鼠标事件

1. 鼠标事件的种类

鼠标事件都继承了 `MouseEvent` 接口，主要有下面这些事件：

1.1. 点击事件

鼠标点击相关的有四个事件：

- `click`：按下鼠标（通常是按下主按钮）时触发。
- `dblclick`：在同一个元素上双击鼠标时触发。
- `mousedown`：按下鼠标键时触发。
- `mouseup`：释放按下的鼠标键时触发。

`click` 事件可以看成是两个事件组成的：用户在同一个位置先触发 `mousedown`，再触发 `mouseup`。因此，触发顺序是，`mousedown` 首先触发，`mouseup` 接着触发，`click` 最后触发。

双击时，`dblclick` 事件则会在 `mousedown`、`mouseup`、`click` 之后触发。

1.2. 移动事件

鼠标移动相关的有五个事件：

- `mousemove`：当鼠标在一个节点内部移动时触发。当鼠标持续移动时，该事件会连续触发。为了避免性能问题，建议对该事件的监听函数做一些限定，比如限定一段时间内只能运行一次（节流）
- `mouseenter`：鼠标进入一个节点时触发，进入子节点不会触发这个事件。
- `mouseover`：鼠标进入一个节点时触发，进入子节点会再一次触发这个事件。
- `mouseout`：鼠标离开一个节点时触发，离开父节点也会触发这个事件。
- `mouseleave`：鼠标离开一个节点时触发，离开父节点不会触发这个事件。

`mouseover` 事件和 `mouseenter` 事件，都是鼠标进入一个节点时触发。两者的区别是，**`mouseenter` 事件只触发一次，而只要鼠标在节点内部移动，`mouseover` 事件会在子节点上触发多次。**

```
/*
 <ul>
   <li>item 1</li>
   <li>item 2</li>
   <li>item 3</li>
 </ul>
*/

let ul = document.querySelector('ul');

// 进入 ul 节点以后，mouseenter 事件只会触发一次，以后只要鼠标在节点内移动，都不会再触发这个事件。 event.target 是 ul 节点
ul.addEventListener('mouseenter', function (event) {
  event.target.style.color = 'purple';
  setTimeout(function () {
```

```
    event.target.style.color = '';
  }, 500);
}, false);

// 进入 ul 节点以后，只要在子节点上移动，mouseover 事件会触发多次。event.target 是 li
节点
ul.addEventListener('mouseover', function (event) {
  event.target.style.color = 'orange';
  setTimeout(function () {
    event.target.style.color = '';
  }, 500);
}, false);
```

上面代码中，在父节点内部进入子节点，不会触发 `mouseenter` 事件，但是会触发 `mouseover` 事件。

`mouseout` 事件和 `mouseleave` 事件，都是鼠标离开一个节点时触发。两者的区别是，**在父元素内部离开一个子元素时，`mouseleave` 事件不会触发，而 `mouseout` 事件会触发。**

```
/*
 <ul>
   <li>item 1</li>
   <li>item 2</li>
   <li>item 3</li>
 </ul>
*/

let ul = document.querySelector('ul');

// 先进入 ul 节点，然后在节点内部移动，不会触发 mouseleave 事件。只有离开 ul 节点时，触
发一次 mouseleave。event.target 是 ul 节点
ul.addEventListener('mouseleave', function (event) {
  event.target.style.color = 'purple';
  setTimeout(function () {
    event.target.style.color = '';
  }, 500);
}, false);

// 先进入 ul 节点，然后在节点内部移动，mouseout 事件会触发多次。event.target 是 li 节点
ul.addEventListener('mouseout', function (event) {
  event.target.style.color = 'orange';
  setTimeout(function () {
    event.target.style.color = '';
  }, 500);
}, false);
```

上面代码中，在父节点内部离开子节点，不会触发 `mouseleave` 事件，但是会触发 `mouseout` 事件。

1.3. 其他事件

- `contextmenu`：按下鼠标右键时（上下文菜单出现前）触发，或者按下“上下文”菜单键时触发。

- `wheel`：滚动鼠标的滚轮时触发，该事件继承的是 `WheelEvent` 接口。

2. `MouseEvent` 接口

`MouseEvent` 接口代表了鼠标相关的事件，单击（`click`）、双击（`dblclick`）、松开鼠标键（`mouseup`）、按下鼠标键（`mousedown`）等动作，所产生的事件对象都是 `MouseEvent` 实例。此外，滚轮事件和拖拉事件也是 `MouseEvent` 实例。`MouseEvent` 接口继承了 `Event` 接口，所以拥有 `Event` 的所有属性和方法，并且还提供鼠标独有的属性和方法。

浏览器原生提供一个 `MouseEvent()` 构造函数，用于新建一个 `MouseEvent` 实例。

```
let event = new MouseEvent(type, options);
```

`MouseEvent()` 构造函数接受两个参数。第一个参数是字符串，表示事件名称；第二个参数是一个事件配置对象，该参数可选。除了 Event 接口的实例配置属性，该对象可以配置以下属性，所有属性都是可选的：

- `screenX`：数值，鼠标相对于屏幕的水平位置（单位像素），默认值为 0，设置该属性不会移动鼠标。
- `screenY`：数值，鼠标相对于屏幕的垂直位置（单位像素），其他与screenX相同。
- `clientX`：数值，鼠标相对于程序窗口的水平位置（单位像素），默认值为0，设置该属性不会移动鼠标。
- `clientY`：数值，鼠标相对于程序窗口的垂直位置（单位像素），其他与clientX相同。
- `ctrlKey`：布尔值，是否同时按下了 Ctrl 键，默认值为false。
- `shiftKey`：布尔值，是否同时按下了 Shift 键，默认值为false。
- `altKey`：布尔值，是否同时按下 Alt 键，默认值为false。
- `metaKey`：布尔值，是否同时按下 Meta 键，默认值为false。
- `relatedTarget`：节点对象，表示事件的相关节点，默认为 `null`。`mouseenter` 和 `mouseover` 事件时，表示鼠标刚刚离开的那个元素节点；`mouseout` 和 `mouseleave` 事件时，表示鼠标正在进入的那个元素节点。

```
function simulateClick() {
  let event = new MouseEvent('click', {
    'bubbles': true,
    'cancelable': true
  }); // 生成一个鼠标点击事件
  let cb = document.getElementById('checkbox');
  cb.dispatchEvent(event); // 触发该事件
}
```

3. MouseEvent 接口的实例属性

3.1. `MouseEvent.altKey`、`MouseEvent.ctrlKey`、`MouseEvent.metaKey`、`MouseEvent.shiftKey`

这四个属性都返回一个布尔值，表示事件发生时，是否按下对应的键。它们都是只读属性。

- `altKey` 属性：`Alt` 键

- `ctrlKey` 属性：`Ctrl` 键
- `metaKey` 属性：`Meta` 键（Mac 键盘是一个四瓣的小花，Windows 键盘是 Windows 键）
- `shiftKey` 属性：`Shift` 键

```
// <body onclick="showKey(event)">
function showKey(e) {
  console.log('ALT key pressed: ' + e.altKey); // 同时按下鼠标左键，和 Alt 键时，这里打印 true
  console.log('CTRL key pressed: ' + e.ctrlKey); // 同时按下鼠标左键，和 Ctrl 键时，这里打印 true
  console.log('META key pressed: ' + e.metaKey); // 同时按下鼠标左键，和 Meta（window 系统是 windows 键） 键时，这里打印 true
  console.log('SHIFT key pressed: ' + e.shiftKey); // 同时按下鼠标左键，和 Shift 键时，这里打印 true
}
```

3.2. MouseEvent.button

`MouseEvent.button` 属性返回一个数值，表示事件发生时按下了鼠标的哪个键。该属性只读。

- 0：按下主键（通常是左键），或者该事件没有初始化这个属性（比如 `mousemove` 事件）。
- 1：按下辅助键（通常是中键或者滚轮键）。
- 2：按下次键（通常是右键）。

```
// <button onmouseup="whichButton(event)">点击</button>
let whichButton = function (e) {
  switch (e.button) {
    case 0:
      console.log('Left button clicked.');
      break;
    case 1:
      console.log('Middle button clicked.');
      break;
    case 2:
      console.log('Right button clicked.');
      break;
    default:
      console.log('Unexpected code: ' + e.button);
  }
}
```

3.3. MouseEvent.clientX，MouseEvent.clientY

`MouseEvent.clientX` 属性返回鼠标位置相对于浏览器窗口左上角的水平坐标（单位像素），`MouseEvent.clientY` 属性返回垂直坐标。这两个属性都是只读属性。

```
// <body onmousedown="showCoords(event)">
function showCoords(evt){
  console.log('clientX value: ' + evt.clientX + '\n' + 'clientY value: ' +
evt.clientY + '\n');
}
```

这两个属性还分别有一个别名 `MouseEvent.x` 和 `MouseEvent.y`。

3.4. MouseEvent.screenX，MouseEvent.screenY

`MouseEvent.screenX` 属性返回鼠标位置相对于屏幕左上角的水平坐标（单位像素），`MouseEvent.screenY` 属性返回垂直坐标。这两个属性都是只读属性。

```
// <body onmousedown="showCoords(event)">
function showCoords(evt) {
  console.log('screenX value: ' + evt.screenX + '\n', 'screenY value: ' +
evt.screenY + '\n');
}
```

3.5. MouseEvent.offsetX，MouseEvent.offsetY

`MouseEvent.offsetX` 属性返回鼠标位置与目标节点左侧的 `padding` 边缘的水平距离（单位像素），`MouseEvent.offsetY` 属性返回与目标节点上方的 `padding` 边缘的垂直距离。这两个属性都是只读属性。

```
/*
  <style>
    p {
      width: 100px;
      height: 100px;
      padding: 100px;
    }
  </style>
  <p>Hello</p>
*/
let p = document.querySelector('p');
p.addEventListener(
  'click',
  function (e) {
    console.log(e.offsetX);
    console.log(e.offsetY);
  },
  false
);
```

上面代码中，鼠标如果在p元素的中心位置点击，会返回150 150。因此中心位置距离左侧和上方的padding边缘，等于padding的宽度（100像素）加上元素内容区域一半的宽度（50像素）。

3.6. MouseEvent.pageX，MouseEvent.pageY

`MouseEvent.pageX` 属性返回鼠标位置与文档左侧边缘的距离（单位像素），`MouseEvent.pageY` 属性返回与文档上侧边缘的距离（单位像素）。它们的返回值都包括文档不可见的部分。这两个属性都是只读。

```
/*
  <style>
    body {
      height: 2000px;
    }
  </style>
*/
document.body.addEventListener(
  'click',
  function (e) {
    console.log(e.pageX);
    console.log(e.pageY);
  },
  false
);
```

上面代码中，页面高度为2000像素，会产生垂直滚动条。滚动到页面底部，点击鼠标输出的pageY值会接近2000。

4. WheelEvent 接口

`WheelEvent` 接口继承了 `MouseEvent` 实例，代表鼠标滚轮事件的实例对象。目前，鼠标滚轮相关的事件只有一个 `wheel` 事件，用户滚动鼠标的滚轮，就生成这个事件的实例。浏览器原生提供 `WheelEvent()` 构造函数，用来生成 `WheelEvent` 实例。

```
let wheelEvent = new WheelEvent(type, options);
```

`WheelEvent()` 构造函数可以接受两个参数，第一个是字符串，表示事件类型，对于滚轮事件来说，这个值目前只能是 `wheel`。第二个参数是事件的配置对象。该对象的属性除了 `Event`、`UIEvent` 的配置属性以外，还可以接受以下几个属性，所有属性都是可选的。

- `deltaX`：数值，表示滚轮的水平滚动量，默认值是 0.0。
- `deltaY`：数值，表示滚轮的垂直滚动量，默认值是 0.0。
- `deltaZ`：数值，表示滚轮的 `Z` 轴滚动量，默认值是 0.0。
- `deltaMode`：数值，表示相关的滚动事件的单位，适用于上面三个属性。0 表示滚动单位为像素，1 表示单位为行，2 表示单位为页，默认为 0。