JSDG 🦏 | Modules | Модули | chapter 10

JavaScript: The Definitive Guide 7th EDITION • Master the World's Most-Used Programming Language • David Flanagan • 2021

Цель модульного программирования - позволить собирать крупные программы с использованием модулей кода от разных авторов и источников и обеспечить корректное выполнение всего кода даже при наличии кода, которые различные авторы модулей не предвидели. На практике модульность главным образом касается инкапсуляции или сокрытия внутренних деталей реализации и поддержания порядка в глобальном пространстве имен, чтобы модули не могли случайно модифицировать переменные, функции и классы, определяемые другими модулями.

Модульность на основе замыканий с поддержкой со стороны инструментов пакетирования кода привела к практической форме модульности, основанной на функции require (), которая была принята в Node.

Модули на основе require () являются фундаментальной частью программной среды Node, но не принимались как официальная часть языка JavaScript. Взамен в ES6 модули определяются с применением ключевых слов import и export.

Хотя import и export считались частью языка в течение многих лет, они лишь относительно недавно были реализованы веб-браузерами и Node. Кроме того, с практической точки зрения модульность JavaScript по-прежнему опирается на инструменты пакетирования кода.

```
const BitSet = (function () { // Установить BitSet в возвращаемое
    // значение этой функции
    // Здесь находятся закрытые детали реализации
    function isValid (set, n) 1 ... )
    function has (set, byte, bit) I ... )
    const BITS = new Uint8Array ([1, 2, 4, 8, 16, 32, 64, 128]);
```

```
    const MASKS = new Uint8Array [~1, ~2, ~4, ~8, ~16, ~32, ~64,
~128]);
    // Открытый API-интерфейс модуля - это просто класс Bitset,
    // который мы здесь определяем и возвращаем.
    // Класс может использовать закрытые функции и константы,
    // определенные выше, но они будут скрыты от
пользователей класса.

    return class BitSet extends AbstractWritableSet {
        // . .. реализация не показана • . •
    };
) () );



// Вот так мы могли бы определить модуль расчета
статистических данных.
const stats = (function () {
    // Служебные функции закрыты по отношению к модулю.
    const sum = (x, y) => x + y;
    const square = x => x * x;

    // Открытая функция, которая будет экспортироваться.
    function mean (data) { •
        return data.reduce (sum) /data. length;
    }

    // Открытая функция, которая будет экспортироваться.
    function stddev (data) {
        let m = mean (data);
        return Math.sart (
            data.map (x => x - m) . map (square) . reduce (sum) /
(data. length-1)
        ) ;
    }

    // Открытые функции экспортируются в виде свойств
объекта.
    return { mean, stddev };
}() ):

// А так мы можем использовать модуль.
```

```
stats.mean ( [1, 3, 5, 7, 9]) [/ => 5
stats.stddev (I1, 3, 5, 7, 91) // => Math.sqrt (10)
```

___

Автоматизация модульности на основе замыканий

```
const modules = {};

function require (moduleName) { return modules [moduleName]; }

modules["sets.¡s"] = (function ()
    const exports = {};
    // Здесь находится содержимое файла sets.js:
    exports.BitSet = class BitSet { ... J;
    return exports;
}());

modules ["stats.js"] = (function () I
    const exports = {};
    // Здесь находится содержимое файла stats.js:
    const sum = (x, y) => x + y;
    const square = x = > x * x;
    exports.mean = function (data) { ... }:
    exports.stddev = function (data) { ... J;
    return exports;
}());
```

Имея модули, пакетированные в единственном файле вроде показанного в предыдущем примере, вы можете также представить, что для использования модулей подойдет код такого вида:

```
// Получить ссылки на необходимые модули (или на содержимое модуля):

const stats = require("stats.js");
const BitSet = require("sets.js"). BitSet;

// Код для использования этих модулей:
let s = new BitSet (100);
s.insert (10);
```

s.insert (20) ;
s.insert (30) ;

let average = stats.mean ([...s]): / / средняя величина равна 20

___

Модули в Node

Каждый файл в Node является независимым модулем с закрытым пространством имен. Константы, переменные, функции и классы, определенные в одном файле, будут закрытыми в рамках этого файла, если только не экспортировать их. Значения, экспортированные одним модулем, будут видны в другом модуле, только если другой модуль явно импортирует их.

Модули Node импортируют другие модули с помощью функции require().
Экспортируют свои открытые API-интерфейсы, устанавливая свойства объекта exports или полностью замещая объект module.exports.

Стандартным значением module.exports является тот же объект, на который ссылается exports.

Импортирование в Node

// Эти модули встроены в Node.
const fs = require ("fs"); // Встроенный модуль файловой системы
const http = require ("http"); // Встроенный модуль HTTP
// фреймворк HTTP-сервера Express - сторонний модуль.
// Не является частью Node, но был установлен локально.
const express = require ("express"):

Когда вы желаете импортировать модуль собственного кода, то имя модуля должно быть путем к файлу, содержащему ваш код, относительно файла текущего модуля. Допускается использовать абсолютные пути, которые начинаются с символа /, но обычно при импортировании модулей, являющихся частью вашей программы, имена модулей будут начинаться с . / либо временами с . . /, указывая на то, что они относительны текущего

или родительского каталога. Например:

```
const stats = require ('./stats.js');
const BitSet = require ('./utils/bitset.js");

(Вы также можете опускать суффикс • js в импортируемых
файлах и Node по-прежнему отыщет файлы, но эти файловые
расширения часто включаются явно.)

// Импортировать целый объект stats со всеми его функциями.
const stats = require (. /stats.js'):
// У нас больше функций, чем нужно, но они аккуратно
// организованы в удобное пространство имен stats.
let average = stats. mean (data):
// В качестве альтернативы мы можем использовать
идиоматическое
// деструктурирующее присваивание для импортирования
именно тех функций,
// которые мы хотим поместить прямо в локально пространство
имен:
const ( stddev } = require ('./stats.js');

// Это элегантно и лаконично, хотя без префикса stats мы теряем
// немного контекста как пространства имен для функции stddev
() .
let sd = stddev (data) :
```

Модули в ES6

Стандарт ES6 добавил в JavaScript ключевые слова import и export и окончательно обеспечил поддержку подлинной модульности как основного языкового средства. Модульность ES6 концептуально такая же, как модульность Node: каждый файл является отдельным модулем, а определенные внутри файла константы, переменные, функции и классы закрыты по отношению к этому модулю, если только явно не экспортируются.
Значения, экспортированные из одного модуля, будут доступны для использования в модулях, которые явно их импортируют.
Модули ES6 отличаются от модулей Node синтаксисом,

применяемым для экспортирования и импортирования, а также способом определения модулей в веб-браузерах.
Все аспекты подробно обсуждаются в последующих подразделах.

Кроме того, в коде модулей нельзя использовать оператор with, объект arguments или необъявленные переменные. Модули ES6 даже немного строже строгого режима: в строгом режиме в функциях, вызываемых как функции, this равно undefined. В модулях this равно undefined даже в коде верхнего уровня. (Напротив, сценарии в веб-браузерах и Node устанавливают this в глобальный объект.)

Модули ES6 в веб-сети и в Node

В случае естественного использования модули ES6 добавляются к HTML-страницам посредством специального дескриптора <script type="module">, который будет описан позже в главе. Между тем, впервые применив модульность JavaScript, среда Node оказалась в неловком положении, когда ей пришлось поддерживать две не полностью совместимые системы модулей. В версии Node 13 поддерживаются модули ES6, но на данный момент подавляющее большинство программ Node по-прежнему используют модули Node.

Экспортирование в ES6

```
export const PI = Math.PI;
export function degreesToRadians (d) | return d * PI / 180; }
export class Circle {
      constructor (r) ( this.r = r; }
      area () I return PI * this.r * this.r;
}
```

or

```
export ( Circle, degreesToRadians, PI }
```

Часто пишут модули, которые экспортируют только одно значение (как правило, функцию или класс), и в таком случае обычно применяется export default, а не export:

```
export default class BitSet {
      // реализация не показана
};
```

Импортирование в ES6

```
import BitSet from './bitset.js*;
```

Строка *спецификатора модуля* должна быть абсолютным путем, начинающимся с /, относительным путем, начинающимся с . / либо .. /, или полным URL с протоколом и именем хоста. Стандарт ES6 не разрешает использовать неуточненную строку спецификатора модуля, такую как "util, js", поскольку неясно, что она задает — имя модуля, находящегося в том же самом каталоге, где и текущий модуль, либо имя системного модуля, который установлен в каком-то особом месте. (Такое ограничение против “голых спецификаторов модулей” не соблюдается инструментами пакетирования кода вроде webpack, которые можно легко сконфигурировать на поиск модулей в указанном библиотечном каталоге.)

До сих пор мы рассматривали случай импортирования одиночного значения из модуля, в котором применяется export default.

Чтобы импортировать значения из модуля, экспортирующего множество значений, мы используем слегка отличающийся синтаксис:

```
import { mean, stddev } from ”./stats.js”;
```

Руководства по стилю иногда рекомендуют явно импортировать каждый символ, который будет применяться в текущем модуле. Тем не менее, при импортировании из модуля, где определено много экспортированных значений, можно легко импортировать их всех посредством следующего оператора import:

```
import * as stats from "./stats.js”;
```

По обыкновению модули определяют либо один экспорт по

умолчанию, либо множество именованных экспортов.
Для модуля вполне законно, хотя и не сколько необычно, применять и export, и export default.
Но когда подобное происходит, то вы можете импортировать значение по умолчанию и именованные значения с помощью оператора import такого вида:

import Histogram, { mean, stddev } from "./histogram-stats.js”;

Чтобы включить модуль без операторов экспорта в свою программу, необходимо просто указать ключевое слово import со спецификатором модуля:
import "./analytics.js”;

Импортирование и экспортирование с переименованием

import { render as renderlmage } from "./imageutils. js";
import { render as renderUI } from "./ui. js”;

Вот как по-другому импортировать экспорт по умолчанию и именованные экспорты данного модуля:

import (default as Histogram, mean, stddev } from ". /histogram-stats. js";

export {
    layout as calculateLayout,
    render as renderLayout
}

___

10.3.4. Повторное экспортирование

С учетом того, что реализации теперь находятся в разных файлах, определить файл "./stat. js" просто:

import { mean } from ". /stats/mean.js";
import { stddev } from " . /stats/stddev. is";

export { mean, stdev };

Модули ES6 предугадывают такой сценарий использования и предлагают для него специальный синтаксис. Вместо импортирования символа всего лишь для того, чтобы экспортировать его вновь, вы можете объединить шаги импорта и экспорта в единственный оператор " повторного экспорта", в котором применятся ключевые слова export и from:

```
export { mean } from ". /stats/mean.js";
export ( stddev } from "./stats/stddev.is";

export * from ". / stats/mean.js";
export * from "./stats/stddev. js";

export ( mean, mean as average } from " . /stats/mean.is"; //
Допустим, нам нужно повторно экспортировать функцию mean (),
но также определить для нее еще одно имя average ().
export ( stddev } from "./stats/stddev.js";

export { default as mean } from " . /stats/mean.is";
export ( default as stddev } from ". /stats/stddev.js";

// Импортировать функцию mean () из ./stats.js и сделать
// ее экспортом по умолчанию этого модуля.
export ( mean as default } from "./stats.is";

// Модуль average.js просто повторно экспортирует экспорт
// по умолчанию stats/mean.js.
export { default } from ". /stats/mean.is";
```

Вспомните, что по умолчанию модули применяют строгий режим, this не ссылается на глобальный объект и объявления верхнего уровня не разделяются глобально.

Из-за того, что модули должны выполняться иначе, чем унаследованный немодульный код, их введение требует внесения изменений в код HTML и JavaScript.
Если вы хотите естественным об разом использовать директивы import в веб-браузере, тогда обязаны сообщить веб-браузеру о том, что ваш код является модулем, с применением дескриптора

<script type="module" />.

Одна из элегантных особенностей модулей ES6 заключается в том, что каж дый модуль имеет статический набор импортов. Таким образом, имея единственный стартовый модуль, веб-браузер может загрузить все импортированные модули, затем все модули, импортированные первым пакетом модулей, и так далее до тех пор, пока не будет загружена полная программа.

Дескриптор <script type="module"> отмечает начальную точку модульной программы.
Тем не менее, импортируемые им модули не должны находиться в дескрипторах <script>:

взамен они загружаются по запросу как обыкновенные файлы JavaScript и выполняются в строгом режиме как обычные модули ES6.
Использовать дескриптор <script type="module"> для определения главной точки входа в модульной программе JavaScript довольно просто, например:

<script type="module">import "./main. js";</script>

Сценарии с атрибутом type="module" загружаются и выполняются подобно сценариям с атрибутом defer. Загрузка кода начинается, как только синтаксический анализатор HTML встречает дескриптор <script> (в случае модулей та кой шаг загрузки кода может быть рекурсивным процессом, загружающим множество файлов JavaScript). Но выполнение кода не начнется до тех пор, пока не закончится синтаксический анализ HTML-разметки. После того, как синтаксический анализ HTML-разметки завершен, сценарии (модульные и немодульные) выполняются в порядке, в котором они появлялись в HTML-документе.

Динамическое импортирование с помощью import()

Итак, вместо импортирования модуля " . / s t a t s . j s " статическим образом:

import * as stats from "./stats.js";

мы могли бы импортировать его и работать с ним динамически:

```
import("./stats.js").then(stats => {
      let average = stats.mean(data);
))
```

или

```
async analyzeData (data) {
      let stats = await import ("./stats.js");
      return {
            average: stats.mean (data),
            stddev: stats.stddev (data)
      }
};
```

Самый простой способ применения инструмента пакетирования кода — сообщить ему главную точку входа для программы и разрешить найти все статические директивы import и собрать все в один крупный файл. Однако, стратегически используя динамические вызовы import (), вы можете разбить такой монолитный пакет на набор пакетов меньшего размера, которые можно загружать по запросу.

——

import.meta.url

Осталось обсудить последнюю особенность системы модулей ES6.

Внутри модуля ES6 (но не в обычном <script> или модуле Node, загруженном с по мощью require ())

Специальный синтаксис import.meta ссылается на объект, который содержит метаданные о выполняющемся в текущий момент модуле.

Свойство url этого объекта хранит URL, из которого модуль был загружен. (В Node это будет URL вида f i l e : //.)

Основной вариант применения import.meta.url — возможность ссылки на изображения, файлы данных и другие ресурсы, которые хранятся в том же самом каталоге, что и модуль (или в каталоге относительно него).

Конструктор URL() облегчает распознавание URL, указанного относительно абсолютного URL наподобие im port.meta.url.

Предположим, например, что вы написали модуль, включающий строки, которые необходимо локализовать, а файлы локализации хранятся в подкаталоге 110п/, находящемся в том же каталоге, что и сам модуль.

Модуль мог бы загружать свои строки с использованием URL, созданного с помощью функции вроде показанной ниже:

```
function localStringsURL (locale) 1
      return new URL (*110n/${locale}.json', import .meta.url):
}
```

___

Цель модульности

Позволить программистам скрывать детали реализации своего кода, чтобы порции кода из различных источников можно было собирать в крупные программы, не беспокоясь о том, что одна порция перезапишет функции или переменные другой порции. В главе объяснялись три системы модулей JavaScript.

• В самом начале существования JavaScript модульности можно было достичь только за счет умелого использования немедленно вызываемых выражений функций.

• Среда Node добавляет собственную систему модулей поверх языка JavaScript. Модули Node импортируются с помощью require () и определяют свои экспорты, устанавливая свойства объекта exports или
module.exports.

• В версии ES6 язык JavaScript, в конце концов, получил

собственную систему модулей с ключевыми словами import и export, а в ES2020 добавилась поддержка динамического импортирования с помощью import ().