

事件模型

1. 监听函数

浏览器的事件模型，就是通过监听函数（`listener`）对事件做出反应。事件发生后，浏览器监听到了这个事件，就会执行对应的监听函数。这是事件驱动编程模式（`event-driven`）的主要编程方式。

JavaScript 有三种方法，可以为事件绑定监听函数。

1.1. HTML 的 `on-` 属性

HTML 语言允许在元素的属性中，直接定义某些事件的监听代码。

```
<body onload="doSomething()"></body>
<div onclick="console.log('触发事件')"></div>
```

上例为 `body` 节点的 `load` 事件、`div` 节点的 `click` 事件，指定了监听代码。一旦事件发生，就会执行这段代码。

元素的事件监听属性，都是 `on` 加上事件名，比如 `onload` 就是 `on` + `load`，表示 `load` 事件的监听代码。

这些属性的值是将会执行的代码，而不是一个函数。

```
<!-- 正确 -->
<body onload="doSomething()">
<!-- 错误 -->
<body onload="doSomething"></body>
</body>
```

一旦指定的事件发生，`on`-属性的值是原样传入 JavaScript 引擎执行。**如果要执行函数，加上一对圆括号。**

使用这个方法指定的监听代码，只会在冒泡阶段触发。

```
<div onclick="console.log(2)">
  <button onclick="console.log(1)">点击</button>
</div>
```

上例中，`<button>` 是 `<div>` 的子元素。`<button>` 的 `click` 事件，也会触发 `<div>` 的 `click` 事件。由于 `on`-属性的监听代码，只在冒泡阶段触发，所以点击结果是先输出 1，再输出 2，即事件从子元素开始冒泡到父元素。

直接设置 `on`-属性，与通过元素节点的 `setAttribute` 方法设置 `on`-属性，效果是一样的。

```
el.setAttribute("onclick", "doSomething()");  
// 等同于  
// <Element onclick="doSomething()">
```

1.2. 元素节点的事件属性

元素节点对象的事件属性，同样可以指定监听函数。

```
window.onload = doSomething;  
  
div.onclick = function (event) {  
    console.log("触发事件");  
};
```

使用这个方法指定的监听函数，也是只会在冒泡阶段触发。

这种方法与 HTML 的 `on-` 属性的差异是，它的值是函数名（`doSomething`），而不像后者，必须给出完整的监听代码（`doSomething()`）。

1.3. EventTarget.addEventListener()

所有 DOM 节点实例都有 `addEventListener()` 方法，用来为该节点定义事件的监听函数。

```
window.addEventListener("load", doSomething, false);
```

- 上面三种方法，第一种“HTML 的 `on-` 属性”，违反了 HTML 与 JavaScript 代码相分离的原则，将两者写在一起，不利于代码分工，不推荐使用。
- 第二种“元素节点的事件属性”的缺点在于，同一个事件只能定义一个监听函数，也就是说，如果定义两次 `onclick` 属性，后一次定义会覆盖前一次，也不推荐使用。
- 第三种 `EventTarget.addEventListener` 是推荐的指定监听函数的方法。它有如下优点：
 - 同一个事件可以添加多个监听函数。
 - 能够指定在哪个阶段（捕获阶段还是冒泡阶段）触发监听函数。
 - 除了 DOM 节点，其他对象（比如 `window`、`XMLHttpRequest` 等）也有这个接口，它等于是整个 JavaScript 统一的监听函数接口。

2. this 的指向

监听函数内部的 `this` 指向触发事件的那个元素节点。

```
<button id="btn" onclick="console.log(this.id)">  
    点击  
</button>
```

执行上例，点击后会输出 `btn`。

其他两种监听函数的写法，`this` 的指向也是如此。

```
// <button id="btn">点击</button>
let btn = document.getElementById("btn");

// 写法一
btn.onclick = function () {
  console.log(this.id);
};

// 写法二
btn.addEventListener(
  "click",
  function (e) {
    console.log(this.id);
  },
  false
);
```

上面两种写法，点击按钮以后也是输出 `btn`。

3. 事件的传播

一个事件发生后，会在子元素和父元素之间传播（`propagation`）。这种传播分成三个阶段：

- 第一阶段：从 window 对象传导到目标节点（上层传到底层），称为“捕获阶段”（`capture phase`）。
- 第二阶段：在目标节点上触发，称为“目标阶段”（`target phase`）。
- 第三阶段：从目标节点传导回 window 对象（从底层传回上层），称为“冒泡阶段”（`bubbling phase`）。

这种三阶段的传播模型，使得同一个事件会在多个节点上触发。

```
<div>
  <p>点击</p>
</div>
```

上例中，`<div>` 节点之中有一个 `<p>` 节点。

如果在这两个节点，都设置 `click` 事件的监听函数（每个节点的捕获阶段和冒泡阶段，各设置一个监听函数），共计设置四个监听函数。然后，对 `<p>` 点击，`click` 事件会触发四次。

```
let phases = {
  1: "capture",
  2: "target",
  3: "bubble",
```

```
};

let div = document.querySelector("div");
let p = document.querySelector("p");

div.addEventListener("click", callback, true);
p.addEventListener("click", callback, true);
div.addEventListener("click", callback, false);
p.addEventListener("click", callback, false);

function callback(event) {
  let tag = event.currentTarget.tagName;
  // event.eventPhase 返回一个代表当前执行阶段的整数值
  let phase = phases[event.eventPhase];
  console.log("Tag: '" + tag + "'. EventPhase: '" + phase + "'");
}

// 点击以后的结果
// Tag: 'DIV'. EventPhase: 'capture'
// Tag: 'P'. EventPhase: 'target'
// Tag: 'P'. EventPhase: 'target'
// Tag: 'DIV'. EventPhase: 'bubble'
```

常量	值	描述
Event.NONE	0	这个时间，没有事件正在被处理
Event.CAPTURING_PHASE	1	事件正在被目标元素的祖先对象处理. 这个处理过程从 Window 开始，然后 Document，然后是 HTMLHtmlElement，一直这样，直到目标元素的父元素。通过 EventTarget.addEventListener() 注册为捕获模式的 Event listeners 被调用。
Event.AT_TARGET	2	事件对象已经抵达 the event's target. 为这个阶段注册的事件监听被调用。如果 Event.bubbles 的值为 false, 对事件对象的处理在这个阶段后就会结束.
Event.BUBBLING_PHASE	3	事件对象逆向向上传播回目标元素的祖先元素, 从父亲元素开始，并且最终到达包含元素 Window. 这就是冒泡，并且只有 Event.bubbles 值为 true 的时候才会发生。为这个阶段注册的 Event listeners 在这个过程中被触发.

上例表示，click 事件被触发了四次：<div> 节点的捕获阶段和冒泡阶段各 1 次，<p>节点的目标阶段触发了 2 次。

- 捕获阶段：事件从 <div> 向 <p> 传播时，触发 <div> 的 click 事件；
- 目标阶段：事件从 <div> 到达 <p> 时，触发 <p> 的 click 事件；
- 冒泡阶段：事件从 <p> 传回 <div> 时，再次触发 <div> 的 click 事件。

其中，<p> 节点有两个监听函数（addEventListener 方法第三个参数的不同，会导致绑定两个监听函数），因此它们都会因为 click 事件触发一次。所以，<p> 会在 target 阶段有两次输出。

浏览器总是假定 `click` 事件的目标节点，就是点击位置嵌套最深的那个节点（本例是 `<div>` 节点里面的 `<p>` 节点）。所以，`<p>` 节点的捕获阶段和冒泡阶段，都会显示为 `target` 阶段。

事件传播的最上层对象是 `window`，接着依次是 `document`，`html` (`document.documentElement`) 和 `body` (`document.body`)。也就是说，上例的事件传播顺序，在捕获阶段依次为 `window`、`document`、`html`、`body`、`div`、`p`，在冒泡阶段依次为 `p`、`div`、`body`、`html`、`document`、`window`。

4. 事件的代理

由于事件会在冒泡阶段向上传播到父节点，因此可以把子节点的监听函数定义在父节点上，由父节点的监听函数统一处理多个子元素的事件。这种方法叫做事件的代理 (`delegation`)。

```
let ul = document.querySelector("ul");

ul.addEventListener("click", function (event) {
  if (event.target.tagName.toLowerCase() === "li") {
    // some code
  }
});
```

上例中，`click` 事件的监听函数定义在 `` 节点，但是实际上，它处理的是子节点 `` 的 `click` 事件。这样做的好处是，只要定义一个监听函数，就能处理多个子节点的事件，而不用在每个 `` 节点上定义监听函数。而且以后再添加子节点，监听函数依然有效。

如果希望事件到某个节点为止，不再传播，可以使用事件对象的 `stopPropagation` 方法。

```
// 事件传播到 p 元素后，就不再向下传播了
p.addEventListener(
  "click",
  function (event) {
    event.stopPropagation();
  },
  true
);

// 事件冒泡到 p 元素后，就不再向上冒泡了
p.addEventListener(
  "click",
  function (event) {
    event.stopPropagation();
  },
  false
);
```

上例中，`stopPropagation` 方法分别在捕获阶段和冒泡阶段，阻止了事件的传播。

但是，`stopPropagation` 方法只会阻止事件的传播，不会阻止该事件触发 `<p>` 节点的其他 `click` 事件的监听函数。也就是说，不是彻底取消 `click` 事件。

```
p.addEventListener("click", function (event) {
  event.stopPropagation();
  console.log(1);
});

p.addEventListener("click", function (event) {
  // 会触发
  console.log(2);
});
```

上例中，`p` 元素绑定了两个 `click` 事件的监听函数。`stopPropagation` 方法只能阻止这个事件的传播，不能取消这个事件，因此，第二个监听函数会触发。输出结果会先是 `1`，然后是 `2`。

如果想要彻底取消该事件，不再触发后面所有 `click` 的监听函数，可以使用 `stopImmediatePropagation` 方法。

```
p.addEventListener("click", function (event) {
  event.stopImmediatePropagation();
  console.log(1);
});

p.addEventListener("click", function (event) {
  // 不会被触发
  console.log(2);
});
```

上例中，`stopImmediatePropagation` 方法可以彻底取消这个事件，使得后面绑定的所有 `click` 监听函数都不再触发。所以，只会输出 `1`，不会输出 `2`。