JS中的DOM

认识DOM和BOM

- **前面我们花了很多时间学习JavaScript的基本语法，但是这些基本语法，但是这些语法好像和做网页没有什么关系，和前面学习的HTML、CSS也没有什么关系呢?**
  - 这是因为我们前面**学习的部分属于ECMAScript**，也就是**JavaScript本身的语法部分**
  - 除了语法部分之外，我们还需要**学习浏览器提供给我们开发者的DOM、BOM相关的API才能对页面、浏览器**进行操作
- **前面我们学习了一个window的全局对象，window上事实上就包含了这些内容:**
  - 我们已经学习了**JavaScript语法部分的Object、Array、Date**等
  - 另外还有**DOM、BOM**部分
- **DOM：文档对象模型（Document Object Model）**
  - 简称 **DOM**，将**页面所有的内容表示为可以修改的对象**
- **BOM：浏览器对象模型（Browser Object Model）**
  - 简称 **BOM**，由**浏览器提供的用于处理文档（document）之外的所有内容的其他对象**
  - 比如**navigator、location、history**等对象

深入理解DOM

- **浏览器会对我们编写的HTML、CSS进行渲染，同时它又要考虑我们可能会通过JavaScript来对其进行操作:**
  - 于是**浏览器将我们编写在HTML中的每一个元素（Element）都抽象成了一个个对象**
  - 所有这些对象都可以**通过JavaScript来对其进行访问**，那么我们就可以**通过JavaScript来操作页面**
  - 所以，我们将**这个抽象过程**称之为 **文档对象模型（Document Object Model）**
- **整个文档被抽象到 document 对象中:**
  - 比如**document.documentElement**对应的是**html元素**
  - 比如**document.body**对应的是**body元素**
  - 比如**document.head**对应的是**head元素**
- **下面的一行代码可以让整个页面变成红色:**

```
document.body.style.background = "red"
```

- **所以我们学习DOM，就是在学习如何通过JavaScript对文档进行操作的;**(通过浏览器提供给我们的接口实现)

> API（Application Programming Interface，应用程序编程接口）是一组预先定义的规则和约定，用于在软件组件、库、框架、操作系统或其他应用程序之间实现通信和互操作。API定义了如何调用和使用各种功能、方法、数据结构等，以便开发者能够更轻松地构建和集成应用程序。

API可以分为以下几类：

1. **库和框架API**：这类API提供了特定编程语言的库、框架或SDK中的功能。开发者可以使用这些API构建应用程序，而无需从零开始编写所有代码。例如，React是一个JavaScript库，提供了一组API，用于构建Web应用程序的用户界面。
2. **操作系统API**：操作系统（如Windows、macOS、Linux等）提供了一组API，供开发者使用系统资源和功能，例如文件操作、网络通信、硬件访问等。
3. **Web API**：Web API通常是一组HTTP端点，允许客户端应用程序（如Web浏览器、移动应用等）与服务器端应用程序进行通信。Web API遵循一定的通信协议，如REST、GraphQL等。客户端可以向服务器发送请求（如GET、POST、PUT、DELETE等HTTP方法），服务器则以特定格式（如JSON、XML等）返回数据。
4. **硬件API**：硬件API提供了与外部硬件设备（如打印机、传感器、摄像头等）进行通信和互操作的接口。这些API通常由硬件制造商提供，并支持特定的设备功能。

API的主要优点是将软件的内部实现细节隐藏起来，仅暴露必要的接口。这有助于降低开发复杂性，提高代码可维护性和可重用性。同时，API还可以为不同的编程语言和平台提供兼容性和互操作性。

DOM Tree的理解

DOM Tree（Document Object Model Tree，文档对象模型树）是一种表示HTML或XML文档结构的树形数据结构。DOM Tree将文档中的各个元素（如标签、属性、文本等）表示为节点（Node），并通过父子关系、兄弟关系等层级关系将这些节点组织起来。在Web浏览器中，DOM Tree是通过JavaScript访问和操作网页内容的主要方式。

DOM Tree的主要组成部分和概念如下：

1. **节点（Node）**：DOM Tree中的基本单位。每个节点都具有一个特定的类型，例如元素节点、文本节点、属性节点等。节点之间存在层级关系，如父子关系和兄弟关系。
2. **元素节点（Element Node）**：表示HTML或XML文档中的元素（如`<div>`、`<span>`、`<p>`等）。元素节点可以包含其他元素节点、文本节点或属性节点。
3. **文本节点（Text Node）**：表示元素节点中的文本内容。文本节点是DOM Tree中的叶子节点，不能包含其他节点。
4. **属性节点（Attribute Node）**：表示元素节点的属性（如`id`、`class`、`style`等）。属性节点为元素节点提供附加信息或配置。
5. **根节点（Root Node）**：表示DOM Tree的起始点。在HTML文档中，根节点通常是`<html>`元素。根节点没有父节点，是整个DOM Tree的最顶层节点。

在Web浏览器中，JavaScript可以通过DOM API访问和操作DOM Tree，实现对网页内容的动态修改、事件处理等功能。DOM API提供了一系列方法和属性，用于遍历、查找、创建、删除和修改节点。这些操作对于实现动态网页、表单验证、动画效果等功能至关重要。

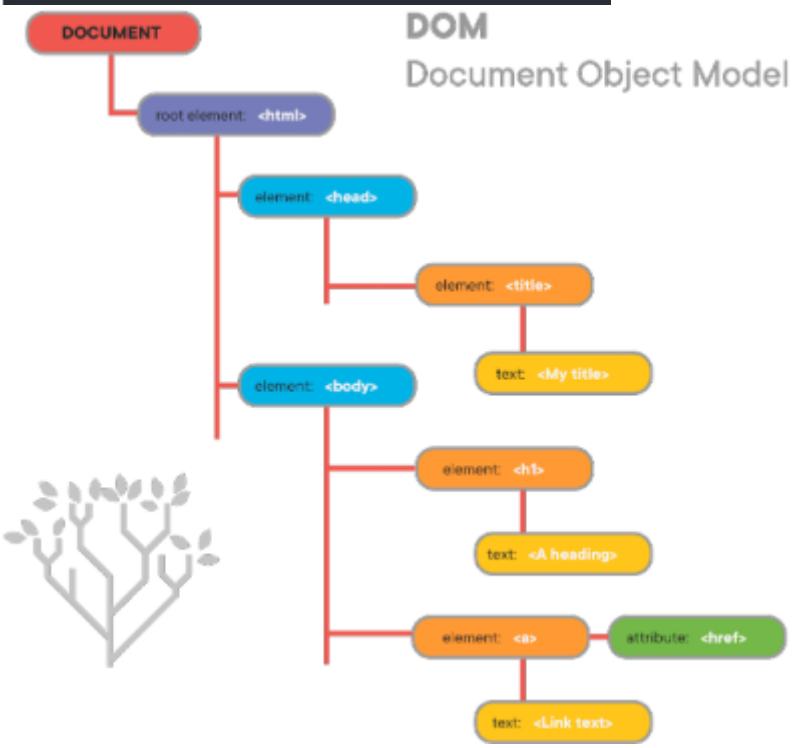
需要注意的是，DOM操作可能会影响网页的性能。频繁地修改DOM Tree可能导致页面重排、重绘等资源消耗较大的操作。因此，在实际开发中，开发者需要权衡性能与功能，确保网页的流畅性和响应速度。

总结：DOM Tree是表示HTML或XML文档结构的树形数据结构，通过JavaScript和DOM API，开发者可以在浏览器中访问和操作网页内容，实现各种动态功能

- **一个页面不只是有html、head、body元素，也包括很多的子元素：**

- 在html结构中，最终会形成一个**树结构**
- 在抽象成DOM对象的时候，它们也会形成一个**树结构**，我们称之为**DOM Tree**

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <title>My Title</title>
</head>
<body>
  <h1>A Heading</h1>
  <a href="#">Link Text</a>
</body>
</html>
```



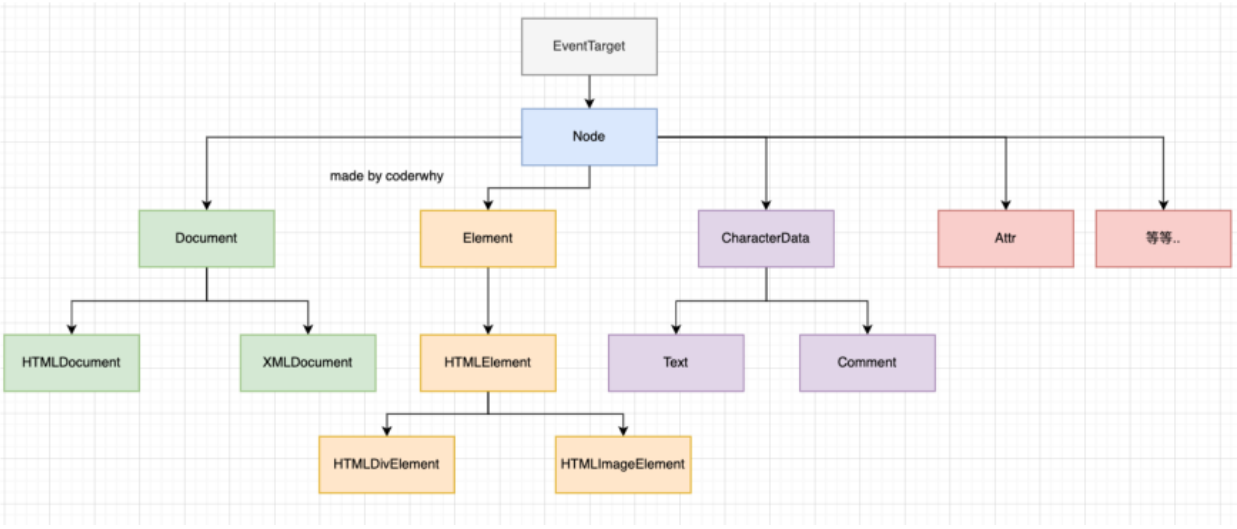
DOM的学习顺序

- **DOM相关的API非常多，我们会通过如下顺序来学习：**
  1. DOM元素之间的关系
  2. 获取DOM元素
  3. DOM节点的type、tag、content
  4. DOM节点的attributes、properies
  5. DOM节点的创建、插入、克隆、删除
  6. DOM节点的样式、类
  7. DOM元素/window的大小、滚动、坐标

DOM的继承关系图

> 1. **Node**：Node 是 DOM 层次结构中的基本对象类型，所有其他对象类型都继承自 Node。Node 提供了一些基本属性和方法，例如 `parentNode`、`nextSibling`、`previousSibling`、`appendChild()` 和 `removeChild()` 等，用于遍历和操作 DOM 树。
> 2. **Document**：Document 对象是整个文档的根节点，继承自 Node。它代表整个 HTML 或 XML 文档，提供了许多用于访问和操作文档内容的方法和属性，如 `getElementById()`、`getElementsByClassName()`、`createElement()` 等。
> 3. **Element**：Element 是所有元素节点的基本类型，继承自 Node。Element 提供了一些通用的属性和方法，如 `getAttribute()`、`setAttribute()`、`classList` 等，用于访问和操作元素及其属性。
> 4. **HTMLElement**：HTMLElement 是 HTML 元素节点的基本类型，继承自 Element。它包含了一些 HTML 特有的属性和方法，如 `innerHTML`、`style`、`click()` 等。
> 5. **HTML*Element**：例如 HTMLAnchorElement、HTMLDivElement、HTMLImageElement 等，这些对象分别代表特定类型的 HTML 元素，如 `<a>`、`<div>`、`<img>` 等。它们继承自 HTMLElement，可能包含一些特定元素相关的属性和方法。
> 6. **SVGElement**：SVGElement 是 SVG 元素节点的基本类型，继承自 Element。它包含了一些 SVG 特有的属性和方法，用于处理矢量图形元素。
> 7. **SVG*Element**：例如 SVGCircleElement、SVGRectElement 等，这些对象分别代表特定类型的 SVG 元素，如 `<circle>`、`<rect>` 等。它们继承自 SVGElement，可能包含一些特定元素相关的属性和方法。
> 8. **Attr**：Attr 对象表示元素的属性节点，继承自 Node。Attr 提供了一些属性和方法，用于访问和操作属性值。
> 9. **Text**：Text 对象表示文本节点，继承自 Node。Text 提供了一些用于操作文本内容的属性和方法，如 `nodeValue`、`data` 和 `splitText()` 等。
> 10. **Comment**：Comment 对象表示注释节点，继承自 Node。Comment 提供了一些用于操作注释内容的属性和方法，类似于 Text 对象。

- **DOM相当于是JavaScript和HTML、CSS之间的桥梁**
  - 通过浏览器提供给我们的**DOM API**，我们可以**对元素以及其中的内容**做任何事情
- **类型之间有如下的继承关系：**




```
Node
├── Document
├── Element
|   ├── HTMLElement
|   |   ├── HTMLAnchorElement
|   |   ├── HTMLDivElement
|   |   ├── HTMLImageElement
|   |   ├── HTMLInputElement
|   |   └── ...
|   └── SVGElement
|       ├── SVGCircleElement
|       ├── SVGRectElement
|       └── ...
├── Attr
├── Text
├── Comment
└── ...
```

document对象

> `document` 对象是Web浏览器中 JavaScript 的一个核心对象，代表整个HTML文档。通过 `document` 对象，开发者可以访问和操作网页中的元素、属性、文本等内容。`document` 对象实现了 DOM（Document Object Model，文档对象模型）的接口，定义了一系列方法和属性，用于与文档进行交互。

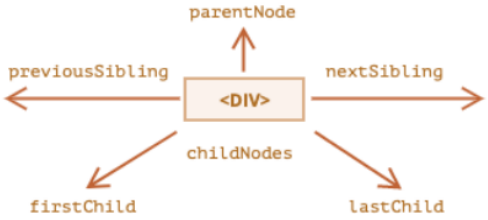
- **Document节点表示的整个载入的网页，它的实例是全局的document对象：**
  - 对DOM的所有操作都是**从 document 对象开始**的
  - 它是**DOM的 入口点**，可以从**document**开始去访问任何节点元素
- **对于最顶层的html、head、body元素，我们可以直接在document对象中获取到：**
  - **html元素**： = document.documentElement
  - **body元素**： = document.body
  - **head元素**： = document.head
  - **文档声明**： = document.doctype

```
console.log(document.doctype)
console.log(document.documentElement)
console.log(document.head)
console.log(document.body)
```

节点（Node）之间的导航（navigator）

- **如果我们获取到一个节点（Node）后，可以根据这个节点去获取其他的节点，我们称之为节点之间的导航**
- **节点之间存在如下的关系：**

- 父节点：parentNode
- 前兄弟节点：previousSibling
- 后兄弟节点：nextSibling
- 子节点：childNodes
- 第一个子节点：firstChild
- 第二个子节点：lastChild



表格（table）元素的导航（navigator）

- **元素支持 (除了上面给出的，之外) 以下这些属性：**
  - table.rows —

  元素的集合
- table.caption/tHead/tFoot — 引用元素

  ，
- table.tBodies —

  元素的集合
- **，， 元素提供了 rows 属性：**
  - **tbody.rows** — 表格内部

  元素的集合
- ：
  - **tr.cells** — 在给定

  中的 在封闭的 // 中的位置（索引）
- **tr.rowIndex** — 在整个表格中

  的编号（包括表格的所有行）
- **中单元格的编号**

获取元素的方法

- **当元素彼此靠近或者相邻时，DOM 导航属性（navigation property）非常有用**
  - **但是，在实际开发中，我们希望可以任意的获取到某一个元素应该如何操作呢?**
- **DOM为我们提供了获取元素的方法：**

以下是 `document` 对象的一些主要方法和属性：

1. getElementById()：通过元素的 `id` 属性获取一个元素节点。例如，`document.getElementById('my-element')` 将返回一个具有 `id` 为 "my-element" 的元素。
2. getElementsByClassName()：通过元素的 `class` 属性获取一个元素节点列表。例如，`document.getElementsByClassName('my-class')` 将返回一个包含所有具有 `class` 为 "my-class" 的元素的列表。
3. getElementsByTagName()：通过元素的标签名获取一个元素节点列表。例如，`document.getElementsByTagName('div')` 将返回一个包含所有 `div` 元素的列表。
4. querySelector()：使用CSS选择器查找单个元素节点。例如，`document.querySelector('#my-element .my-class')` 将返回与指定选择器匹配的第一个元素。
5. querySelectorAll()：使用CSS选择器查找多个元素节点。例如，`document.querySelectorAll('div.my-class')` 将返回一个包含所有与指定选择器匹配的元素的列表。
6. createElement()：创建一个新的元素节点。例如，`document.createElement('div')` 将创建一个新的 `div` 元素。
7. createTextNode()：创建一个新的文本节点。例如，`document.createTextNode('Hello, World!')` 将创建一个包含 "Hello, World!" 文本的新文本节点。
8. appendChild()：将一个节点添加为另一个节点的子节点。例如，`document.body.appendChild(newElement)` 将将 `newElement` 添加到 `body` 元素的子节点中。
9. removeChild()：从其父节点中删除一个节点。例如，`parentNode.removeChild(childNode)` 将从 `parentNode` 中删除 `childNode`。
10. addEventListener()：为元素添加事件监听器。例如，`document.addEventListener('click', handleClick)` 将为整个文档添加一个点击事件监听器。
11. removeEventListener()：移除元素的事件监听器。例如，`document.removeEventListener('click', handleClick)` 将移除整个文档的点击事件监听器。
12. body：表示文档的 `<body>` 元素。
13. head：表示文档的 `<head>` 元素。
14. title：表示文档的标题，即 `<title>` 标签的内容。可以读取和修改。

方法名	搜索方式	可以在元素上调用?	实时的?
querySelector	CSS-selector	✓	-
querySelectorAll	CSS-selector	✓	-
getElementById	id	-	-
getElementsByName	name	-	✓
getElementsByTagName	tag or '*'	✓	✓
getElementsByClassName	class	✓	✓

> **图片中实时的意思是：**
>
> - **当我们说 `querySelector` 不是实时的，是指它在调用时会立即查询 DOM，返回与给定选择器匹配的第一个元素（如果存在）。这意味着 `querySelector` 返回的结果是对当时 DOM 结构的一个静态"快照"。**
> - **与之相对的概念是实时更新的 NodeList，例如 `getElementsByClassName` 方法返回的 NodeList。这种 NodeList 会随着 DOM 的变化而实时更新。也就是说，当你在 DOM 中添加、删除或修改与 NodeList 中元素相关的节点时，NodeList 会自动反映这些变化。**
> - **下面是一个例子，说明 `querySelector` 不是实时的：**
>
>   ```
>   <ul id="myList">
>     <li>Item 1</li>
>     <li>Item 2</li>
>   </ul>
>
>   <script>
>     const firstListItem = document.querySelector("#myList li");
>     console.log(firstListItem.textContent); // 输出 "Item 1"
>
>     // 现在删除第一个列表项
>     firstListItem.remove();
>
>     // 再次使用 querySelector 查找第一个列表项
>     const newFirstListItem = document.querySelector("#myList li");
>     console.log(newFirstListItem.textContent); // 输出 "Item 2"
>   </script>
>   ```
>
> - **在这个例子中，当我们第一次调用 `querySelector` 时，它返回的是当前存在于 DOM 中的第一个列表项。我们删除了第一个列表项后，再次调用 `querySelector`，它返回的是现在的第一个列表项（原来的第二个列表项）。`querySelector` 的结果不会因为 DOM 的变化而自动更新，所以我们需要再次调用它以获取新的第一个列表项。**

元素上调用querySelector

- **`querySelector` 是一个非常实用的 DOM 方法，可以用于在一个元素的子元素中查找与给定 CSS 选择器匹配的第一个元素。`querySelector` 不仅可以在 `document` 对象上调用，还可以在任何元素节点上调用。**

  当我们在元素节点上调用 `querySelector` 时，它将从调用元素的子元素中查找与给定选择器匹配的元素。这样，你可以更精确地查找特定部分的子元素，而无需在整个文档中进行搜索。

下面是一个例子，说明如何在元素上调用 `querySelector`：

假设我们的 HTML 结构如下：

```
<div id="container">
  <div class="item">Item 1</div>
  <div class="item">Item 2</div>
  <div class="sub-container">
    <div class="item">Item 3</div>
    <div class="item">Item 4</div>
  </div>
</div>
```

- 现在我们希望找到 `sub-container` 中的第一个具有类名 `item` 的元素。可以使用以下代码：

```
// 首先，获取 sub-container 元素的引用
const subContainer = document.querySelector(".sub-container");

// 在 sub-container 元素上调用 querySelector，查找第一个具有类名 item 的元素
const firstItemInSubContainer = subContainer.querySelector(".item");

console.log(firstItemInSubContainer.textContent); // 输出 "Item 3"
```

- 在这个例子中，我们首先使用 `querySelector` 从 `document` 中查找具有类名 `sub-container` 的元素。然后，我们在找到的 `sub-container` 元素上调用 `querySelector`，查找第一个具有类名 `item` 的子元素。这样，我们可以精确地找到 `sub-container` 中的第一个 `item` 元素。
- 开发中如何选择呢?
  - 目前最常用的是querySelector和querySelectAll
  - getElementById偶尔也会使用或者在适配一些低版本浏览器时

节点的属性 - nodeType

- 目前，我们已经可以获取到节点了，接下来我们来看一下节点中有哪些常见的属性：
  - 当然，不同的节点类型有可能有不同的属性
  - 这里我们主要讨论节点共有的属性
- nodeType属性：
  - nodeType 属性提供了一种获取节点类型的方法
  - 它有一个数值型值（numeric value）
- 常见的节点类型有如下：

常量	值	描述
Node.ELEMENT_NODE	1	一个 元素 节点，例如 <p> 和 。
Node.TEXT_NODE	3	Element 或者 Attr 中实际的 文字
Node.COMMENT_NODE	8	一个 Comment 节点。
Node.DOCUMENT_NODE	9	一个 Document 节点。
Node.DOCUMENT_TYPE_NODE	10	描述文档类型的 DocumentType 节点。例如 <!DOCTYPE html> 就是用于 HTML5 的。

- 其他类型可以查看MDN文档： https://developer.mozilla.org/zh-CN/docs/Web/API/Node/nodeType

节点的属性 – nodeName、tagName

> `nodeName` 和 `tagName` 都是 DOM 节点的属性，它们用于表示节点的名称。但是，这两个属性在不同类型的节点上有不同的表现。
>
> 1. nodeName：`nodeName` 是一个只读属性，用于表示节点的名称。对于不同类型的节点，`nodeName` 的值可能有所不同。以下是 `nodeName` 在不同类型节点上的表现：
>   - 元素节点：`nodeName` 返回元素的标签名（大写形式）。
>   - 属性节点：`nodeName` 返回属性的名称。
>   - 文本节点：`nodeName` 的值是 `#text`。
>   - 文档节点：`nodeName` 的值是 `#document`。
>   - 注释节点：`nodeName` 的值是 `#comment`。
>   - 文档片段节点：`nodeName` 的值是 `#document-fragment`。
>
>   示例：

```
const div = document.createElement("div");
console.log(div.nodeName); // 输出 "DIV"
```

> tagName：`tagName` 是一个只读属性，仅用于表示元素节点（`nodeType` 为 1）的标签名。`tagName` 的值是元素的标签名（大写形式）。对于非元素节点，`tagName` 属性为 `undefined`。
>
> 示例：

```
const div = document.createElement("div");
console.log(div.tagName); // 输出 "DIV"
```

> 总结一下，`nodeName` 属性在所有类型的节点上都有值，而 `tagName` 属性仅在元素节点上有效。当处理元素节点时，`nodeName` 和 `tagName` 属性的值是相同的。

- 目前，我们已经可以获取到节点了，接下来我们来看一下节点中有哪些常见的属性：
  - 当然，不同的节点类型有可能有不同的属性；
  - 这里我们主要讨论节点共有的属性
- nodeType属性：
  - nodeType 属性提供了一种获取节点类型的方法
  - 它有一个数值型值（numeric value）
- 常见的节点类型有如下：

> 我们就能够根据返回的值来判断这是一个什么类型的节点

```
//两种方式都可以
if(node.nodeType === Node.COMMENT_NODE)
或者
if(node.nodeType === 8)
```

常量	值	描述
Node.ELEMENT_NODE	1	一个 元素 节点，例如 <p> 和 。
Node.TEXT_NODE	3	Element 或者 Attr 中实际的 文字
Node.COMMENT_NODE	8	一个 Comment 节点。
Node.DOCUMENT_NODE	9	一个 Document 节点。
Node.DOCUMENT_TYPE_NODE	10	描述文档类型的 DocumentType 节点。例如 <!DOCTYPE html> 就是用于 HTML5 的。

- **其他类型可以查看MDN文档：** https://developer.mozilla.org/zh-CN/docs/Web/API/Node/nodeType
- **nodeName：获取node节点的名字**
- **tagName：获取元素的标签名词**

```
var textNode = document.body.firstChild
var itemNode = document.body.childNodes[3]
console.log(textNode.nodeName)
console.log(itemNode.nodeName)
```

- **tagName 和 nodeName 之间有什么不同呢?**
  - **tagName 属性仅适用于 Element 节点**
  - **nodeName 是为任意 Node 定义的：**

    ✓ 对于元素，它的意义与 tagName 相同，所以使用哪一个都是可以的

    ✓ 对于其他节点类型（text，comment 等），它拥有一个对应节点类型的字符串

节点的属性 - innerHTML、textContent

- **innerHTML 属性**
  - **将元素中的 HTML 获取为字符串形式**
  - **设置元素中的内容**
- **outerHTML 属性**
  - **包含了元素的完整 HTML**
  - **innerHTML 加上元素本身一样**
- **textContent 属性**
  - **仅仅获取元素中的文本内容**
- **innerHTML和textContent的区别：**
  - **使用 innerHTML，我们将其“作为 HTML”插入，带有所有 HTML 标签**
  - **使用 textContent，我们将其“作为文本”插入，所有符号（symbol）均按字面意义处理**

节点的属性 - nodeValue

- nodeValue/data
  - 用于获取非元素节点的文本内容

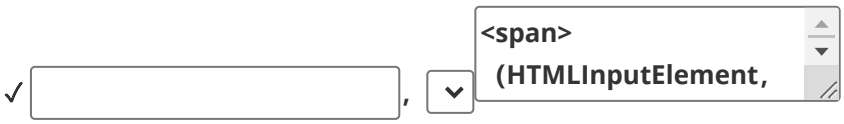
```
coderwhy
<!-- 注释内容 -->

<script>
  var text = document.body.firstChild
  console.log(text.nodeValue)

  var comment = text.nextSibling
  console.log(comment.nodeValue)
</script>
```

节点的其他属性

- hidden属性：也是一个全局属性，可以用于设置元素隐藏

```
<div class="box">哈哈哈哈</div>

<script>
  var box = document.querySelector(".box")
  box.hidden = true
</script>
```

- DOM 元素还有其他属性：
  - value




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

if (div.hasAttribute('class')) {

  console.log('该元素具有 class 属性');

} else {

  console.log('该元素不具有 class 属性');

}
```

  - elem.getAttribute(name) — 获取这个特性值
    - `elem.getAttribute(name)`：假设有一个 `<img>` 元素，我们要获取其 `src` 属性的值

```
const img = document.querySelector('img');
const srcValue = img.getAttribute('src');
console.log(`该图片的地址为：${srcValue}`);
```

- elem.setAttribute(name, value) — 设置这个特性值
  - `elem.setAttribute(name, value)`：假设有一个 `<input>` 元素，我们要将其 `value` 属性的值设置为 `hello world`

```
const input = document.querySelector('input');
input.setAttribute('value', 'hello world');
console.log(`该输入框的值为：${input.value}`);
```

- elem.removeAttribute(name) — 移除这个特性
  - `elem.removeAttribute(name)`：假设有一个 `<a>` 元素，我们要移除其 `href` 属性

```
const a = document.querySelector('a');
a.removeAttribute('href');
console.log('已经移除了 href 属性');
```

- attributes：attr对象的集合，具有name、value属性
  - `attributes`：假设有一个 `<div>` 元素，我们要获取其所有属性的名称和值

```
const div = document.querySelector('div');
const attrs = div.attributes;
for (let i = 0; i < attrs.length; i++) {
  console.log(`属性名：${attrs[i].name}，属性值：${attrs[i].value}`);
}
```

```
for (var attr of boxEl.attributes) {
  console.log(attr.name, attr.value)
}
console.log(boxEl.hasAttribute("age"))
console.log(boxEl.getAttribute("name"))
boxEl.setAttribute("name", "kobe")
boxEl.removeAttribute("abc")
```

- attribute具备以下特征：
  - 它们的名字是大小写不敏感的（id 与 ID 相同）
  - 它们的值总是字符串类型的，而这其实算一个缺陷。
    - 类型转换：当我们使用 DOM 操作修改某个属性的值时，需要将其转换为字符串类型，然后再赋值。这可能会导致一些类型转换的问题，例如当我们将一个数字类型的属性设置为字符串类型的值时，可能会丢失一些精度。而如果该属性本身就是一个字符串类型，我们在对其进行操作时也需要注意类型转换的问题。

- 性能问题：由于属性和特性的值总是字符串类型，因此当我们需要对其进行一些数值计算时，需要先将其转换为数值类型，这可能会导致一些性能问题。在某些情况下，如果我们能直接使用数值类型的属性和特性，可能会更加高效。

元素的属性（property）

- 对于标准的attribute，会在DOM对象上创建与其对应的property属性：
  1. 对于标准的 HTML 属性（例如 `class`、`id`、`name` 等），它们会在 DOM 元素对象上创建对应的 JavaScript 对象属性，这些 JavaScript 对象属性通常被称为 DOM 元素的 property 属性。这样，我们就可以使用 JavaScript 对象的属性访问方式来访问 DOM 元素的属性，例如 `element.className` 可以用于获取元素的 `class` 属性的值，`element.id` 可以用于获取元素的 `id` 属性的值，等等。
  2. 这种特性使得我们可以像操作普通 JavaScript 对象一样来操作 DOM 元素，更加方便地对其进行修改和查询。但需要注意的是，并不是所有的 HTML 属性都会被映射为 DOM 元素的 property 属性，例如 `data-*` 属性和自定义属性就不会被自动映射。对于这些属性，我们可以使用 `getAttribute()` 和 `setAttribute()` 方法来操作它们。
  3. 需要注意的是，DOM 元素的 property 属性和 attribute 属性的值并不总是相等的。当我们使用 `element.propertyName` 的方式来访问属性时，获取到的是该属性在元素对象上对应的 property 属性的值，而不是 attribute 属性的值。当我们修改了元素的 property 属性时，attribute 属性的值也会被相应地修改。但是，当我们修改了元素的 attribute 属性时，property 属性的值不会自动更新，需要手动进行同步。

举个例子来说明 DOM 元素的 property 属性和 attribute 属性的区别和联系：

```
<div id="myDiv" class="myClass" data-custom="customValue"></div>
```

- 我们可以使用以下代码来访问和修改元素的属性：

```
const div = document.getElementById('myDiv');

// 获取元素的 id 和 class 属性值，使用属性访问方式
console.log(div.id);    // 输出：myDiv
console.log(div.className);   // 输出：myClass

// 获取元素的自定义属性值，使用 getAttribute() 方法
console.log(div.getAttribute('data-custom'));   // 输出：customValue

// 修改元素的 id 和 class 属性值，使用属性赋值方式
div.id = 'newId';
div.className = 'newClass';

// 修改元素的自定义属性值，使用 setAttribute() 方法
div.setAttribute('data-custom', 'newValue');
```

- 在上述代码中，我们使用属性访问方式来获取和修改元素的 `id` 和 `class` 属性值，使用 `getAttribute()` 和 `setAttribute()` 方法来获取和修改元素的 `data-custom` 自定义属性的值。需要注意的是，当我们修改了元素的 `id`、`class` 和 `data-custom` 属性的值时，它们对应的 attribute 属性的值也会相应地修改。但是，当我们修改了 attribute 属性的值时，property 属性的值不会自动更新，需要手动进行同步。
- 例如，我们可以使用以下代码来手动同步元素的 `id` 和 `class` 属性的值：

```
// 同步元素的 id 和 class 属性值
div.id = div.getAttribute('id');
div.className = div.getAttribute('class');
```

- 这样，当我们修改了元素的 `id` 和 `class` 属性的值时，它们对应的 attribute 属性的值也会被相应地修改，并且我们手动同步后，property 属性的值也会更新。
- 在大多数情况下，它们是相互作用的
  - 改变property，通过attribute获取的值，会随着改变
  - 通过attribute操作修改，property的值会随着改变
- 除非特别情况，大多数情况下，设置、获取attribute，推荐使用property的方式：
  - 这是因为它默认情况下是有类型的

```
toggleBtn.onclick = function() {
  checkBoxInput.checked = !checkBoxInput.checked
}
```

HTML5的data-*自定义属性

- 前面我们有学习HTML5的data-*自定义属性，那么它们也是可以在dataset属性中获取到的：

```
<div class="box" data-name="why" data-age="18"></div>

<script>
  var boxEl = document.querySelector(".box")
  console.log(boxEl.dataset.name)
  console.log(boxEl.dataset.age)
</script>
```

JavaScript动态修改样式

- 有时候我们会通过JavaScript来动态修改样式，这个时候我们有两个选择：
  - 选择一：在CSS中编写好对应的样式，动态的添加class；
  - 选择二：动态的修改style属性
- 开发中如何选择呢？
  - 在大多数情况下，如果可以动态修改class完成某个功能，更推荐使用动态class

- 如果对于某些情况，无法通过动态修改class（比如精准修改某个css属性的值），那么就可以修改style属性

元素的className和classList

- 元素的class attribute，对应的property并非叫class，而是className：
  - 这是因为JavaScript早期是不允许使用class这种关键字来作为对象的属性，所以DOM规范使用了className
  - 虽然现在JavaScript已经没有这样的限制，但是并不推荐，并且依然在使用className这个名称
- 我们可以对className进行赋值，它会替换整个类中的字符串

```
var boxEl = document.querySelector(".box")
boxEl.className = "why abc"
```

- 如果我们需要添加或者移除单个的class，那么可以使用classList属性
- elem.classList 是一个特殊的对象：
  - elem.classList.add (class)：添加一个类
  - elem.classList.remove(class)：添加/移除类
  - elem.classList.toggle(class)：如果类不存在就添加类，存在就移除它
  - elem.classList.contains(class)：检查给定类，返回 true/false
- classList是可迭代对象，可以通过for of进行遍历

```
<!DOCTYPE html>
<html>
<head>
  <title>classList方法示例</title>
  <style>
    .active {
      background-color: red;
      color: white;
    }
  </style>
</head>
<body>
  <button id="btn">点击我</button>
  <script>
    // 获取按钮元素
    var btn = document.getElementById('btn');

    // 给按钮添加一个类名
    btn.classList.add('btn-primary');
    console.log(btn.classList); // 输出：DOMTokenList ["btn-primary"]

    // 移除按钮的类名
    btn.classList.remove('btn-primary');
    console.log(btn.classList); // 输出：DOMTokenList []
```

```
    // 切换按钮的类名
    btn.classList.toggle('active');
    console.log(btn.classList); // 输出：DOMTokenList ["active"]

    // 检查按钮是否包含指定的类名
    var hasClass = btn.classList.contains('active');
    console.log(hasClass); // 输出：true
  </script>
</body>
</html>
```

元素的style属性

- 如果需要单独修改某一个CSS属性，那么可以通过style来操作：
  - 对于多词（multi-word）属性，使用驼峰式 camelCase

```
boxEl.style.width = "100px"
boxEl.style.height = "50px"
boxEl.style.backgroundColor = "red"
```

- 如果我们将值设置为空字符串，那么会使用CSS的默认样式：

```
boxEl.style.display = ""
```

- 多个样式的写法，我们需要使用cssText属性：
  - 不推荐这种用法，因为它会替换整个字符串；

```
boxEl.style.cssText = `
            width: 100px;
            height: 100px;
            background-color: red;`
```

元素style的读取 - getComputedStyle

> `getComputedStyle()` 是一个用于获取元素计算后样式的 API。它返回一个对象，其中包含了指定元素的所有计算后的样式属性及其对应的值。
>
> `getComputedStyle()` 接受两个参数，第一个参数是要获取样式的元素，第二个参数是伪元素（可选）。伪元素参数可以用来获取指定元素的伪元素计算后的样式，例如获取元素的 `::before` 或 `::after` 伪元素的样式。
>
> `getComputedStyle()` 方法返回的对象是一个只读对象，其中包含了指定元素的所有计算后的样式属性及其对应的值。这些样式属性包括字体、颜色、边框、内边距、外边距、宽度、高度等等。
>
> 用得少，了解就行

- 如果我们需要读取样式：
  - 对于内联样式，是可以通过style.*的方式读取到的；

- 对于style、css文件中的样式，是读取不到的

- 这个时候，我们可以通过getComputedStyle的全局函数来实现：

```
console.log(getComputedStyle(boxEl).width)
console.log(getComputedStyle(boxEl).height)
console.log(getComputedStyle(boxEl).backgroundColor)
```

```
<!DOCTYPE html>
<html>
<head>
  <title>getComputedStyle() 方法示例</title>
  <style>
    #example {
      background-color: red;
      color: white;
      font-size: 20px;
      padding: 10px;
      margin: 5px;
      border: 1px solid black;
      width: 200px;
      height: 100px;
    }
  </style>
</head>
<body>
  <div id="example">Hello, world!</div>
  <script>
    // 获取元素
    var example = document.getElementById('example');

    // 获取元素的计算后样式
    var computedStyle = window.getComputedStyle(example);

    // 获取元素的背景颜色
    var backgroundColor = computedStyle.backgroundColor;
    console.log('背景颜色：' + backgroundColor); // 输出：背景颜色：rgb(255, 0, 0)

    // 获取元素的字体大小
    var fontSize = computedStyle.fontSize;
    console.log('字体大小：' + fontSize); // 输出：字体大小：20px

    // 获取元素的边框宽度
    var borderWidth = computedStyle.borderWidth;
    console.log('边框宽度：' + borderWidth); // 输出：边框宽度：1px
  </script>
</body>
</html>
```

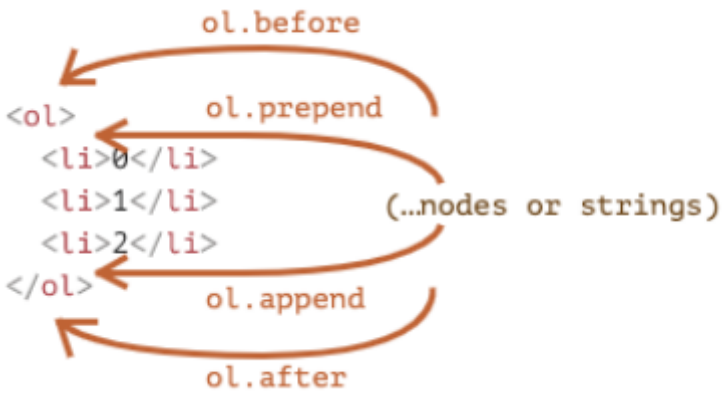
创建元素

- 前面我们使用过 document.write 方法写入一个元素：
  - 这种方式写起来非常便捷，但是对于复杂的内容、元素关系拼接并不方便
  - 它是在早期没有DOM的时候使用的方案，目前依然被保留了下来
- 那么目前我们想要插入一个元素，通常会按照如下步骤：
  - 步骤一：创建一个元素
  - 步骤二：插入元素到DOM的某一个位置
- 创建元素： document.createElement(tag)

```
var boxEl = document.querySelector(".box")
var h2El = document.createElement("h2")
h2El.innerHTML = "我是标题"
h2El.classList.add("title")
boxEl.append(h2El)
```

插入元素

- 插入元素的方式如下：
  - node.append(...nodes or strings) —— 在 node 末尾 插入节点或字符串
  - node.prepend(...nodes or strings) —— 在 node 开头 插入节点或字符串
  - node.before(...nodes or strings) —— 在 node 前面 插入节点或字符串
  - node.after(...nodes or strings) —— 在 node 后面 插入节点或字符串
  - node.replaceWith(...nodes or strings) —— 将 node 替换为给定的节点或字符串



移除和克隆元素

- 移除元素我们可以调用元素本身的remove方法：

```
setTimeout(() => {
  h2El.remove()
}, 2000);
```

- 如果我们想要复制一个现有的元素，可以通过cloneNode方法：

- 可以传入一个Boolean类型的值，来决定是否是深度克隆
- 深度克隆会克隆对应元素的子元素，否则不会

```
var cloneBoxEl = boxEl.cloneNode(true)
document.body.append(cloneBoxEl)
```

旧的元素操作方法

- 在很多地方我们也会看到一些旧的操作方法：
  - parentElem.appendChild(node)：

    ✓ 在parentElem的父元素最后位置添加一个子元素
  - parentElem.insertBefore(node, nextSibling)：

    ✓ 在parentElem的nextSibling前面插入一个子元素
  - parentElem.replaceChild(node, oldChild)：

    ✓ 在parentElem中，新元素替换之前的oldChild元素
  - parentElem.removeChild(node)：

    ✓ 在parentElem中，移除某一个元素

Window

元素的大小、滚动

> client：
>
> - `client`这个词主要是指代"客户端"，即浏览器客户端中的元素。在这种情况下，`clientWidth`表示元素在客户端浏览器中的宽度，包括内容区域和内边距（padding），但不包括边框（border）、外边距（margin）以及滚动条（如果有的话）。
>
> offset：
>
> - 偏移量

- clientWidth：contentWith+padding（不包含滚动条）
- clientHeight：contentHeight+padding
- clientTop：border-top的宽度
- clientLeft：border-left的宽度

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Client Properties Example</title>
  <style>
    /* 设置div元素的样式 */
    .box {
```

```
      width: 200px;
      height: 100px;
      padding: 20px;
      border: 10px solid #333;
      overflow: auto;
    }
  </style>
</head>
<body>
  <div class="box" id="box">
    This is a sample text inside the box.
  </div>
  <script>
    // 获取div元素的引用
    const box = document.getElementById('box');

    // clientWidth: 获取元素的宽度，包括content和padding，但不包括滚动条和border
    const clientWidth = box.clientWidth;
    console.log('clientWidth:', clientWidth); // 输出: clientWidth: 240

    // clientHeight: 获取元素的高度，包括content和padding，但不包括滚动条和border
    const clientHeight = box.clientHeight;
    console.log('clientHeight:', clientHeight); // 输出: clientHeight: 140

    // clientTop: 获取元素的border-top宽度
    const clientTop = box.clientTop;
    console.log('clientTop:', clientTop); // 输出: clientTop: 10

    // clientLeft: 获取元素的border-left宽度
    const clientLeft = box.clientLeft;
    console.log('clientLeft:', clientLeft); // 输出: clientLeft: 10
  </script>
</body>
</html>
```

- **offsetWidth：元素完整的宽度**
- **offsetHeight：元素完整的高度**
- **offsetLeft：距离父元素的x**
- **offsetHeight：距离父元素的y**

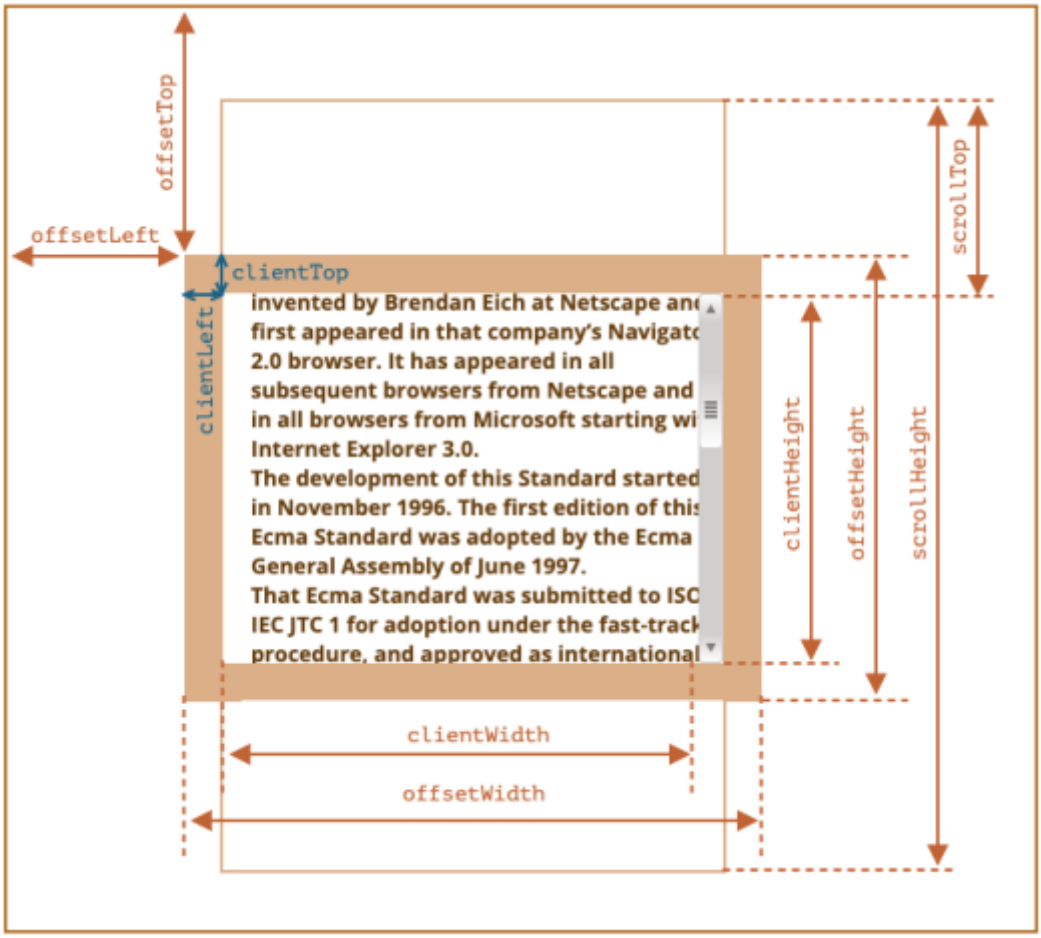
```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Offset Properties Example</title>
  <style>
    /* 设置外层div元素的样式 */
    .container {
```

```
      position: relative;
      width: 400px;
      height: 300px;
      background-color: #eee;
      padding: 10px;
    }

    /* 设置内层div元素的样式 */
    .box {
      position: absolute;
      top: 30px;
      left: 20px;
      width: 200px;
      height: 100px;
      padding: 20px;
      border: 10px solid #333;
      background-color: #fff;
    }
  </style>
</head>
<body>
  <div class="container">
    <div class="box" id="box">
      This is a sample text inside the box.
    </div>
  </div>
  <script>
    // 获取div元素的引用
    const box = document.getElementById('box');

    // offsetWidth: 获取元素的完整宽度，包括content、padding、border，但不包括滚动条
    const offsetWidth = box.offsetWidth;
    console.log('offsetWidth:', offsetWidth); // 输出: offsetWidth: 240

    // offsetHeight: 获取元素的完整高度，包括content、padding、border，但不包括滚动条
    const offsetHeight = box.offsetHeight;
    console.log('offsetHeight:', offsetHeight); // 输出: offsetHeight: 140

    // offsetLeft: 获取元素距离其offsetParent（距离最近的有定位属性的父元素）的左边距
    const offsetLeft = box.offsetLeft;
    console.log('offsetLeft:', offsetLeft); // 输出: offsetLeft: 20

    // offsetTop: 获取元素距离其offsetParent（距离最近的有定位属性的父元素）的上边距
    const offsetTop = box.offsetTop;
    console.log('offsetTop:', offsetTop); // 输出: offsetTop: 30
  </script>
</body>
</html>
```

- **scrollHeight：整个可滚动的区域高度**

- **scrollTop：滚动部分的高度**

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Scroll Properties Example</title>
  <style>
    /* 设置div元素的样式 */
    .box {
      width: 300px;
      height: 200px;
      padding: 20px;
      border: 2px solid #333;
      overflow: auto;
    }
  </style>
</head>
<body>
  <div class="box" id="box">
    <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit. Integer nec odio. Praesent libero. Sed cursus ante dapibus diam. Sed nisi. Nulla quis sem at nibh elementum imperdiet. Duis sagittis ipsum. Praesent mauris. Fusce nec tellus sed augue semper porta. Mauris massa. Vestibulum lacinia arcu eget nulla. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos himenaeos.</p>
  </div>
  <script>
    // 获取div元素的引用
    const box = document.getElementById('box');

    // scrollHeight: 获取元素的整个可滚动区域的高度，包括不可见的部分
    const scrollHeight = box.scrollHeight;
    console.log('scrollHeight:', scrollHeight);

    // 获取元素的滚动部分的高度（即滚动条向下滚动的距离）
    box.addEventListener('scroll', () => {
      const scrollTop = box.scrollTop;
      console.log('scrollTop:', scrollTop);
    });
  </script>
</body>
</html>
```



- window的width和height
  - innerWidth、innerHeight：获取window窗口的宽度和高度（包含滚动条）
  - outerWidth、outerHeight：获取window窗口的整个宽度和高度（包括调试工具、工具栏）
  - documentElement.clientHeight、documentElement.clientWidth：获取html的宽度和高度（不包含滚动条）
- window的滚动位置：
  - scrollX：X轴滚动的位置（别名pageXOffset）
  - scrollY：Y轴滚动的位置（别名pageYOffset）
- 也有提供对应的滚动方法：
  - 方法 scrollBy(x,y)：将页面滚动至 相对于当前位置的 (x, y) 位置
  - 方法 scrollTo(pageX,pageY) 将页面滚动至 绝对坐标

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Smooth Scroll Example</title>
  <style>
    /* 设置div元素的样式，固定宽高，内容溢出时出现滚动条 */
```

```
    .box {
      width: 300px;
      height: 200px;
      padding: 20px;
      border: 2px solid #333;
      overflow: auto;
    }

    /* 设置按钮的样式 */
    .scroll-btn {
      display: inline-block;
      margin-top: 10px;
      padding: 5px 10px;
      background-color: #f1c40f;
      color: #333;
      cursor: pointer;
    }
  </style>
</head>
<body>
  <!-- 创建一个带有滚动条的div元素 -->
  <div class="box" id="box">
    <p>Lorem ipsum ...</p>
  </div>
  <!-- 创建一个按钮，用于触发滚动操作 -->
  <button class="scroll-btn" id="scrollBtn">Smooth Scroll Down 100px</button>

  <script>
    // 获取div元素和按钮的引用
    const box = document.getElementById('box');
    const scrollBtn = document.getElementById('scrollBtn');
    // 定义滚动距离和动画持续时间
    const scrollDistance = 100;
    const animationDuration = 300; // 动画持续时间（毫秒）

    // 给按钮添加点击事件监听器，触发平滑滚动函数
    scrollBtn.addEventListener('click', () => {
      smoothScrollDown(box, scrollDistance, animationDuration);
    });

    // 平滑滚动函数，接受3个参数：滚动的元素，滚动距离，动画持续时间
    function smoothScrollDown(element, distance, duration) {
      // 获取元素当前的滚动距离和动画开始时间
      const startScrollTop = element.scrollTop;
      const startTime = performance.now();

      // 动画滚动函数
      function animateScroll(currentTime) {
        // 计算动画已经执行的时间和进度
        const elapsedTime = currentTime - startTime;
        const progress = Math.min(elapsedTime / duration, 1); // 计算动画进度
（0-1）
```

```
        // 使用easeInOutQuad缓动函数计算当前进度下的缓动值
        const easeProgress = progress < 0.5 ? 2 * progress * progress : 1 -
Math.pow(-2 * progress + 2, 2) / 2;

        // 计算当前应该滚动到的位置
        const currentScroll = startScrollTop + easeProgress * distance;

        // 更新元素的滚动位置
        element.scrollTop = currentScroll;

        // 如果动画未完成，继续执行动画
        if (progress < 1) {
          requestAnimationFrame(animateScroll);
        }
      }

      // 启动动画
      requestAnimationFrame(animateScroll);
    }
  </script>
</body>
</html>
```

24小时倒计时案例

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>24 Hours Countdown</title>
  <style>
    body {
      display: flex;
      justify-content: center;
      align-items: center;
      min-height: 100vh;
      background-color: #f5f5f5;
      font-family: Arial, sans-serif;
    }

    .countdown {
      display: flex;
      background-color: #333;
      padding: 20px;
      border-radius: 5px;
      box-shadow: 0 0 10px rgba(0, 0, 0, 0.2);
    }

    .countdown span {
```

```
      display: inline-block;
      margin: 0 10px;
      padding: 10px 20px;
      font-size: 24px;
      font-weight: bold;
      color: #f1c40f;
      background-color: #222;
      border-radius: 3px;
      animation: blink 1s infinite;
    }

    @keyframes blink {
      0%, 100% { opacity: 1; }
      50% { opacity: 0.5; }
    }
  </style>
</head>
<body>
  <div class="countdown">
    <span id="hours">00</span>
    <span id="minutes">00</span>
    <span id="seconds">00</span>
  </div>
  <script>
    // 获取显示倒计时的元素
    const hoursElement = document.getElementById('hours');
    const minutesElement = document.getElementById('minutes');
    const secondsElement = document.getElementById('seconds');

    // 设置倒计时的结束时间
    const endTime = new Date().getTime() + 24 * 60 * 60 * 1000;

    // 更新倒计时显示的函数
    function updateCountdown() {
      // 计算剩余时间
      const currentTime = new Date().getTime();
      const remainingTime = endTime - currentTime;

      // 将剩余时间转换为小时、分钟和秒
      const hours = Math.floor(remainingTime / (1000 * 60 * 60));
      const minutes = Math.floor((remainingTime % (1000 * 60 * 60)) / (1000 *
60));
      const seconds = Math.floor((remainingTime % (1000 * 60)) / 1000);

      // 更新倒计时显示
      hoursElement.textContent = hours.toString().padStart(2, '0');
      minutesElement.textContent = minutes.toString().padStart(2, '0');
      secondsElement.textContent = seconds.toString().padStart(2, '0');
    }

    // 立即更新倒计时，确保页面加载时即可看到正确的倒计时
    updateCountdown();
```

```
    // 每秒更新一次倒计时
    setInterval(updateCountdown, 1000);
  </script>
</body>
</html>
```

事件监听

认识事件（Event）

- Web页面需要经常和用户之间进行交互，而交互的过程中我们可能想要捕捉这个交互的过程：
  - 比如用户点击了某个按钮、用户在输入框里面输入了某个文本、用户鼠标经过了某个位置
  - 浏览器需要搭建一条JavaScript代码和事件之间的桥梁
  - 当某个事件发生时，让JavaScript可以相应（执行某个函数），所以我们需要针对事件编写处理程序（handler）
- 如何进行事件监听呢?
  - 事件监听方式一：在script中直接监听（很少使用）

```
  <button id="myButton">Click me!</button>

  <script>
    document.getElementById('myButton').onclick = function() {
      alert('Button clicked (Method 1)');
    }
  </script>
```

  - 事件监听方式二：DOM属性，通过元素的on来监听事件

```
  <script>
    function buttonClicked() {
      alert('Button clicked (Method 2)');
    }
  </script>
</head>
<body>
  <button id="myButton" onclick="buttonClicked()">Click me!</button>
</body>
```

  - 事件监听方式三：通过EventTarget中的addEventListener来监听

```
  <script>
    function buttonClicked() {
      alert('Button clicked (Method 3)');
    }

    document.addEventListener('DOMContentLoaded', function() {
      document.getElementById('myButton').addEventListener('click',
buttonClicked);
    });
  </script>
</head>
<body>
  <button id="myButton">Click me!</button>
</body>
```

常见的事件列表

- 鼠标事件：
  - click —— 当鼠标点击一个元素时（触摸屏设备会在点击时生成）
  - mouseover / mouseout —— 当鼠标指针移入/离开一个元素时
  - mousedown / mouseup —— 当在元素上按下/释放鼠标按钮时
  - mousemove —— 当鼠标移动时
- 键盘事件：
  - keydown 和 keyup —— 当按下和松开一个按键时
- 表单（form）元素事件：
  - submit —— 当访问者提交了一个
    时
  - focus —— 当访问者聚焦于一个元素时，例如聚焦于一个
- Document 事件：
  - DOMContentLoaded —— 当 HTML 的加载和处理均完成，DOM 被完全构建完成时
- CSS 事件：
  - transitionend —— 当一个 CSS 动画完成时

认识事件流

- 事实上对于事件有一个概念叫做事件流，为什么会产生事件流呢?
  - 我们可以想到一个问题：当我们在浏览器上对着一个元素点击时，你点击的不仅仅是这个元素本身
  - 这是因为我们的HTML元素是存在父子元素叠加层级的
  - 比如一个span元素是放在div元素上的，div元素是放在body元素上的，body元素是放在html元素上的

```html
<div class="box">
  <span class="word">哈哈哈哈</span>
</div>
```

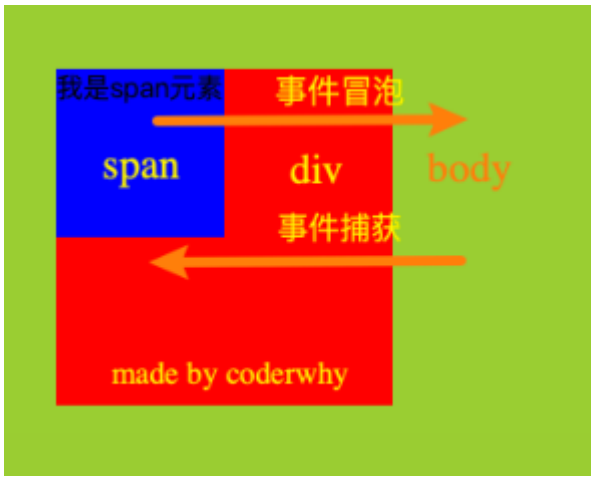
```js
// 1.获取元素
var spanEl = document.querySelector(".word")
var divEl = document.querySelector(".box")
var bodyEl = document.body

// 2.添加监听
spanEl.addEventListener("click", function() {
  console.log("span被点击~")
})
divEl.addEventListener("click", function() {
  console.log("div被点击~")
})
bodyEl.addEventListener("click", function() {
  console.log("body被点击~")
})
```

事件冒泡和事件捕获

> 事件冒泡和事件捕获是浏览器中的两种事件传播机制。当一个事件（例如点击、鼠标移入等）触发时，浏览器会将这个事件按照一定的顺序传递给页面中的元素，以便对事件进行处理。下面我们详细介绍事件冒泡和事件捕获的过程。
>
> 1. 事件捕获（Event Capturing）： 事件捕获是从最外层的祖先元素开始，逐层向内（从上到下）传递直至触发事件的目标元素。事件捕获的目的是在事件到达目标元素之前，提前捕获到这个事件，从而可以在事件到达目标元素之前进行一些处理。需要注意的是，并非所有事件都支持捕获，例如：`mouseenter`、`mouseleave` 和 `focus` 等事件不支持捕获。
> 2. 事件冒泡（Event Bubbling）： 与事件捕获相反，事件冒泡是从触发事件的目标元素开始，逐层向外（从下到上）传递直至最外层的祖先元素。事件冒泡的目的是让父元素或祖先元素能够对子元素的事件作出反应。大多数事件都支持冒泡，例如：`click`、`mousedown` 和 `mouseup` 等。

- **我们会发现默认情况下事件是从最内层的span向外依次传递的顺序，这个顺序我们称之为事件冒泡（Event Bubble）**
- **事实上，还有另外一种监听事件流的方式就是从外层到内层（body -> span），这种称之为事件捕获（Event Capture）**
- **为什么会产生两种不同的处理流呢?**
  - **这是因为早期浏览器开发时，不管是IE还是Netscape公司都发现了这个问题**
  - **但是他们采用了完全相反的事件流来对事件进行了传递**
  - **IE采用了事件冒泡的方式，Netscape采用了事件捕获的方式**
- **那么我们如何去监听事件捕获的过程呢?**
  - **在第三个参数传true，表明我们采用事件监听而不是事件捕获**

```
spanEl.addEventListener("click", function() {
  console.log("span被点击~")
}, true)
divEl.addEventListener("click", function() {
  console.log("div被点击~")
}, true)
bodyEl.addEventListener("click", function() {
  console.log("body被点击~")
}, true)
```



事件捕获和冒泡的过程

> 事件捕获和冒泡是浏览器事件处理的两个阶段。当你在一个元素上触发某个事件（如点击）时，浏览器会首先在捕获阶段从根节点向目标元素传播事件，然后在冒泡阶段从目标元素回到根节点。这种机制被称为事件流。

- 如果我们都监听，那么会按照如下顺序来执行：(简略)
  - 捕获阶段（Capturing phase）：
    - 事件（从 Window）向下走近元素。
  - 目标阶段（Target phase）：
    - 事件到达目标元素。
  - 冒泡阶段（Bubbling phase）：
    - 事件从元素上开始冒泡。

> 详细版本：
>
> 1. 事件捕获阶段（Event Capturing Phase）：事件捕获阶段从根节点（通常是`document`对象）开始，沿着DOM树向下传播，通过每一个祖先节点，一直到达目标元素。在这个阶段，可以在捕获阶段的祖先节点上注册事件处理程序，以在事件达到目标元素之前就对其进行处理。然而，大多数情况下我们并不在捕获阶段处理事件，因为在实际应用中，这种用途并不常见。
> 2. 目标阶段（Target Phase）：当事件到达目标元素，就进入了目标阶段。在这个阶段，会触发在目标元素上注册的事件处理程序。

> 3. 事件冒泡阶段（Event Bubbling Phase）：事件冒泡阶段从目标元素开始，沿着DOM树向上传播，通过每一个祖先节点，直到根节点。在这个阶段，可以在冒泡阶段的祖先节点上注册事件处理程序，这样当事件从目标元素冒泡上来时就可以进行处理。这种处理事件的方式更常见，因为冒泡阶段提供了在更高级别（如`document`对象）处理事件的机会，这样就可以用少量的事件处理程序来处理大量的事件，这种技术叫做事件委托或事件代理。
>
> 注意，有些事件（例如`focus`和`blur`）不冒泡，但是它们仍然经历了捕获和目标阶段。另外，你也可以使用`event.stopPropagation()`来阻止事件的进一步冒泡或捕获，以防止事件处理程序被过度触发

- 事实上，我们可以通过event对象来获取当前的阶段：
  - eventPhase
- 开发中通常会使用事件冒泡，所以事件捕获了解即可

事件对象

> 当你在 JavaScript 中使用事件时，你会发现事件处理函数中通常会有一个参数，我们通常命名为`event`或`e`，这就是事件对象。事件对象是一个包含了与发生的事件相关的信息和方法的对象。

- 当一个事件发生时，就会有和这个事件相关的很多信息：
  - 比如事件的类型是什么，你点击的是哪一个元素，点击的位置是哪里等等相关的信息

- 那么这些信息会被封装到一个Event对象中，这个对象由浏览器创建，称之为event对象
- 该对象给我们提供了想要的一些属性，以及可以通过该对象进行某些操作

- 如何获取这个event对象呢?
  - event对象会在传入的事件处理（event handler）函数回调时，被系统传入
  - 我们可以在回调函数中拿到这个event对象

```
spanEl.onclick = function(event) {
  console.log("事件对象:", event)
}
spanEl.addEventListener("click", function(event) {
  console.log("事件对象:", event)
})
```

event常见的属性和方法

> 如果我们的函数有多个参数，那么 `event` 对象应当是最后一个参数。例如：
>
> ```
> element.addEventListener('click', function(arg1, arg2, event) {
> console.log(arg1);  // 打印第一个参数
> console.log(arg2);  // 打印第二个参数
> console.log(event.target);  // 打印点击的元素
> });
> ```
>
> 在这个例子中，`event` 对象是作为函数的第三个参数传入的。但请注意，除非你有明确的理由使用多个参数，通常我们都会将 `event` 作为事件处理函数的第一个（也是唯一的）参数。

属性：

- type：事件的类型，例如"click"，"mouseover"等。
- target：触发事件的最深层元素，也就是真正的目标元素。
- currentTarget：绑定事件处理函数的元素，也就是正在处理事件的元素。
- eventPhase：事件当前所处的阶段，1表示捕获阶段，2表示处于目标阶段，3表示冒泡阶段。
- offsetX/offsetY：事件相对于事件目标的内部坐标位置，即相对于触发元素内部的X和Y坐标。
- clientX/clientY：事件相对于浏览器窗口可视区（viewport）的X和Y坐标。
- pageX/pageY：事件相对于整个文档的X和Y坐标。
- screenX/screenY：事件相对于电脑屏幕左上角的X和Y坐标。

target和currentTarget的区别

> `target` 和 `currentTarget` 是 JavaScript 中事件对象的两个属性，它们在事件处理中表示不同的元素：
>
> - target：`target` 属性返回触发事件的元素，也就是事件最初发生的地方，这个元素被称为目标元素或源元素。例如，如果你在一个按钮元素上点击，那么按钮就是事件的 `target`。

> - currentTarget：`currentTarget` 属性返回其事件监听器被触发的元素。换句话说，`currentTarget` 是绑定事件处理器的元素。例如，如果你在一个按钮元素（这个按钮元素在一个 div 元素内部）上点击，而事件监听器是绑定在这个 div 上的，那么按钮是 `target`，div 是 `currentTarget`。
>
> 在事件冒泡或者事件捕获的过程中，`target` 是固定不变的，始终是事件最初发生的地方，而 `currentTarget` 是随着事件的传递而改变的。如果在事件处理函数中没有使用到 `event.currentTarget`，那么通常可以不必关注它。但是在处理一些复杂的事件逻辑，尤其是涉及到事件代理的时候，`currentTarget` 就非常有用了。

方法：

- **preventDefault()：阻止事件的默认行为。例如，阻止链接的默认点击行为，即阻止链接打开一个新页面。**
- **stopPropagation()：阻止事件的进一步传播，包括冒泡和捕获。例如，在嵌套元素中阻止点击事件冒泡到父元素。**

> 这个例子创建了一个父元素和一个子元素，并在父元素上添加了一个点击事件监听器。当你点击子元素或父元素时，监听器将打印出关于事件的各种信息。
>
> 注意，当你点击子元素时，虽然子元素上的监听器会阻止事件冒泡，但是父元素的监听器仍然会被触发。这是因为，当一个事件发生时，浏览器首先在捕获阶段将事件传递到最深层的目标元素，然后在冒泡阶段将事件传回去。虽然 `stopPropagation` 阻止了事件的冒泡，但是在捕获阶段，事件仍然会到达父元素。

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Event Object Example</title>
  <style>
    .container {
      width: 300px;
      height: 300px;
      background-color: #eee;
      position: relative;
    }
    .box {
      width: 100px;
      height: 100px;
      background-color: #f66;
      position: absolute;
      left: 50px;
      top: 50px;
    }
  </style>
</head>
<body>
  <div class="container" id="container">
    <div class="box" id="box"></div>
  </div>
```

```
  <script>
    const container = document.getElementById('container');
    const box = document.getElementById('box');

    container.addEventListener('click', function(event) {
      // 打印事件类型
      console.log('Event Type: ', event.type); // "click"

      // 打印触发事件的元素和处理事件的元素
      console.log('Target Element: ', event.target); // box or container，取决于点击的元素
      console.log('Current Target Element: ', event.currentTarget); // container

      // 打印事件所处阶段
      console.log('Event Phase: ', event.eventPhase); // 2 or 3，取决于点击的元素

      // 打印相对于事件目标的坐标和相对于视窗、文档和屏幕的坐标
      console.log('OffsetX: ', event.offsetX, 'OffsetY: ', event.offsetY);
      console.log('ClientX: ', event.clientX, 'ClientY: ', event.clientY);
      console.log('PageX: ', event.pageX, 'PageY: ', event.pageY);
      console.log('ScreenX: ', event.screenX, 'ScreenY: ', event.screenY);
    });

    // 阻止默认行为和事件传播
    box.addEventListener('click', function(event) {
      event.preventDefault(); // 在此例中没有明显效果，因为 div 的点击没有默认行为
      event.stopPropagation(); // 阻止事件冒泡到父元素
    });
  </script>
</body>
</html>
```

事件处理中函数的this

- **在函数中，我们也可以通过this来获取当前的发生元素：**

```
boxEl.addEventListener("click", function(event) {
  console.log(this === event.target) // true
})
```

- **这是因为在浏览器内部，调用event handler是绑定到当前的target上的，后续JS高级笔记中有详细讲解this的笔记**

EventTarget类

> **在JavaScript中，EventTarget是一个表示可以接收和处理事件的对象类。它是一个抽象类，通常被其他类继承，例如DOM中的Element、Document和Window对象。**
>
> **EventTarget类提供了以下三个主要方法：**

1. `addEventListener(type, listener, options)`：用于向对象添加事件监听器。参数`type`指定事件类型，`listener`是一个回调函数，`options`是一个可选的配置对象，用于指定事件监听的一些选项，例如是否使用捕获模式。
2. `removeEventListener(type, listener, options)`：用于从对象中移除事件监听器。参数和用法与`addEventListener`方法相同。移除事件监听器时，需要提供与添加时相同的类型、回调函数和选项。
3. `dispatchEvent(event)`：用于触发指定类型的事件。参数`event`是一个事件对象，它包含了要触发的事件类型和相关的数据。当调用`dispatchEvent`方法时，会将事件传递给事件目标对象，然后根据事件的冒泡或捕获机制，在DOM树中进行传播，并依次触发相应的事件监听器。

EventTarget类的实例可以是任何可以接收和处理事件的对象，例如DOM元素、XMLHttpRequest对象、Worker对象等。通过使用EventTarget类的方法，我们可以向对象添加事件监听器，移除事件监听器，以及手动触发事件。这使得我们能够在JavaScript中处理用户交互、网络请求和其他事件驱动的操作

- 我们会发现，所有的节点、元素都继承自EventTarget
  - 事实上Window也继承自EventTarget
- 那么这个EventTarget是什么呢?
  - EventTarget是一个DOM接口，主要用于添加、删除、派发Event事件
- EventTarget常见的方法：
  - addEventListener：注册某个事件类型以及事件处理函数
  - removeEventListener：移除某个事件类型以及事件处理函数
  - dispatchEvent：派发某个事件类型到EventTarget上

```
// 创建一个自定义的EventTarget实例
const myEventTarget = new EventTarget();

// 定义事件处理函数
function handleEvent(event) {
  console.log(`事件处理函数被触发：${event.type}`);
}

// 注册事件监听器
myEventTarget.addEventListener('click', handleEvent);
// 通过addEventListener方法，将'click'事件类型和handleEvent函数关联起来。
// 当myEventTarget对象触发'click'事件时，handleEvent函数将被调用。

// 派发自定义事件
const myEvent = new Event('click');
myEventTarget.dispatchEvent(myEvent);
// 通过dispatchEvent方法，派发一个名为'click'的自定义事件到myEventTarget对象。
// 这会触发之前注册的事件监听器，并执行handleEvent函数。
// 控制台输出：事件处理函数被触发：click

// 移除事件监听器
myEventTarget.removeEventListener('click', handleEvent);
```

```
// 通过removeEventListener方法，移除之前注册的事件监听器。
// 这样，当myEventTarget对象触发'click'事件时，handleEvent函数不再被调用。

// 再次派发事件，但由于监听器已被移除，不会触发事件处理函数
myEventTarget.dispatchEvent(myEvent);
// 尝试再次派发'click'事件，但由于之前的事件监听器已被移除，
// 所以不会执行handleEvent函数。
```

事件委托（event delegation）

> **事件委托（Event Delegation）是一种在JavaScript中常用的事件处理技术，它允许我们将事件监听器绑定到一个父元素上，以处理其子元素触发的事件。这种技术的核心思想是利用事件的冒泡机制，将事件处理的责任委托给父元素，从而减少了在子元素上添加事件监听器的数量，提高了性能和代码的简洁性。**
>
> **事件委托的工作原理如下：**
>
> 1. **选择一个共同的父元素，该父元素包含我们感兴趣的子元素。**
> 2. **将事件监听器绑定到父元素上，监听父元素上发生的事件。**
> 3. **当子元素触发该事件时，事件会向上冒泡到父元素。**
> 4. **父元素的事件监听器捕获到事件，并执行相应的处理逻辑。**
>
> **通过使用事件委托，我们可以实现以下优势：**
>
> 1. **动态添加和删除子元素：由于事件监听器绑定在父元素上，当添加或删除子元素时，无需手动重新绑定事件监听器。**
> 2. **减少事件监听器的数量：只需在父元素上添加一个事件监听器，而不是在每个子元素上都添加，从而减少了事件处理的代码量。**
> 3. **提高性能：减少了事件监听器的数量，可以减少内存占用和事件处理的开销。**
> 4. **处理动态生成的元素：对于通过JavaScript动态生成的元素，事件委托可以方便地处理它们的事件。**

- **事件冒泡在某种情况下可以帮助我们实现强大的事件处理模式 - 事件委托模式（也是一种设计模式）**
- **那么这个模式是怎么样的呢?**
  - **因为当子元素被点击时，父元素可以通过冒泡可以监听到子元素的点击**
  - **并且可以通过event.target获取到当前监听的元素**
- **案例：一个ul中存放多个li，点击某一个li会变成红色**
  - **方案一：监听每一个li的点击，并且做出相应**
  - **方案二：在ul中监听点击，并且通过event.target拿到对应的li进行处理**

  ✓ 因为这种方案并不需要遍历后给每一个li上添加事件监听，所以它更加高效；

```
var listEl = document.querySelector(".list")
var currentActive = null
listEl.addEventListener("click", function(event) {
  if (currentActive) currentActive.classList.remove("active")
  event.target.classList.add("active")
  currentActive = event.target
})
```

```
<ul id="myList">
  <li>Item 1</li>
  <li>Item 2</li>
  <li>Item 3</li>
</ul>
```

```
const myList = document.getElementById('myList');

// 在父元素上添加事件监听器
myList.addEventListener('click', function(event) {
  // 检查事件目标是否为子元素
  if (event.target.tagName === 'LI') {
    // 子元素被点击，执行处理逻辑
    console.log(`点击了子元素：${event.target.textContent}`);
  }
});
```

- **在上述示例中，我们将事件监听器绑定到父元素`myList`上。当列表中的任何一个`<li>`元素被点击时，事件会冒泡到父元素，并被父元素的事件监听器捕获到。然后，我们可以通过检查事件的目标元素(`event.target`)来确定是哪个子元素被点击，并执行相应的处理逻辑。**

事件委托的标记

- **某些事件委托可能需要对具体的子组件进行区分，这个时候我们可以使用data-*对其进行标记：**
- **比如多个按钮的点击，区分点击了哪一个按钮：**

```
<div class="btn-list">
  <button data-action="new">新建</button>
  <button data-action="search">搜索</button>
  <button data-action="delete">删除</button>
</div>
```

```
var btnListEl = document.querySelector(".btn-list")
btnListEl.addEventListener("click", function(event) {
  var action = event.target.dataset.action
  switch (action) {
    case "new":
      console.log("点击了新建~")
      break
    case "search":
      console.log("点击了搜索~")
      break
    case "delete":
      console.log("点击了删除~")
      break
    default:
      console.log("位置action")
  }
})
```

常见的鼠标事件

- **接下来我们来看一下常见的鼠标事件（不仅仅是鼠标设备，也包括模拟鼠标的设备，比如手机、平板电脑）**
- **常见的鼠标事件：**

属性	描述
click	当用户点击某个对象时调用的事件句柄。
contextmenu	在用户点击鼠标右键打开上下文菜单时触发
dblclick	当用户双击某个对象时调用的事件句柄。
mousedown	鼠标按钮被按下。
mouseup	鼠标按键被松开。
mouseover	鼠标移到某元素之上。（支持冒泡）
mouseout	鼠标从某元素移开。（支持冒泡）
mouseenter	当鼠标指针移动到元素上时触发。（不支持冒泡）
mouseleave	当鼠标指针移出元素时触发。（不支持冒泡）
mousemove	鼠标被移动。

mouseover和mouseenter的区别

`mouseover` 和 `mouseenter` 都是 JavaScript 事件，用于检测鼠标指针进入一个元素的边界。然而，它们之间有一些关键区别：

1. 事件冒泡（Event Bubbling）：

`mouseover` 事件支持事件冒泡，这意味着当鼠标指针移动到一个元素上时，`mouseover` 事件会从该元素一直向上触发，直到到达 DOM 树的根节点。在这个过程中，所有的父元素都会收到这个事件。

相反，`mouseenter` 事件不支持事件冒泡。当鼠标指针进入一个元素时，只有该元素会触发 `mouseenter` 事件，而它的父元素不会收到此事件。

1. 频率：

由于 `mouseover` 事件在整个 DOM 树中冒泡，因此它可能会比 `mouseenter` 事件触发得更频繁。当鼠标在子元素与父元素之间移动时，`mouseover` 事件可能在父元素和子元素之间来回触发，而 `mouseenter` 仅在进入每个元素时触发一次。

简而言之，`mouseover` 和 `mouseenter` 的主要区别在于事件冒泡。在选择使用它们时，我们需要考虑实际需求。如果希望在整个 DOM 树中监听鼠标进入事件，那么 `mouseover` 可能更适合。然而，如果只关心特定元素的鼠标进入事件，那么 `mouseenter` 可能是更好的选择，因为它不会触发额外的事件冒泡。

常见的键盘事件

> 在JavaScript中，有许多与键盘事件相关的事件。这些事件能够帮助我们检测用户与键盘的交互，并根据需要执行特定的操作。以下是一些常见的键盘事件：
>
> 1. `keydown`：当用户按下一个键时，触发此事件。`keydown` 事件在按键被按下时立即触发，无论是否释放按键。如果按住按键不放，`keydown` 事件将重复触发。
> 2. `keyup`：当用户释放一个按下的键时，触发此事件。`keyup` 事件在按键被释放后触发。
> 3. `keypress`：当用户按下一个字符键时，触发此事件。`keypress` 事件在按键被按下并产生字符时触发，这意味着仅当按下的键实际上可以产生字符时（例如字母、数字、标点符号等）才会触发。特殊键（如功能键、方向键、Shift、Ctrl、Alt等）不会触发 `keypress` 事件。需要注意的是，`keypress` 事件在现代浏览器中已被废弃，推荐使用 `keydown` 代替。
>
> 在处理这些事件时，事件对象会包含一些有用的属性，例如：
>
> - `key`：表示按下的键的值。对于字符键，`key` 的值将是按键对应的字符；对于特殊键，将是一个代表按键的字符串，例如"Shift"、"Control"、"ArrowUp"等。
> - `code`：表示按下的键的物理键码。这个值在任何键盘布局下都是相同的。
> - `altKey`、`ctrlKey`、`shiftKey`、`metaKey`：布尔值，表示按下时是否同时按下了Alt、Ctrl、Shift或Meta（例如，在Mac上的Command键）键。

- 常见的键盘事件：

属性	描述
onkeydown	某个键盘按键被按下。
onkeypress	某个键盘按键被按下。
onkeyup	某个键盘按键被松开。

- 事件的执行顺序是 onkeydown、onkeypress、onkeyup
  - down事件先发生
  - press发生在文本被输入
  - up发生在文本输入完成
- 我们可以通过key和code来区分按下的键：

- code：“按键代码”（"KeyA"，"ArrowLeft" 等），特定于键盘上按键的物理位置
- key：字符（"A"，"a" 等），对于非字符（non-character）的按键，通常具有与 code 相同的值。）

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Keyboard Events</title>
</head>
<body>
  <input type="text" id="input-field">
  <script>
    const inputField = document.querySelector('#input-field');

    inputField.addEventListener('keydown', (event) => {
      console.log('keydown:', event.key);
    });

    inputField.addEventListener('keyup', (event) => {
      console.log('keyup:', event.key);
    });

    // 不推荐使用，仅作为示例
    inputField.addEventListener('keypress', (event) => {
      console.log('keypress:', event.key);
    });
  </script>
</body>
</html>
```

常见的表单事件

> 在HTML中，表单事件通常与表单元素（如`<input>`、`<select>`和`<textarea>`）一起使用。这些事件用于处理用户与表单交互，例如输入数据、选择选项或提交表单。

- 针对表单也有常见的事件：

属性	描述
onchange	该事件在表单元素的内容改变时触发(<input>, <keygen>, <select>, 和 <textarea>)
oninput	元素获取用户输入时触发
onfocus	元素获取焦点时触发
onblur	元素失去焦点时触发
onreset	表单重置时触发
onsubmit	表单提交时触发

1. `change`：当表单元素的值发生改变并失去焦点时，触发`change`事件。这个事件在`<input>`、`<select>`和`<textarea>`元素上都适用。

   示例：

   ```
   codeconst inputElement = document.querySelector('input');
   inputElement.addEventListener('change', (event) => {
     console.log('Input value changed to:', event.target.value);
   });
   ```

2. `input`：当表单元素的值发生改变时，触发`input`事件。与`change`事件不同，`input`事件在元素值改变时立即触发，而不是在失去焦点时。这个事件在`<input>`、`<select>`和`<textarea>`元素上都适用。

   示例：

   ```
   codeconst inputElement = document.querySelector('input');
   inputElement.addEventListener('input', (event) => {
     console.log('Input value changed to:', event.target.value);
   });
   ```

3. `focus`：当表单元素获得焦点时，触发`focus`事件。这个事件在`<input>`、`<select>`和`<textarea>`元素上都适用。

   示例：

   ```
   codeconst inputElement = document.querySelector('input');
   inputElement.addEventListener('focus', () => {
     console.log('Input element is focused');
   });
   ```

4. `blur`：当表单元素失去焦点时，触发`blur`事件。这个事件在`<input>`、`<select>`和`<textarea>`元素上都适用。

   示例：

   ```
   codeconst inputElement = document.querySelector('input');
   inputElement.addEventListener('blur', () => {
     console.log('Input element lost focus');
   });
   ```

5. `submit`：当表单被提交时，触发`submit`事件。这个事件仅适用于`<form>`元素。通常，我们使用这个事件来验证表单数据并阻止默认的提交行为，这样我们就可以使用JavaScript来处理表单数据，例如发送Ajax请求。

   示例：

```
codeconst formElement = document.querySelector('form');
formElement.addEventListener('submit', (event) => {
  event.preventDefault(); // 阻止表单的默认提交行为
  console.log('Form submitted');
  // 在此处处理表单数据
});
```

这些表单事件使得我们可以轻松地响应用户在表单中的交互，例如实时验证用户输入、显示自定义错误消息或处理表单数据。

文档加载事件

文档加载事件是浏览器在加载和解析HTML文档时触发的事件。这些事件通常用于在文档加载完成后执行一些操作，例如初始化组件、绑定事件监听器或执行其他需要等待DOM可用的任务。

以下是两个重要的文档加载事件：

1. `DOMContentLoaded`：**当HTML文档被完全加载和解析后，浏览器会触发`DOMContentLoaded`事件。此时，DOM已经构建完毕，但外部资源（如图像、样式表和其他脚本文件）可能仍在加载。这个事件对于在文档可用时立即执行代码非常有用，而无需等待所有外部资源加载完成。**

   示例：

```
document.addEventListener('DOMContentLoaded', () => {
  console.log('DOM fully loaded and parsed');
  // 在此处执行需要在DOM可用时执行的代码
});
```

2. `load`：**当整个页面及其所有相关资源（如图像、样式表和脚本文件）都已加载完成后，`window`对象上的`load`事件会被触发。通常，这个事件在`DOMContentLoaded`事件之后发生。这意味着页面上所有的内容和资源都已加载，可以进行更复杂的操作，如处理图像或依赖于其他脚本的功能。**

   示例：

```
window.addEventListener('load', () => {
  console.log('Page fully loaded');
  // 在此处执行需要在所有资源加载完成后执行的代码
});
```

在大多数情况下，`DOMContentLoaded`事件足以满足我们的需求，因为它能确保DOM已经构建完成，我们可以安全地访问和操作它。但是，在需要等待外部资源（如图像或其他脚本文件）加载完成的情况下，使用`load`事件可能更合适。

```html
<div>哈哈哈</div>
<img src="https://ossweb-img.qq.com/upload/webplat/info/yxzj/20220510/28345686899485.jpg" alt="">

<script>
  window.addEventListener("DOMContentLoaded", function() {
    var imgEl = document.querySelector("img")
    console.log("页面内容加载完毕", imgEl.offsetWidth, imgEl.offsetHeight)
  })
  window.addEventListener("load", function() {
    var imgEl = document.querySelector("img")
    console.log("页面所有内容加载完毕", imgEl.offsetWidth, imgEl.offsetHeight)
  })
</script>
```

- **事件类型：https://developer.mozilla.org/zh-CN/docs/Web/Events**

window定时器方法

> **这部分应该要属于BOM部分的，但是接下来的轮播图案例会用到，所以放到了这里**

- **有时我们并不想立即执行一个函数，而是等待特定一段时间之后再执行，我们称之为“计划调用（scheduling a call）”**
- **目前有两种方式可以实现：**
  - **setTimeout 允许我们将函数推迟到一段时间间隔之后再执行**
  - **setInterval 允许我们重复运行一个函数，从一段时间间隔之后开始运行，之后以该时间间隔连续重复运行该函数**
- **并且通常情况下有提供对应的取消方法：**
  - **clearTimeout：取消setTimeout的定时器**
  - **clearInterval：取消setInterval的定时器**
- **大多数运行环境都有内置的调度程序，并且提供了这些方法：**
  - **目前来讲，所有浏览器以及 Node.js 都支持这两个方法**
  - **所以我们后续学习Node的时候，也可以在Node中使用它们**

setTimeout的使用

> **`setTimeout`是JavaScript中的一个全局函数，它允许我们在指定的延迟时间后执行某个函数。这个函数在Web浏览器和Node.js环境中都可以使用。`setTimeout`的工作原理是基于事件循环和任务队列，它将指定的函数推迟到指定的时间后执行。需要注意的是，`setTimeout`并不能保证精确执行时间，因为它仅在任务队列中排定任务，而任务的实际执行时间取决于其他任务的完成情况。**

`setTimeout`的基本用法如下：

```
setTimeout(function, delay, arg1, arg2, ...);
```

参数说明：

- **`function`：在指定延迟后执行的函数。**

- `delay`：以毫秒为单位的延迟时间。注意，最小延迟时间通常为4毫秒，但这可能因浏览器或环境而异。
- `arg1, arg2, ...`：（可选）传递给执行函数的参数。

`setTimeout`返回一个数字ID，可用于取消尚未执行的定时器。

以下是一些`setTimeout`的使用示例：

基本用法：

```
setTimeout(() => {
  console.log('This message will be displayed after 2 seconds.');
}, 2000);
```

使用参数：

```
function greet(name, age) {
  console.log(`Hello, ${name}! You are ${age} years old.`);
}

setTimeout(greet, 3000, 'John', 25);
```

取消定时器：

```
const timeoutId = setTimeout(() => {
  console.log('This message will not be displayed.');
}, 5000);

setTimeout(() => {
  clearTimeout(timeoutId);
  console.log('Timeout has been cancelled.');
}, 2000);
```

在这个例子中，第一个`setTimeout`被取消了，所以它的回调函数不会执行。

案例练习

> 前三个案例不展示，较为简单

消息滚动切换

m站头部关闭

侧边栏展示

轮播图(重要)

，

和	**单元格的集合** • **tr.sectionRowIndex — 给定的**
和	： ◦ **td.cellIndex — 在封闭的**