Web APIs - 第4天

> 进一步学习 DOM 相关知识，实现可交互的网页特效

- 能够插入、删除和替换元素节点
- 能够依据元素节点关系查找节点

日期对象

掌握 Date 日期对象的使用，动态获取当前计算机的时间。

ECMAScript 中内置了获取系统时间的对象 Date，使用 Date 时与之前学习的内置对象 console 和 Math 不同，它需要借助 new 关键字才能使用。

实例化

```
// 1. 实例化
// const date = new Date(); // 系统默认时间
const date = new Date('2020-05-01') // 指定时间
// date 变量即所谓的时间对象

console.log(typeof date)
```

方法

```
 // 1. 实例化
const date = new Date();
// 2. 调用时间对象方法
// 通过方法分别获取年、月、日，时、分、秒
const year = date.getFullYear(); // 四位年份
const month = date.getMonth(); // 0 ~ 11
```

getFullYear 获取四位年份

getMonth 获取月份，取值为 0 ~ 11

getDate 获取月份中的每一天，不同月份取值也不相同

getDay 获取星期，取值为 0 ~ 6

getHours 获取小时，取值为 0 ~ 23

getMinutes 获取分钟，取值为 0 ~ 59

getSeconds 获取秒，取值为 0 ~ 59

时间戳

时间戳是指1970年01月01日00时00分00秒起至现在的总秒数或毫秒数，它是一种特殊的计量时间的方式。

注：ECMAScript 中时间戳是以毫秒计的。

```
    // 1. 实例化
  const date = new Date()
  // 2. 获取时间戳
  console.log(date.getTime())
// 还有一种获取时间戳的方法
  console.log(+new Date())
  // 还有一种获取时间戳的方法
  console.log(Date.now())
```

获取时间戳的方法，分别为 getTime 和 Date.now 和 +new Date()

DOM 节点

> 掌握元素节点创建、复制、插入、删除等操作的方法，能够依据元素节点的结构关系查找节点

回顾之前 DOM 的操作都是针对元素节点的属性或文本的，除此之外也有专门针对元素节点本身的操作，如插入、复制、删除、替换等。

插入节点

在已有的 DOM 节点中插入新的 DOM 节点时，需要关注两个关键因素：首先要得到新的 DOM 节点，其次在哪个位置插入这个节点。

如下代码演示：

```
<body>
  <h3>插入节点</h3>
  <p>在现有 dom 结构基础上插入新的元素节点</p>
  <hr>
  <!-- 普通盒子 -->
  <div class="box"></div>
  <!-- 点击按钮向 box 盒子插入节点 -->
  <button class="btn">插入节点</button>
  <script>
    // 点击按钮，在网页中插入节点
    const btn = document.querySelector('.btn')
    btn.addEventListener('click', function () {
      // 1. 获得一个 DOM 元素节点
      const p = document.createElement('p')
      p.innerText = '创建的新的p标签'
      p.className = 'info'

      // 复制原有的 DOM 节点
      const p2 = document.querySelector('p').cloneNode(true)
      p2.style.color = 'red'

      // 2. 插入盒子 box 盒子
      document.querySelector('.box').appendChild(p)
      document.querySelector('.box').appendChild(p2)
    })
  </script>
```

```
</body>
```

结论：

- `createElement` 动态创建任意 DOM 节点
- `cloneNode` 复制现有的 DOM 节点，传入参数 true 会复制所有子节点
- `appendChild` 在末尾（结束标签前）插入节点

再来看另一种情形的代码演示：

```
<body>
  <h3>插入节点</h3>
  <p>在现有 dom 结构基础上插入新的元素节点</p>
    <hr>
  <button class="btn1">在任意节点前插入</button>
  <ul>
    <li>HTML</li>
    <li>CSS</li>
    <li>JavaScript</li>
  </ul>
  <script>
    // 点击按钮，在已有 DOM 中插入新节点
    const btn1 = document.querySelector('.btn1')
    btn1.addEventListener('click', function () {

      // 第 2 个 li 元素
      const relative = document.querySelector('li:nth-child(2)')

      // 1. 动态创建新的节点
      const li1 = document.createElement('li')
      li1.style.color = 'red'
      li1.innerText = 'Web APIs'

      // 复制现有的节点
      const li2 = document.querySelector('li:first-child').cloneNode(true)
      li2.style.color = 'blue'

      // 2. 在 relative 节点前插入
      document.querySelector('ul').insertBefore(li1, relative)
      document.querySelector('ul').insertBefore(li2, relative)
    })
  </script>
</body>
```

结论：

- `createElement` 动态创建任意 DOM 节点
- `cloneNode` 复制现有的 DOM 节点，传入参数 true 会复制所有子节点
- `insertBefore` 在父节点中任意子节点之前插入新节点

删除节点

删除现有的 DOM 节点，也需要关注两个因素：首先由父节点删除子节点，其次是要删除哪个子节点。

```
<body>
  <!-- 点击按钮删除节点 -->
```

```
  <button>删除节点</button>
  <ul>
    <li>HTML</li>
    <li>CSS</li>
    <li>Web APIs</li>
  </ul>

  <script>
    const btn = document.querySelector('button')
    btn.addEventListener('click', function () {
      // 获取 ul 父节点
      let ul = document.querySelector('ul')
      // 待删除的子节点
      let lis = document.querySelectorAll('li')

      // 删除节点
      ul.removeChild(lis[0])
    })
  </script>
</body>
```

结论：`removeChild` 删除节点时一定是由父子关系。

查找节点

DOM 树中的任意节点都不是孤立存在的，它们要么是父子关系，要么是兄弟关系，不仅如此，我们可以依据节点之间的关系查找节点。

父子关系

```
<body>
  <button class="btn1">所有的子节点</button>
  <!-- 获取 ul 的子节点 -->
  <ul>
    <li>HTML</li>
    <li>CSS</li>
    <li>JavaScript 基础</li>
    <li>Web APIs</li>
  </ul>
  <script>
    const btn1 = document.querySelector('.btn1')
    btn1.addEventListener('click', function () {
      // 父节点
      const ul = document.querySelector('ul')

      // 所有的子节点
      console.log(ul.childNodes)
      // 只包含元素子节点
      console.log(ul.children)
    })
  </script>
</body>
```

结论：

- `childNodes` 获取全部的子节点，回车换行会被认为是空白文本节点
- `children` 只获取元素类型节点

```
<body>
  <table>
    <tr>
      <td width="60">序号</td>
      <td>课程名</td>
      <td>难度</td>
      <td width="80">操作</td>
    </tr>
    <tr>
      <td>1</td>
      <td><span>HTML</span></td>
      <td>初级</td>
      <td><button>变色</button></td>
    </tr>
    <tr>
      <td>2</td>
      <td><span>CSS</span></td>
      <td>初级</td>
      <td><button>变色</button></td>
    </tr>
    <tr>
      <td>3</td>
      <td><span>Web APIs</span></td>
      <td>中级</td>
      <td><button>变色</button></td>
    </tr>
  </table>
  <script>
    // 获取所有 button 节点，并添加事件监听
    const buttons = document.querySelectorAll('table button')
    for(let i = 0; i < buttons.length; i++) {
      buttons[i].addEventListener('click', function () {
        // console.log(this.parentNode); // 父节点 td
        // console.log(this.parentNode.parentNode); // 爷爷节点 tr
        this.parentNode.parentNode.style.color = 'red'
      })
    }
  </script>
</body>
```

结论：`parentNode` 获取父节点，以相对位置查找节点，实际应用中非常灵活。

兄弟关系

```
<body>
  <ul>
    <li>HTML</li>
    <li>CSS</li>
    <li>JavaScript 基础</li>
    <li>Web APIs</li>
  </ul>
  <script>
    // 获取所有 li 节点
    const lis = document.querySelectorAll('ul li')

    // 对所有的 li 节点添加事件监听
```

```
    for(let i = 0; i < lis.length; i++) {
      lis[i].addEventListener('click', function () {
        // 前一个节点
        console.log(this.previousSibling)
        // 下一下节点
        console.log(this.nextSibling)
      })
    }
  </script>
</body>
```

结论：

- `previousSibling` 获取前一个节点，以相对位置查找节点，实际应用中非常灵活。
- `nextSibling` 获取后一个节点，以相对位置查找节点，实际应用中非常灵活。