JavaScript события введение

Введение в события, Всплытие и погружение, Делегирование событий, Действия браузера по умолчанию

Браузерные события

Событие – это сигнал от браузера о том, что что-то произошло. Все DOM-узлы подают такие сигналы (хотя события бывают и не только в DOM).

Обработчики событий
Событию можно назначить обработчик, то есть функцию, которая сработает, как только событие произошло.

Именно благодаря обработчикам JavaScript-код может реагировать на действия пользователя.

Есть несколько способов назначить событию обработчик.

- onclick в html разметке у элемента (тега)
- onclick в script части у элемента (html-элемента)
- используя addEventListener

События мыши:

- click – происходит, когда кликнули на элемент левой кнопкой мыши (на устройствах с сенсорными экранами оно происходит при касании).
- contextmenu – происходит, когда кликнули на элемент правой кнопкой мыши.
- mouseover / mouseout – когда мышь наводится на / покидает элемент.
- mousedown / mouseup – когда нажали / отжали кнопку мыши на элементе.
- mousemove – при движении мыши.

События на элементах управления:

- submit – пользователь отправил форму <form>.
- focus – пользователь фокусируется на элементе, например нажимает на <input>.
- Клавиатурные события:
- keydown и keyup – когда пользователь нажимает / отпускает клавишу.

События документа:

- DOMContentLoaded – когда HTML загружен и обработан, DOM документа полностью построен и доступен.

CSS events:

- transitionend – когда CSS-анимация завершена.

Использование атрибута HTML

Обработчик может быть назначен прямо в разметке, в атрибуте, который называется on<событие>.

Например, чтобы назначить обработчик события click на элементе input, можно использовать атрибут onclick

Атрибут HTML-тега – не самое удобное место для написания большого количества кода, поэтому лучше создать отдельную JavaScript-функцию и вызвать её там.

Атрибут HTML-тега не чувствителен к регистру, поэтому ONCLICK будет работать так же, как onClick и onCLICK… Но, как правило, атрибуты пишут в нижнем регистре: onclick.

```
<input value="Нажми меня" onclick="alert('Клик!')" type="button">
```

```
<script>
  function countRabbits() {
    for(let i=1; i<=3; i++) {
      alert("Кролик номер " + i);
    }
  }
</script>

<input type="button" onclick="countRabbits()" value="Считать кроликов!">
```

Использование свойства DOM-объекта

Можно назначить обработчик, используя свойство DOM-элемента on<событие>.

Если обработчик задан через атрибут, то браузер читает HTML-разметку, создает новую функцию из содержимого атрибута и записывает в свойство.

Этот способ, по сути, аналогичен предыдущему.

Обработчик всегда хранится в свойстве DOM-объекта, а атрибут – лишь один из способов его инициализации.

Так как у элемента DOM может быть только одно свойство с именем onclick, то назначить более одного обработчика так нельзя.

Обработчикам можно назначить и уже существующую функцию

Убрать обработчик можно назначением elem.onclick = null.

```
<input type="button" onclick="alert('Клик!')" value="Кнопка">
```

```
<input type="button" id="button" value="Кнопка">
<script>
  button.onclick = function() {
    alert('Клик!');
  };
</script>
```

```
<!doctype html>
<body>
<input type="button" id="elem" onclick="alert('Было')" value="Нажми меня">
<script>
  elem.onclick = function() { // перезапишет существующий обработчик
    alert('Станет'); // выведется только это
  };
</script>
</body>
```

Доступ к элементу через this

Внутри обработчика события this ссылается на текущий элемент, то есть на тот, на котором, как говорят, «висит» (т.е. назначен) обработчик.

```
<button onclick="alert(this.innerHTML)">Нажми меня</button>
```

Частые ошибки

Функция должна быть присвоена как sayThanks, а не sayThanks().

Если добавить скобки, то sayThanks() – это уже вызов функции, результат которого (равный undefined, так как функция ничего не возвращает) будет присвоен onclick. Так что это не будет работать.

Используйте именно функции, а не строки.

Назначение обработчика строкой elem.onclick = "alert(1)" также сработает. Это сделано из соображений совместимости, но делать так не рекомендуется.

Не используйте setAttribute для обработчиков.

Регистр DOM-свойства имеет значение.

Используйте elem.onclick, а не elem.ONCLICK, потому что DOM-свойства чувствительны к регистру.

```
// правильно
button.onclick = sayThanks;

// неправильно
button.onclick = sayThanks();
```

```
<input type="button" id="button" onclick="sayThanks()">
```

```
button.onclick = function() {
  sayThanks(); // содержимое атрибута
};
```

```
// при нажатии на body будут ошибки,
// атрибуты всегда строки, и функция станет строкой
document.body.setAttribute('onclick', function() { alert(1) });
```

addEventListener

Фундаментальный недостаток описанных выше способов назначения обработчика – невозможность повесить несколько обработчиков на одно событие.

Разработчики стандартов достаточно давно это поняли и предложили альтернативный способ назначения обработчиков при помощи специальных методов addEventListener и removeEventListener. Они свободны от указанного недостатка.

event - Имя события, например "click".

handler - Ссылка на функцию-обработчик.

options - Дополнительный объект со свойствами:

- once: если true, тогда обработчик будет автоматически удалён после выполнения.
- capture: фаза, на которой должен сработать обработчик, подробнее об этом будет рассказано в главе Всплытие и погружение. Так исторически сложилось, что options может быть false/true, это то же самое, что {capture: false/true}.
- passive: если true, то указывает, что обработчик никогда не вызовет preventDefault()

Для удаления обработчика следует использовать removeEventListener

```
element.addEventListener(event, handler, [options]);
```

```
element.removeEventListener(event, handler, [options]);
```

```
input.onclick = function() { alert(1); }
// ...
input.onclick = function() { alert(2); } // заменит предыдущий обработчик
```

```
<input id="elem" type="button" value="Нажми меня"/>

<script>
  function handler1() {
    alert('Спасибо!');
  };

  function handler2() {
    alert('Спасибо ещё раз!');
  }

  elem.onclick = () => alert("Привет");
  elem.addEventListener("click", handler1); // Спасибо!
  elem.addEventListener("click", handler2); // Спасибо ещё раз!
</script>
```

Удаление требует именно ту же функцию

Обратим внимание – если функцию обработчик не сохранить где-либо, мы не можем ее удалить. Нет метода, который позволяет получить из элемента обработчики событий, назначенные через addEventListener.

Вот так не сработает:

```
elem.addEventListener( "click" , () => alert('Спасибо!'));
// ....
elem.removeEventListener( "click", () => alert('Спасибо!'));
```

Вот так правильно:

```
function handler() {
  alert( 'Спасибо!' );
}

input.addEventListener("click", handler);
// ....
input.removeEventListener("click", handler);
```

Обработчики некоторых событий можно назначать только через addEventListener

Существуют события, которые нельзя назначить через DOM-свойство, но можно через addEventListener.

Например, таково событие DOMContentLoaded, которое срабатывает, когда завершена загрузка и построение DOM документа.

Так что addEventListener более универсален. Хотя заметим, что таких событий меньшинство, это скорее исключение, чем правило.

```
document.onDOMContentLoaded = function() {
  alert("DOM построен"); // не будет работать
};
```

```
document.addEventListener("DOMContentLoaded", function() {
  alert("DOM построен"); // а вот так сработает
});
```

Объект события

Некоторые свойства объекта event:

event.type - Тип события, в данном случае "click".

event.currentTarget - Элемент, на котором сработал обработчик. Значение – обычно такое же, как и у this, но если обработчик является функцией-стрелкой или при помощи bind привязан другой объект в качестве this, то мы можем получить элемент из event.currentTarget.

event.clientX / event.clientY - Координаты курсора в момент клика относительно окна, для событий мыши.

```
<input type="button" value="Нажми меня" id="elem">

<script>
  elem.onclick = function(event) {
    // вывести тип события, элемент и координаты клика
    alert(event.type + " на " + event.currentTarget);
    alert("Координаты: " + event.clientX + ":" + event.clientY);
  };
</script>
```

Объект-обработчик: handleEvent

Мы можем назначить обработчиком не только функцию, но и объект при помощи addEventListener. В этом случае, когда происходит событие, вызывается метод объекта handleEvent.

```
<button id="elem">Нажми меня</button>

<script>
  elem.addEventListener('click', {
    handleEvent(event) {
      alert(event.type + " на " + event.currentTarget);
    }
  });
</script>
```

```
<button id="elem">Нажми меня</button>

<script>
  class Menu {
    handleEvent(event) {
      switch(event.type) {
        case 'mousedown':
          elem.innerHTML = "Нажата кнопка мыши";
          break;
        case 'mouseup':
          elem.innerHTML += "...и отжата.";
          break;
      }
    }
  }

  let menu = new Menu();
  elem.addEventListener('mousedown', menu);
  elem.addEventListener('mouseup', menu);
</script>
```

Всплытие и погружение

- Всплытие
- event.target
- Прекращение всплытия
- Погружение

Всплытие

Когда на элементе происходит событие, обработчики сначала срабатывают на нём, потом на его родителе, затем выше и так далее, вверх по цепочке предков.

Клик по внутреннему <p> вызовет обработчик onclick:

- Сначала на самом <p>.
- Потом на внешнем .
- Затем на внешнем <form>.
- И так далее вверх по цепочке до самого document.

Почти все события всплывают.

Ключевое слово в этой фразе – «почти».

Например, событие focus не всплывает. В дальнейшем мы увидим и другие примеры. Однако, стоит понимать, что это скорее исключение, чем правило, всё-таки большинство событий всплывают.


```
<style>
  body * {
    margin: 10px;
    border: 1px solid blue;
  }
</style>

<form onclick="alert('form')">FORM
  <div onclick="alert('div')">DIV
    <p onclick="alert('p')">P</p>
  </div>
</form>
```

event.target

Всегда можно узнать, на каком конкретно элементе произошло событие.

Самый глубокий элемент, который вызывает событие, называется целевым элементом, и он доступен через event.target.

Отличия от this (=event.currentTarget):

- event.target – это «целевой» элемент, на котором произошло событие, в процессе всплытия он неизменен.
- this – это «текущий» элемент, до которого дошло всплытие, на нём сейчас выполняется обработчик.

Например, если стоит только один обработчик form.onclick, то он «поймает» все клики внутри формы. Где бы ни был клик внутри – он всплывет до элемента <form>, на котором сработает обработчик.

При этом внутри обработчика form.onclick:

this (=event.currentTarget) всегда будет элемент <form>, так как обработчик сработал на ней.

event.target будет содержать ссылку на конкретный элемент внутри формы, на котором произошёл клик.

ПРИМЕР

Прекращение всплытия

Всплытие идёт с «целевого» элемента прямо наверх. По умолчанию событие будет всплывать до элемента <html>, а затем до объекта document, а иногда даже до window, вызывая все обработчики на своём пути.

Но любой промежуточный обработчик может решить, что событие полностью обработано, и остановить всплытие.

Для этого нужно вызвать метод event.stopPropagation().

```
<body onclick="alert(`сюда всплытие не дойдёт`)">
  <button onclick="event.stopPropagation()">Кликни меня</button>
</body>
```

event.stopImmediatePropagation()

Если у элемента есть несколько обработчиков на одно событие, то даже при прекращении всплытия все они будут выполнены.

То есть, event.stopPropagation() препятствует продвижению события дальше, но на текущем элементе все обработчики будут вызваны.

Для того, чтобы полностью остановить обработку, существует метод event.stopImmediatePropagation(). Он не только предотвращает всплытие, но и останавливает обработку событий на текущем элементе.

Не прекращайте всплытие без необходимости!

Всплытие – это удобно. Не прекращайте его без явной нужды, очевидной и архитектурно прозрачной.

Зачастую прекращение всплытия через event.stopPropagation() имеет свои подводные камни, которые со временем могут стать проблемами.

Например:

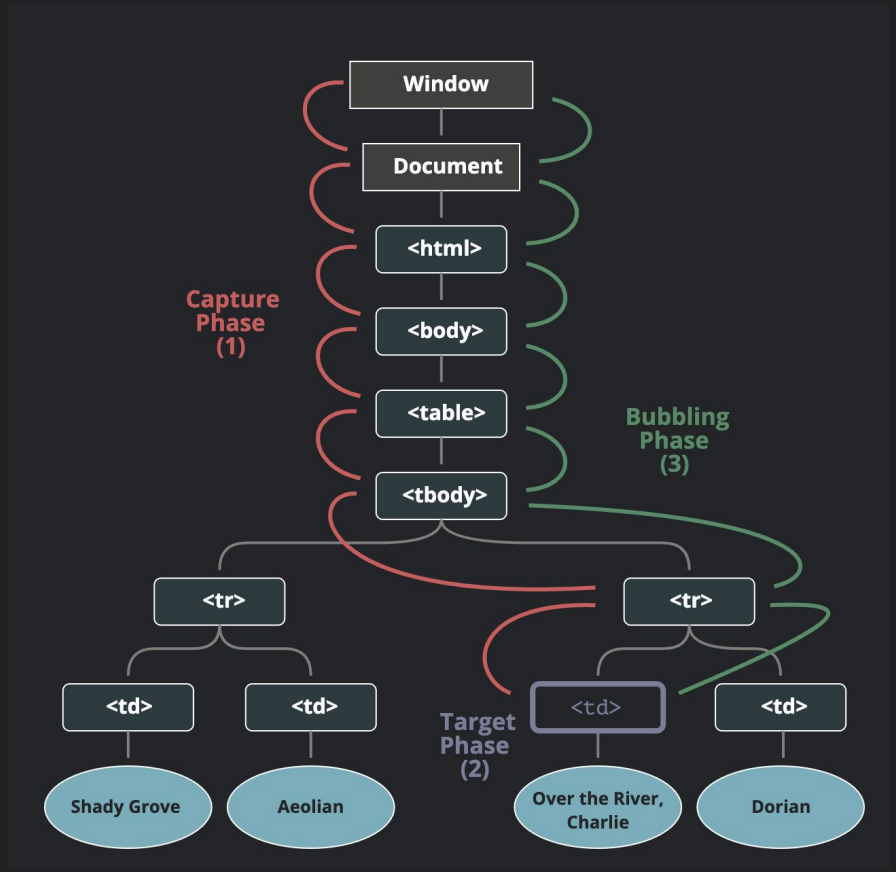
- Мы делаем вложенное меню. Каждое подменю обрабатывает клики на своих элементах и делает для них stopPropagation, чтобы не срабатывало внешнее меню.
- Позже мы решили отслеживать все клики в окне для какой-то своей функциональности, к примеру, для статистики – где вообще у нас кликают люди. Некоторые системы аналитики так делают. Обычно используют document.addEventListener('click'…), чтобы отлавливать все клики.
- Наша аналитика не будет работать над областью, где клики прекращаются stopPropagation. Увы, получилась «мёртвая зона».

Погружение

Существует еще одна фаза из жизненного цикла события – «погружение» (иногда её называют «перехват»). Она очень редко используется в реальном коде, однако тоже может быть полезной.

Стандарт DOM Events описывает 3 фазы прохода события:

- Фаза погружения (capturing phase) – событие сначала идёт сверху вниз.
- Фаза цели (target phase) – событие достигло целевого(исходного) элемента.
- Фаза всплытия (bubbling stage) – событие начинает всплывать.



capture

```
elem.addEventListener(..., {capture: true})
// или просто "true", как сокращение для {capture: true}
elem.addEventListener(..., true)
```

Ранее мы говорили только о всплытии, потому что другие стадии, как правило, не используются и проходят незаметно для нас.

Обработчики, добавленные через on<event>-свойство или через HTML-атрибуты, или через addEventListener(event, handler) с двумя аргументами, ничего не знают о фазе погружения, а работают только на 2-ой и 3-ей фазах.

Чтобы поймать событие на стадии погружения, нужно использовать третий аргумент capture.

Существуют два варианта значений опции capture:

- Если аргумент false (по умолчанию), то событие будет поймано при всплытии.
- Если аргумент true, то событие будет перехвачено при погружении.

Особенности погружения

Чтобы убрать обработчик removeEventListener, нужна та же фаза

Если мы добавили обработчик вот так addEventListener(..., true), то мы должны передать то же значение аргумента capture в removeEventListener(..., true), когда снимаем обработчик.

На каждой фазе разные обработчики на одном элементе срабатывают в порядке назначения

Если у нас несколько обработчиков одного события, назначенных addEventListener на один элемент, в рамках одной фазы, то их порядок срабатывания – тот же, в котором они установлены:

```
// всегда сработает перед следующим
elem.addEventListener("click", e => alert(1));

elem.addEventListener("click", e => alert(2));
```

Делегирование событий

Всплытие и перехват событий позволяет реализовать один из самых важных приемов разработки – делегирование.

Идея в том, что если у нас есть много элементов, события на которых нужно обрабатывать похожим образом, то вместо того, чтобы назначать обработчик каждому, мы ставим один обработчик на их общего предка.

Из него можно получить целевой элемент event.target, понять на каком именно потомке произошло событие и обработать его.

Пример - ТЫК

Наша задача – реализовать подсветку ячейки <td> при клике.

Вместо того, чтобы назначать обработчик onclick для каждой ячейки <td> (их может быть очень много) – мы повесим «единый» обработчик на элемент <table>.

Он будет использовать event.target, чтобы получить элемент, на котором произошло событие, и подсветить его.

Такому коду нет разницы, сколько ячеек в таблице. Мы можем добавлять, удалять <td> из таблицы динамически в любое время, и подсветка будет стабильно работать.

Разберём пример:

1. Метод elem.closest(selector) возвращает ближайшего предка, соответствующего селектору. В данном случае нам нужен <td>, находящийся выше по дереву от исходного элемента.
2. Если event.target не содержится внутри элемента <td>, то вызов вернет null, и ничего не произойдёт.
3. Если таблицы вложенные, event.target может содержать элемент <td>, находящийся вне текущей таблицы. В таких случаях мы должны проверить, действительно ли это <td> нашей таблицы.
4. И если это так, то подсвечиваем его.

В итоге мы получили короткий код подсветки, быстрый и эффективный, которому совершенно не важно, сколько всего в таблице <td>.

```
<table>
  <tr>
    <th colspan="3">Квадрат <em>Bagua</em>: Направление, Элемент, Цвет, Значение</th>
  </tr>
  <tr>
    <td>...<strong>Северо-Запад</strong>...</td>
    <td>...</td>
    <td>...</td>
  </tr>
  <tr>...ещё 2 строки такого же вида...</tr>
  <tr>...ещё 2 строки такого же вида...</tr>
</table>
```

```
<script>
  let table = document.getElementById('bagua-table');

  let selectedTd;

  table.onclick = function(event) {
    let target = event.target;

    while (target != this) {
      if (target.tagName == 'TD') {
        highlight(target);
        return;
      }
      target = target.parentNode;
    }
  }

  function highlight(node) {
    if (selectedTd) {
      selectedTd.classList.remove('highlight');
    }
    selectedTd = node;
    selectedTd.classList.add('highlight');
  }
</script>
```

Приём проектирования «поведение»

Делегирование событий можно использовать для добавления элементам «поведения» (behavior), декларативно задавая хитрые обработчики установкой специальных HTML-атрибутов и классов.

Приём проектирования «поведение» состоит из двух частей:

Элементу ставится пользовательский атрибут, описывающий его поведение.

При помощи делегирования ставится обработчик на документ, который ловит все клики (или другие события) и, если элемент имеет нужный атрибут, производит соответствующее действие.

Поведение: «Счётчик»

Если нажать на кнопку – значение увеличится. Конечно, нам важны не счётчики, а общий подход, который здесь продемонстрирован.

Элементов с атрибутом data-counter может быть сколько угодно. Новые могут добавляться в HTML-код в любой момент. При помощи делегирования мы фактически добавили новый «псевдостандартный» атрибут в HTML, который добавляет элементу новую возможность («поведение»).

```
Счётчик: <input type="button" value="1" data-counter>
Ещё счётчик: <input type="button" value="2" data-counter>

<script>
  document.addEventListener('click', function(event) {

    if (event.target.dataset.counter != undefined) { // если есть атрибут
      event.target.value++;
    }

  });
</script>
```

Поведение: «Переключатель» (Toggler)

Ещё один пример поведения. Сделаем так, что при клике на элемент с атрибутом data-toggle-id будет скрываться/показываться элемент с заданным id:

```
<button data-toggle-id="subscribe-mail">
  Показать форму подписки
</button>

<form id="subscribe-mail" hidden>
  Ваша почта: <input type="email">
</form>

<script>
  document.addEventListener('click', function(event) {
    let id = event.target.dataset.toggleId;
    if (!id) return;

    let elem = document.getElementById(id);

    elem.hidden = !elem.hidden;
  });
</script>
```

Действия браузера по умолчанию

Многие события автоматически влекут за собой действие браузера.

Например:

Клик по ссылке инициирует переход на новый URL.

Нажатие на кнопку «отправить» в форме – отсылку её на сервер.

Нажатие кнопки мыши над текстом и её движение в таком состоянии – инициирует его выделение.

Если мы обрабатываем событие в JavaScript, то зачастую такое действие браузера нам не нужно. К счастью, его можно отменить.

- Отмена действия браузера
- Опция «passive» для обработчика
- event.defaultPrevented

Отмена действия браузера

Есть два способа отменить действие браузера:

- Основной способ – это воспользоваться объектом event. Для отмены действия браузера существует стандартный метод event.preventDefault().
- Если же обработчик назначен через on<событие> (не через addEventListener), то также можно вернуть false из обработчика.

```
<a href="/" onclick="return false">Нажми здесь</a>
или
<a href="/" onclick="event.preventDefault()">здесь</a>
```

Опция «passive» для обработчика - оптимизация

Необязательная опция passive: true для addEventListener сигнализирует браузеру, что обработчик не собирается выполнять preventDefault().

Почему это может быть полезно?

Есть некоторые события, как touchmove на мобильных устройствах (когда пользователь перемещает палец по экрану), которое по умолчанию начинает прокрутку, но мы можем отменить это действие, используя preventDefault() в обработчике.

Поэтому, когда браузер обнаружит такое событие, он должен для начала запустить все обработчики и после, если preventDefault не вызывается нигде, он может начать прокрутку. Это может вызвать ненужные задержки в пользовательском интерфейсе.

Опция passive: true сообщает браузеру, что обработчик не собирается отменять прокрутку. Тогда браузер начинает её немедленно, обеспечивая максимально плавный интерфейс, параллельно обрабатывая событие.

Для некоторых браузеров (Firefox, Chrome) опция passive по умолчанию включена в true для таких событий, как touchstart и touchmove.

event.defaultPrevented - знание сила!

event.defaultPrevented — это свойство объекта события (event object) в JavaScript, которое указывает, было ли предотвращено стандартное поведение события. Оно возвращает true, если вызван event.preventDefault(), и false в противном случае.

event.defaultPrevented позволяет разработчикам определить, было ли стандартное поведение события отменено другим обработчиком. Это может быть полезно, когда у вас есть несколько обработчиков для одного и того же события, и вам нужно принять решение на основе того, было ли предотвращено стандартное поведение.

Когда вызывается метод event.preventDefault(), свойство event.defaultPrevented устанавливается в true. Это позволяет другим обработчикам событий проверить, было ли отменено стандартное поведение и соответствующим образом отреагировать.

Практическое применение:

- Множественные обработчики событий: При наличии нескольких обработчиков на одном элементе проверка event.defaultPrevented может помочь избежать ненужных действий или дублирования логики.
- Библиотеки и фреймворки: Многие библиотеки и фреймворки используют это свойство для управления поведением событий, обеспечивая гибкость и контроль для разработчиков.
- Пользовательские события: При создании пользовательских событий и их обработчиков, event.defaultPrevented позволяет контролировать и изменять поведение на основе пользовательского ввода или других условий.

Ресурсы

Введение в события - ТЫК

Справочник по событиям - ТЫК

MDN документация метода EventTarget.addEventListener() - ТЫК

event MDN документация - ТЫК

Всплытие и погружение - ТЫК

Делегирование событий - ТЫК

Действия браузера по умолчанию - ТЫК