Работа с событиями

Прерывание распространения события

В процессе разработки иногда возникает задача не вызывать часть слушателей событий при некотором условии. Конечно, можно добавить проверку условия в код каждого обработчика, но лучшим решением станет добавление ещё одного слушателя перед остальными или в более раннюю фазу захвата. В нём мы будем проверять условие и останавливать дальнейшее распространение события. Для этого у события есть два метода: **Event.stopPropagation()** и **Event.stopImmediatePropagation()**.

Первый метод остановит дальнейшее распространение события вверх по DOM-дереву в текущей и последующих фазах.

В дополнение к этому, если несколько обработчиков прикреплены к одному и тому же элементу с одинаковым типом события, они вызываются в порядке своего добавления. Если один из этих обработчиков вызовет **event.stopImmediatePropagation(),** то события оставшихся обработчиков вызваны не будут.

Рассмотрим пример:

```
<form>
  <label><input type="radio" name="propagation-control"
value="stopPropagation">Stop propagation</label>
  <label><input type="radio" name="propagation-control"
value="stopImmediatePropagation">Stop Immediate propagation</label>
</form>

<ul>
  <li>Первый</li>
  <li>Второй</li>
  <li>Третий</li>
</ul>

<script>
  const form = document.forms[0]
  const list = document.querySelector('ul')
  list.addEventListener('click', (e) => {
    console.log('В первом UL в фазе захвата')
    const propagationControlMethodName =
form.elements['propagation-control'].value
    if (propagationControlMethodName) {
      e[propagationControlMethodName]()
    }
  }, true)
  list.addEventListener('click', () => {
    console.log('Во втором UL в фазе захвата')
```

```
  }, true)
  list.addEventListener('click', () => {
    console.log('В первом UL в фазе всплытия')
  })
  Array.from(list.children).forEach((child) => {
    child.addEventListener('click', () => {
      console.log('В каждом LI в фазе всплытия')
    })
  })
</script>
```

Выбор одного из переключателей (радиоинпутов) ведёт к тому, что обработчики, назначенные на элементы списка и на сам список в фазе всплытия, перестают срабатывать при клике.

Если выбирается stopImmediatePropagation, то перестаёт срабатывать и второй capture-слушатель списка.

Действия по умолчанию и отменяемые события

События обычно отправляются браузерной реализацией DOM в результате действий пользователя в ответ на завершение задачи или для сигнализации прогресса во время асинхронной активности, например, сетевого запроса.

Некоторые события используются для управления поведением, которое реализация может предпринять в следующий раз. Или это происходит для отмены действия, уже предпринятого реализацией. События в этой категории называются отменяемыми, а поведение, которое они отменяют, — действием по умолчанию. Отменяемые объекты событий часто связаны с одним или несколькими действиями по умолчанию. Чтобы отменить действие по умолчанию, надо вызвать метод preventDefault() на объекте события.

Например, событие mousedown отправляются сразу после того, как пользователь нажимает кнопку мыши или другого указательного устройства. Одним из возможных действий по умолчанию, предпринимаемых реализацией, считается переход в режим. Это действие позволяет пользователю перетаскивать изображения или выбирать текст.

Действие по умолчанию зависит от сопутствующих условий. Например, если указывающее устройство пользователя находится над текстом, начнётся выделение текста. Если указывающее устройство пользователя располагается над изображением, произойдёт действие перетаскивания изображения. Отмена действия по умолчанию для mousedown-события предотвращает выполнение этих действий.

Действия по умолчанию обычно выполняются после завершения отправки события, но в исключительных случаях они также могут выполняться прямо перед отправкой события.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Привет, DOM-события!</title>
  </head>
  <body>
    <form name="myForm">
      <label>Тестовый checkbox: <input type="checkbox"></label>
    </form>
  </body>
</html>
```

Действие по умолчанию, связанное с событием **click** для элементов <input type="checkbox">, переключает значение атрибута **checked** этого элемента. Если действие **click** по умолчанию для события отменяется, значение возвращается в прежнее состояние.

```
const checkbox = document.querySelector('input[type=checkbox]')

checkbox.addEventListener('click', (event) => {
  console.log(event.target.checked) // true
  event.preventDefault()
})
```

Когда событие отменяется, условные действия по умолчанию, связанные с событием, пропускаются. Или, как упомянуто выше, если действия по умолчанию выполняются до отправки, их эффект отменяется. Возможность отмены действия по умолчанию указывается в атрибуте cancelable объекта события. Вызов **Event.preventDefault()** останавливает все связанные действия по умолчанию объекта события. Атрибут **Event.defaultPrevented** указывает, отменялось ли уже это событие, например, через предварительного слушателя событий.

Многие реализации дополнительно интерпретируют возвращаемое из слушателя событий значение, например, **false**. Это означает, что действие по умолчанию для отменяемых событий отменится, хотя обработчики **window.onerror** отменяются возвратом **true**.

```
const checkbox = document.querySelector('input[type=checkbox]')

checkbox.addEventListener('click', (event) => {
  console.log(event.target.checked) // true
  return false
})
```

Генерация событий, пользовательские события

Помимо подписки на встроенные события мы можем сами создавать уникальные пользовательские и встроенные события на DOM-элементе. Это полезно при реализации компонентов пользовательского интерфейса или для задач тестирования.

Объект события создаётся с применением базового конструктора Event или одного из наиболее подходящих конструкторов из иерархии классов событий.

Генерация встроенных событий

Следующий код демонстрирует генерацию события click на DOM-элементе.

```
<button>Кликни меня!</button>

<script>
  const eventOptions = {bubbles: true, cancelable: true}
  const event = new Event('click', eventOptions)
  event.view = window

  const mouseEvent = new MouseEvent('click', {
    ...eventOptions,
    view: window,
  })

  document.addEventListener('click', (event) => {
    console.log(event.isTrusted)
  })

  const button = document.querySelector('button')
  button.dispatchEvent(event)
  button.dispatchEvent(mouseEvent)
  button.click()
</script>
```

Специализированные конструкторы типа MouseEvent, в отличие от Event, поддерживают такие дополнительные поля, как **view**, **clientX**, **clientY** и другие. В случае с конструктором Event, который эти поля не поддерживает, потребуется дополнительный код для их добавления в объект события.

Встроенные события также генерируются методами **HTMLElement.click()**, **HTMLElement.focus()** и другими.

Чтобы сгенерированное событие всплывало и было отменяемым, надо передать в опции поля **bubbles** и **cancelable**.

Отличить сгенерированное событие от собственного можно по полю **isTrusted** в обработчике события.

Важно! *Обработчики сгенерированных событий выполняются синхронно.*

В отличие от собственных событий, которые запускаются реализацией DOM и вызывают обработчики событий асинхронно через цикл событий (event loop), **dispatchEvent()** вызывает обработчики событий синхронно. Все применимые обработчики событий будут выполняться и возвращаться до того, как код продолжится после вызова **dispatchEvent()**.

Генерация пользовательских событий

Как и встроенные события, пользовательские генерируются с применением базового класса Event или более специализированного класса CustomEvent. Единственное отличие между ними заключается в поддержке дополнительного поля **detail**:

```
<button>Click me!</button>

<script>
  const eventOptions = {bubbles: true, cancelable: true}
  const event = new Event('foo', eventOptions)
  event.detail = {text: 'Произвольный текст'}

  const customEvent = new CustomEvent('foo', {
    ...eventOptions,
    detail: {text: 'Произвольный текст'},
  })

  document.addEventListener('foo', (event) => {
    console.log(event.detail)
  })

  const button = document.querySelector('button')
  button.dispatchEvent(event)
  button.dispatchEvent(customEvent)
</script>
```

Отмена действия по умолчанию в пользовательских событиях

Как и встроенные события, пользовательские поддерживают отмену действий по умолчанию с применением опций конструктора события **cancelable** и вызова **Event.preventDefault()**. Код, который создаёт событие, сам решает, что делать, если один из обработчиков вызовет **Event.preventDefault()**. Для этого используется результат вызова метода **dispatchEvent().**

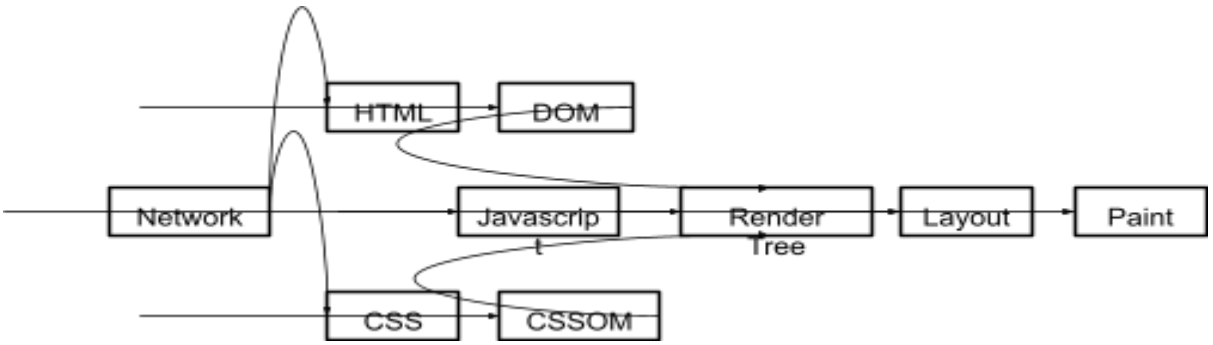
```
<pre id="rabbit">
  |\   /|
   \|_|/
   /. .\
  =\_Y_/=
   {>o<}
```

```
</pre>

<button>Спрятать кролика</button>

<script>
  const rabbit = document.querySelector('#rabbit')
  const button = document.querySelector('button')

  rabbit.addEventListener('hide', (event) => {
    if (confirm('Вызвать event.preventDefault()?')) {
      event.preventDefault()
    }
  })

  button.addEventListener(() => {
    const event = new CustomEvent('hide', {cancelable: true})
    const defaultPrevented = !rabbit.dispatchEvent(event)
    if (defaultPrevented) {
      console.log('Отменено обработчиком события');
    } else {
      rabbit.hidden = true
    }
  }
</script>
```

Важно, чтобы вызов конструктора события **hide** содержал флаг **cancelable: true**. В противном случае вызов **Event.preventDefault()** будет проигнорирован.

Как браузер использует DOM

До этого раздела мы рассматривали работу с DOM только с точки зрения пользователя DOM API для JavaScript. Для более глубокого понимания объектной модели документа (DOM) и её роли разберём, какие шаги проходит браузер при отображении веб-страницы после получения ответа от сервера или из локального файла на компьютере и до отрисовки пикселей на нашем экране. Эта последовательность шагов называется **критическим путём рендеринга** (Critical Rendering Path).



Он разбивается на два больших этапа:

1. Анализ кода и построение модели визуализации (render tree):
    - Получение HTML-документа по сети или чтение с локального диска.
    - Обработка HTML-кода и создание DOM, отправка дополнительных запросов.
    - Обработка CSS-стилей и создание CSSOM.
    - Выполнение синхронного JavaScript-кода.
2. Прямая отрисовка страницы:
    - Создание макета страницы (Layout).
    - Растеризация или визуализация страницы на экране (Painting).

Построение модели визуализации (Render tree)

Дерево рендера, или модель визуализации, — это специальная структура, состоящая из элементов, которые отобразятся на странице, и связанных с ними стилей. Такое дерево собирается из двух независимых структур данных (моделей):

1. DOM — объектная модель всех элементов страницы. Собирается на основе HTML-разметки страницы.
2. CSSOM — объектная модель таблицы стилей страницы. Собирается на основе CSS-разметки страницы.

Важно! *Модель визуализации, состоящая из DOM и CSSOM, содержит только видимые элементы и не включает скрытые. Например, те, в стилях которых указывается display: none.*

Построение DOM-модели

Построение DOM-модели делится на следующие этапы:

1. **Преобразование.** Браузер преобразует байты из HTML-файла, размещённого на диске или в сети, в символы, основываясь на приведённой в файле кодировке (например, UTF-8).
2. **Разметка.** На основании стандарта W3C HTML5 браузер выделяет среди символов теги в угловых скобках, такие как <html> и <body>. У каждого тега есть своё значение и набор правил.

3. **Создание объектов.** HTML-тегами браузер выделяет в документе объекты с конкретными свойствами.

4. **Формирование DOM.** Объекты образуют древовидную структуру, повторяющую иерархию HTML-файла, в котором одни теги помещаются в другие. Так, объект p помещается под body, а объект body, в свою очередь, — под html, и так далее.

В результате создаётся объектная модель документа (DOM), которую использует браузер, чтобы продолжить обрабатывать страницу.

Итак, модель DOM готова. Но чтобы вывести страницу на экран, одной структуры объектов недостаточно. Браузер также определяет, как эти объекты выглядят.

Рассмотрим следующий этап — формирование объектной модели, относящейся к таблице стилей (CSSOM).

Построение CSSOM-модели

Объектная модель стилей (CSSOM) так же, как DOM, представлена в виде дерева объектов, со связанными стилями для каждого узла, независимо от того, объявлены они явно или унаследованы неявно.

При схожести в обработке кода HTML и CSS стоит упомянуть, что CSS — блокирующий рендеринг ресурс. Это означает, что модель визуализации не может быть построена до полной обработки всех стилей на странице из-за особенностей их (стилей) каскадной реализации. Стили, указанные далее в документе, переопределяют те, что определены до них. CSS также блокирует выполнение JavaScript-кода, который ждёт формирования CSSOM.

Выполнение JavaScript

JavaScript-код, размещённый на странице, блокирует процесс обработки самого HTML-документа. Когда браузер встречает тег <script> вне зависимости, считается ли он внешним, указывая на внешний файл в атрибуте src, или внутренним, включая JavaScript-код, то останавливается для загрузки (если внешний) и выполнения. Поэтому, если на странице есть JavaScript-код, ссылающийся на элементы в DOM, желательно размещать его ближе к концу документа.

Чтобы избежать блокирования обработки браузером документа, внешние JavaScript-теги помечаются как асинхронные путём добавления атрибута async: <script async src="script.js">.

Мы узнали, как на основании файлов HTML и CSS строятся модели DOM и CSSOM. Эти независимые друг от друга модели отвечают за разные аспекты страницы: DOM описывает контент, а CSSOM — стили, которые к нему применяются. Рассмотрим, как браузер объединяет эти модели и выводит страницу на экран.

Для формирования модели визуализации браузер выполняет следующие действия:

1. Начиная с основания модели DOM, находит все видимые объекты.
    a. Этот этап не затрагивает элементы, которые не будут видны на странице, например, теги скриптов, метатеги и т. д.
    b. Он не затрагивает объекты, помеченные как невидимые, используя CSS.
2. Находит в CSSOM наборы стилей и присваивает их соответствующим объектам.
3. Формирует модель из видимых объектов, их содержания и стилей.

Создав модель визуализации, браузер на шаг приблизился к выводу страницы на экран! Теперь можно приступить к формированию макета.

Создание макета (Layout)

Макет — то, что определяет размер области просмотра и обеспечивает контекст для стилей CSS, которые зависят от него, например, проценты или единицы просмотра. Размер области просмотра определяется метатегом области просмотра, указанным в заголовке документа, или, если тег не указан, используется ширина области просмотра, по умолчанию равная 980 пикселям.

Наиболее распространённое значение метаобласти просмотра — размер области просмотра, установленный в соответствии с шириной устройства:

```
<meta name="viewport" content="width=device-width,initial-scale=1">
```

Если пользователь посещает веб-страницу на устройстве шириной 1 000 пикселей, размеры будут основаны на этой единице. Половина области просмотра станет равной 500 пикселям, 10vw — 100 пикселям, и так далее.

Итак, браузер уже определил:

- какие объекты будут видны на странице;
- какие стили надо им присвоить.

Пришло время создать макет, то есть выяснить:

- какого размера окажутся объекты;
- как их надо расположить в области просмотра.

Для этого браузер вычислит геометрическую форму объектов, проанализировав модель визуализации с самого начала. Рассмотрим простой пример:

```
<!DOCTYPE html>
<html>
  <head>
    <meta name="viewport" content="width=device-width,initial-scale=1">
    <title>Здравствуй, мир!</title>
  </head>
  <body>
    <div style="width: 50%">
      <div style="width: 50%">Здравствуй, мир!</div>
    </div>
  </body>
</html>
```

В теле этой страницы есть два блока div. Ширина родительского блока — 50% от области просмотра, а вложенного — 50% от родительского, то есть 25% экрана.

Сформировав макет, браузер получает блочную модель, точно отражающую расположение и размер каждого объекта в области просмотра. Все относительные показатели преобразуются в абсолютное положение пикселей на экране.

Наконец, когда браузеру станет известно, какие объекты отобразятся на странице, где их разместить и какие стили им надо присвоить, можно приступать к следующему этапу — выводу страницы на экран. Этот этап также называется визуализацией или растеризацией.

Растеризация (Painting)

Наконец, на этапе растеризации видимое содержимое страницы преобразуется в пиксели для отображения на экране.

То, сколько времени занимает этап рисования, зависит от размера модели DOM, а также от применяемых стилей. Воплощать одни стили труднее, другие — проще. Например, отрисовка сложного градиентного фонового изображения требует больше времени, чем простая заливка фона одним цветом.