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 ведёт 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 объёмный 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-asy

<!-- Это не отобразится, пока скрипт не загрузится -->
<p>...содержимое после скрипта...</p>
```

Конечно, есть пути, как это обойти. Например, мы можем поместить скрипт внизу страницы. Тогда он сможет видеть элементы над ним и не будет препятствовать отображению содержимого страницы:

```
<body>
  ...всё содержимое над скриптом...

  <script src="https://javascript.info/article/script-a
</body>
```

Но это решение далеко от идеального. Например, браузер замечает скрипт (и может начать загружать его) только после того, как он полностью загрузил HTML-документ. В случае с длинными HTML-страницами это может создать заметную задержку.

Такие вещи незаметны людям, у кого очень быстрое соединение, но много кто в мире имеет медленное подключение к интернету или использует не такой хороший мобильный интернет.

К счастью, есть два атрибута тега `<script>`, которые решают нашу проблему: `defer` и `async`.

defer

Атрибут `defer` сообщает браузеру, что он должен продолжать обрабатывать страницу и загружать скрипт в фоновом режиме, а затем запустить этот скрипт, когда он загрузится.

Вот тот же пример, что и выше, но с `defer`:

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

<script defer src="https://javascript.info/article/scri


```

```
<!-- отображается сразу же -->
<p>...содержимое после скрипта...</p>
```


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

Следующий пример это показывает:

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

<script>
  document.addEventListener('DOMContentLoaded', () => a
</script>

<script defer src="https://javascript.info/article/scri

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

1. Содержимое страницы отобразится мгновенно.
2. Событие `DOMContentLoaded` подождёт отложенный скрипт. Оно будет сгенерировано, только когда скрипт `(2)` будет загружен и выполнен.

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

Поэтому, если сначала загружается большой скрипт, а затем меньшего размера, то последний будет ждать.

```
<script defer src="https://javascript.info/article/scri
<script defer src="https://javascript.info/article/scri
```

Маленький скрипт загрузится первым, но выполнится вторым

Браузеры сканируют страницу на предмет скриптов и загружают их параллельно в целях увеличения производительности. Поэтому и в примере выше оба скрипта скачиваются параллельно. `small.js` скорее всего загрузится первым.

Но спецификация требует последовательного выполнения скриптов согласно порядку в документе, поэтому он подождёт выполнения `long.js`.

Атрибут `defer` предназначен только для внешних скриптов

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

async

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.

Так что если у нас есть несколько скриптов с `async`, они могут выполняться в любом порядке. То, что первое загрузится – запустится в первую очередь:

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

<script>
  document.addEventListener('DOMContentLoaded', () => a
</script>

<script async src="https://javascript.info/article/scri
<script async src="https://javascript.info/article/scri

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

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. Меньший скрипт `small.js` идёт вторым, но скорее всего загрузится раньше `long.js`, поэтому и запустится первым. То есть, скрипты выполняются в порядке загрузки.

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

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

Динамически загружаемые скрипты

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

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

Скрипт начнёт загружаться, как только он будет добавлен в документ `(*)`.

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

То есть:

- Они никого не ждут, и их никто не ждёт.
- Скрипт, который загружается первым – запускается первым (в порядке загрузки).

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";

script.async = false;

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

Например, здесь мы добавляем два скрипта. Без `script.async=false` они запускались бы в порядке загрузки (`small.js` скорее всего запустился бы раньше). Но с этим флагом порядок будет как в документе:

```
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");
```

Итого

У `async` и `defer` есть кое-что общее: они не блокируют отрисовку страницы. Так что пользователь может просмотреть содержимое страницы и ознакомиться с ней сразу же.

Но есть и значимые различия:

	Порядок	DOMContentLoaded
`async`	*Порядок загрузки* (кто загрузится первым, тот и сработает).	Не имеет значения. Может загрузиться и выполниться до того, как страница полностью загрузится. Такое случается, если скрипты маленькие или хранятся в кеше, а документ достаточно большой.
`defer`	*Порядок документа* (как расположены в документе).	Выполняется после того, как документ загружен и обработан (ждёт), непосредственно перед `DOMContentLoaded`.

Страница без скриптов должна быть рабочей

Пожалуйста, помните, что когда вы используете `defer`, страница видна до того, как скрипт загрузится.

Пользователь может знакомиться с содержимым страницы, читать её, но графические компоненты пока отключены.

Поэтому обязательно должна быть индикация загрузки, нерабочие кнопки – отключены с помощью CSS или другим образом. Чтобы пользователь явно видел, что уже готово, а что пока нет.

На практике `defer` используется для скриптов, которым требуется доступ ко всему DOM и/или важен их относительный порядок выполнения.

А `async` хорош для независимых скриптов, например счётчиков и рекламы, относительный порядок выполнения которых не играет роли.

Комментарии

перед тем как писать...