Всплытие и погружение

Давайте начнём с примера.

Этот обработчик для `<div>` сработает, если вы кликните по любому из вложенных тегов, будь то `<em>` или `<code>`:

```
<div onclick="alert('Обработчик!')">
  <em>Если вы кликните на <code>EM</code>, сработает об
</div>
```

Если вы кликните на `EM`, сработает обработчик на `DIV`

Вам не кажется это странным? Почему же сработал обработчик на `<div>`, если клик произошёл на `<em>`?

Всплытие

Принцип всплытия очень простой.

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

Например, есть 3 вложенных элемента `FORM > DIV > P` с обработчиком на каждом:

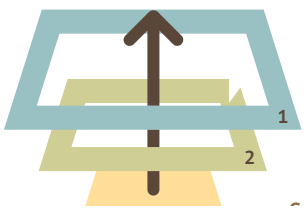
```
<style>
  body * {
    margin: 10px;
    border: 1px solid blue;
  }
</style>

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

FORM
DIV
P

Клик по внутреннему `<p>` вызовет обработчик `onclick`:

1. Сначала на самом `<p>`.
2. Потом на внешнем `<div>`.
3. Затем на внешнем `<form>`.
4. И так далее вверх по цепочке до самого `document`.



Поэтому если кликнуть на `<p>`, то мы увидим три оповещения: `p` → `div` → `form`.

Этот процесс называется «всплытием», потому что события «всплывают» от внутреннего элемента вверх через родителей подобно тому, как всплывает пузырёк воздуха в воде.

> ***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.

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 всплывёт 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.

Попробуйте сами:

Результат script.js example.css index.html

Клик покажет оба: и `event.target`, и `this` для сравнения:

FORM
DIV

P

Возможна и ситуация, когда `event.target` и `this` – один и тот же элемент, например, если клик был непосредственно на самом элементе `<form>`, а не на его подэлементе.

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()`.

Например, здесь при клике на кнопку `<button>` обработчик `body.onclick` не сработает:

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

Кликни меня

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:

1. Мы делаем вложенное меню. Каждое подменю обрабатывает клики на своих элементах и делает для них `stopPropagation`, чтобы не срабатывало внешнее меню.
2. 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.
3. Наша аналитика не будет работать над областью, где клики прекращаются `stopPropagation`. Увы, получилась «мёртвая зона».

Зачастую нет никакой необходимости прекращать всплытие. Задача, которая, казалось бы, требует этого, может быть решена иначе. Например, с помощью создания своего уникального события, о том, как это делать, мы поговорим позже. Также мы можем записывать какую-то служебную информацию в объект `event` в одном обработчике, а читать в другом, таким образом мы можем сообщить обработчикам на родительских элементах информацию о том, что событие уже было как-то обработано.

Погружение

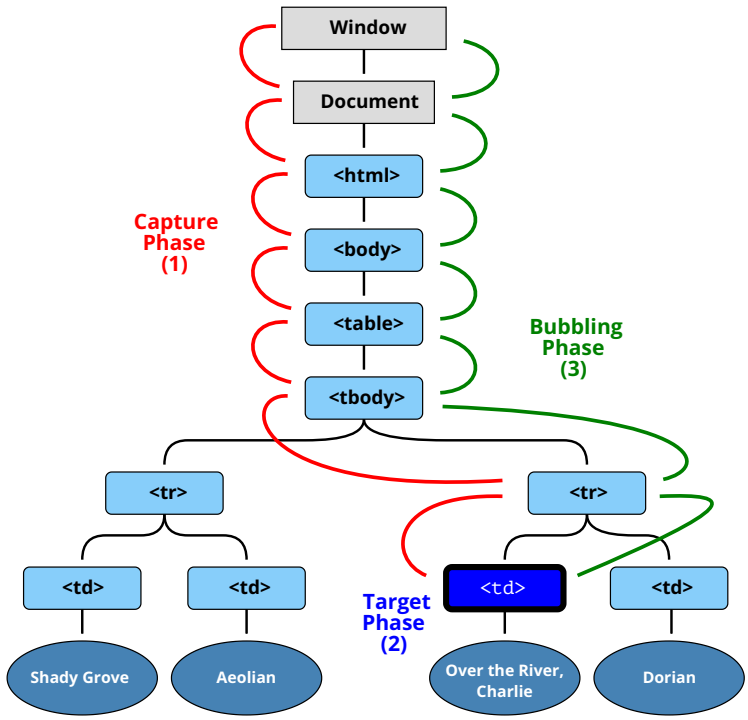
Существует ещё одна фаза из жизненного цикла события – «погружение» (иногда её называют «перехват»). Она очень редко используется в реальном коде, однако тоже может быть полезной.

Стандарт DOM Events описывает 3 фазы прохода события:

1. Фаза погружения (capturing phase) – событие сначала идёт сверху вниз.

2. Фаза цели (target phase) – событие достигло целевого(исходного) элемента.
3. Фаза всплытия (bubbling stage) – событие начинает всплывать.

Картинка из спецификации демонстрирует, как это работает при клике по ячейке `<td>`, расположенной внутри таблицы:



То есть при клике на `<td>` событие путешествует по цепочке родителей сначала вниз к элементу (погружается), затем оно достигает целевой элемент (фаза цели), а потом идёт наверх (всплытие), вызывая по пути обработчики.

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` вот так:

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

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.

Обратите внимание, что хоть и формально существует 3 фазы, 2-ую фазу («фазу цели»: событие достигло элемента) нельзя обработать отдельно, при её достижении вызываются все обработчики: и на всплытие, и на погружение.

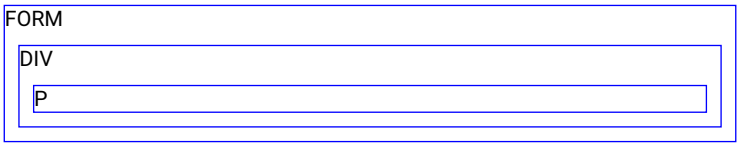
Давайте посмотрим и всплытие и погружение в действии:

```
<style>
  body * {
    margin: 10px;
```

```
    border: 1px solid blue;
  }
</style>

<form>FORM
  <div>DIV
    <p>P</p>
  </div>
</form>

<script>
  for(let elem of document.querySelectorAll('*')) {
    elem.addEventListener("click", e => alert(`Погружен
    elem.addEventListener("click", e => alert(`Всплытие
  }
</script>
```



Здесь обработчики навешиваются на *каждый* элемент в документе, чтобы увидеть в каком порядке они вызываются по мере прохода события.

Если вы кликните по `<p>`, то последовательность следующая:

1. `HTML` → `BODY` → `FORM` → `DIV` (фаза погружения, первый обработчик)
2. `P` (фаза цели, срабатывают обработчики, установленные и на погружение и на всплытие, так что выведется два раза)
3. `DIV` → `FORM` → `BODY` → `HTML` (фаза всплытия, второй обработчик)

Существует свойство `event.eventPhase`, содержащее номер фазы, на которой событие было поймано. Но оно используется редко, мы обычно и так знаем об этом в обработчике.

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:

```
elem.addEventListener("click", e => alert(1)); //
elem.addEventListener("click", e => alert(2));
```

Итого

При наступлении события – самый глубоко вложенный элемент, на котором оно произошло, помечается как «целевой» (`event.target`).

- Затем событие сначала двигается вниз от корня документа к `event.target`, по пути вызывая обработчики, поставленные через `addEventListener(...., true)`, где `true` – это сокращение для `{capture: true}`.
- Далее обработчики вызываются на целевом элементе.

- Далее событие двигается от `event.target` вверх к корню документа, по пути вызывая обработчики, поставленные через `on<event>` и `addEventListener` без третьего аргумента или с третьим аргументом равным `false`.

Каждый обработчик имеет доступ к свойствам события `event`:

- `event.target` – самый глубокий элемент, на котором произошло событие.
- `event.currentTarget` (`= this`) – элемент, на котором в данный момент сработал обработчик (тот, на котором «висит» конкретный обработчик)
- `event.eventPhase` – на какой фазе он сработал (погружение=1, фаза цели=2, всплытие=3).

Любой обработчик может остановить событие вызовом `event.stopPropagation()`, но делать это не рекомендуется, так как в дальнейшем это событие может понадобиться, иногда для самых неожиданных вещей.

В современной разработке стадия погружения используется очень редко, обычно события обрабатываются во время всплытия. И в этом есть логика.

В реальном мире, когда происходит чрезвычайная ситуация, местные службы реагируют первыми. Они знают лучше всех местность, в которой это произошло, и другие детали. Вышестоящие инстанции подключаются уже после этого и при необходимости.

Тоже самое справедливо для обработчиков событий. Код, который «навесил» обработчик на конкретный элемент, знает максимум деталей об элементе и его предназначении. Например, обработчик на определённом `<td>` скорее всего подходит только для этого конкретного `<td>`, он знает все о нём, поэтому он должен отработать первым. Далее имеет смысл передать обработку события родителю – он тоже понимает, что происходит, но уже менее детально, далее – выше, и так далее, до самого объекта `document`, обработчик на котором реализовывает самую общую функциональность уровня документа.

Всплытие и погружение являются основой для «делегирования событий» – очень мощного приёма обработки событий. Его мы изучим в следующей главе.

Комментарии

перед тем как писать...