

JavaScript - файлы

Загрузка документа и ресурсов;
Бинарные данные и файлы.

Загрузка документа и ресурсов

- Страница: DOMContentLoaded, load, beforeunload, unload
- Скрипты: async, defer
- Загрузка ресурсов: onload и onerror

Жизненный цикл

Жизненный цикл — это период времени, на протяжении которого происходит последовательность изменений и стадий в развитии чего-либо.

В контексте HTML и веб-разработки, жизненный цикл относится к этапам создания, загрузки, взаимодействия и удаления веб-страницы или элементов на странице. Этот процесс управляет браузером и JavaScript-кодом, который разработчик может написать для управления этими этапами.

1. Создание документа

|
v

2. Загрузка и парсинг

|
v

3. Загрузка внешних ресурсов

|
v

4. Создание визуального дерева

|
v

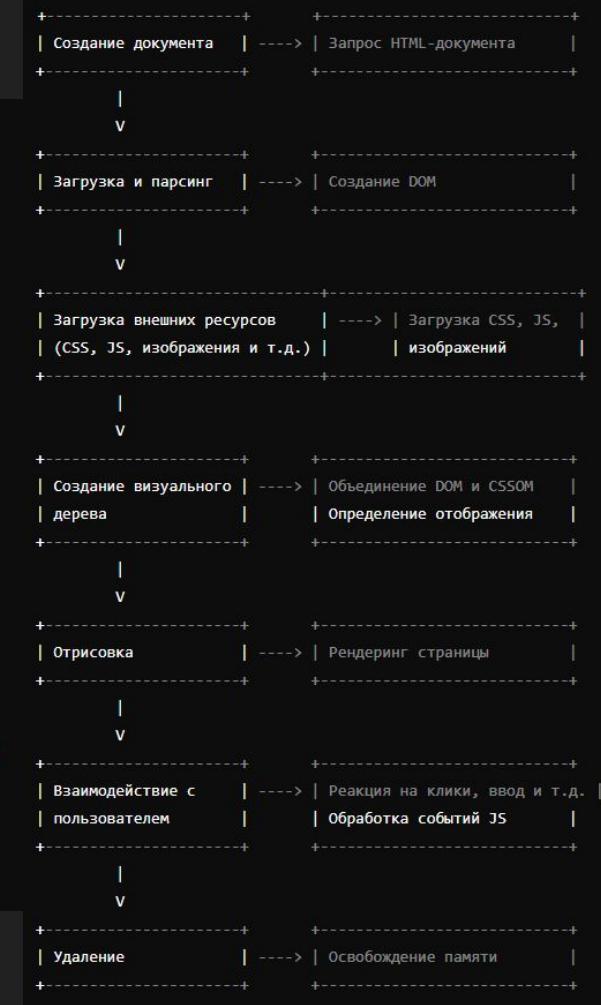
5. Отрисовка

|
v

6. Взаимодействие с пользователем

|
v

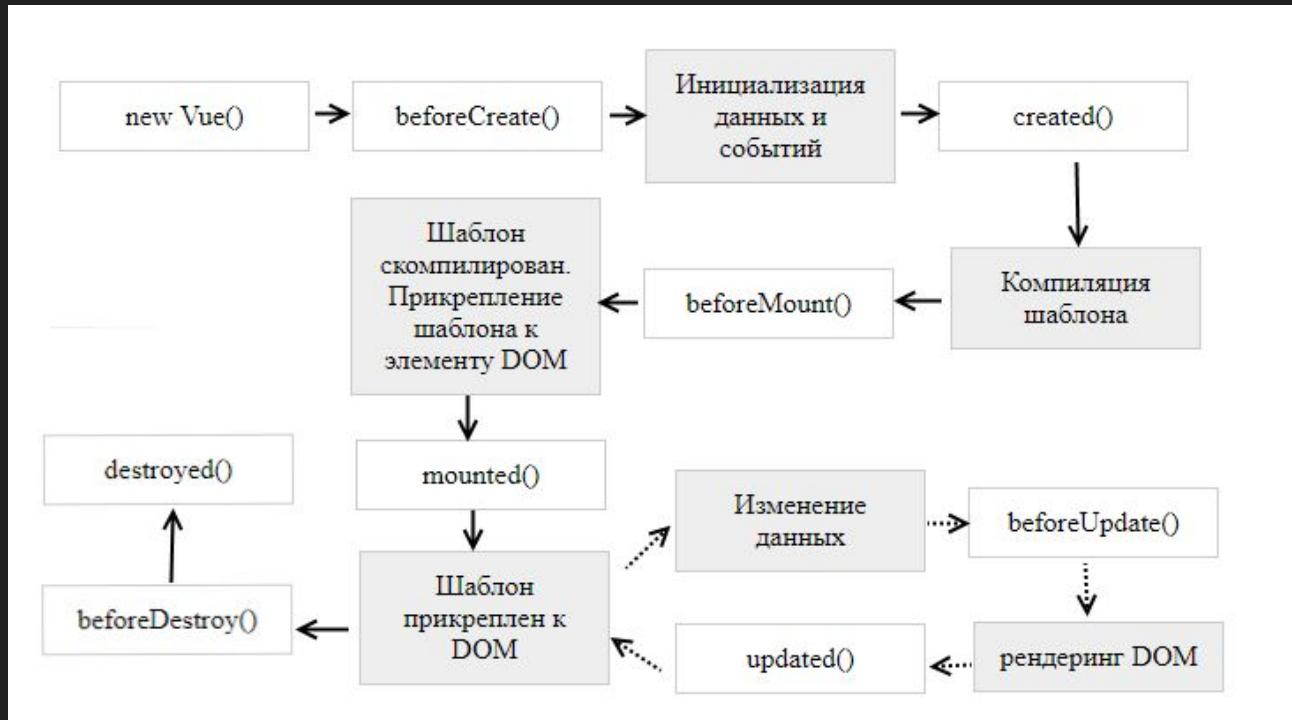
7. Удаление



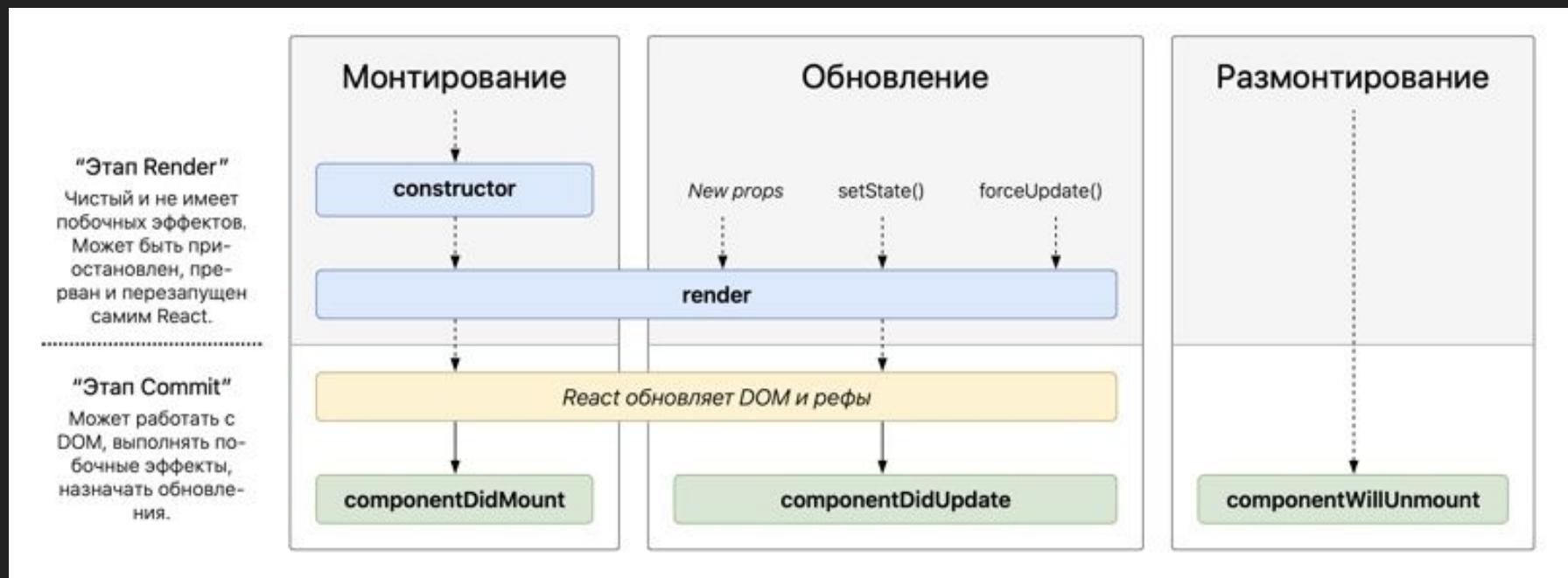
Этапы жизненного цикла HTML-документа (теория)

- 1. Создание документа:**
 - Когда пользователь переходит на веб-страницу, браузер запрашивает HTML-документ с сервера.
- 2. Загрузка и парсинг:**
 - Браузер загружает HTML-документ и начинает его парсить (читать и интерпретировать).
 - HTML разбивается на элементы, создаются DOM (Document Object Model) объекты.
- 3. Загрузка внешних ресурсов:**
 - Браузер загружает CSS, JavaScript, изображения и другие ресурсы, указанные в HTML-документе.
- 4. Создание визуального дерева:**
 - Браузер создает визуальное дерево, соединяя DOM и CSSOM (CSS Object Model), чтобы определить, как элементы будут отображаться на экране.
- 5. Отрисовка:**
 - Браузер отрисовывает страницу на экране пользователя.
- 6. Взаимодействие с пользователем:**
 - Пользователь взаимодействует с веб-страницей (клики, ввод данных и т.д.).
 - JavaScript может изменять DOM и CSSOM в ответ на действия пользователя.
- 7. Удаление:**
 - Когда пользователь закрывает страницу или переходит на другую, браузер удаляет все элементы, связанные с текущей страницей, освобождая память.

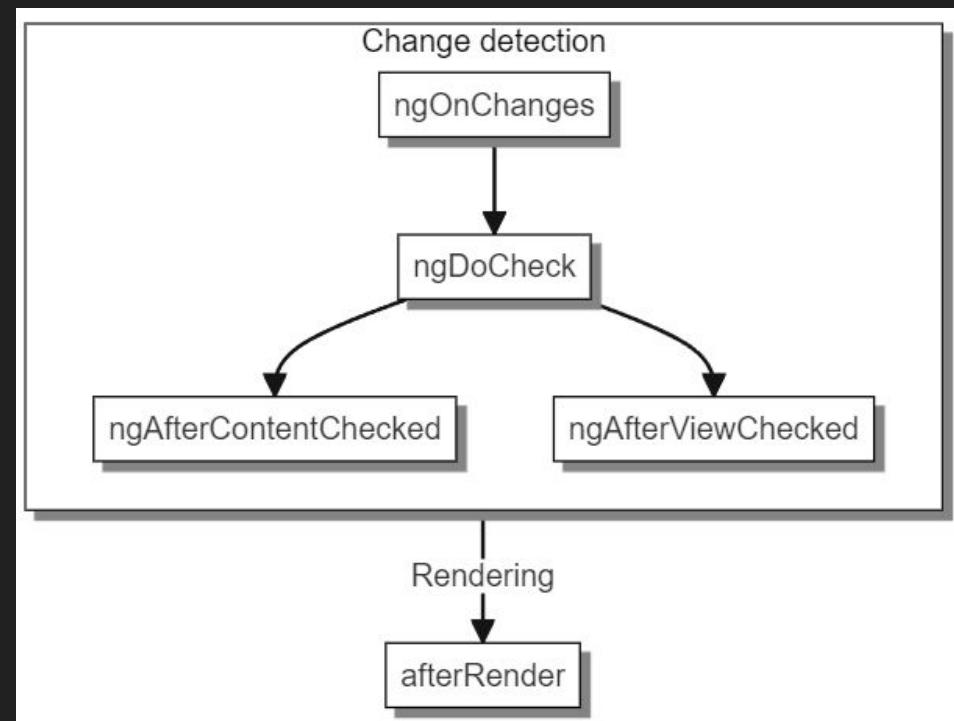
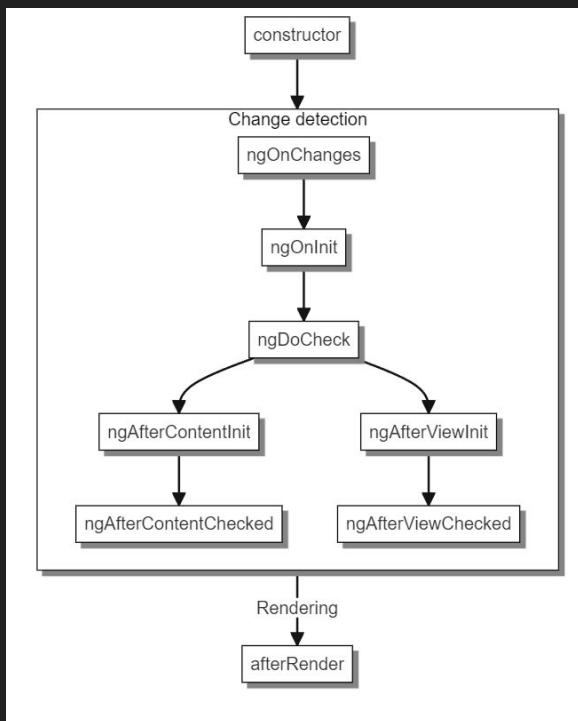
Пример жизненного цикла компонента в VUE JS



Пример жизненного цикла компонента React js



Пример жизненного цикла компонента Angular - ТЫК



Страница: DOMContentLoaded, load, beforeunload, unload

У жизненного цикла HTML-страницы есть три важных события:

- DOMContentLoaded – браузер полностью загрузил HTML, было построено DOM-дерево, но внешние ресурсы, такие как картинки `` и стили, могут быть ещё не загружены.
- load – браузер загрузил HTML и внешние ресурсы (картинки, стили и т.д.).
- beforeunload/unload – пользователь покидает страницу.

Каждое из этих событий может быть полезно:

- Событие DOMContentLoaded – DOM готов, так что обработчик может искать DOM-узлы и инициализировать интерфейс.
- Событие load – внешние ресурсы были загружены, стили применены, размеры картинок известны и т.д.
- Событие beforeunload – пользователь покидает страницу. Мы можем проверить, сохранил ли он изменения и спросить, на самом ли деле он хочет уйти.
- unload – пользователь почти ушёл, но мы всё ещё можем запустить некоторые операции, например, отправить статистику.

DOMContentLoaded

Событие DOMContentLoaded срабатывает на объекте document. Мы должны использовать addEventListener, чтобы поймать его.

```
document.addEventListener("DOMContentLoaded", ready);
// не "document.onDOMContentLoaded = ..."
```

```
<script>
  document.addEventListener("DOMContentLoaded", () => {
    alert("DOM готов!");
  });
</script>

<script src="https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.3.0/lodash.js"></script>

<script>
  alert("Библиотека загружена, встроенный скрипт выполнен");
</script>
```

DOMContentLoaded и скрипты

Когда браузер обрабатывает HTML-документ и встречает тег `<script>`, он должен выполнить его перед тем, как продолжить строить DOM. Это делается на случай, если скрипт захочет изменить DOM или даже дописать в него (`document.write`), так что `DOMContentLoaded` должен подождать.

Сначала увидим «Библиотека загружена...», а затем «DOM готов!» (все скрипты выполнены).

Пример

```
<script>
    function ready() {
        alert('DOM готов');

        // изображение ещё не загружено (если не было закешировано), так что размер будет 0x0
        alert(`Размер изображения: ${img.offsetWidth}x${img.offsetHeight}`);
    }

    document.addEventListener("DOMContentLoaded", ready);
</script>


```

В этом примере обработчик DOMContentLoaded запустится, когда документ загрузится, так что он увидит все элементы, включая расположенный ниже ``.

Но он не дожидается, пока загрузится изображение.
Поэтому `alert` покажет нулевой размер.

На первый взгляд событие `DOMContentLoaded` очень простое.
DOM-дерево готово – получаем событие. Хотя тут есть несколько особенностей.

Скрипты, которые не блокируют DOMContentLoaded

Есть два исключения из этого правила:

1. Скрипты с атрибутом `async` не блокируют `DOMContentLoaded`.
2. Скрипты, сгенерированные динамически при помощи `document.createElement('script')` и затем добавленные на страницу, также не блокируют это событие.

DOMContentLoaded и стили

Внешние таблицы стилей не затрагивают DOM, поэтому `DOMContentLoaded` их не ждёт.

Но здесь есть подводный камень. Если после стилей у нас есть скрипт, то этот скрипт должен дождаться, пока загрузятся стили

Причина в том, что скрипту может понадобиться получить координаты или другие свойства элементов, зависящих от стилей, как в примере выше. Естественно, он должен дождаться, пока стили загрузятся.

Так как `DOMContentLoaded` дожидается скриптов, то теперь он так же дожидается и стилей перед ними.

```
<link type="text/css" rel="stylesheet" href="style.css">
<script>
    // скрипт не выполняется, пока не загрузятся стили
    alert(getComputedStyle(document.body).marginTop);
</script>
```

Встроенное в браузер автозаполнение

Firefox, Chrome и Opera автоматически заполняют поля при наступлении `DOMContentLoaded`.

Например, если на странице есть форма логина и пароля и браузер запомнил значения, то при наступлении `DOMContentLoaded` он попытается заполнить их (если получил разрешение от пользователя).

Так что, если `DOMContentLoaded` откладывается из-за долгой загрузки скриптов, в свою очередь – откладывается автозаполнение. Вы наверняка замечали, что на некоторых сайтах (если вы используете автозаполнение в браузере) поля логина и пароля не заполняются мгновенно, есть некоторая задержка до полной загрузки страницы. Это и есть ожидание события `DOMContentLoaded`.

window.onload

Событие load на объекте window наступает, когда загрузилась вся страница, включая стили, картинки и другие ресурсы. Это событие доступно через свойство onload.

```
<script>
  window.onload = function() { // можно также использовать window.addEventListener('load', (event) => {
    alert('Страница загружена');

    // к этому моменту картинка загружена
    alert(`Image size: ${img.offsetWidth}x${img.offsetHeight}`);
  });
</script>


```

window.onunload

```
let analyticsData = { /* объект с собранными данными */ };

window.addEventListener("unload", function() {
  navigator.sendBeacon("/analytics", JSON.stringify(analyticsData));
});
```

Когда посетитель покидает страницу, на объекте window генерируется событие unload. В этот момент стоит совершать простые действия, не требующие много времени, вроде закрытия связанных всплывающих окон.

Обычно здесь отсылают статистику.

Предположим, мы собрали данные о том, как используется страница: клики, прокрутка, просмотры областей страницы и так далее.

Естественно, событие unload – это тот момент, когда пользователь нас покидает и мы хотим сохранить эти данные.

Для этого существует специальный метод navigator.sendBeacon(url, data), описанный в спецификации <https://w3c.github.io/beacon/>.

Он посылает данные в фоне. Переход к другой странице не задерживается: браузер покидает страницу, но всё равно выполняет sendBeacon.

- Отсылается POST-запрос.
- Мы можем послать не только строку, но так же формы и другие форматы, но обычно это строковый объект.
- Размер данных ограничен 64 Кб.

Если мы хотим отменить переход на другую страницу, то здесь мы этого сделать не сможем. Но сможем в другом месте – в событии onbeforeunload.

window.onbeforeunload

Если посетитель собирается уйти со страницы или закрыть окно, обработчик beforeunload попросит дополнительное подтверждение.

Если мы отменим это событие, то браузер спросит посетителя, уверен ли он.

По историческим причинам возврат непустой строки так же считается отменой события. Когда-то браузеры использовали её в качестве сообщения, но, как указывает современная спецификация, они не должны этого делать.

Поведение было изменено, потому что некоторые веб-разработчики злоупотребляли этим обработчиком события, показывая вводящие в заблуждение и надоедливые сообщения. Так что, прямо сейчас старые браузеры всё ещё могут показывать строку как сообщение, но в остальных – нет возможности настроить показ сообщения пользователям.

```
window.onbeforeunload = function() {  
    return false;  
};
```

```
window.onbeforeunload = function() {  
    return "Есть несохранённые изменения. Всё равно уходим?";  
};
```

readyState

Что произойдет, если мы установим обработчик DOMContentLoaded после того, как документ загрузился?

Естественно, он никогда не запустится.

Есть случаи, когда мы не уверены, готов документ или нет. Мы бы хотели, чтобы наша функция исполнилась, когда DOM загрузился, будь то сейчас или позже.

Свойство `document.readyState` показывает нам текущее состояние загрузки.

Есть три возможных значения:

- "loading" – документ загружается.
- "interactive" – документ был полностью прочитан.
- "complete" – документ был полностью прочитан и все ресурсы (такие как изображения) были тоже загружены.

Пример

Так что мы можем проверить `document.readyState` и, либо установить обработчик, либо, если документ готов, выполнить код сразу же.

Событие `readystatechange` – альтернативный вариант отслеживания состояния загрузки документа, который появился очень давно. На сегодняшний день он используется редко.

```
// текущее состояние
console.log(document.readyState);

// вывести изменения состояния
document.addEventListener('readystatechange', () => console.log(document.readyState));
```



```
function work() { /*...*/ }

if (document.readyState == 'loading') {
    // ещё загружается, ждём события
    document.addEventListener('DOMContentLoaded', work);
} else {
    // DOM готов!
    work();
}
```

Здесь документ с <iframe>, и обработчиками, которые логируют события:

Типичный вывод:

- [1] начальный readyState:loading
- [2] readyState:interactive
- [2] DOMContentLoaded
- [3] iframe onload
- [4] img onload
- [4] readyState:complete
- [4] window onload

```
<script>
  log('начальный readyState: ' + document.readyState);

  document.addEventListener('readystatechange', () => log('readyState: ' + document.readyState));
  document.addEventListener('DOMContentLoaded', () => log('DOMContentLoaded'));

  window.onload = () => log('window onload');
</script>

<iframe src="iframe.html" onload="log('iframe onload')"></iframe>


<script>
  img.onload = () => log('img onload');
</script>
```

Цифры в квадратных скобках обозначают примерное время события. События, отмеченные одинаковой цифрой, произойдут примерно в одно и то же время (\pm несколько миллисекунд).

document.readyState станет interactive прямо передDOMContentLoaded. Эти две вещи, на самом деле, обозначают одно и то же.

document.readyState станет complete, когда все ресурсы (iframe и img) загрузятся. Здесь мы видим, что это произойдёт примерно в одно время с img.onload (img последний ресурс) и window.onload. Переключение на состояние complete означает то же самое, что и window.onload. Разница заключается в том, что window.onload всегда срабатывает после всех load других обработчиков.

Скрипты: async, defer

В современных сайтах скрипты обычно «тяжелее», чем HTML: они весят больше, дольше обрабатываются.

Когда браузер загружает HTML и доходит до тега `<script>...</script>`, он не может продолжать строить DOM. Он должен сначала выполнить скрипт. То же самое происходит и с внешними скриптами `<script src="..."></script>`: браузер должен подождать, пока загрузится скрипт, выполнить его, и только затем обработать остальную страницу.

Это ведет к двум важным проблемам:

1. Скрипты не видят DOM-элементы ниже себя, поэтому к ним нельзя добавить обработчики и т.д.
2. Если вверху страницы объемный скрипт, он «блокирует» страницу. Пользователи не видят содержимое страницы, пока он не загрузится и не запустится.

```
<p>...содержимое перед скриптом...</p>
```

```
<script src="https://javascript.info/article/script-async-defer/long.js?speed=1"></script>
```

```
<!-- Это не отобразится, пока скрипт не загрузится -->
```

```
<p>...содержимое после скрипта...</p>
```

defer

Атрибут defer сообщает браузеру, что он должен продолжать обрабатывать страницу и загружать скрипт в фоновом режиме, а затем запустить этот скрипт, когда DOM дерево будет полностью построено.

- Скрипты с defer никогда не блокируют страницу.
- Скрипты с defer всегда выполняются, когда дерево DOM готово, но до события DOMContentLoaded.

Отложенные с помощью defer скрипты сохраняют порядок относительно друг друга, как и обычные скрипты.

defer не только говорит браузеру «не блокировать рендеринг», он также обеспечивает правильную последовательность выполнения скриптов.

Атрибут defer предназначен только для внешних скриптов

Атрибут defer будет проигнорирован, если в теге <script> нет src.

```
<p>...содержимое до скрипта...</p>

<script>
| document.addEventListener('DOMContentLoaded', () => alert("Дерево DOM готово после скрипта с 'defer'!"));
</script>

<script defer src="https://javascript.info/article/script-async-defer/long.js?speed=1"></script> // (2)

<p>...содержимое после скрипта...</p>
```

async

```
<!-- Типичное подключение скрипта Google Analytics -->
<script async src="https://google-analytics.com/analytics.js"></script>
```

Атрибут `async` означает, что скрипт абсолютно независим:

- Страница не ждёт асинхронных скриптов, содержимое обрабатывается и отображается.
- Событие `DOMContentLoaded` и асинхронные скрипты не ждут друг друга:
 - `DOMContentLoaded` может произойти как до асинхронного скрипта (если асинхронный скрипт завершит загрузку после того, как страница будет готова),
 - ...так и после асинхронного скрипта (если он короткий или уже содержится в HTTP-кеше)
- Остальные скрипты не ждут `async`, и скрипты с `async` не ждут другие скрипты.

Асинхронные скрипты очень полезны для добавления на страницу сторонних скриптов: счетчиков, рекламы и т.д. Они не зависят от наших скриптов, и мы тоже не должны ждать их.

1. Содержимое страницы отображается сразу же : `async` его не блокирует.
2. `DOMContentLoaded` может произойти как до, так и после `async`, никаких гарантий нет.
3. Асинхронные скрипты не ждут друг друга.

Атрибут `async` предназначен только для внешних скриптов

Как и в случае с `defer`, атрибут `async` будет проигнорирован, если в теге `<script>` нет `src`.

Динамически загружаемые скрипты

Мы можем также добавить скрипт и динамически, с помощью JavaScript:

Динамически загружаемые скрипты по умолчанию ведут себя как «`async`».

Мы можем изменить относительный порядок скриптов с «первый загрузился – первый выполнился» на порядок, в котором они идут в документе (как в обычных скриптах) с помощью явной установки свойства `async` в `false`:

```
let script = document.createElement('script');
script.src = "/article/script-async-defer/long.js";
document.body.append(script); // (*)
```

```
let script = document.createElement('script');
script.src = "/article/script-async-defer/long.js";

script.async = false;

document.body.append(script);
```

```
function loadScript(src) {
  let script = document.createElement('script');
  script.src = src;
  script.async = false;
  document.body.append(script);
}
```

```
// long.js запускается первым, так как async=false
loadScript("/article/script-async-defer/long.js");
loadScript("/article/script-async-defer/small.js");
```

Загрузка ресурсов: onload и onerror

Браузер позволяет отслеживать загрузку сторонних ресурсов: скриптов, ифреймов, изображений и др.

Для этого существуют два события:

- load – успешная загрузка,
- error – во время загрузки произошла ошибка.

Загрузка скриптов

Допустим, нам нужно загрузить сторонний скрипт и вызвать функцию, которая объявлена в этом скрипте.

...Но как нам вызвать функцию, которая объявлена внутри того скрипта? Нам нужно подождать, пока скрипт загрузится, и только потом мы можем её вызвать.

Главный помощник – это событие `load`. Оно срабатывает после того, как скрипт был загружен и выполнен.

Ошибки, которые возникают во время загрузки скрипта, могут быть отслежены с помощью события `error`.

Обработчики `onload/onerror` отслеживают только сам процесс загрузки.

Ошибки обработки и выполнения загруженного скрипта ими не отслеживаются. Чтобы «поймать» ошибки в скрипте, нужно воспользоваться глобальным обработчиком `window.onerror`.

```
let script = document.createElement('script');

// мы можем загрузить любой скрипт с любого домена
script.src = "https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.3.0/lodash.js"
document.head.append(script);

script.onload = function() {
    // в скрипте создаётся вспомогательная переменная с именем "_"
    alert(_.VERSION); // отображает версию библиотеки
};
```

```
let script = document.createElement('script');
script.src = "https://example.com/404.js"; // такого файла не существует
document.head.append(script);

script.onerror = function() {
    alert("Ошибка загрузки " + this.src); // Ошибка загрузки https://example.com/404.js
};
```

Другие ресурсы

События load и error также срабатывают и для других ресурсов, а вообще, для любых ресурсов, у которых есть внешний src.

```
let img = document.createElement('img');
img.src = "https://js.cx/clipart/train.gif"; // (*)

img.onload = function() {
    alert(`Изображение загружено, размеры ${img.width}x${img.height}`);
};

img.onerror = function() {
    alert("Ошибка во время загрузки изображения");
};
```

Однако есть некоторые особенности:

- Большинство ресурсов начинают загружаться после их добавления в документ. За исключением тега . Изображения начинают загружаться, когда получают src (*).
- Для <iframe> событие load срабатывает по окончании загрузки как в случае успеха, так и в случае ошибки.

Такое поведение сложилось по историческим причинам.

Бинарные данные и файлы.

- ArrayBuffer, бинарные массивы
- TextDecoder и TextEncoder
- Blob
- File и FileReader

ArrayBuffer, бинарные массивы

ArrayBuffer — это специальный тип данных в JavaScript, который представляет собой бинарный массив фиксированной длины. Это означает, что он может хранить только двоичные данные (числа или символы), и его размер не может быть изменен после создания.

Зачем нужен ArrayBuffer на фронте?

ArrayBuffer может использоваться для работы с большими объемами двоичных данных, таких как изображения, аудио- и видеофайлы. На фронте это может потребоваться для следующих задач:

Загрузка и обработка изображений.

Работа с аудио- и видеоданными.

Передача больших объемов данных между сервером и клиентом.

В целом, ArrayBuffer является мощным инструментом для работы с двоичными данными в JavaScript. Однако его использование требует понимания основ работы с памятью и бинарными данными.

Работа с ArrayBuffer:

Создание:

Для создания ArrayBuffer используется конструктор `new ArrayBuffer(length)`. Здесь `length` — длина массива в байтах.

Чтение и запись:

Доступ к данным в ArrayBuffer осуществляется через методы `slice()` и `subarray()`. Они позволяют получить подмассив из исходного массива.

Запись в ArrayBuffer выполняется с помощью метода `set()`. Он принимает два аргумента: индекс и значение.

Преобразование в другие типы данных:

Чтобы преобразовать ArrayBuffer в строку, можно использовать метод `TextDecoder`.

Если нужно преобразовать в типизированный массив (например, `Uint8Array`), можно использовать методы `DataView` или `TypedArray`.

TextDecoder

Что если бинарные данные фактически являются строкой? Например, мы получили файл с текстовыми данными.

Встроенный объект TextDecoder позволяет декодировать данные из бинарного буфера в обычную строку.

Для этого прежде всего нам нужно создать сам декодер:

- `label` – тип кодировки, utf-8 используется по умолчанию, но также поддерживаются big5, windows-1251 и многие другие.
- `options` – объект с дополнительными настройками:
 - `fatal` – boolean, если значение `true`, тогда генерируется ошибка для невалидных (не декодируемых) символов, в ином случае (по умолчанию) они заменяются символом `\uFFFD`.
 - `ignoreBOM` – boolean, если значение `true`, тогда игнорируется BOM (дополнительный признак, определяющий порядок следования байтов), что необходимо крайне редко.

...и после использовать его метод `decode`:

- `input` – бинарный буфер (`BufferSource`) для декодирования.
- `options` – объект с дополнительными настройками:
 - `stream` – `true` для декодирования потока данных, при этом `decoder` вызывается вновь и вновь для каждого следующего фрагмента данных. В этом случае многобайтовый символ может иногда быть разделён и попасть в разные фрагменты данных. Это опция указывает `TextDecoder` запомнить символ, на котором остановился процесс, и декодировать его со следующим фрагментом.

```
let decoder = new TextDecoder([label], [options]);
```

```
let str = decoder.decode([input], [options]);
```

```
let uint8Array = new Uint8Array([72, 101, 108, 108, 111]);
```

```
alert( new TextDecoder().decode(uint8Array) ); // Hello
```

```
let uint8Array = new Uint8Array([228, 189, 160, 229, 165, 189]);
```

```
alert( new TextDecoder().decode(uint8Array) ); // 你好
```

TextEncoder

TextEncoder поступает наоборот – кодирует строку в бинарный массив.

Поддерживается только кодировка «utf-8».

Кодировщик имеет следующие два метода:

- encode(str) – возвращает бинарный массив Uint8Array, содержащий закодированную строку.
- encodeInto(str, destination) – кодирует строку (str) и помещает её в destination, который должен быть экземпляром Uint8Array.

```
let encoder = new TextEncoder();
```

```
let uint8Array = encoder.encode("Hello");
alert(uint8Array); // 72,101,108,108,111
```

Blob

ArrayBuffer и бинарные массивы являются частью ECMA-стандартта и, соответственно, частью JavaScript.

Кроме того, в браузере имеются дополнительные высокоуровневые объекты, описанные в File API.

Объект Blob состоит из необязательной строки type (обычно MIME-тип) и blobParts – последовательности других объектов Blob, строк и BufferSource.

Благодаря type мы можем загружать и скачивать Blob-объекты, где type естественно становится Content-Type в сетевых запросах.

Конструктор имеет следующий синтаксис:

new Blob(blobParts, options);

- blobParts – массив значений Blob/BufferSource/String.
- options – необязательный объект с дополнительными настройками:
 - type – тип объекта, обычно MIME-тип, например. image/png.
 - endings – если указан, то окончания строк создаваемого Blob будут изменены в соответствии с текущей операционной системой (\r\n или \n). По умолчанию "transparent" (ничего не делать), но также может быть "native" (изменять).



```
// создадим Blob из строки
let blob = new Blob(["<html>...</html>"], {type: 'text/html'});
// обратите внимание: первый аргумент должен быть массивом [...]
```

```
// создадим Blob из типизированного массива и строк
let hello = new Uint8Array([72, 101, 108, 108, 111]); // "hello" в бинарной форме

let blob = new Blob([hello, ' ', 'world'], {type: 'text/plain'});
```

Blob не изменяем (immutable)

Мы не можем изменять данные напрямую в Blob, но мы можем делать срезы и создавать новый Blob на их основе, объединять несколько объектов в новый и так далее.

Это поведение аналогично JavaScript-строке: мы не можем изменить символы в строке, но мы можем создать новую исправленную строку на базе имеющейся.

Blob как URL

Blob может быть использован как URL для `<a>`, `` или других тегов, для показа содержимого.

Мы также можем создать ссылку динамически, используя только JavaScript, и эмулировать на ней клик, используя `link.click()`, тогда загрузка начнётся автоматически.

`URL.createObjectURL` берёт Blob и создаёт уникальный URL для него в формате `blob:<origin>/<uuid>`.

```
<!-- download атрибут указывает браузеру делать загрузку вместо навигации -->
<a download="hello.txt" href="#" id="link">Загрузить</a>
```

```
<script>
let blob = new Blob(['Hello, world!'], {type: 'text/plain'});

link.href = URL.createObjectURL(blob);
</script>
```

```
let link = document.createElement('a');
link.download = 'hello.txt';

let blob = new Blob(['Hello, world!'], {type: 'text/plain'});

link.href = URL.createObjectURL(blob);

link.click();

URL.revokeObjectURL(link.href);
```

`blob:https://javascript.info/1e67e00e-860d-40a5-89ae-6ab0cbee6273`

Blob to base64

```

```

Альтернатива URL.createObjectURL – конвертация Blob-объекта в строку с кодировкой base64.

data url имеет форму data:[<mediatype>][:<base64>],<data>. Мы можем использовать такой url где угодно наряду с «обычным» url.

Для трансформации Blob в base64 мы будем использовать встроенный в браузер объект типа FileReader.

```
let link = document.createElement('a');
link.download = 'hello.txt';

let blob = new Blob(['Hello, world!'], {type: 'text/plain'});

let reader = new FileReader();
reader.readAsDataURL(blob); // конвертирует Blob в base64 и вызывает onload

reader.onload = function() {
  link.href = reader.result; // url с данными
  link.click();
};
```

URL.createObjectURL(blob)

- Нужно отрывать объект для освобождения памяти.
- Прямой доступ к Blob, без «кодирования/декодирования».

Blob to data url

- Нет необходимости что-либо отрывать.
- Потеря производительности и памяти при декодировании больших Blob-объектов.

Изображение в Blob

Мы можем создать Blob для изображения, части изображения или даже создать скриншот страницы. Что удобно для последующей загрузки куда-либо.

Операции с изображениями выполняются через элемент <canvas>:

- Для отрисовки изображения (или его части) на холсте (canvas) используется canvas.drawImage.
- Вызов canvas-метода .toBlob(callback, format, quality) создаёт Blob и вызывает функцию callback при завершении.

Или если вы предпочитаете async/await вместо колбэка:

```
// берём любое изображение
let img = document.querySelector('img');

// создаём <canvas> того же размера
let canvas = document.createElement('canvas');
canvas.width = img.clientWidth;
canvas.height = img.clientHeight;

let context = canvas.getContext('2d');

// копирую изображение в canvas (метод позволяет вырезать часть изображения)
context.drawImage(img, 0, 0);
// мы можем вращать изображение при помощи context.rotate() и делать множество других преобразований

// toBlob является асинхронной операцией, для которой callback-функция вызывается при завершении
canvas.toBlob(function(blob) {
    // после того, как Blob создан, загружаем его
    let link = document.createElement('a');
    link.download = 'example.png';

    link.href = URL.createObjectURL(blob);
    link.click();

    // удаляем внутреннюю ссылку на Blob, что позволит браузеру очистить память
    URL.revokeObjectURL(link.href);
}, 'image/png');

async function blobResolve () {
    let blob = await new Promise(resolve => canvasElem.toBlob(resolve, 'image/png'));
}
```

Из Blob в ArrayBuffer

Конструктор Blob позволяет создать Blob-объект практически из чего угодно, включая BufferSource.

Но если нам нужна производительная низкоуровневая обработка, мы можем использовать ArrayBuffer из FileReader:

```
// получаем ArrayBuffer из Blob
let fileReader = new FileReader();

fileReader.readAsArrayBuffer(blob);

fileReader.onload = function(event) {
  let arrayBuffer = fileReader.result;
};
```

File и FileReader

Объект File наследуется от объекта Blob и обладает возможностями по взаимодействию с файловой системой.

```
<input type="file" onchange="showFile(this)">

<script>
function showFile(input) {
  let file = input.files[0];

  alert(`File name: ${file.name}`); // например, my.png
  alert(`Last modified: ${file.lastModified}`); // например, 1552830408824
}
</script>
```

Есть два способа его получить.

Во-первых, есть конструктор, похожий на Blob:

- fileParts – массив значений Blob/BufferSource/строки.
- fileName – имя файла, строка.
 - options – необязательный объект со свойством:
 - lastModified – дата последнего изменения в формате таймстамп (целое число).

Во-вторых, чаще всего мы получаем файл из <input type="file"> или через перетаскивание с помощью мыши, или из других интерфейсов браузера. В этом случае файл получает эту информацию из ОС.

Так как File наследует от Blob, у объектов File есть те же свойства плюс:

- name – имя файла,
- lastModified – таймстамп для даты последнего изменения.

FileReader

FileReader объект, цель которого читать данные из Blob (и, следовательно, из File тоже).

Данные передаются при помощи событий, так как чтение с диска может занять время.

```
let reader = new FileReader(); // без аргументов
```

Основные методы:

- `readAsArrayBuffer(blob)` – считать данные как ArrayBuffer
- `readAsText(blob, [encoding])` – считать данные как строку (кодировка по умолчанию: utf-8)
- `readAsDataURL(blob)` – считать данные как base64-кодированный URL.
- `abort()` – отменить операцию.

Методы чтения

Выбор метода для чтения зависит от того, какой формат мы предпочитаем, как мы хотим далее использовать данные.

- `readAsArrayBuffer` – для бинарных файлов, для низкоуровневой побайтовой работы с бинарными данными. Для высокоуровневых операций у `File` есть свои методы, унаследованные от `Blob`, например, `slice`, мы можем вызвать их напрямую.
- `readAsText` – для текстовых файлов, когда мы хотим получить строку.
- `readAsDataURL` – когда мы хотим использовать данные в `src` для `img` или другого тега. Есть альтернатива – можно не читать файл, а вызвать `URL.createObjectURL(file)`, детали в главе `Blob`.

В процессе чтения происходят следующие события:

- loadstart – чтение начато.
- progress – срабатывает во время чтения данных.
- load – нет ошибок, чтение окончено.
- abort – вызван abort().
- error – произошла ошибка.
- loadend – чтение завершено (успешно или нет).

Когда чтение закончено, мы сможем получить доступ к его результату следующим образом:

- reader.result результат чтения (если оно успешно)
- reader.error объект ошибки (при неудаче).

```
<input type="file" onchange="readFile(this)">

<script>
function readFile(input) {
    let file = input.files[0];

    let reader = new FileReader();

    reader.readAsText(file);

    reader.onload = function() {
        console.log(reader.result);
    };

    reader.onerror = function() {
        console.log(reader.error);
    };
}
</script>
```

FileReader для Blob

Как упоминалось в главе Blob, FileReader работает для любых объектов Blob, а не только для файлов.

Поэтому мы можем использовать его для преобразования Blob в другой формат:

- `readAsArrayBuffer(blob)` – в `ArrayBuffer`,
- `readAsText(blob, [encoding])` – в строку (альтернатива `TextDecoder`),
- `readAsDataURL(blob)` – в формат `base64`-кодированного URL.

Итого касательно File

File объекты наследуют от Blob.

Помимо методов и свойств Blob, объекты File также имеют свойства name и lastModified плюс внутреннюю возможность чтения из файловой системы. Обычно мы получаем объекты File из пользовательского ввода, например, через <input> или перетаскиванием с помощью мыши, в событии dragend.

Объекты FileReader могут читать из файла или Blob в одном из трёх форматов:

- Стока (readAsText).
- ArrayBuffer (readAsArrayBuffer).
- URL в формате base64 (readAsDataURL).

Однако, во многих случаях нам не нужно читать содержимое файла. Как и в случае с Blob, мы можем создать короткий URL с помощью URL.createObjectURL(file) и использовать его в теге <a> или . Таким образом, файл может быть загружен или показан в виде изображения, как часть canvas и т.д.

Ресурсы

Страница: DOMContentLoaded, load, beforeunload, unload - [ТыК](#)

Скрипты: async, defer - [ТыК](#)

Загрузка ресурсов: onload и onerror - [ТыК](#)

ArrayBuffer, бинарные массивы - [ТыК](#)

TextDecoder и TextEncoder - [ТыК](#)

Blob - [ТыК](#)

File и FileReader - [ТыК](#)

Пример работы с файлами на фронте - [ТыК](#)