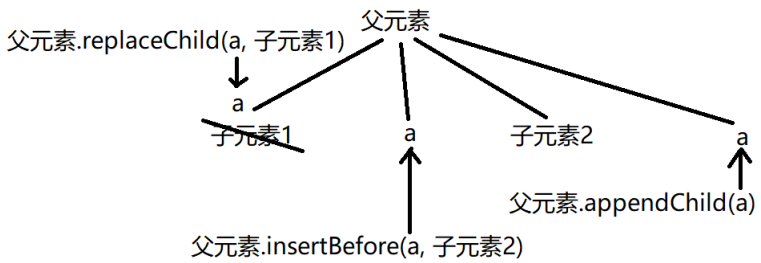
父元素.appendChild(新元素)

2). 在一个指定父元素下的某个子元素之前插入新元素：

父元素.insertBefore(新元素，现有子元素)

3). 替换一个指定父元素下的某个现有子元素

父元素.replaceChild(新元素，现有子元素)



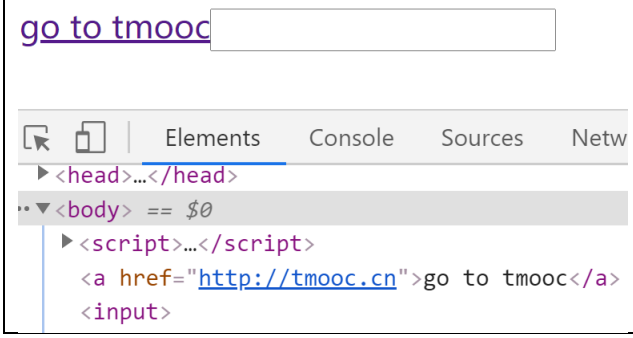
2. 示例：在页面上添加一个 a 元素和一个 input 元素

1_createElement.html

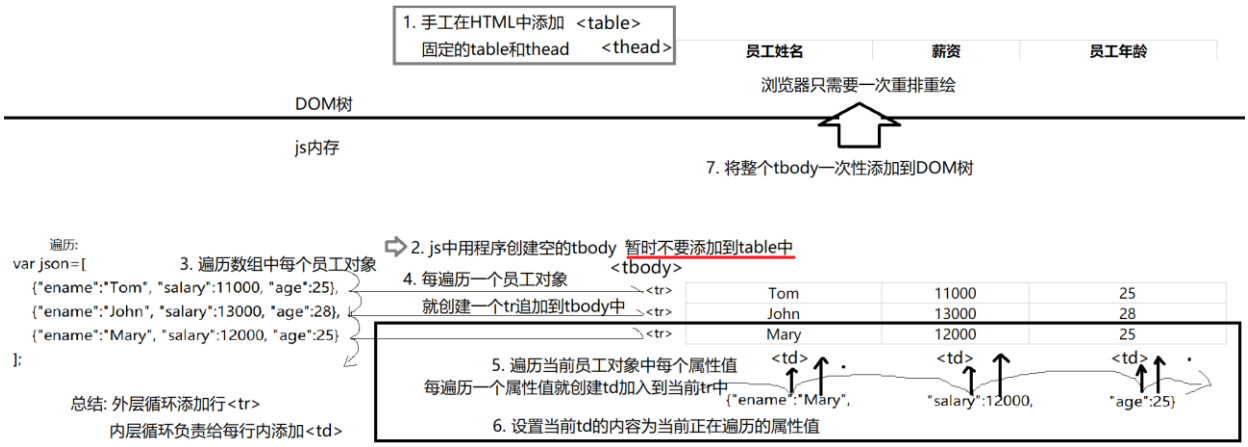
```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>
  <script>
    //想创建 a
    var a=document.createElement("a");
    console.log(a.outerHTML);
    a.href="http://tmooc.cn";
    a.innerHTML="go to tmooc";
    console.log(a);
    //想把 a 追加到 body 中
    document.body.appendChild(a);

    //创建一个文本框
    var input=document.createElement("input");
```

```
    //省略第二步
    //希望 input 出现在 a 之后:
    document.body.appendChild(input);
    //希望 input 出现在 a 之前:
    //document.body.insertBefore(input,a
);
    //希望用 input 替换 a
    //document.body.replaceChild(input,a
);
  </script>
</body>
</html>
```

运行结果:



3. 示例: 动态创建表格



1_createTable.html

```
<!DOCTYPE HTML>
<html>
<head>
<title>动态创建表格</title>
<meta charset="utf-8" />
<style>
  table{width:600px; border-
collapse:collapse;
    text-align:center;
  }
  td,th{border:1px solid #ccc}
</style>

</head>
<body>
```

```
  <div id="data">
  <!-- table>thead>tr>th*3 -->
    <table>
      <thead>
        <tr>
          <th>姓名</th>
          <th>薪资</th>
          <th>年龄</th>
        </tr>
      </thead>
    </table>
  </div>
  <script>
  var json=[
    {"ename":"Tom", "salary":11000, "age
":25},
    {"ename":"John", "salary":13000, "ag
e":28},
    {"ename":"Mary", "salary":12000, "ag
e":25}
  ];
```

```
  //1. 在 HTML 中手工添加固定不变的 table 和表头行内容

  //查找 id 为 data 的 div 下的 table 元素
  var table=document.querySelector("#data>table")
  //2. 创建一个 tbody 元素
  var tbody=document.createElement("tbody");
  //暂时不要添加到 table 中
  //table.appendChild(tbody);
  //3. 遍历 json 数组中每个员工对象
  for(var emp of json){
    //4. 每遍历一个员工对象，就创建一个 tr 元素，追加到 tbody 中
    var tr=document.createElement("tr");
    tbody.appendChild(tr);
    //5. 遍历当前员工对象中每个属性
    for(var key in emp){
      //每遍历一个属性值，就创建 td，追加到 tr 中
```

```
        var td=document.createElement("td"
);
        tr.appendChild(td);
        //6. 设置 td 的内容为当前属性的属性值
        td.innerHTML=emp[key];
      }
    }

    //7. 当 tbody 元素下所有子内容填充完成之
后，再将 tbody 一次性添加到 dom 树
    table.appendChild(tbody);
    </script>
</body>
</html>
```

运行结果：

员工姓名	薪资	员工年龄
Tom	11000	25
John	13000	28
Mary	12000	25

4. 优化:

(1). 问题: 每修改一次 DOM 树, 浏览器都要重新排版，重新绘制。如果频繁修改 DOM 树，浏览器就被

迫重新反复排版，反复绘制！——加载效率极低

(2). 解决: 今后，应该尽量减少操作 DOM 树的次数!

(3). 如何: 2 种情况:

a. 如果同时添加父元素和子元素时，应该现在内存中，将所有子元素添加到父元素中。最后再一次性将父元素添加到 DOM 树。

b. 父元素已经在页面上了! 要添加多个平级子元素。应该借助于一种新的对象——文档片段对象（临时托盘）

1). 什么是文档片段对象： 内存中临时保存多个平级子元素的虚拟父元素对象

2). 何时: 父元素已经在页面上了！要添加多个平级子元素时，都要用文档片段对象。

3). 如何: 3 步:

i. 先创建文档片段对象:

var 文档片段=document.createDocumentFragment();

创建文档片段

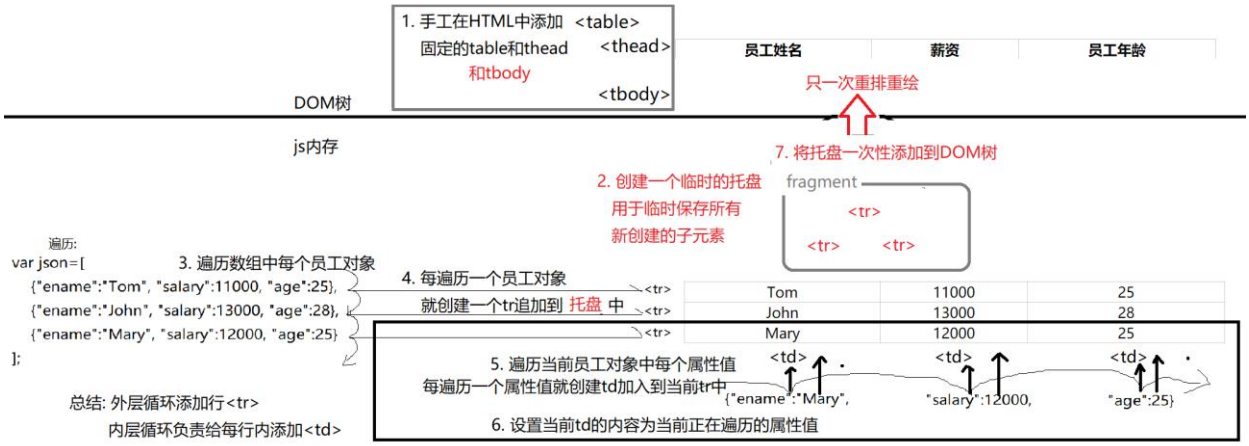
ii. 再将子元素添加到文档片段对象中:

文档片段.appendChild(子元素)

iii. 最后再将整个文档片段一次性添加到 DOM 树中

父元素.appendChild(文档片段);

4). 强调: 临时创建的文档片段对象，再将子元素追加到他们真正的父元素下之后，文档片段对象就释放了！不会成为页面上真正的元素占用页面空间!



5). 示例: 如果 tbody 已经在页面上了，如何添加多个 tr

1_createTable2.html

```
<!DOCTYPE HTML>
<html>
<head>
<title>动态创建表格</title>
<meta charset="utf-8" />
<style>
```

```
  table{width:600px; border-
collapse:collapse;
    text-align:center;
  }
  td,th{border:1px solid #ccc}
</style>

</head>
<body>
  <div id="data">
  <!-- table>thead>tr>th*3 -->
    <table>
      <thead>
        <tr>
          <th>姓名</th>
          <th>薪资</th>
          <th>年龄</th>
        </tr>
      </thead>
      <tbody>

      </tbody>
```

```
    </table>
  </div>
  <script>
  var json=[
    {"ename":"Tom", "salary":11000, "age
":25},
    {"ename":"John", "salary":13000, "ag
e":28},
    {"ename":"Mary", "salary":12000, "ag
e":25}
  ];
  //1. 在 HTML 中手工添加固定不变的 table 和
表头行内容，并提前添加<tbody>元素

  //2. 创建一个文档片段对象
  var frag=document.createDocumentFragme
nt();

  //3. 遍历 json 数组中每个员工对象
  for(var emp of json){
    //4. 每遍历一个员工对象，就创建一个 tr
元素，追加到文档片段中
```

```
    var tr=document.createElement("tr");
    frag.appendChild(tr);
    //5. 遍历当前员工对象中每个属性
    for(var key in emp){
      //每遍历一个属性值，就创建 td，追加到
tr 中
      var td=document.createElement("td"
);
      tr.appendChild(td);
      //6. 设置 td 的内容为当前属性的属性值
      td.innerHTML=emp[key];
    }
  }

  //7. 将文档片段一次性添加到 table>tbody 下
  var tbody=
    document.querySelector("#data>table>
tbody");
  tbody.appendChild(frag);
  </script>
</body>
</html>
```

运行结果：

员工姓名	薪资	员工年龄
Tom	11000	25
John	13000	28
Mary	12000	25

5. 删除元素：父元素.removeChild(子元素)

三. HTML DOM 常用对象：对常用的个别复杂的元素操作提供的简化版访问方式

——了解——鸡肋

1. <img>：创建 img 时：

(1). 核心 DOM: var img=document.createElement("img")

(2). HTML DOM: var img=new Image();

(3). 强调：绝不是所有元素都能 new，最常用的元素中只有两个元素可以 new:

a. var img=new Image()

b. var opt=new Option(文本，值)

2. <table>：逐级管理：

(1). table 管着行分组:

a. table 可以添加行分组:

1). var thead=table.createTHead();

一句话做了两件事: 1. 既创建 thead 元素，2. 同时又将 thead 追加到 table 上

2). var tbody=table.createTBody();

3). var tfoot=table.createTFoot();

b. table 可以删除行分组

1). table.deleteTHead();

2). table.deleteTFoot();

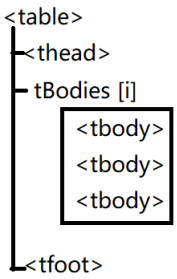
~~真的没有 table.deleteTBody()~~

c. table 可以获取行分组

1). table.tHead

2). table.tFoot

3). table.tBodies[i]



(2). 行分组管着行:

a. 行分组可添加一行:

1). var tr=行分组.insertRow(i)

2). 意为: 在当前行分组中第 i 行位置插入一个新行

3). 也是一句话办 2 件事: 1. 既创建， 2. 又插入

4). 说明: 如果 i 位置是行分组内的中间位置，i 位置已经有旧行了。那么，insertRow(i)，添加的新行会将原 i 位置的行向后挤，新行占据 i 位置。

5). 两个位置用的特别多:

开头: 想在开头插入一个新行: var tr=行分组.insertRow(0)

结尾: 想在结尾插入一个新行: var tr=行分组.insertRow() //默认就是末尾追加

b. 行分组可删除一行:

1). 行分组.deleteRow(i)

2). 意为: 删除当前行分组下的第 i 行

3). 强调: i, 应该是要删除的行在行分组内的相对下标位置

4). 问题: 如果用户随便点一行，要删除，又因为页面中行很多，不容易一眼看出要删除的是第几行，行分组.deleteRow(?)中下标应该写几

5). 解决: 每个行对象上都自动有一个.rowIndex 属性记录着该行在整个 table 中的下标位置!

6). 问题: 行分组.deleteRow(i)要求行在行分组内的相对下标。而.rowIndex 提供的是行在整个表中的下标。行在整个表中的下标和行在行分组内的相对下标，因为受表头行的影响，几乎都不相等! 如果用行

分组.deleteRow(tr.rowIndex)删除一行，则实际删除的是 tr 下方的某一行!

7). 解决：其实 table 元素对象上也有一个 deleteRow(i)函数，因为调用时.前的主语换成了 table，所以需要的参数 i，也变成了要删除的行在整个表中的下标位置。刚好和 tr.rowIndex 配对!

8). 结论：今后只要在行分组中删除行，都应该用:

table.deleteRow(tr.rowIndex)

c. 行分组可获取一行: var tr=行分组.rows[i]

(3). 行管着格:

a. 行可以添加格：一行末尾追加格：var td=tr.insertCell()

b. 行可以删除格: tr.deleteCell(i)

c. 行可以获取: tr.cells[i]

(4). 示例：使用 HTML DOM 简写实现动态加载表格

内容，以及删除行

1_createTable3_HTMLDOM.html

```
<!DOCTYPE HTML>
<html>
<head>
<title>动态创建表格</title>
<meta charset="utf-8" />
<style>
  table{width:600px; border-
collapse:collapse;
    text-align:center;
  }
  td,th{border:1px solid #ccc}
</style>

</head>
<body>
  <div id="data">
  <!-- table>thead>tr>th*3 -->
    <table>
      <thead>
        <tr>
```

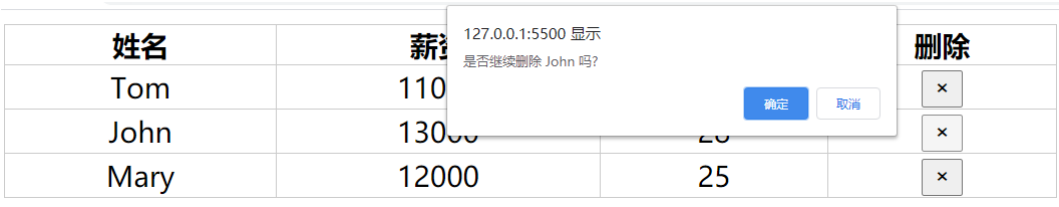
```
            <th>姓名</th>
            <th>薪资</th>
            <th>年龄</th>
            <th>删除</th>
          </tr>
        </thead>
      </table>
    </div>
    <script>
    var json=[
      {"ename":"Tom", "salary":11000, "age":25},
      {"ename":"John", "salary":13000, "age":28},
      {"ename":"Mary", "salary":12000, "age":25}
    ];
    //1. 在 HTML 中手工添加固定不变的 table 和表头行内容

    //查找 id 为 data 的 div 下的 table 元素
```

```
  var table=document.querySelector("#data>table")
  //2. 创建一个 tbody 元素
  var tbody=document.createElement("tbody");
  //3. 遍历 json 数组中每个员工对象
  for(var emp of json){
    //4. 每遍历一个员工对象，就创建一个 tr 元素，追加到 tbody 中
    //旧核心 DOM:
    // var tr=document.createElement("tr");
    // tbody.appendChild(tr);
    //HTML DOM:
    var tr=tbody.insertRow();
    //5. 遍历当前员工对象中每个属性
    for(var key in emp){
      //每遍历一个属性值，就创建 td，追加到 tr 中
      //旧核心 DOM:
      // var td=document.createElement("td");
```

```
        // tr.appendChild(td);
        //HTML DOM:
        var td=tr.insertCell();
        //6. 设置 td 的内容为当前属性的属性值
        td.innerHTML=emp[key];
    }//内层 for 循环之后
    //在添加完每行的格之后，再额外多添加一个格
    var td=tr.insertCell();
    //想 td 中添加一个按钮
    var btn=document.createElement("button");
    td.appendChild(btn);
    btn.innerHTML="×";
    btn.onclick=function(){
        // alert("疼!");
        // confirm("是否继续删除?");
        //查找当前按钮所在行
        //       btn    td          tr
        var tr=this.parentNode.parentNode;
        //获得当前行中的员工姓名:
        var ename=tr.cells[0].innerHTML;
```

```
        //希望先弹出确认框，用户点击确认后，才
        if(confirm(`是否继续删
除 ${ename} 吗?`)==true){
            //最终：删除当前按钮所在的行
            table.deleteRow(tr.rowIndex);
        }
        //否则如果用户点击取消，则我们也不做！
    }
  }//外层 for 循环里！

  //7. 当 tbody 元素下所有子内容填充完成之
后，再将 tbody 一次性添加到 dom 树
  table.appendChild(tbody);
  </script>
</body>
</html>
```

运行结果：

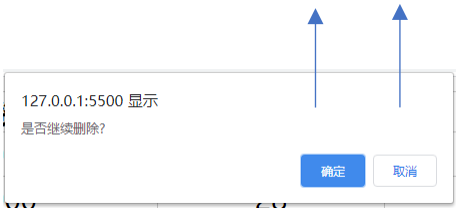


浏览器三大对话框:

1. 输入框: var 用户输入的内容=prompt("提示信息")
2. 警告框: alert("警告信息")

3. 确认框: var bool 是否同意继续=confirm("提示信息")

true false



3. <form>元素及其内部表单元素的简写:

(1). 获取<form>元素时:

a. document 已经将页面中所有<form>元素都集中存储在 document.forms 集合中。可通过下标，方式来获取

b. var form=document.forms[i]

(2). 获取<form>元素内的表单元素时

a. 标准: <form>元素内的所有表单元素(input, textarea, button, select)都已经被收集到了 form 对象的 elements 集合中。

var 表单元素=form.elements[i 或 id 或 name]

b. 简写: 如果这个表单元素带有 name 属性, 可直接:

var 表单元素=form.name 名

(3). 让表单元素自动获得焦点: 表单元素.focus()

(4). 示例: 使用 HTMLDOM 简化带样式的表单验证, 并让姓名文本框自动获得焦点

0_valiWithCss_HTMLDOM.html

```
<!doctype html>
<html>
 <head>
  <meta charset="UTF-8">
  <title>实现带样式的表单验证</title>
  <style>
    table{width:700px}
    td:first-child{width:60px}
    td:nth-child(2){width:200px}
    td:first-child+td{width:200px}
    td span{color:red}
    /*文本框旁边 div 隐藏的根源*/
    .vali_info{
      display:none;
```

```
    }
    .txt_focus{
      border-top:2px solid black;
      border-left:2px solid black;
    }

    .vali_success,.vali_fail{
      background-repeat:no-repeat;
      background-position:left center;
      display:block;
    }
    /*文本框验证通过时 div 的样式*/
    .vali_success{
        background-
image:url("../images/ok.png");
      padding-left:20px;
      width:0px;height:20px;
      overflow:hidden;
    }
    /*文本框验证失败时 div 的样式*/
    .vali_fail{
```

```
        background-
image:url("../images/err.png");
        border:1px solid red;
        background-color:#ddd;
        color:Red;
        padding-left:30px;
    }


  </style>
 </head>
 <body>
  <form id="form1">
    <h2>增加管理员</h2>
    <table>
      <tr>
        <td>姓名：</td>
        <td>
          <input name="username"/>
          <span>*</span>
        </td>
        <td>
```

```
            <div class="vali_info">
              10 个字符以内的字母、数字或下划线的组合
            </div>
          </td>
        </tr>
        <tr>
          <td>密码：</td>
          <td>
            <input type="password" name="pwd"/>
            <span>*</span>
          </td>
          <td>
            <div class="vali_info">6 位数字</div>
          </td>
        </tr>
        <tr>
          <td></td>
          <td colspan="2">
```

```
            <input type="submit" value="保存"/>
            <input type="reset" value="重填"/>
          </td>
        </tr>
        </table>
    </form>
<script>
//只实现姓名文本框的验证:
//2 个需求:
//1. 文本框获得焦点时，显示右边的提示
//2. 文本框失去焦点时，验证文本款中的内容
    //如果验证通过，就修改右边的提示为正确的样式
    //如果验证不通过，就修改右边的提示为错误的样式

//实现第一个需求: 文本框获得焦点时，显示右边的提示
//DOM 4 步:
//1. 查找触发事件的元素
```

```
//本例中：name 属性=username 的文本框获得焦点触发变化
//先找到 form 元素对象：
var form=document.forms[0];
//元素缩写  业务单词
//    ↓      ↓
var txtName=form.username;
            //form.elements["username"];
            //form.elements[0];

//希望一加载网页，就让姓名文本框默认获得焦点！
txtName.focus();

//2. 绑定事件处理函数
//本例中：文本框获得焦点时触发变化
txtName.onfocus=function(){
  //3. 查找要修改的元素
  //本例中：当前文本框获得焦点，却要修改当前文本框的父元素 td 的下一个兄弟元素 td 的第一个孩子元素 div 的样式
```

```
  var div=this.parentNode.nextElementSib
ling.children[0];
  //4. 修改元素
  //本例中: div 因为带有 vali_info class 才
默认隐藏，所以，我们需要将 div 的 class 暂时清
除，让 div 显示出来
  div.className="";
}

//实现第二个需求: 当文本框失去焦点时，验证文
本框的内容，修改旁边 div 的样式
//DOM 4 步
//1. 查找触发事件的元素
//本例中: 还是姓名文本框失去焦点——上边已经找
过了 txtName
//2. 绑定事件处理函数
//本例中: 文本框失去焦点时，触发验证和修改
txtName.onblur=function(){
  //3. 查找要修改的元素
  //本例中: 当前文本框失去焦点，却要修改当前
文本框的父元素 td 的下一个兄弟元素 td 的第一个
孩子元素 div 的样式
```

```
  var div=this.parentNode.nextElementSib
ling.children[0];
  //4. 修改元素
  //定义正则表达式
  var reg=/^\w{1,10}$/;
  //如果验证文本框的内容通过
  if(reg.test(this.value)==true){
    //就修改 div 为正确的样式
    div.className="vali_success";
  }else{//否则如果验证文本框的内容不通过
    //就修改 div 为错误的样式
    div.className="vali_fail";
  }
}
</script>
 </body>
</html>
```

运行结果：

增加管理员

姓名： * 10个字符以内的字母、数字或下划线的组合

密码： *

保存 重填

#DOMday04:

正课:

1. BOM
2. ***事件绑定***

一. BOM: Browser Object Model

1. 什么是: 一套专门操作浏览器窗口和软件信息的对象、属性和方法的集合。
2. vs DOM: DOM 专门操作网页内容; BOM 专门操作浏览器窗口和软件
3. 何时: 只要想操作浏览器窗口或想获得浏览器窗口和软件的信息时, 都用 BOM
4. 问题: 没有标准的! 每种浏览器可能都不一样! ——极大的兼容性问题! ——用的越来越少了!
5. 包括:

window, location, history, navigator, event, document, screen

6. window 对象: 3 个角色:

(1). 代替 ES 标准中的 global, 充当全局作用域对象

(2). 集中包含所有原生的对象和函数: ES 标准的内

置对象+DOM+BOM

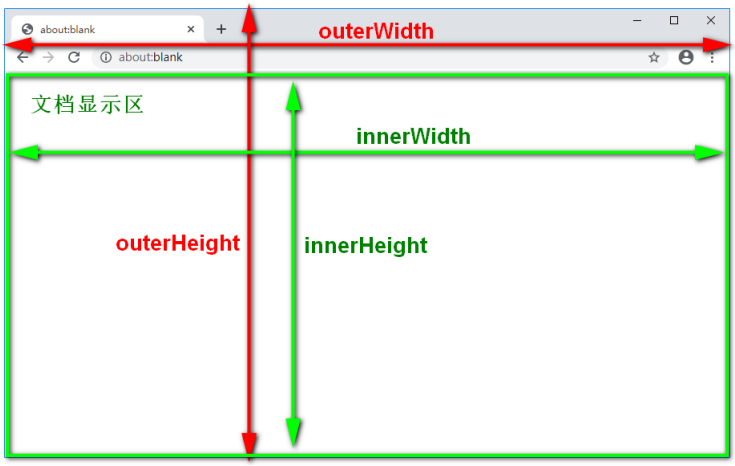
(3). 代表当前浏览器窗口:

a. window 可获得当前浏览器窗口的大小

1). 完整窗口大小: window.outerWidth window.outerHeight

2). 仅内部文档显示区的大小: window.innerWidth window.innerHeight

文档显示区: 浏览器窗口中, 专门用于显示网页的区域



b. 用 window 还可打开和关闭窗口

1). 打开一个新窗口: window.open()

2). 关闭当前窗口: window.close()

7. 打开和关闭新链接: 4 种情况:

(1). 在当前窗口打开, 可后退

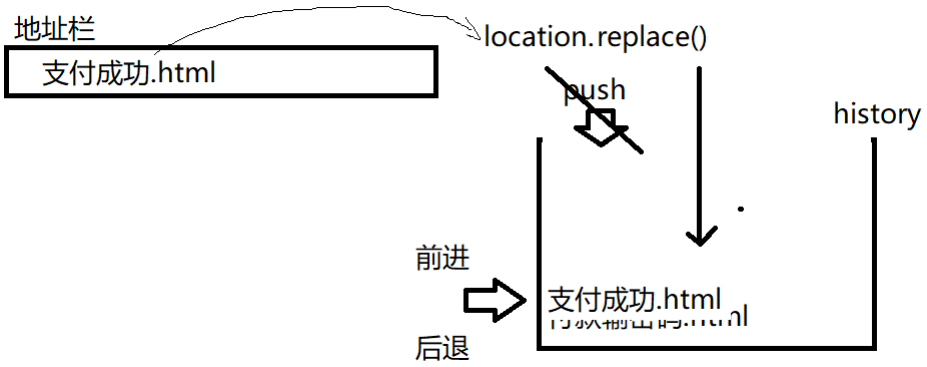
a. html: <a href="url" target="_self">

b. js: window.open("url", "_self");

(2). 在当前窗口打开，禁止后退

a. js: location.replace("新 url")

b. 原理: 用新 url 代替 history 中旧 url，因为没有旧 url 了，所以无法后退!



(3). 在新窗口打开，可同时打开多个

a. html: <a href="url" target="_blank">

b. js: window.open("url", "_blank");

(4). 在新窗口打开，只能打开一个

a. html: <a href="url" target="自定义窗口名">

b. js: window.open("url", "自定义窗口名")

(5). 原理:

a. 每个打开的浏览器窗口在内存中都有一个 name 名来唯一表示这个窗口对象——一般用户看不见!

b. 浏览器规定，相同 name 名的窗口只能打开一个。后打开的同 name 名的窗口会覆盖先打开的同 name

名的窗口。

c. 在我们用a元素或window.open()打开新窗口时，target 属性值和 open()的第二个参数其实都是在为新窗口指定一个自定义的窗口名

d. 强调: 今后自己起变量名或属性名时，禁止使用"name"作为变量名或属性名!

e. 预定义窗口名:

1). _self: 表示将当前旧窗口自己的 name 名，设置给新打开窗口的name名——新窗口覆盖/替换同名旧窗口——在当前窗口打开

2). _blank: 表示不给新窗口指定 name 名，但是浏览器不会让 name 名空着，而会在底层自动为新窗口随机生成 name 名！因为随机生成，所以每个新窗口的 name 名各不相同!——同时打开多个而不冲突!

(6). 示例: 打开新链接 4 种方式:

day03/4_open.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8"/>
    <title>打开新链接方式总结</title>
    <script>
```

```
        /*打开新链接方式总结：
        1. 在当前窗口打开，可后退
        2. 在当前窗口打开，禁止后退
        3. 在新窗口打开，可同时打开多个
        4. 在新窗口打开，只能打开一个
        */
    </script>
  </head>
  <body>
    <h3>1. 在当前窗口打开，可后退</h3>
    <a href="http://tmooc.cn" target="_self">go to tmooc</a><br/>
    <button id="btn1">go to tmooc</button>
    <h3>2. 在当前窗口打开，禁止后退</h3>
    <button id="btn2">go to tmooc</button>
    <h3>3. 在新窗口打开，可同时打开多个</h3>
    <a href="http://tmooc.cn" target="_blank">go to tmooc</a><br/>
```

```
    <button id="btn3">go to tmooc</button>
    <h3>4. 在新窗口打开，只能打开一个</h3>
    <a href="http://tmooc.cn" target="abc">go to tmooc</a><br/>
    <button id="btn4">go to tmooc</button>
    <script>
      var btn1=document.getElementById("btn1");
      btn1.onclick=function(){
        //window.open("http://tmooc.cn","_self");
        //location.assign("http://tmooc.cn");
        location.href="http://tmooc.cn";
      }
      var btn2=document.getElementById("btn2");
      btn2.onclick=function(){
        location.replace("http://tmooc.cn");
```

```
        }
        var btn3=document.getElementById("
btn3");
        btn3.onclick=function(){
          window.open("http://tmooc.cn","_
blank");
        }
        var btn4=document.getElementById("
btn4");
        btn4.onclick=function(){
          window.open("http://tmooc.cn","a
bc");
        }
    </script>
  </body>
</html>
```

运行结果:

1. 在当前窗口打开，可后退

go to tmooc
go to tmooc

2. 在当前窗口打开，可后退

go to tmooc

3. 在新窗口打开，可同时打开多个

go to tmooc
go to tmooc

4. 在新窗口打开，可同时打开多个

go to tmooc
go to tmooc

8. history:

(1). 什么是: 每个窗口中自带的记录当前窗口打开后成功访问过的所有 url 的历史记录数组。

(2). 原理:

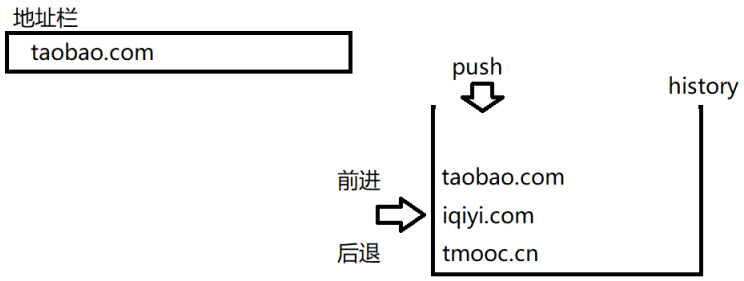
a. 首次打开一个 url 时，将 url 压入(push)history 数组中保存

b. 之后每打开一个新 url 时，都会再次将新 url 压入 history 中上层保存

(3). 前进后退的原理:

a. 能否前进取决于当前正在看的页面的 url 在 history 中前边有没有比它新的 url

b. 能否后退取决于当前正在看的页面的 url 在 history 中后边有没有比它旧的 url



(5). history 只能前进，后退，刷新:

a. 前进: history.go(1) 前进一步

b. 后退: history.go(-1) 后退一步

history.go(-2)

c. 刷新: history.go(0)

(6). 示例: 演示前进后退

a. 如何:

1). 先从 9-1_history.html 运行起来

—— 此时 history 中只有一个 url: 9-1，既不能前进又不能后退

2). 再点击页面中的 2、3 两个链接，让 history 中增加两个 url

—— 此时 history 中有了三个 url: 9-1 9-2 9-3 就可以验证前进后退

3). 再测试前进后退!

b. 代码:

day03/9-1_history.html

```
<!DOCTYPE HTML>
```

```
<html>
<head>
<title>使用 history 对象</title>
<meta charset="utf-8" />
</head>
<body>
  <h2>9-1.html</h2>
  <div>
    <a href="9-
1_history.html">1</a>  
    <a href="9-
2_history.html">2</a>  
    <a href="9-
3_history.html">3</a>  
  </div><br/>
  <div>
    <!--a 元素的 href 属性以 javascript:开
头，就不再执行跳转操作，而是执行一条 js 语句-
->
    <a href="javascript: history.go(1)">
前进一次</a>  
```

```
    <a href="javascript: history.go(2)">
前进二次</a>  
  </div>
</body>
</html>
```

day03/9-2_history.html

```
<!DOCTYPE HTML>
<html>
<head>
<title>使用 history 对象</title>
<meta charset="utf-8" />
</head>
<body>
  <h2>9-2.html</h2>
  <div>
    <a href="9-
1_history.html">1</a>  
    <a href="9-
2_history.html">2</a>  
    <a href="9-
3_history.html">3</a>  
```

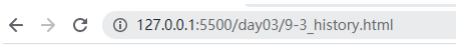
```
  </div><br/>
  <div>
    <a href="javascript: history.go(-
1)">后退一次</a>  
    <a href="javascript: history.go(1)">
前进一次</a>  
  </div>
</body>
</html>
```

day03/9-3_history.html

```
<!DOCTYPE HTML>
<html>
<head>
<title>使用 history 对象</title>
<meta charset="utf-8" />
</head>
<body>
  <h2>9-3.html</h2>
  <div>
    <a href="9-
1_history.html">1</a>  
```

```
    <a href="9-2_history.html">2</a>  
    <a href="9-3_history.html">3</a>  
  </div><br/>
  <div>
    <a href="javascript: history.go(-2)">后退二次</a>  
    <a href="javascript: history.go(-1)">后退一次</a>  
  </div>
</body>
</html>
```

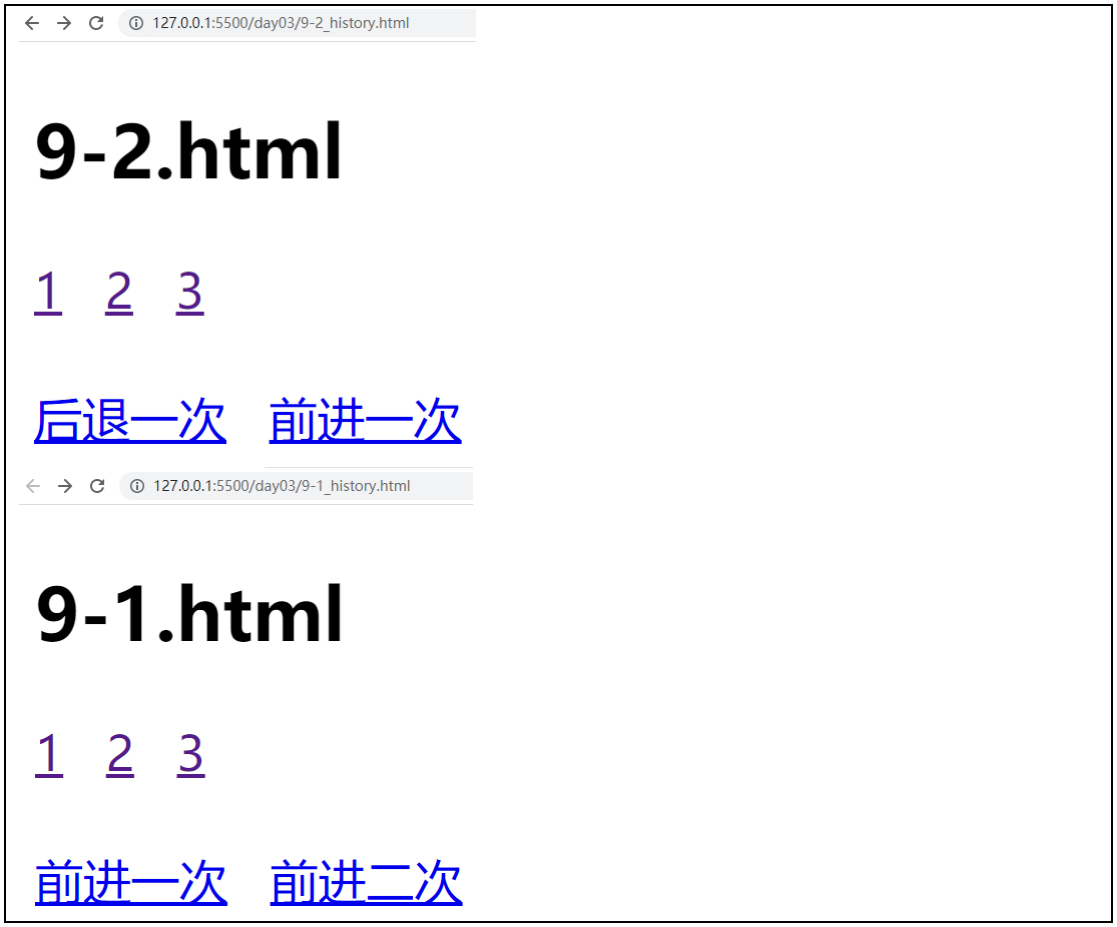
运行结果：



9-3.html

1 2 3

后退二次 后退一次



9. location:

(1). 什么是: 专门保存当前窗口正在打开的 url 信息的对象

(2). 何时: 要获得 url 中相关信息时, 或者希望执行跳转操作时

(3). 属性: 分段获得 url 中各个部分:

a. location.href 完整 url

b. location.protocol 协议

c. location.host 主机名+端口号

d. location.hostname 主机名

e. location.port　　端口号

f. location.pathname　相对路径

g. location.search　　?及其之后的查询字符串参数列表

h. location.hash　　#锚点地址

(4). 示例: 输出 location 的各个部分:

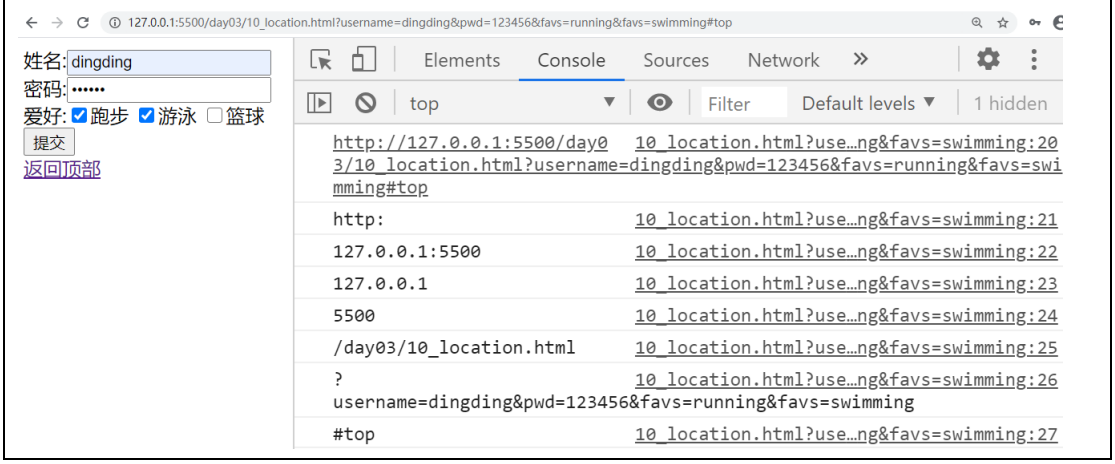
day03/10_location.html

```
<!DOCTYPE HTML>
<html>
<head>
<title>事件处理</title>
<meta charset="utf-8" />

</head>
<body>
  <form>
    姓名:<input name="username"/><br>
    密
码:<input type="password" name="pwd"/><br
>
```

```
    爱
好:<input type="checkbox" name="favs" val
ue="running"/>跑步
          <input type="checkbox" name="f
avs" value="swimming"/>游泳
          <input type="checkbox" name="f
avs" value="basketball"/>篮球
          <br>
    <input type="submit"/>
  </form>
  <a href="#top">返回顶部</a>
  <script>
    console.log(location.href);
    console.log(location.protocol);
    console.log(location.host);
    console.log(location.hostname);
    console.log(location.port);
    console.log(location.pathname);
    console.log(location.search);
    console.log(location.hash);
  </script>
</body>
```

```
</html>
```

运行结果:



(5). 方法:

a. 也可以实现在当前窗口打开, 可后退:

location.assign("新 url") 或 location.href="新 url"

b. 只有 location 可以实现在当前窗口打开, 禁止后退:

location.replace("新 url")

c. 也可以实现刷新: location.reload();

10. navigator:

(1). 什么是: 保存浏览器配置信息的对象

(2). 何时: 今后只要想获得浏览器中的配置信息, 都用 navigator 对象

(3). 常用:

a. 查看浏览器的名称和版本号: navigator.userAgent

b. 查看浏览器中安装的插件列表: navigator.plugins

二. 事件:

1. 事件绑定: 3 种:

(1). 在 HTML 中绑定:

a. html 中: <元素 on 事件名="事件处理函数()">

b. js 中: function 事件处理函数(){ ... }

c. 问题: 因为事件绑定随元素分散在网页的各个角落, 极其不便于维护

(2). 在 js 中用赋值方式绑定:

a. 元素对象.on 事件名=function(){ ... }

b. 好处: 所有事件绑定都集中在 js 中, 非常便于维护

c. 问题: 一个元素的一个事件属性上, 只能保留一个事件处理函数。无法同时保留多个事件处理函数。重复给一个元素的一个属性上赋值多个事件处理函数, 结果只有最后一个事件处理函数才能留下来!

(3). 在 js 中用添加事件监听对象的方式绑定:

a. 元素对象.addEventListener("事件名", 事件处理函数)

添加 事件 监听对象

b. 强调: 如果使用 addEventListener, 事件名之前不用加"on"。因为原本 DOM 标准中规定的事件名都是没有 on 的! 比如: click, focus, change, ...

c. 原理:

1). 在浏览器内存中有一个巨大的事件队列

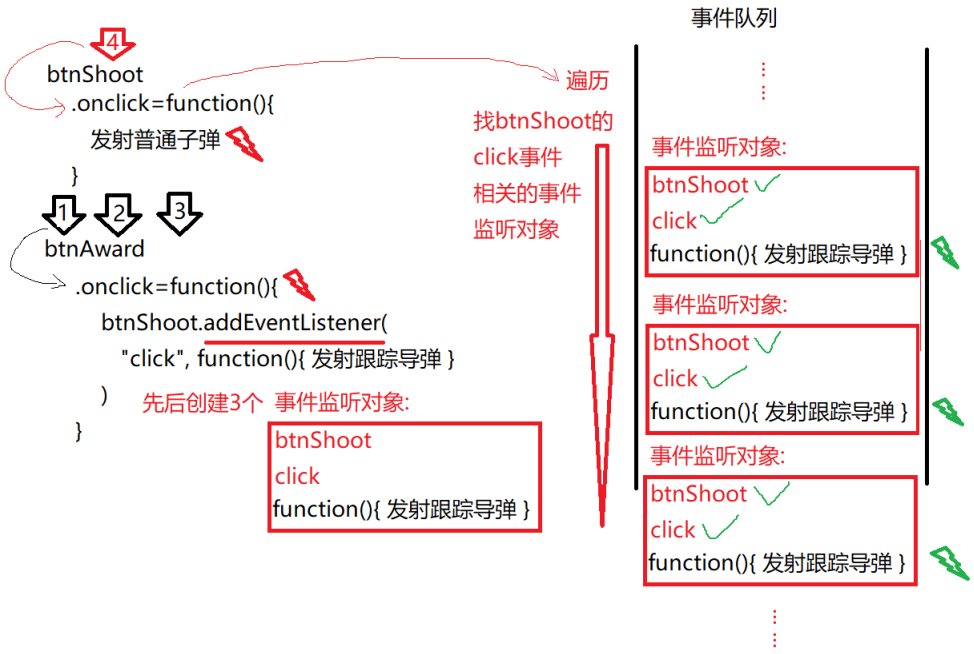
2). addEventListener() 会做 2 件事:

i. 创建一个事件监听对象, 其中保存三个内容:

当前元素+当前事件名+事件处理函数对象

ii. 将事件监听对象添加到浏览器的事件队列中保存起来

3). 当某个元素上发生某个事件时, 浏览器会通过遍历事件队列的方式查找到符合条件的事件监听对象, 自动执行其中的事件处理函数。找到几个事件监听对象, 就执行几个事件处理函数.



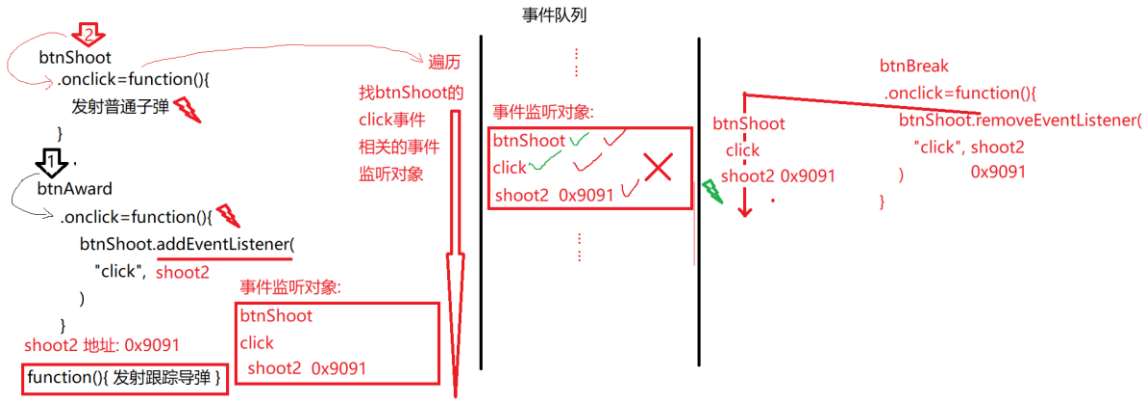
d. 移除事件监听:

1). 元素.removeEventListener("事件名", 原事件处理函数对象)

2). 坑: 移除事件监听对象时, 仅仅将事件处理函数写的和添加事件监听时一模一样, 是无法移除原事件处理函数:

3). 原因: function 是 new Function , 是创建新函数对象的意思。绑定时 function(){}创建了一个函数, 移除时 function(){}又会创建一个新函数, 两个函数地址绝不相同! 所以, 在查找要删除的事件监听对象时, 不可能用新函数对象地址, 匹配旧的函数对象的。——所以, 只要绑定时使用匿名函数绑定, 移除时也使用匿名函数移除, 则都移除不成功!

4). 解决: 今后如果一个函数有可能被移除, 则绑定时就不能用匿名函数, 应该用有名称的函数绑定。移除时, 才可以用函数名变量获得原处理函数的地址, 移除原处理函数。



5). 问题: 事件监听队列规定完全相同的事件监听对象(元素相同, 事件名相同, 事件处理函数对象的地址也相同), 只能添加一个, 不能重复添加! ——结果, 如果我们使用有名称函数绑定事件, 无论点几次添加事件监听, 永远只有一个事件监听!

e. 示例: 实现为发射按钮添加多种子弹并能移除子弹

1_addEventListener.html

```
<!DOCTYPE html>
<html>

<head>
    <meta charset="utf-8" />
    <title>...</title>
    <script>
    </script>
</head>

<body>
    <button id="btnShoot">shoot</button><
br>
    <button id="btnAward">获得跟踪导弹
</button><br>
    <button id="btnBreak">失去跟踪导弹
</button><br>
    <script>
        var btnShoot=document.getElementBy
Id("btnShoot");
```

```
        var btnAward=document.getElementBy
Id("btnAward");
        var btnBreak=document.getElementBy
Id("btnBreak");
        //需求:
        //开局: 点 shoot 按钮，只能发射一种普
通子弹
        btnShoot.onclick=function(){
            console.log(`发射普通子
弹......`);
        }
        //如果一个事件处理函数有可能被移除，则
绑定时必须用有名称的函数
        function shoot2(){
            alert(`发射跟踪导弹=>=>=>`);
        }
        //点获得跟踪导弹按钮: 给 shoot 按钮再
添加一种跟踪导弹
        btnAward.onclick=function(){
            btnShoot.addEventListener("clic
k",shoot2)
        }
```

```
            //结果：再点 shoot 按钮时，可以发射两种子弹！
            //点失去跟踪导弹按钮：从 shoot 按钮上移除跟踪导弹
            btnBreak.onclick=function(){
                btnShoot.removeEventListener("click",shoot2)
            }
            //结果：再点 shoot 按钮时，恢复成只发射一种子弹

    </script>
</body>

</html>
```

运行结果:

f. 扩展示例：添加多个跟踪导弹，并能移除多个跟踪导弹

1_addEventListener2.html

```
<!DOCTYPE html>
<html>

<head>
    <meta charset="utf-8" />
    <title>...</title>
    <script>
    </script>
</head>

<body>
    <button id="btnShoot">shoot</button><
br>
    <button id="btnAward">获得跟踪导弹
</button><br>
    <button id="btnBreak">失去跟踪导弹
</button><br>
    <script>
        var btnShoot=document.getElementBy
Id("btnShoot");
        var btnAward=document.getElementBy
Id("btnAward");
```

```
        var btnBreak=document.getElementBy
Id("btnBreak");
        //需求:
        //开局: 点 shoot 按钮，只能发射一种普
通子弹
        btnShoot.onclick=function(){
            console.log(`发射普通子
弹......`);
        }
        //定义数组保存所有添加的跟踪导弹函
数:
        var funs=[];
        //点获得跟踪导弹按钮: 给 shoot 按钮再
添加一种跟踪导弹
        btnAward.onclick=function(){
            //每点击一次添加就创建一个新函数，
加入数组中
            var shoot=function(){
                alert(`发射跟踪导弹=>=>=>`);
            }
            funs.push(shoot)
```

```
        btnShoot.addEventListener("clic
k",shoot)
    }
        //结果：再点 shoot 按钮时，可以发射
两种子弹！
    //点失去跟踪导弹按钮：从 shoot 按钮上
移除跟踪导弹
    btnBreak.onclick=function(){
        for(var fun of funs){
            btnShoot.removeEventListener
(
                "click", fun
            )
        }
    }
        //结果：再点 shoot 按钮时，恢复成只
发射一种子弹

  </script>
</body>

</html>
```

运行结果: 同上

2. 事件模型:

(1). 什么是: 从触发事件到所有事件处理函数执行完所经历的过程

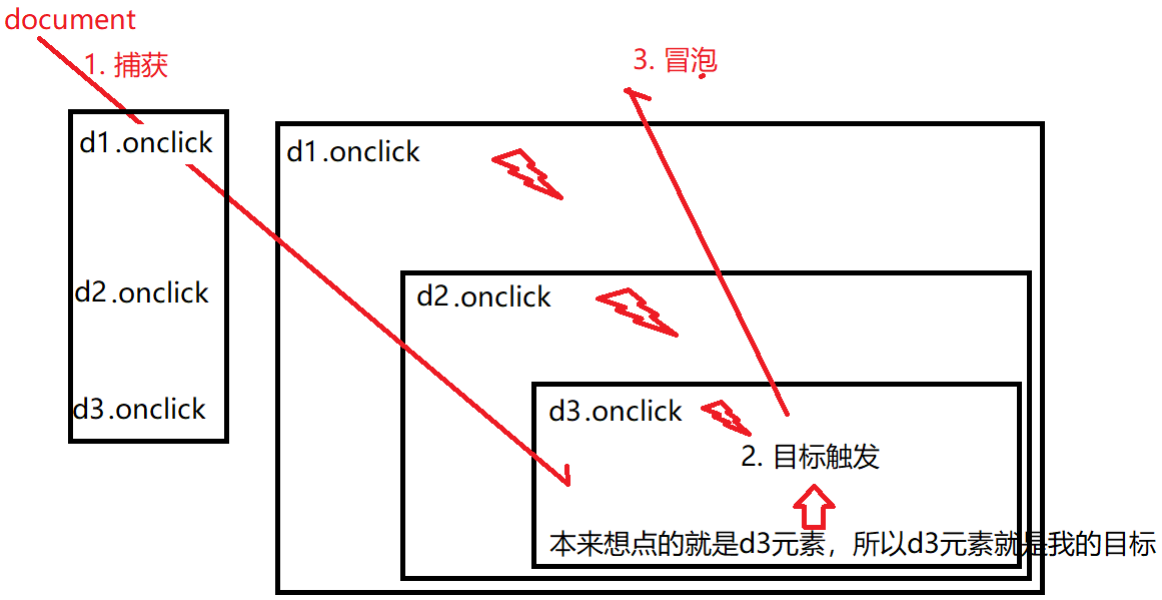
(2). DOM 标准事件模型包括: 3 个阶段:

a. 捕获: 从 document 开始, 从外向内依次记录实际触发事件的元素的所有父级元素的事件处理函数——只记录, 不执行

b. 目标触发: 首先触发最早触发事件的那个元素上的事件处理函数

最早触发事件的元素——称为目标元素(target)

c. 冒泡: 然后依次由内向外触发捕获阶段记录的各级父元素上的事件处理函数



3. 事件对象:

(1). 事件对象: 事件发生时，浏览器自动创建的(不用我们自己创建)保存事件信息的对象

(2). 何时: 2 种:

a. 想获得事件相关的信息时，比如鼠标坐标

b. 想改变事件默认的行为时

(3). 如何获得事件对象: 事件对象总是在事件发生时，自动作为事件处理函数的第一个参数自动传入。所以，我们只要在事件处理函数上，提前定义一个形参 e，等着接就行了！——信任

事件发生时自动创建 event 对象

↓

元素.onclick=function(e){ ... }

(4). 能做什么:

a. 阻止冒泡:

1). e.stopPropagation()

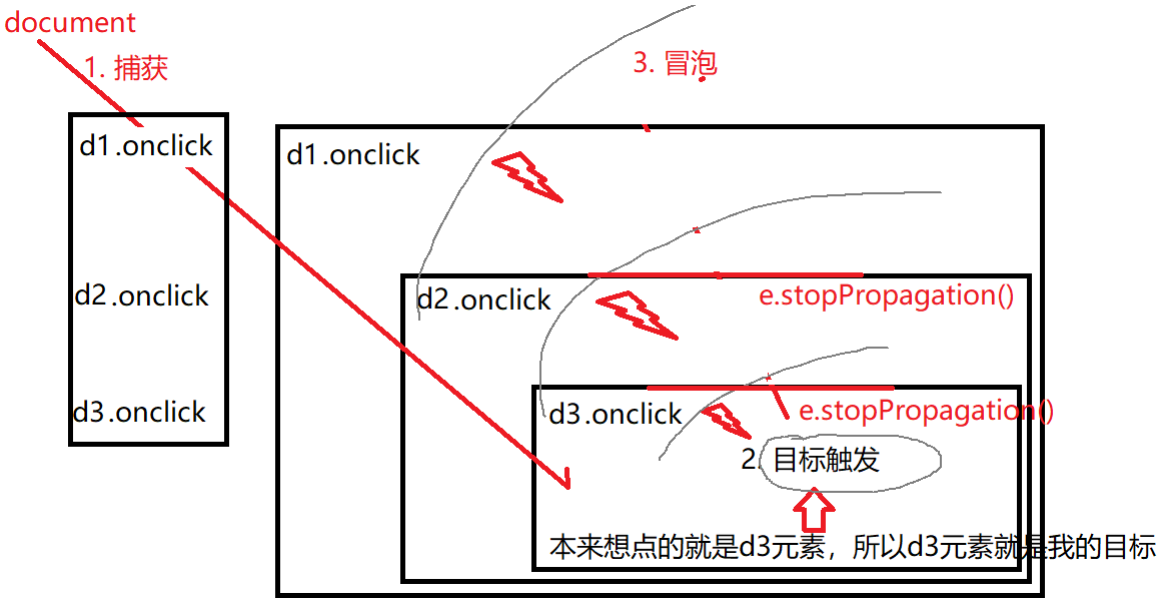
停止 漫延

2). 强调:

i. 最外层的元素上不用阻止冒泡

ii. 在一个事件处理函数中，e.stopPropagation()写前写后无所谓，因为 e.stopPropagation()阻止的是父元素的事件继续触发，而不是阻止自己!

3). 示例: 使用 e.stopPropagation()停止漫延



2_bubble.html

```
<!DOCTYPE HTML>
<html>
  <head>
```

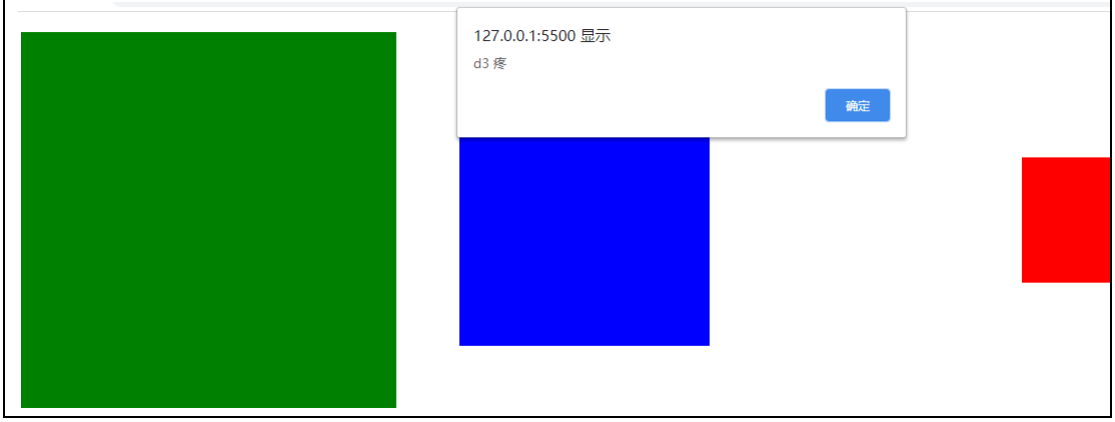
```
<title>事件处理</title>
<meta charset="utf-8"/>
<style>
  #d1 #d2 #d3{cursor:pointer}
  #d1 {
    background-color: green;
    position: relative;
    width: 150px;
    height: 150px;
    text-align: center;
    cursor: pointer;
  }

  #d2 {
    background-color: blue;
    position: absolute;
    top: 25px;
    left: 175px;
    width: 100px;
    height: 100px;
  }
```

```
      #d3 {
        background-color: red;
        position: absolute;
        top: 25px;
        left: 225px;
        width: 50px;
        height: 50px;
        line-height: 50px;
      }

    </style>

  </head>
  <body>
    <div id="d1">
      <div id="d2">
        <div id="d3">
        </div>
      </div>
    </div>
    <script>
```

```
        //需求：为每个 div 都绑定单击事件，希望点哪个 div，哪个 div 就喊疼！
        var d1=document.getElementById("d1");
        var d2=document.getElementById("d2");
        var d3=document.getElementById("d3");
        d1.onclick=function(){
          //因为 d1 外边没有东西了，所以没必要阻止冒泡了！
          alert("d1 疼")
        }
        d2.onclick=function(e){
          //e.stopPropagation();
          alert("d2 疼");
          e.stopPropagation();
        }
        d3.onclick=function(e){
          e.stopPropagation();
          alert("d3 疼")
        }
```

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

运行结果:



b. 利用冒泡/事件委托:

1). 优化: 尽量减少事件监听对象的个数

2). 原因: 因为浏览器触发事件时, 是用遍历的方式查找事件监听对象的。所以, 事件监听对象多, 则查找速度慢。事件监听对象少, 查找速度就快!

3). 何时: 多个平级的子元素都需要绑定相同的事件时, 都要利用冒泡优化

4). 如何: 3 步:

i. 第一步: 事件只在父元素上集中绑定一个!

所有子元素触发事件时, 都通过冒泡自动触发父元素的事件处理函数, 共享使用——所以这个技巧也称为"事件委托", 意为, 所以子元素将自己想做的事儿托付父元素集中代为保管!

ii. 问题：当事件冒泡到父元素上才执行时，this 也跟着指向了实际触发事件处理函数的父元素，不再指向最初点击的子元素——在事件委托中，this 用不了了！

iii. 第二步：事件委托中所有 this 都要用 e.target 代替

e.target：是事件对象中专门保存最初触发事件的元素的特殊属性

优点：不随冒泡而改变！

iv. 问题：事件绑定在父元素上，万一点父元素也会触发事件——我们不希望看到的

v. 第三步：如果使用事件委托，则事件处理函数中必须先判断触发事件的目标元素是不是想要的元素。只有判断通过的使我们想要的元素，才能继续执行事件处理函数中后续代码！

5). 示例：使用事件委托实现计算器效果：

3_calc.html

```
<!DOCTYPE HTML>
<html>
  <head>
    <title>取消与利用冒泡</title>
    <meta charset="utf-8"/>
```

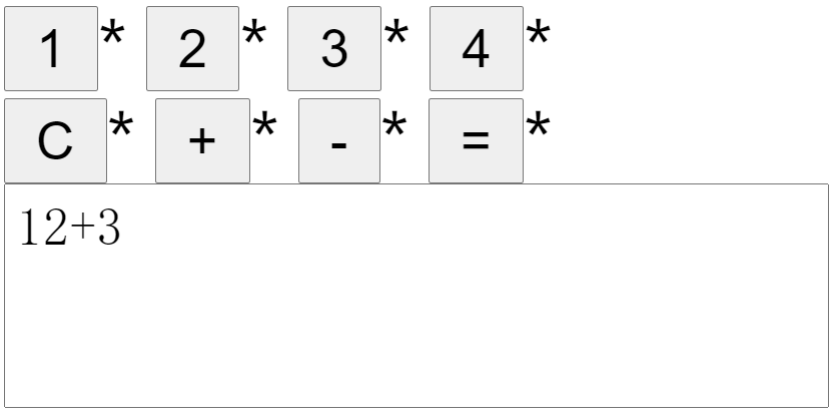
```
  </head>
  <body>
    <div id="keys">
      <button>1</button><span>*</span>
      <button>2</button><span>*</span>
      <button>3</button><span>*</span>
      <button>4</button><span>*</span><br>
      <button>C</button><span>*</span>
      <button>+</button><span>*</span>
      <button>-</button><span>*</span>
      <button>=</button><span>*</span>
    </div>
    <textarea id="sc" style="resize:none;width:200px; height:50px;" readonly></textarea>
    <script>
      //1. 只给父元素 div 绑定单击事件
      var div=document.getElementById("keys");
      div.onclick=function(e){
```

```
        //需求 1：无论点哪个子元素按钮，都喊一样的疼！
        //alert("疼!");
        //需求 2：点哪个按钮，就让当前按钮的内容变为❀
        //错误：this->父元素
        //this.innerHTML="❀";
        //正确：e.target
        //需求 3：点父元素或其他无关的子元素，不允许触发事件
        //判断：只希望 button 元素触发事件
        //如果当前目标元素是 button 元素
        //                不能改名
        if(e.target.nodeName=="BUTTON"){
          //才修改内容为❀
          //e.target.innerHTML="❀"
          //3. 查找要修改的元素
          //本例中：每次点击按钮，都要修改下方的文本框
          var sc=document.getElementById("sc");
          //4. 修改元素
```

```
            //先判断点击的按钮的内容
            switch(e.target.innerHTML){
              //如果是 C，就清空显示屏文本框
              case "C":
                sc.value="";
                break;
              //如果是=，就将文本框内容，交给 eval 计算结果，再将结果放回显示屏中
              case "=":
                //错误处理：复习第一阶段
                try{//尝试执行
                  //eval：可计算字符串类型的 js 表达式的值.——复习第一阶段
                  sc.value=eval(sc.value);
                }catch(err){//如果出错
                  sc.value=err;//就把错误信息显示在文本框中
                }
                break;
              //点击其余所有数字按钮和+ -号按钮，都只将按钮内容追加到文本框中算式结尾即可，不做计算
```

```
                default:
                    sc.value+=e.target.innerHTML;
                }
            }
            //nodeName 属性专门保存一个元素全大写的标签名
        }
    </script>
</body>
</html>
```

运行结果:



c. 阻止默认行为

1). 问题: 个别元素身上带有一些默认的行为, 而这些默认行为是我们不想要的!

2). 比如: <a href="#xxx">点击这个 a, 总是去自

动修改地址栏里的 url——和今后框架中的路由发生冲突，所以，我们不希望 a 元素擅自修改地址栏中的 url

3). 如何: e.preventDefault()

阻止 默认

4). 针对 a，还有另外一种方法:

将来只要只想使用一个 a 当按钮用时，href="javascript:;"即可

javascript: 让 a 不要跳转，转而执行一条 js 语句

; 是空语句的意思，什么也不执行

5). 示例: 使用两种方法阻止 a 的默认行为:

4_preventDefault.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>
  <a id="a1" href="#">click me</a><br/>
```

```
  <a id="a2" href="javascript:;">click m
e</a>
  <script>
    //只想拿 a 当做按钮:
    var a1=document.getElementById("a1")
;
    a1.onclick=function(e){
      e.preventDefault();
      alert("疼!");
    }
    var a2=document.getElementById("a2")
;
    a2.onclick=function(){
      alert("疼!");
    }
  </script>
</body>
</html>
```

运行结果:

d. 获取鼠标位置: 当事件发生时，我们可以获得三组鼠标位置

1). 相对于屏幕左上角的 x，y 坐标: e.screenX, e.screenY

屏 幕

屏幕

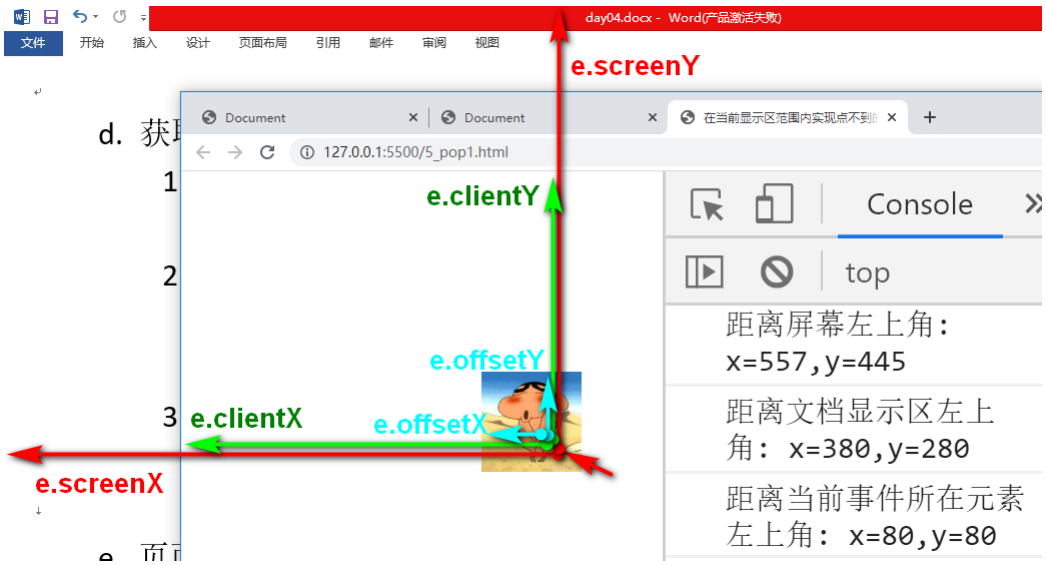
2). 相对于文档显示区左上角的 x，y 用坐标: e.clientX, e.clientY

客户端 客户端

浏览器 浏览器

3). 相对于事件所在元素左上角的 x，y 坐标: e.offsetX e.offsetY

偏移 偏移



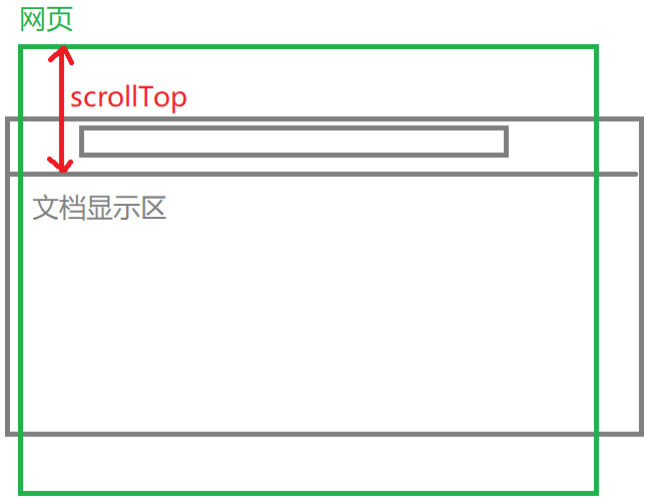
e. 页面滚动事件:

1). 在窗口发生滚动时触发一个事件

window.onscroll=function(){

2). 可获得当前窗口滚动过的距离

var scrollTop=document.documentElement.scrollTop||document.body.scrollTop



3). 根据滚动过的距离执行不同的操作

}

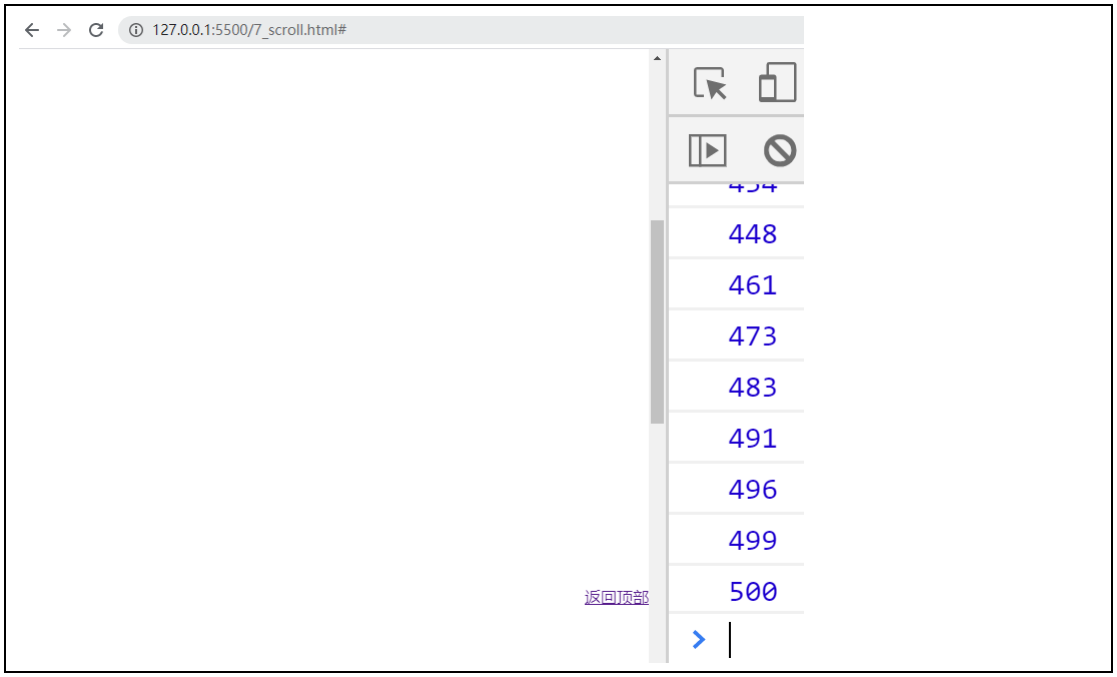
4). 示例：页面滚动到 500 位置时，显示返回顶部

7_scrollTop.html

```
<!doctype html>
<html>
 <head>
  <meta charset="UTF-8">
  <title>根据页面滚动位置显示浮动框</title>
  <style>
    body{height:2000px;}
    #toTop{
      position:fixed;
      bottom:100px;
      right:0;
      display:none;
    }
  </style>

 </head>
 <body>
  <div id="toTop">
```

```
    <a href="#">返回顶部</a>
  </div>
  <script>
    window.onscroll=function(){
      var scrollTop=document.documentEle
ment.scrollTop||document.body.scrollTop;
      console.log(scrollTop);
      var toTop=document.getElementById(
"toTop");
      //希望当滚动距离>=500 时，让 toTop 显
示
      if(scrollTop>=500){
        toTop.style.display="block";
      }else{//否则如果滚动距离<500 时，让
toTop 隐藏
        toTop.style.display="none";
      }
    }
  </script>
 </body>
</html>
```

运行结果：



总结:

Day01:

总结: 查找方法的返回值规律: 3 种

1. 如果原方法返回下标位置 i, 则如果找不到返回-1

2. 如果原方法返回一个数组或对象, 则如果找不到返回 null

3. 如果原方法返回类数组对象, 则如果找不到返回空类数组对象: { length:0 }

总结: DOM 4 步:

1. 先查找可能触发事件的元素
2. 再为元素绑定事件处理函数
    3. 当事件发生时，查找要修改的元素
    4. 修改元素

总结: DOM 5 件事: 增删改查+事件绑定:

1. 查找元素:

(1). 不需要查找就可直接获得的节点对象: 4 种:

document

document.documentElement —— <html>

document.head —— <head>

document.body —— <body>

(2). 如果已经获得一个元素对象，找周围附近的元素对象时，就用按节点间关系查找: 2 大类关系，6 个属性

父子关系: 4 种

元素的父元素: 元素.parentNode 或元素.parentElement

元素下的所有直接子元素: 元素.children

元素下的第一个直接子元素: 元素.firstElementChild

元素下的最后一个直接子元素: 元

素.lastElementChild

兄弟关系: 2 种

元素的前一个兄弟: 元素.previousElementSibling

元素的后一个兄弟: 元素.nextElementSibling

(3). 如果用一个特征就能找到想要的元素, 就用按 HTML 特征查找: 4 个方法:

按 id 查找: var 一个元素对象=document.getElementById("id 名");

按标签名查找: var 类数组对象=任意父元素.getElementsByTagName("标签名")、

按 class 名查找: var 类数组对象=任意父元素.getElementsByClassName("class 名")

按 name 名查找表单元素: var 类数组对象=document.getElementsByName("name 名")

(4). 如果通过复杂的查找条件, 才能找到想要的元素时, 就用按选择器查找: 2 个方法

只查找一个符合条件的元素: var 一个元素=任意父元素.querySelector("任意选择器")

查找多个符合条件的元素: var 类数组对象=任意父元素.querySelectorAll("任意选择器")

2. 修改元素: 3 种东西可修改

(1). 修改内容: 3 种内容可修改:

获取或修改元素的 HTML 内容: 元素.innerHTML

(2). 修改属性: (待续...)

(3). 修改样式:

修改元素的内联样式: 元素.style.css 属性="属性值"

3. 添加/删除元素:(待续...)

4. 事件绑定: 3 种:

(1). 在 HTML 中绑定: DOM 和 jquery 中几乎不用

HTML 中: <元素 on 事件名="事件处理函数()">

JS 中: function 事件处理函数(){ ... }

(2). 使用赋值方式绑定: 元素对象.on 事件名=function(){ ... }

(3). (待续...)

总结: this 5 种: 判断 this, ~~一定不要看定义在哪儿!~~ 只看调用时!

1. obj.fun() this->obj

2. fun() 或 (function(){ ... })() 或 多数回调函数 this->window

3. new Fun() this->new 正在创建的新对象

4. 类型名.prototype.共有方法=function(){ ... } this->将来谁调用指谁, 同第一种情况

5. 事件处理函数中的 this->当前正在触发事件的 DOM 元素对象

Day02:

总结: DOM 4 步:

1. 先查找可能触发事件的元素
2. 再为元素绑定事件处理函数
   3. 查找要修改的元素
   4. 修改元素
   4.1 取出元素的旧值或旧内容
   4.2 计算/修改
   4.3 将新值放回元素的内容或属性上

总结: DOM 5 件事: 增删改查+事件绑定:

1. 查找元素:

(1). 不需要查找就可直接获得的节点对象: 4 种:

document —— 唯一的树根节点

document.documentElement —— <html>

document.head —— <head>

document.body —— <body>

(2). 如果已经获得一个元素对象，找周围附近的元素对象时，就用按节点间关系查找: 2 大类关系，6 个属性

父子关系: 4 种

元素的父元素：元素.parentNode 或元素.parentElement

元素下的所有直接子元素: 元素.children

元素下的第一个直接子元素：元素.firstElementChild

元素下的最后一个直接子元素：元素.lastElementChild

兄弟关系: 2 种

元素的前一个兄弟: 元素.previousElementSibling

元素的后一个兄弟: 元素.nextElementSibling

(3). 如果用一个特征就能找到想要的元素，就用按 HTML 特征查找：4 个方法:

按 id 查找：var 一个元素对象=document.getElementById("id 名");

按标签名查找: var 类数组对象=任意父元素.getElementsByTagName("标签名")、

按 class 名查找: var 类数组对象=任意父元素.getElementsByClassName("class 名")

按 name 名查找表单元素: var 类数组对象=document.getElementsByName("name 名")

(4). 如果通过复杂的查找条件，才能找到想要的元素

时，就用按选择器查找: 2 个方法

只查找一个符合条件的元素: var 一个元素=任意父元素.querySelector("任意选择器")

查找多个符合条件的元素: var 类数组对象=任意父元素.querySelectorAll("任意选择器")

总结: 查找方法的返回值规律: 3 种

1. 如果原方法返回下标位置 i，则如果找不到返回-1

2. 如果原方法返回一个数组或对象，则如果找不到返回 null

3. 如果原方法返回类数组对象，则如果找不到返回空类数组对象: { length:0 }

2. 修改元素: 3

(1). 修改内容: 3 种

a. 获取或修改元素的原始 HTML 内容: 元素.innerHTML

b. 获取或修改元素的纯文本内容: 元素.textContent

c. 获取或修改表单元素的值: 元素.value

(2). 修改属性: 3 种:

a. 字符串类型的 HTML 标准属性: 2 种方式

1). 核心 DOM 4 个函数:

i. 获取属性值: var 属性值=元素.getAttribute("属性名")

ii. 修改属性值: 元素.setAttribute("属性名","属性值")

iii. 判断是否包含某个属性: var bool=元素.hasAttribute("属性名")

iv. 移除属性: 元素.removeAttribute("属性名")

2). HTML DOM 简写:

i. 获取属性值:元素.属性名

ii. 修改属性值: 元素.属性名="属性值"

iii. 判断是否包含某个属性:元素.属性名!==""

iv. 移除属性: 元素.属性名=""

b. bool 类型 HTML 标准属性:

只能用"元素.属性名", 且值为 bool 类型

c. 自定义扩展属性:

1) HTML 中: <元素 data-自定义属性名="属性值">

2). js 中: 2 种:

i. 核心 DOM:

var 属性值=元素.getAttribute("data-自定义属性名")

元素.setAttribute("data-自定义属性名","属性值")

ii. HTML5 标准:　元素.dataset.自定义属性名

(3). 修改样式:

a. 修改元素的内联样式: 元素.style.css 属性="属性值"

b. 获取元素的完整样式: (待续...)

c. 使用 class 批量修改元素的样式: 元素.className="class 名"

3. 添加/删除元素:(待续...)

4. 事件绑定: 3 种:

(1). 在 HTML 中绑定: DOM 和 jquery 中几乎不用

HTML 中: <元素 on 事件名="事件处理函数()">

JS 中: function 事件处理函数(){ ... }

(2). 使用赋值方式绑定: 元素对象.on 事件名=function(){ ... }

(3). (待续...)

总结: this 5 种: 判断 this, ~~一定不要看定义在哪儿!~~ 只看调用时!

1. obj.fun()　this->obj

2. fun() 或 (function(){ ... })() 或 多数回调函数 this->window

3. new Fun()　　this->new 正在创建的新对象

4. 类型名.prototype.共有方法=function(){ … }

this->将来谁调用指谁，同第一种情况

5. 事件处理函数中的 this->当前正在触发事件的 DOM 元素对象

Day03:

总结:

2. 修改元素: 3 种

(1). 修改内容: 3 种

a. 获取或修改元素的原始 HTML 内容: 元素.innerHTML

b. 获取或修改元素的纯文本内容: 元素.textContent

c. 获取或修改表单元素的值: 元素.value

(2). 修改属性: 3 种:

a. 字符串类型的 HTML 标准属性: 2 种方式

1). 核心 DOM 4 个函数:

i. 获取属性值: var 属性值=元素.getAttribute("属性名")

ii. 修改属性值: 元素.setAttribute("属性名","属性值")

iii. 判断是否包含某个属性：var bool= 元素.hasAttribute("属性名")

iv. 移除属性: 元素.removeAttribute("属性名")

2). HTML DOM 简写:

i. 获取属性值:元素.属性名

ii. 修改属性值: 元素.属性名="属性值"

iii. 判断是否包含某个属性:元素.属性名!==""

iv. 移除属性: 元素.属性名=""

b. bool 类型 HTML 标准属性: 只能用"元素.属性名", 且值为 bool 类型

c. 自定义扩展属性:

1) HTML 中: <元素 data-自定义属性名="属性值">

2). js 中: 2 种:

i. 核心 DOM:

var 属性值=元素.getAttribute("data-自定义属性名")

元素.setAttribute("data-自定义属性名","属性值")

ii. HTML5 标准: 元素.dataset.自定义属性名

(3). 修改样式:

a. 修改元素的内联样式: 元素.style.css 属性="属性值"

b. 获取元素的完整样式:

var style=getComputedStyle(元素对象);

style.css 属性

c. 批量修改元素的样式时，都用 class:

元素.className="class 名"

3. 添加/删除元素:

(1). 只添加一个新元素: 3 步

a. 创建一个新元素:

var 新元素=document.createElement("标签名")

b. 为元素设置关键属性:

新元素.属性名="属性值";

c. 将新元素添加到 DOM 树: 3 种:

1). 末尾追加:

父元素.appendChild(新元素)

2). 在某个元素前插入:

父元素.insertBefore(新元素，现有元素)

3). 替换某个元素:

父元素.replaceChild(新元素，现有元素)

(2). 优化: 尽量减少操作 DOM 树的次数

a. 如果同时添加父元素和子元素，应该先将子元素添加到父元素，最后再将父元素一次性添加到 DOM 树

b. 如果父元素已经在页面上，要添加多个平级子元素。

应该利用文档片段对象

1). 创建文档片段对象:

var frag=document.createDocumentFragment()

2). 将子元素添加到文档片段对象中:

frag.appendChild(子元素)

3). 最后将文档片段对象一次性添加到 DOM 树上父元素下

父元素.appendChild(frag);

(3). 删除元素: 父元素.removeChild(子元素)

4. HTML DOM 常用对象:（了解即可）

(1). var img=new Image()

(2). table

a. table 管着行分组:

1). 添加行分组:

var thead=table.createTHead()

var tbody=table.createTBody()

var tfoot=table.createTFoot()

2) 删除行分组:

table.deleteTHead(); table.deleteTFoot()

3). 获取行分组:

table.tHead table. tFoot table.tBodies[i]

b. 行分组管着行:

1). 添加行:

i. 任意行插入新行: var tr=行分组.insertRow(i);

ii. 开头插入新行: var tr=行分组.insertRow(0)

iii. 末尾追加新行: var tr=行分组.insertRow()

2). 删除行: table.deleteRow(tr.rowIndex)

3). 获取行: 行分组.rows[i]

c. 行管着格:

1). 添加格: var td=tr.insertCell()

2). 删除格: tr.deleteCell(i)

3). 获取格: tr.cells[i]

(3). form:

a. 获取 form 元素: document.forms[i]

b. 获取 form 中的表单元素:

1). 标准: form.elements[i 或 id 或 name 名]

2). 简写: 如果有 name 属性: form.name 名

c. 让表单元素自动获得焦点: 表单元素.focus()

Day04:

总结: 不要背英文名字! 反而应该记中文能做哪些事儿!

3. 添加/删除元素:

(1). 只添加一个新元素: 3 步

a. 创建一个新元素:

var 新元素=document.createElement("标签名")

b. 为元素设置关键属性:

新元素.属性名="属性值";

c. 将新元素添加到 DOM 树: 3 种:

1). 末尾追加:

父元素.appendChild(新元素)

2). 在某个元素前插入:

父元素.insertBefore(新元素, 现有元素)

3). 替换某个元素:

父元素.replaceChild(新元素, 现有元素)

(2). 优化: 尽量减少操作 DOM 树的次数, 2 种:

a. 如果同时添加父元素和子元素, 应该先将子元素添加到父元素, 最后再将父元素一次性添加到 DOM 树

b. 如果父元素已经在页面上, 要添加多个平级子元素。应该利用文档片段对象

1). 创建文档片段对象:

var frag=document.createDocumentFragment()

2). 将子元素添加到文档片段对象中:

frag.appendChild(子元素)

3). 最后将文档片段对象一次性添加到 DOM 树上

父元素下

父元素.appendChild(frag);

(3). 删除元素: 父元素.removeChild(子元素)

4. HTML DOM 常用对象:（了解即可）

(1). var img=new Image()

(2). table

a. table 管着行分组:

1). 添加行分组:

var thead=table.createTHead()

var tbody=table.createTBody()

var tfoot=table.createTFoot()

2) 删除行分组:

table.deleteTHead(); table.deleteTFoot()

3). 获取行分组:

table.tHead table. tFoot table.tBodies[i]

b. 行分组管着行:

1). 添加行:

i. 任意行插入新行: var tr=行分组.insertRow(i);

ii. 开头插入新行: var tr=行分组.insertRow(0)

iii. 末尾追加新行: var tr=行分组.insertRow()

2). 删除行: table.deleteRow(tr.rowIndex)

3). 获取行: 行分组.rows[i]

c. 行管着格:

1). 添加格: var td=tr.insertCell()

2). 删除格: tr.deleteCell(i)

3). 获取格: tr.cells[i]

(3). form:

a. 获取 form 元素: document.forms[i]

b. 获取 form 中的表单元素:

1). 标准: form.elements[i 或 id 或 name 名]

2). 简写: 如果有 name 属性: form.name 名

c. 让表单元素自动获得焦点: 表单元素.focus()

总结:

BOM:

1. window:

(1). 获得窗口大小:

a. 获得完整窗口大小:

window.outerWidth 和 window.outerHeight

b. 获得文档显示区大小:

window.innerWidth 和 window.innerHeight

(2). 打开和关闭窗口:

window.open()和 window.close()

2. 打开新链接 4 种方式:

(1). 在当前窗口打开，可后退

a. html: <a href="url" target="_self">

b. js: window.open("url", "_self");

(2). 在当前窗口打开，禁止后退

a. js: location.replace("新 url")

(3). 在新窗口打开，可同时打开多个

a. html: <a href="url" target="_blank">

b. js: window.open("url", "_blank");

(4). 在新窗口打开，只能打开一个

a. html: <a href="url" target="自定义窗口名">

b. js: window.open("url", "自定义窗口名")

3. history:

(1). 前进: history.go(n)

(2). 后退: history.go(-n)

(3). 刷新: history.go(0)

4. location:

(1). 属性: 分段获得 url 中各个部分:

a. location.href　完整 url

b. location.protocol　协议

c. location.host　　主机名+端口号

d. location.hostname　主机名

e. location.port　　端口号

f. location.pathname　相对路径

g. location.search　　?及其之后的查询字符串参数列表

h. location.hash　　#锚点地址

(2). 方法:

a. 在当前窗口打开，可后退:

location.assign("新 url") 或 location.href="新 url"

b. 在当前窗口打开，禁止后退:

location.replace("新 url")

c. 刷新: location.reload();

5. navigator

(1). 查看浏览器的名称和版本号: navigator.userAgent

(2). 查看浏览器中安装的插件列表: navigator.plugins

事件:

1. 绑定事件: js 中:

(1). 一个事件只绑定一个处理函数

元素.on 事件名=function(){ ... }

(2). 一个事件绑定多个处理函数

元素.addEventListener("事件名", 事件处理函数)

(3). 移除一个事件监听:

元素.removeEventListener("事件名"，原事件处理函数对象)

2. 事件模型：捕获，目标触发，冒泡

3. 事件对象:

(1). 获得事件对象:

元素.on 事件名=function(e){ … }

(2). 阻止冒泡: e.stopPropagation()

(3). 当多个子元素都要绑定相同事件时，利用冒泡/事件委托 3 步:

a. 事件只在父元素上绑定一次

b. e.target 代替 this

c. 判断 e.target 的任意特征是否是我们想要的元素

(4). 阻止元素默认行为:

e.preventDefault()

(5). 获取鼠标位置:

a. 相对于屏幕左上角的 x，y 坐标:

e.screenX,　e.screenY

b. 相对于文档显示区左上角的 x，y 用坐标:

e.clientX,　e.clientY

c. 相对于事件所在元素左上角的 x，y 坐标:

e.offsetX　　e.offsetY

(6). 页面滚动事件:

```
window.onscroll=function(){
    var
scrollTop=document.documentElement.scrollTop||
            document.body.scrollTop
    //如果 scrollTop>多少，就执行 xx 操作
    //否则就恢复原样
}
```

张景元