Event 对象

目录
概述
实例属性
Event.bubbles， Event.eventPhase
Event.cancelable， Event.cancelBubble，event.defaultPrevented Event.currentTarget， Event.target
Event.type
Event.timeStamp
Event.isTrusted
Event.detail
实例方法
Event.preventDefault()
Event.stopPropagation()
Event.stopImmediatePropagation()
Event.composedPath()

概述
事件发生以后，会产生一个事件对象，作为参数传给监听函数。浏览器原生提供一个 Event
对象，所有的事件都是这个对象的实例，或者说继承了 Event.prototype对象。

Event 对象本身就是一个构造函数，可以用来生成新的实例。

event = new Event(type, options);

Event 构造函数接受两个参数。第一个参数 type是字符串，表示事件的名称；第二个参数 options 是一个对象，表示事件对象的配置。该对象主要有下面两个属性。

bubbles ：布尔值，可选，默认为 false，表示事件对象是否冒泡。
cancelable：布尔值，可选，默认为 false，表示事件是否可以被取消，即能否用
Event.preventDefault()

取消这个事件。一旦事件被取消，就好像从来没有发生过，不会触发浏览器对该事件的默认行为。

var ev = new Event(
'look',
{
'bubbles': true,
'cancelable': false
}
);
document.dispatchEvent(ev);

上面代码新建一个 look事件实例，然后使用 dispatchEvent方法触发该事件。

注意，如果不是显式指定 bubbles属性为 true，生成的事件就只能在 “捕获阶段 ”触发监听函数。

// HTML 代码为
// <div><p>Hello</p></div>
var div = document.querySelector('div');
var p = document.querySelector('p');

function callback(event) {
var tag = event.currentTarget.tagName;
console.log('Tag: ' + tag); // 没有任何输出
}

div.addEventListener('click', callback, false);

var click = new Event('click');
p.dispatchEvent(click);

上面代码中， p元素发出一个 click事件，该事件默认不会冒泡。 div.addEventListener
方法指定在冒泡阶段监听，因此监听函数不会触发。如果写成 div.addEventListener('click', callback, true) ，那么在 “捕获阶段 ”可以监听到这个事件。

另一方面，如果这个事件在 div元素上触发。

div.dispatchEvent(click);

那么，不管 div元素是在冒泡阶段监听，还是在捕获阶段监听，都会触发监听函数。因为这时 div
元素是事件的目标，不存在是否冒泡的问题， div元素总是会接收到事件，因此导致监听函数生效。

实例属性

Event.bubbles，Event.eventPhase

`Event.bubbles`属性返回一个布尔值，表示当前事件是否会冒泡。该属性为只读属性，一般用来了解 Event 实例是否可以冒泡。前面说过，除非显式声明，`Event`构造函数生成的事件，默认是不冒泡的。

`Event.eventPhase`属性返回一个整数常量，表示事件目前所处的阶段。该属性只读。

```
var phase = event.eventPhase;
```

`Event.eventPhase`的返回值有四种可能。

- 0，事件目前没有发生。
- 1，事件目前处于捕获阶段，即处于从祖先节点向目标节点的传播过程中。
- 2，事件到达目标节点，即`Event.target`属性指向的那个节点。
- 3，事件处于冒泡阶段，即处于从目标节点向祖先节点的反向传播过程中。

Event.cancelable，Event.cancelBubble，event.defaultPrevented

`Event.cancelable`属性返回一个布尔值，表示事件是否可以取消。该属性为只读属性，一般用来了解 Event 实例的特性。

大多数浏览器的原生事件是可以取消的。比如，取消`click`事件，点击链接将无效。但是除非显式声明，`Event`构造函数生成的事件，默认是不可以取消的。

```
var evt = new Event('foo');
evt.cancelable  // false
```

当`Event.cancelable`属性为`true`时，调用`Event.preventDefault()`就可以取消这个事件，阻止浏览器对该事件的默认行为。

如果事件不能取消，调用`Event.preventDefault()`会没有任何效果。所以使用这个方法之前，最好用`Event.cancelable`属性判断一下是否可以取消。

```
function preventEvent(event) {
  if (event.cancelable) {
    event.preventDefault();
```

```
  } else {
    console.warn('This event couldn\'t be canceled.');
    console.dir(event);
  }
}
```

`Event.cancelBubble`属性是一个布尔值，如果设为`true`，相当于执行`Event.stopPropagation()`，可以阻止事件的传播。

`Event.defaultPrevented`属性返回一个布尔值，表示该事件是否调用过`Event.preventDefault`方法。该属性只读。

```
if (event.defaultPrevented) {
  console.log('该事件已经取消了');
}
```

Event.currentTarget，Event.target

事件发生以后，会经过捕获和冒泡两个阶段，依次通过多个DOM节点。因此，任意事件都有两个与事件相关的节点，一个是事件的原始触发节点（`Event.target`），另一个是事件当前正在通过的节点（`Event.currentTarget`）。前者通常是后者的后代节点。

`Event.currentTarget`属性返回事件当前所在的节点，即事件当前正在通过的节点，也就是当前正在执行的监听函数所在的那个节点。随着事件的传播，这个属性的值会变。

`Event.target`属性返回原始触发事件的那个节点，即事件最初发生的节点。这个属性不会随着事件的传播而改变。

事件传播过程中，不同节点的监听函数内部的`Event.target`与`Event.currentTarget`属性的值是不一样的。

```
// HTML 代码为
// <p id="para">Hello <em>World</em></p>
function hide(e) {
  // 不管点击 Hello 或 World，总是返回 true
  console.log(this === e.currentTarget);

  // 点击 Hello，返回 true
  // 点击 World，返回 false
  console.log(this === e.target);
}
```

```
document.getElementById('para').addEventListener('click', hide, false);
```

上面代码中，`<em>`是`<p>`的子节点，点击`<em>`或者点击`<p>`，都会导致监听函数执行。这时，`e.target`总是指向原始点击位置的那个节点，而`e.currentTarget`指向事件传播过程中正在经过的那个节点。由于监听函数只有事件经过时才会触发，所以`e.currentTarget`总是等同于监听函数内部的`this`。

Event.type

`Event.type`属性返回一个字符串，表示事件类型。事件的类型是在生成事件的时候指定的。该属性只读。

```
var evt = new Event('foo');
evt.type // "foo"
```

Event.timeStamp

`Event.timeStamp`属性返回一个毫秒时间戳，表示事件发生的时间。它是相对于网页加载成功开始计算的。

```
var evt = new Event('foo');
evt.timeStamp // 3683.6999999995896
```

它的返回值有可能是整数，也有可能是小数（高精度时间戳），取决于浏览器的设置。

下面是一个计算鼠标移动速度的例子，显示每秒移动的像素数量。

```
var previousX;
var previousY;
var previousT;

window.addEventListener('mousemove', function(event) {
  if (
    previousX !== undefined &&
    previousY !== undefined &&
    previousT !== undefined
  ) {
    var deltaX = event.screenX - previousX;
    var deltaY = event.screenY - previousY;
    var deltaD = Math.sqrt(Math.pow(deltaX, 2) + Math.pow(deltaY, 2));
```

```
    var deltaT = event.timeStamp - previousT;
    console.log(deltaD / deltaT * 1000);
  }

  previousX = event.screenX;
  previousY = event.screenY;
  previousT = event.timeStamp;
});
```

Event.isTrusted

`Event.isTrusted` 属性返回一个布尔值，表示该事件是否由真实的用户行为产生。比如，用户点击链接会产生一个 `click` 事件，该事件是用户产生的；`Event` 构造函数生成的事件，则是脚本产生的。

```
var evt = new Event('foo');
evt.isTrusted // false
```

上面代码中，`evt` 对象是脚本产生的，所以 `isTrusted` 属性返回 `false`。

Event.detail

`Event.detail` 属性只有浏览器的 UI（用户界面）事件才具有。该属性返回一个数值，表示事件的某种信息。具体含义与事件类型相关。比如，对于 `click` 和 `dblclick` 事件，`Event.detail` 是鼠标按下的次数（`1` 表示单击，`2` 表示双击，`3` 表示三击）；对于鼠标滚轮事件，`Event.detail` 是滚轮正向滚动的距离，负值就是负向滚动的距离，返回值总是 3 的倍数。

```
// HTML 代码如下
// <p>Hello</p>
function giveDetails(e) {
  console.log(e.detail);
}

document.querySelector('p').onclick = giveDetails;
```

实例方法

Event.preventDefault()
Event.preventDefault
方法取消浏览器对当前事件的默认行为。比如点击链接后，浏览器默认会跳转到另一个页面，使用这个方法以
后，就不会跳转了；再比如，按一下空格键，页面向下滚动一段距离，使用这个方法以后也不会滚动了。该方
法生效的前提是，事件对象的 cancelable属性为 true，如果为 false，调用该方法没有任何效果。

注意，该方法只是取消事件对当前元素的默认影响，不会阻止事件的传播。如果要阻止传播，可以使用
stopPropagation() 或 stopImmediatePropagation() 方法。

// HTML 代码为
// <input type="checkbox" id="my-checkbox" />
var cb = document.getElementById('my-checkbox');

cb.addEventListener(
'click',
function (e){ e.preventDefault(); },
false
);

上面代码中，浏览器的默认行为是单击会选中单选框，取消这个行为，就导致无法选中单选框。

利用这个方法，可以为文本输入框设置校验条件。如果用户的输入不符合条件，就无法将字符输入文本框。

// HTML 代码为
// <input type="text" id="my-input" />
var input = document.getElementById('my-input'); input.addEventListener('keypress', checkName, false);

function checkName(e) {
if (e.charCode < 97 || e.charCode > 122) { e.preventDefault();
}
}

上面代码为文本框的 keypress
事件设定监听函数后，将只能输入小写字母，否则输入事件的默认行为（写入文本框）将被取消，导致不能向
文本框输入内容。

Event.stopPropagation()

stopPropagation方法阻止事件在 DOM
中继续传播，防止再触发定义在别的节点上的监听函数，但是不包括在当前节点上其他的事件监听函数。

function stopEvent(e) {
e.stopPropagation();
}

el.addEventListener('click', stopEvent, false);

上面代码中， click事件将不会进一步冒泡到 el节点的父节点。

Event.stopImmediatePropagation()
Event.stopImmediatePropagation
方法阻止同一个事件的其他监听函数被调用，不管监听函数定义在当前节点还是其他节点。也就是说，该方法
阻止事件的传播，比 Event.stopPropagation()更彻底。

如果同一个节点对于同一个事件指定了多个监听函数，这些函数会根据添加的顺序依次调用。只要其中有一个
监听函数调用了 Event.stopImmediatePropagation方法，其他的监听函数就不会再执行了。

function l1(e){
e.stopImmediatePropagation();
}

function l2(e){
console.log('hello world');
}

el.addEventListener('click', l1, false);
el.addEventListener('click', l2, false);

上面代码在 el节点上，为 click事件添加了两个监听函数 l1和 l2。由于 l1调用了 event.stopImmediatePropagation方法，所以 l2不会被调用。

Event.composedPath()
Event.composedPath() 返回一个数组，成员是事件的最底层节点和依次冒泡经过的所有上层节点。

// HTML 代码如下
// <div>
// <p>Hello</p>

```
// </div>
var div = document.querySelector('div');
var p = document.querySelector('p');

div.addEventListener('click', function (e) {
  console.log(e.composedPath());
}, false);
// [p, div, body, html, document, Window]
```

□□□□□□ `click`□□□□□□□□□ `p`□□□□□□ `div`□ `body`□ `html`□ `document`□ `Window`□

□□□ #2
□ □□□ □□□ 2018-04-23 21:40:23 UTC
□ □□□ □□□ 2020-10-01 20:06:24 UTC