# 事件

事件的本质是程序各个组成部分之间的一种通信方式，也是异步编程的一种实现。DOM 支持大量的事件。

## 概述

DOM 的事件操作（监听和触发），都定义在EventTarget接口。所有节点对象都部署了这个接口，其他一些需要事件通信的浏览器内置对象（比如，XMLHttpRequest、AudioNode、AudioContext）也部署了这个接口。

该接口主要提供三个实例方法。

* addEventListener：绑定事件的监听函数
* removeEventListener：移除事件的监听函数
* dispatchEvent：触发事件

## EventTarget.addEventListener()

EventTarget.addEventListener()用于在当前节点或对象上，定义一个特定事件的监听函数。一旦这个事件发生，就会执行监听函数。该方法没有返回值。

target.addEventListener(type, listener[, useCapture]);

该方法接受三个参数。

* type：事件名称，大小写敏感。
* listener：监听函数。事件发生时，会调用该监听函数。
* useCapture：布尔值，表示监听函数是否在捕获阶段（capture）触发，默认为false（监听函数只在冒泡阶段被触发）。该参数可选。

如果希望事件监听函数只执行一次，可以打开属性配置对象的once属性。

element.addEventListener('click', function (event) {

// 只执行一次的代码

}, {once: true});

addEventListener方法可以为针对当前对象的同一个事件，添加多个不同的监听函数。这些函数按照添加顺序触发，即先添加先触发。

如果希望向监听函数传递参数，可以用匿名函数包装一下监听函数。

## EventTarget.removeEventListener()

EventTarget.removeEventListener方法用来移除addEventListener方法添加的事件监听函数。该方法没有返回值。

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

div.removeEventListener('click', listener, false);

removeEventListener方法的参数，与addEventListener方法完全一致。

注意，removeEventListener方法移除的监听函数，**必须是addEventListener方法添加的那个监听函数**，而且必须在同一个元素节点，否则无效。

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

div.removeEventListener('click', function (e) {}, false);

上面代码中，removeEventListener方法无效，因为监听函数不是同一个匿名函数。

## EventTarget.dispatchEvent()

EventTarget.dispatchEvent方法在当前节点上触发指定事件，从而触发监听函数的执行。

dispatchEvent方法的参数是一个Event对象的实例。

para.addEventListener('click', hello, false);

var event = new Event('click');

para.dispatchEvent(event);

上面代码在当前节点触发了click事件。

# 事件模型

## 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。

### 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()">

<div onclick="console.log('触发事件')">

上面代码为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。

注意，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">

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

### 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');

};

注意，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()）。

### 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);

### 小结

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，因此不推荐使用。

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。因此，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。

## 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>

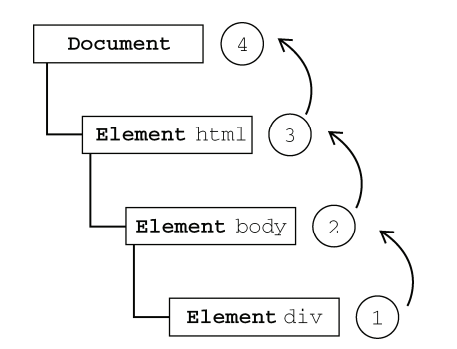
执行上面代码，点击后会输出btn。

其他两种监听函数的写法，this的指向也是如此。

## 事件的传播

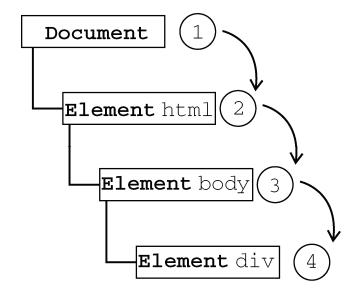
事件流描述的是从页面中接收事件的顺序。但有意思的是， IE 和 Netscape 开发团队居然提出了差不多是完全相反的事件流的概念。 IE 的事件流是事件冒泡流，而 Netscape Communicator 的事件流是事件捕获流 。

IE 的事件流叫做事件冒泡（event bubbling），即事件开始时由最具体的元素（文档中嵌套层次最深的那个节点）接收，然后逐级向上传播到较为不具体的节点（文档）



所有现代浏览器都支持事件冒泡 。

Netscape Communicator 团队提出的另一种事件流叫做事件捕获（event capturing）。事件捕获的思想是不太具体的节点应该更早接收到事件，而最具体的节点应该最后接收到事件。



IE9、 Safari、 Chrome、 Opera和 Firefox 目前也都支持这种事件流模型。

由于老版本的浏览器不支持，因此很少有人使用事件捕获。建议放心地使用事件冒泡，  
在有特殊需要时再使用事件捕获。

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();

});

上面代码中，stopPropagation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。

# Event对象

## 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。

## 实例属性

### Event.bubbles

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。

### Event.cancelBubble

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。

## 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。 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。

// HTML 代码为// <input type="checkbox" id="my-checkbox" />

var cb = document.getElementById('my-checkbox');

cb.addEventListener(

'click',

function (e){ e.preventDefault(); },

false

);

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。