

Раздел

Модули

Навигация по уроку

Что такое модуль?

Основные возможности модулей

Особенности в браузерах

Инструменты сборки

Итого

Комментарии

Поделиться



Редактировать на GitHub

 → [Язык программирования JavaScript](#) → [Модули](#)  18-го февраля 2020

Модули, введение

По мере роста нашего приложения, мы обычно хотим разделить его на много файлов, так называемых «модулей». Модуль обычно содержит класс или библиотеку с функциями.

Долгое время в JavaScript отсутствовал синтаксис модулей на уровне языка. Это не было проблемой, потому что первые скрипты были маленькими и простыми. В модулях не было необходимости.

Но со временем скрипты становились всё более и более сложными, поэтому сообщество придумало несколько вариантов организации кода в модули. Появились библиотеки для динамической подгрузки модулей.

Например:

- **AMD** – одна из самых старых модульных систем, изначально реализована библиотекой [require.js](#).
- **CommonJS** – модульная система, созданная для сервера Node.js.
- **UMD** – ещё одна модульная система, предлагается как универсальная, совместима с AMD и CommonJS.

Теперь все они постепенно становятся частью истории, хотя их и можно найти в старых скриптах.

Система модулей на уровне языка появилась в стандарте JavaScript в 2015 году и постепенно эволюционировала. На данный момент она поддерживается большинством браузеров и Node.js. Далее мы будем изучать именно её.



Что такое модуль?



Модуль – это просто файл. Один скрипт – это один модуль.

Модули могут загружать друг друга и использовать директивы `export` и `import`, чтобы обмениваться функциональностью, вызывать функции одного модуля из другого:

- `export` отмечает переменные и функции, которые должны быть доступны вне текущего модуля.
- `import` позволяет импортировать функциональность из других модулей.

Например, если у нас есть файл `sayHi.js`, который экспортирует функцию:

```
1 // sayHi.js
2 export function sayHi(user) {
3   alert(`Hello, ${user}!`);
4 }
```

...Тогда другой файл может импортировать её и использовать:

```
1 // main.js
2 import {sayHi} from './sayHi.js';
3
4 alert(sayHi); // function...
5 sayHi('John'); // Hello, John!
```

Директива `import` загружает модуль по пути `./sayHi.js` относительно текущего файла и записывает экспортированную функцию `sayHi` в соответствующую переменную.

Давайте запустим пример в браузере.

Раздел

Модули

Навигация по уроку

Что такое модуль?

Основные возможности модулей

Особенности в браузерах

Инструменты сборки

Итого

Комментарии

Поделиться



Редактировать на GitHub



Так как модули поддерживают ряд специальных ключевых слов, и у них есть ряд особенностей, то необходимо явно сказать браузеру, что скрипт является модулем, при помощи атрибута `<script type="module">`.

Вот так:

```
Результат  say.js  index.html     
  
<!doctype html> 1  
<script type="module"> 2  
  import {sayHi} from './say.js'; 3  
  4  
  document.body.innerHTML = sayHi('John'); 5  
</script> 6
```

Браузер автоматически загрузит и запустит импортированный модуль (и те, которые он импортирует, если надо), а затем запустит скрипт.

Основные возможности модулей

Чем отличаются модули от «обычных» скриптов?

Есть основные возможности и особенности, работающие как в браузере, так и в серверном JavaScript.

Всегда «use strict»

В модулях всегда используется режим `use strict`. Например, присваивание к необъявленной переменной вызовет ошибку.

```
1 <script type="module">  
2   a = 5; // ошибка  
3 </script>
```




Своя область видимости переменных



Каждый модуль имеет свою собственную область видимости. Другими словами, переменные и функции, объявленные в модуле, не видны в других скриптах.

В следующем примере импортированы 2 скрипта, и `hello.js` пытается использовать переменную `user`, объявленную в `user.js`. В итоге ошибка:

```
Результат  hello.js  user.js  index.html     
  
<!doctype html> 1  
<script type="module" src="user.js"></script> 2  
<script type="module" src="hello.js"></script> 3
```

Модули должны экспортировать функциональность, предназначенную для использования извне. А другие модули могут её импортировать.

Так что нам надо импортировать `user.js` в `hello.js` и взять из него нужную функциональность, вместо того чтобы полагаться на глобальные переменные.

Правильный вариант:

```
Результат  hello.js  user.js  index.html     
  
import {user} from './user.js'; 1  
2  
document.body.innerHTML = user; // John 3
```

В браузере также существует независимая область видимости для каждого скрипта `<script type="module">`:

Раздел

Модули

Навигация по уроку

Что такое модуль?

Основные возможности модулей

Особенности в браузерах

Инструменты сборки

Итого

Комментарии

Поделиться



Редактировать на GitHub



```
1 <script type="module">
2   // Переменная доступна только в этом модуле
3   let user = "John";
4 </script>
5
6 <script type="module">
7   alert(user); // Error: user is not defined
8 </script>
```

Если нам нужно сделать глобальную переменную уровня всей страницы, можно явно присвоить её объекту `window`, тогда получить значение переменной можно обратившись к `window.user`. Но это должно быть исключением, требующим веской причины.

Код в модуле выполняется только один раз при импорте

Если один и тот же модуль используется в нескольких местах, то его код выполнится только один раз, после чего экспортируемая функциональность передаётся всем импортёрам.

Это очень важно для понимания работы модулей. Давайте посмотрим примеры.

Во-первых, если при запуске модуля возникают побочные эффекты, например выдаётся сообщение, то импорт модуля в нескольких местах покажет его только один раз – при первом импорте:

```
1 // alert.js
2 alert("Модуль выполнен!");
```



```
1 // Импорт одного и того же модуля в разных файлах
2
3 // 1.js
4 import './alert.js'; // Модуль выполнен!
5
6 // 2.js
7 import './alert.js'; // (ничего не покажет)
```



На практике, задача кода модуля – это обычно инициализация, создание внутренних структур данных, а если мы хотим, чтобы что-то можно было использовать много раз, то экспортируем это.

Теперь более продвинутый пример.

Давайте представим, что модуль экспортирует объект:

```
1 // admin.js
2 export let admin = {
3   name: "John"
4 };
```

Если модуль импортируется в нескольких файлах, то код модуля будет выполнен только один раз, объект `admin` будет создан и в дальнейшем будет передан всем импортёрам.

Все импортёры получают один-единственный объект `admin`:

```
1 // 1.js
2 import {admin} from './admin.js';
3 admin.name = "Pete";
4
5 // 2.js
6 import {admin} from './admin.js';
```

Раздел

Модули

Навигация по уроку

Что такое модуль?

Основные возможности модулей

Особенности в браузерах

Инструменты сборки

Итого

Комментарии

Поделиться



Редактировать на GitHub



```
7 alert(admin.name); // Pete
8
9 // Оба файла, 1.js и 2.js, импортируют один и тот же об
10 // Изменения, сделанные в 1.js, будут видны в 2.js
```

Ещё раз заметим – модуль выполняется только один раз. Генерируется экспорт и после передаётся всем импортёрам, поэтому, если что-то изменится в объекте `admin`, то другие модули тоже увидят эти изменения.

Такое поведение позволяет *конфигурировать* модули при первом импорте. Мы можем установить его свойства один раз, и в дальнейших импортах он будет уже настроенным.

Например, модуль `admin.js` предоставляет определённую функциональность, но ожидает передачи учётных данных в объект `admin` извне:

```
1 // 📄 admin.js
2 export let admin = { };
3
4 export function sayHi() {
5   alert(`Ready to serve, ${admin.name}!`);
6 }
```

В `init.js`, первом скрипте нашего приложения, мы установим `admin.name`. Тогда все это увидят, включая вызовы, сделанные из самого `admin.js`:

```
1 // 📄 init.js
2 import {admin} from './admin.js';
3 admin.name = "Pete";
```



Другой модуль тоже увидит `admin.name`:



```
1 // 📄 other.js
2 import {admin, sayHi} from './admin.js';
3
4 alert(admin.name); // Pete
5
6 sayHi(); // Ready to serve, Pete!
```

import.meta

Объект `import.meta` содержит информацию о текущем модуле.

Содержимое зависит от окружения. В браузере он содержит ссылку на скрипт или ссылку на текущую веб-страницу, если модуль встроен в HTML:

```
1 <script type="module">
2   alert(import.meta.url); // ссылка на html страницу дл
3 </script>
```

В модуле «this» не определён

Это незначительная особенность, но для полноты картины нам нужно упомянуть об этом.

В модуле на верхнем уровне `this` не определён (`undefined`).

Сравним с не-модульными скриптами, там `this` – глобальный объект:

```
1 <script>
2   alert(this); // window
3 </script>
```

Раздел

Модули

Навигация по уроку

Что такое модуль?

Основные возможности модулей

Особенности в браузерах

Инструменты сборки

Итого

Комментарии

Поделиться



Редактировать на GitHub



```
4
5 <script type="module">
6   alert(this); // undefined
7 </script>
```

Особенности в браузерах

Есть и несколько других, именно браузерных особенностей скриптов с `type="module"` по сравнению с обычными скриптами.

Если вы читаете материал в первый раз или, если не собираетесь использовать модули в браузерах, то сейчас можете пропустить эту секцию.

Модули являются отложенными (deferred)

Модули *всегда* выполняются в отложенном (deferred) режиме, точно так же, как скрипты с атрибутом `defer` (описан в главе [Скрипты: `async`, `defer`](#)). Это верно и для внешних и встроенных скриптов-модулей.

Другими словами:

- загрузка внешних модулей, таких как `<script type="module" src="...">`, не блокирует обработку HTML.
- модули, даже если загрузились быстро, ожидают полной загрузки HTML документа, и только затем выполняются.
- сохраняется относительный порядок скриптов: скрипты, которые идут раньше в документе, выполняются раньше.

Как побочный эффект, модули всегда видят полностью загруженную HTML-страницу, включая элементы под ними.

Например:

```
1 <script type="module">
2   alert(typeof button); // object: скрипт может 'видеть'
3   // так как модули являются отложенными, то скрипт нач
4 </script>
5
6 Сравните с обычным скриптом ниже:
7
8 <script>
9   alert(typeof button); // Ошибка: кнопка не определена
10  // обычные скрипты запускаются сразу, не дожидаясь по
11 </script>
12
13 <button id="button">Кнопка</button>
```

Пожалуйста, обратите внимание: второй скрипт выполнится раньше, чем первый! Поэтому мы увидим сначала `undefined`, а потом `object`.

Это потому, что модули начинают выполняться после полной загрузки страницы. Обычные скрипты запускаются сразу же, поэтому сообщение из обычного скрипта мы видим первым.

При использовании модулей нам стоит иметь в виду, что HTML-страница будет показана браузером до того, как выполнятся модули и JavaScript-приложение будет готово к работе. Некоторые функции могут ещё не работать. Нам следует разместить «индикатор загрузки» или что-то ещё, чтобы не смутить этим посетителя.

Атрибут `async` работает во встроенных скриптах

Для не-модульных скриптов атрибут `async` работает только на внешних скриптах. Скрипты с ним запускаются сразу по готовности, они не ждут другие скрипты или HTML-документ.

Для модулей атрибут `async` работает на любых скриптах.

Например, в скрипте ниже есть `async`, поэтому он выполнится сразу после загрузки, не ожидая других скриптов.

Раздел

Модули

Навигация по уроку

Что такое модуль?

Основные возможности модулей

Особенности в браузерах

Инструменты сборки

Итого

Комментарии

Поделиться



Редактировать на GitHub



Скрипт выполнит импорт (загрузит `./analytics.js`) и сразу запустится, когда будет готов, даже если HTML документ ещё не будет загружен, или если другие скрипты ещё загружаются.

Это очень полезно, когда модуль ни с чем не связан, например для счётчиков, рекламы, обработчиков событий.

```
1 <!-- загружаются зависимости (analytics.js) и скрипт за
2 <!-- модуль не ожидает загрузки документа или других тэ
3 <script async type="module">
4   import {counter} from './analytics.js';
5
6   counter.count();
7 </script>
```

Внешние скрипты

Внешние скрипты с атрибутом `type="module"` имеют два отличия:

1. Внешние скрипты с одинаковым атрибутом `src` запускаются только один раз:

```
1 <!-- скрипт my.js загрузится и будет выполнен только
2 <script type="module" src="my.js"></script>
3 <script type="module" src="my.js"></script>
```

2. Внешний скрипт, который загружается с другого домена, требует указания заголовков [CORS](#). Другими словами, если модульный скрипт загружается с другого домена, то удалённый сервер должен установить заголовок `Access-Control-Allow-Origin` означающий, что загрузка скрипта разрешена.



```
1 <!-- another-site.com должен указать заголовок Access
2 <!-- иначе, скрипт не выполнится -->
3 <script type="module" src="http://another-site.com/th
```



Это обеспечивает лучшую безопасность по умолчанию.

Не допускаются «голые» модули

В браузере `import` должен содержать относительный или абсолютный путь к модулю. Модули без пути называются «голыми» (`bare`). Они не разрешены в `import`.

Например, этот `import` неправильный:

```
1 import {sayHi} from 'sayHi'; // Ошибка, "голый" модуль
2 // путь должен быть, например './sayHi.js' или абсолютн
```

Другие окружения, например Node.js, допускают использование «голых» модулей, без путей, так как в них есть свои правила, как работать с такими модулями и где их искать. Но браузеры пока не поддерживают «голые» модули.

Совместимость, «nomodule»

Старые браузеры не понимают атрибут `type="module"`. Скрипты с неизвестным атрибутом `type` просто игнорируются. Мы можем сделать для них «резервный» скрипт при помощи атрибута `nomodule`:

```
1 <script type="module">
2   alert("Работает в современных браузерах");
3 </script>
4
```



Раздел

Модули

Навигация по уроку

Что такое модуль?

Основные возможности модулей

Особенности в браузерах

Инструменты сборки

Итого

Комментарии

Поделиться



Редактировать на GitHub



```
5 <script nomodule>
6   alert("Современные браузеры понимают оба атрибута - и
7   alert("Старые браузеры игнорируют скрипты с неизвестн
8 </script>
```

Инструменты сборки

В реальной жизни модули в браузерах редко используются в «сыром» виде. Обычно, мы объединяем модули вместе, используя специальный инструмент, например [Webpack](#) и после выкладываем код на рабочий сервер.

Одно из преимуществ использования сборщика – он предоставляет больший контроль над тем, как модули ищутся, позволяет использовать «голые» модули и многое другое «свое», например CSS/HTML-модули.

Сборщик делает следующее:

1. Берёт «основной» модуль, который мы собираемся поместить в `<script type="module">` в HTML.
2. Анализирует зависимости (импорты, импорты импортов и так далее)
3. Собирает один файл со всеми модулями (или несколько файлов, это можно настроить), перезаписывает встроенный `import` функцией импорта от сборщика, чтобы всё работало. «Специальные» типы модулей, такие как HTML/CSS тоже поддерживаются.
4. В процессе могут происходить и другие трансформации и оптимизации кода:
 - Недостижимый код удаляется.
 - Неиспользуемые экспорты удаляются («tree-shaking»).
 - Специфические операторы для разработки, такие как `console` и `debugger`, удаляются.
 - Современный синтаксис JavaScript также может быть трансформирован в предыдущий стандарт, с похожей функциональностью, например, с помощью [Babel](#).
 - Полученный файл можно минимизировать (удалить пробелы, заменить названия переменных на более короткие и т.д.).

Если мы используем инструменты сборки, то они объединяют модули вместе в один или несколько файлов, и заменяют `import/export` на свои вызовы. Поэтому итоговую сборку можно подключать и без атрибута `type="module"`, как обычный скрипт:

```
1 <!-- Предположим, что мы собрали bundle.js, используя н
2 <script src="bundle.js"></script>
```

Хотя и «как есть» модули тоже можно использовать, а сборщик настроить позже при необходимости.

Итого

Подводя итог, основные понятия:

1. Модуль – это файл. Чтобы работал `import/export`, нужно для браузеров указывать атрибут `<script type="module">`. У модулей есть ряд особенностей:
 - Отложенное (`deferred`) выполнение по умолчанию.
 - Атрибут `async` работает во встроенных скриптах.
 - Для загрузки внешних модулей с другого источника, он должен ставить заголовки CORS.
 - Дублирующиеся внешние скрипты игнорируются.
2. У модулей есть своя область видимости, обмениваться функциональностью можно через `import/export`.
3. В модулях всегда включена директива `use strict`.
4. Код в модулях выполняется только один раз. Экспортируемая функциональность создаётся один раз и передаётся всем импортёрам.

Раздел

[Модули](#)

Навигация по уроку

Что такое модуль?

Основные возможности
модулей

Особенности в браузерах

Инструменты сборки

Итого

Комментарии

Поделиться



Редактировать на GitHub



Когда мы используем модули, каждый модуль реализует свою функциональность и экспортирует её. Затем мы используем `import`, чтобы напрямую импортировать её туда, куда необходимо. Браузер загружает и анализирует скрипты автоматически.

В реальной жизни часто используется сборщик [Webpack](#), чтобы объединить модули: для производительности и других «плюшек».

В следующей главе мы увидим больше примеров и вариантов импорта/экспорта.

Проводим [курсы по JavaScript и фреймворкам](#). ✕



Комментарии

перед тем как писать...

