JSDG 🦏 | The JavaScript Standard Library | Стандартная библиотека JavaScript | chapter 11

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

Некоторые типы данных, такие как числа и строки (см. главу 3), объекты (см. главу 6) и массивы (см. главу 7), настолько фундаментальны в JavaScript, что мы можем считать их частью самого языка.

Множества и отображения

Класс Set

Подобно массиву множество является коллекцией значений. Однако в отличие от массивов множества не упорядочиваются, не индексируются и не допускают наличия дубликатов: значение либо будет членом множества, либо нет; запрашивать, сколько раз значение встречается внутри множества, невозможно.

let t = new Set (s); / Новое множество, которое копирует элементы s.
let unique = new Set ("Mississippi"); //4 элемента: "M", "i", "s" „ "p"

unique.size 1 => 4 // Свойство size множества похоже на свойство length массива: оно сообщает нам, сколько значений содержит множество

let s = new Set () ; / / Начать с пустого множества
s.size // => 0
s.add(1); // Добавить число
s.size //> 1; теперь множество имеет один член
s.add (1) // добавить то же самое число еще раз
s.size // 2
s.add (true); // Добавить другое значение; обратите внимание на

```
допустимость смешивания типов
s.size // 2
s.add ([1,2,3]); //
s.size // 3
s.delete (1) // true: успешное удаление элемента 1
s.size // => 2: размер уменьшился до 2
s.delete ("test"')  // false: "test" не является ченом, удаление
терпит неудачу
s.delete (true)  // true: удаление прошло успешно
s.delete ([1,2,3]) // => false: массив во множестве отличается
s.size // 1 во множестве по-прежнему присутствует один массив
s.clear (); // Удалить все из множества
s.size // => 0
```

Множество может содержать число 1 и строку "1", потому что считает их несовпадающими значениями. Когда значения являются объектами (либо массивами или функциями), они также сравниваются, как если бы применялась операция ===. Вот почему мы не смогли удалить элемент типа массива из множества в показанном выше коде. Мы добавили к множеству массив и затем попытались удалить его, передавая методу delete () другой массив (хотя с теми же самыми элементами). Чтобы все работало, нам нужно было передавать ссылку на точно тот же массив.

```
let oneDigitPrimes = new Set ( [2,3,5,7]);
oneDigitPrimes.has (2) // => true: 2 - простое число с одной
цифрой
oneDigitPrimes.has (3) // => true: то же касается 3
oneDigitPrimes. has (4) // => false: 4 - не простое число
oneDigitPrimes.has ("5") // => false: "5" - даже не число
```

Относительно множеств важнее всего понимать то, что они оптимизированы для проверки членства, и независимо от того, сколько членов содержится во множестве, метод has () будет очень быстрым. Метод includes () массива тоже выполняет проверку членства, но необходимое для проверки время пропорционально размеру массива, а применение массива в качестве множества может оказаться гораздо медленнее, чем использование настоящего объекта Set.

```
[...oneDigitPrimes]  // => [2,3,5,7]: множество, преобразованное в
```

Array
Math.max(...oneDigitPrimes) // => 7: элементы множества, передаваемые как аргументы

Множества часто описывают как "неупорядоченные коллекции". Тем не менее, это не совсем верно для класса Set в JavaScript. Множество JavaScript не поддерживает индексацию: вы не можете запрашивать, скажем, первый или третий элемент у множества, как могли бы делать в случае массива. Но класс set В JavaScript всегда запоминает порядок, в котором вставлялись элементы, и он всегда использует такой порядок при проходе по множеству: первый вставленный элемент будет первым пройденным (при условии, что вы его сначала не удалили), а последний вставленный элемент - последним пройденным.

Метод forEach () массива передает индексы массива в виде второго аргу мента указанной вами функции. Множества не имеют индексов, а потому вер сия forEach () класса Set просто передает значение элемента как первый и второй аргументы.

Класс Map

Объект Map представляет набор значений, называемых *ключами*, где с каждым ключом ассоциировано (или “отображено” на него) еще одно значение. В некотором смысле отображение похоже на массив, но вместо применения набора последовательных целых чисел для ключей отображения позволяют использовать в качестве “индексов” произвольные значения. Подобно массивам отображения характеризуются высокой скоростью работы: поиск значения, ассоциированного с ключом, будет быстрым (хотя и не настолько быстрым, как индексация в массиве) независимо от размера отображения.

```
let m = new Map ( ) ; / / Новое пустое отображение
let n = new Map ( [ / Новое отображение, которое
инициализировано строковыми ключами, отображенными на
числа
      ["one", 11,
      ["two", 2]
]);

let o = { x: 1, y: 2}; / Объект с двумя свойствами
```

```
let p = new Map (Object. entries (0)) ; // То же, что и new map([["x", 1], ["y", 2]])

m.get ("three") // => undefined: этот ключ отсутствует в отображении
m. set ("one", true); //
m.size
m. has ("one")
m.has (true)
m.delete ("one")
m.size
m.delete ("three")
m.clear () ; // Удалить все ключи и значения из отображения

let m = new Map () . set ("one", 1). set ("two", 2) . set ("three", 3) ;
m.size // => 3
m.get("two") // => 2
```

Как и в Set, в качестве ключа или значения Map можно применять любое значение JavaScript, включая null, undefined и NaN, а также ссылочные типы вроде объектов и массивов. И подобно классу Set класс Map сравнивает ключи на предмет идентичности, а не равенства, так что если вы используете для ключа объект или массив, то он будет считаться отличающимся от любого другого объекта или массива даже при наличии в нем одинаковых свойств или элементов:

```
let m = new Map () ; / Начать с пустого отображения
m. set ({), 1); // Отобразить один пустой объект на число 1
m.set ({), 2); // Отобразить другой пустой объект на число 2
m.size // => 2: в отображении есть два ключа
m.get (t]) // => undefined: но этот пустой объект не является ключом
m.set (m, undefined); // Отобразить само отображение на значение undefined
m. has (m) => true: отображение т - ключ в самом себе
m.get (m) => undefined: такое же значение мы получили бы, если бы т не было ключом
```

При проходе по отображению с помощью цикла for/of идиоматично использовать деструктурирующее присваивание, чтобы присвоить ключ и значение отдельным переменным:

```
let m = new Map (I["x", 1], ["y", 2]]);
[...m] // => [I"x", 1], ["y", 2]];
for (let [key, value] of m) {
      // На первой итерации ключом будет "" • а значением - 1
      // На второй итерации ключом будет "y", а значением - 2
}
```

Если вы хотите проходить только по ключам или только по ассоциированным значениям отображения, тогда применяйте методы keys () и values (): они возвращают итерируемые объекты, которые обеспечивают проход по ключам и значениям в порядке вставки. (Метод entries () возвращает итерируемый объект, который делает возможным проход по парам ключ/значение, но результат будет в точности таким же, как итерация непосредственно по отображению.)

```
[...m. keys () // ["x", "y"]: только ключи
[...m.values ()] // [1, 2]: только значения
[...m.entries () ] // => [["x", 1], ["y", 2]]: то же, что и [...m]
```

WeakMap и WeakSet

Класс WeakMap является разновидностью (но не фактическим подклассом) класса Map, которая не препятствует обработке значений своих ключей сборщиком мусора.

Сборка мусора — это процесс, посредством которого интерпретатор JavaScript восстанавливает память, занимаемую объектами, которые перестали быть “достижимыми” и не могут использоваться программой.

Обыкновенное отображение хранит “сильные” ссылки на значения своих ключей, и они остаются достижимыми через отображение, даже если все прочие ссылки на них исчезли. Напротив, WeakMap хранит “слабые” ссылки на значения своих ключей, так что они не достижимы через WeakMap, а их присутствие в отображении не препятствует восстановлению занимаемой ими памяти.

Конструктор WeakMap () похож на конструктор Map (), но между

WeakMap и Map существует несколько важных отличий.
• Ключи WeakM обязаны быть объектами или массивами; элементарные значения не подлежат сборке мусора и не могут применяться в качестве ключей.
• Класс WeakM реализует только методы get (), set () , has () и delete (). В частности, WeakM не является итерируемым и не определяет keys (), values () или forEach () . Если бы класс WeakM был итерируемым, тогда его ключи оказались бы достижимыми и не слабыми.
• Аналогично WeakM не реализует свойство size, потому что размер WeakM может измениться в любой момент из-за обработки обьектов сборщиком мусора.

=====
Если вы реализуете кеш с использованием объекта Map, то воспрепятствуете восстановлению памяти, занимаемой любым объектом, но в случае применения WeakMap проблема не возникнет.
(Часто вы можете добить ся похожего результата, используя закрытое свойство Symbol для кеширования вычисленного значения прямо в объекте. См. подраздел 6.10.3.)

Класс WeakSet реализует множество объектов, которые не препятствуют обработке этих объектов сборщиком мусора. Конструктор WeakSet() работает подобно конструктору Set(), но объекты WeakSet отличаются от объектов Set , в тех же аспектах, в каких объекты WeakMap отличаются от объектов Map.

• Класс WeakSet не разрешает элементарным значениям быть членами.
• Класс WeakSet реализует только методы add (), has () и delete () и не является итерируемым.
• Класс WeakSet не имеет свойства size.

Типизированные массивы и двоичные данные

Типизированные массивы появившиеся в ES6, намного ближе к низкоуровневым массивам в упомянутых языках.
Типизированные массивы формально не являются массивами (Array.isArray () возвращает для них false), но они реализуют все методы массивов, описанные в разделе 7.8, плюс несколько собственных методов

• Все элементы типизированного массива представляют собой числа. Тем не менее, в отличие от обыкновенных чисел JavaScript типизированные массивы позволяют указывать тип (целочисленный со знаком и без знака и с плавающей точкой IEEE-754) и размер (от 8 до 64 бит) чисел, которые будут храниться в массиве.
• Вы должны указывать длину типизированного массива при его создании, и эта длина никогда не изменяется.
• Элементы типизированного массива всегда инициализируются при создании массива.

Типизированные массивы впервые появились в JavaScript на стороне клиента, когда в веббраузеры была добавлена поддержка графики WebGL.
Нововведением ES6 стало то, что они были ПоДняты до уровня базового языкового средства.

Типы типизированных массивов

Таблица 11.1. Виды типизированных массивов

Таблица 11.1. Виды типизированных массивов

Конструктор	Числовой тип
`Int8Array()`	байты со знаком
`Uint8Array()`	байты без знака
`Uint8ClampedArray()`	байты без знака и без переноса бит
`Int16Array()`	16-битные короткие целые числа со знаком
`Uint16Array()`	16-битные короткие целые числа без знака
`Int32Array()`	32-битные целые числа со знаком
`Uint32Array()`	32-битные целые числа без знака
`BigInt64Array()`	64-битные значения `BigInt` со знаком (ES2020)
`BigUint64Array()`	64-битные значения `BigInt` без знака (ES2020)
`Float32Array()`	32-битные значения с плавающей точкой
`Float64Array()`	64-битные значения с плавающей точкой: обыкновенные числа JavaScript

---

Сопоставление с образцом с помощью регулярных выражений

Регулярное выражение — это объект, который описывает

текстовый шаблон. Регулярные выражения в JavaScript представляются посредством класса **RegExp** а в классах **String** и **RegExp** определены методы, которые применяют регулярные выражения для выполнения мощных функций сопоставления с образцом и поиска с заменой в тексте.

Определение регулярных выражений

let pattern = / s$/;

Приведенная выше строка создает новый объект RegE и присваивает его переменной pattern. Созданный объект RegExp соответствует любой строке, которая заканчивается буквой s

То же самое регулярное выражение можно было бы создать посредством конструктора RegExp () :

let pattern = new RegExp ("s$");

let pattern = /s$/i;

Несколько символов пунктуации имеют особый смысл в регулярных выра жениях. Вот они:

^ $. * + ? = | \ / ( ) [ ] { }

Таблица 11.2. Буквальные символы в регулярных выражениях
Таблица 11.3. Классы символов в регулярных выражениях
Таблица 11.4. Символы повторения в регулярных выражениях

Символ	Соответствие
Алфавитно-цифровой символ	Сам себе
`\0`	Символ NUL (`\u0000`)
`\t`	Табуляция (`\u0009`)
`\n`	Новая строка (`\u000A`)
`\v`	Вертикальная табуляция (`\u000B`)
`\f`	Перевод страницы (`\u000C`)
`\r`	Возврат каретки (`\u000D`)
`\x`*nn*	Символ Latin, указанный шестнадцатеричным числом *nn*; например, `\x0A` — то же самое, что и `\n`
`\u`*xxxx*	Символ Unicode, указанный шестнадцатеричным числом *xxxx*; например, `\u0009` — то же самое, что и `\t`
`\u{`*n*`}`	Символ Unicode, указанный кодовой точкой *n*, где *n* — от одной до шести шестнадцатеричных цифр между 0 и 10FFFF. Обратите внимание, что такой синтаксис поддерживается только в регулярных выражениях, использующих флаг `u`
`\c`*X*	Управляющий символ `^`*X*; например, `\cJ` эквивалентно символу новой строки `\n`

Таблица 11.3. Классы символов в регулярных выражениях

Класс символов	Соответствие
`[...]`	Любой один символ, находящийся между квадратными скобками
`[^...]`	Любой один символ, не находящийся между квадратными скобками
`.`	Любой символ за исключением символа новой строки или другого разделителя строк Unicode. Или если `RegExp` использует флаг `s`, тогда точка соответствует любому символу, в том числе разделителям строк
`\w`	Любой символ слов ASCII. Эквивалентно `[a-zA-Z0-9_]`
`\W`	Любой символ, который не является символом слов ASCII. Эквивалентно `[^a-zA-Z0-9_]`
`\s`	Любой пробельный символ Unicode
`\S`	Любой символ, не являющийся пробельным символом Unicode
`\d`	Любая цифра ASCII. Эквивалентно `[0-9]`
`\D`	Любой символ, отличающийся от цифры ASCII. Эквивалентно `[^0-9]`
`[\b]`	Буквальный забой (особый случай)

Таблица 11.4. Символы повторения в регулярных выражениях

Символ	Что означает
{*n*,*m*}	Соответствует предшествующему элементу, по крайней мере, *n* раз, но не более *m* раз
{*n*,}	Соответствует предшествующему элементу *n* или больше раз
{*n*}	Соответствует в точности *n* вхождениям предшествующего элемента
?	Соответствует нулю или одному вхождению предшествующего элемента. То есть предшествующий элемент необязателен. Эквивалентно {0,1}
+	Соответствует одному или большему количеству вхождений предшествующего элемента. Эквивалентно {1,}
*	Соответствует нулю или большему количеству вхождений предшествующего элемента. Эквивалентно {0,}

Ниже приведены примеры:

```
let r = /\d{2,4}/; // Соответствует цифрам в количестве
                   // от двух до четырех
r = /\w{3}\d?/;    // Соответствует в точности трем символам слов
                   // и необязательной цифре
r = /\s+java\s+/;  // Соответствует "java" с одним
                   // и более пробелов до и после
r = /[^(]*/;   // Соответствует нулю или большему количеству символов,
               // которые не являются открывающими круглыми скобками
```

Таблица 11.5. Символы чередования, группирования и ссылки в регулярных выражениях

Символ	Что означает
|	Чередование: соответствует либо подвыражению слева, либо подвыражению справа
(...)	Группирование: группирует элементы в единое целое, которое может использоваться с *, +, ?, | и т.д. Также запоминает символы, которые соответствуют этой группе, для применения в последующих ссылках
(?:...)	Только группирование: группирует элементы в единое целое, но не запоминает символы, которые соответствуют этой группе
\n	Соответствует тем же символам, которые дали совпадение, когда впервые сопоставлялась группа номер *n*. Группы — это подвыражения внутри (возможно вложенных) круглых скобок. Номера назначаются группам путем подсчета левых круглых скобок слева направо. Группы, сформированные с помощью (?:, не нумеруются

Таблица 11.6. Якорные символы в регулярных выражениях

Символ	Что означает
^	Соответствует началу строки или при наличии флага m началу линии в строке
$	Соответствует концу строки или при наличии флага m концу линии в строке
\b	Соответствует границе слова, т.е. позиции между символом \w и символом \W либо между символом \w и началом или концом строки. (Тем не менее, имейте в виду, что [\b] соответствует символу забоя.)
\B	Соответствует позиции не в границе слова
(?=*p*)	Утверждение положительного просмотра вперед. Требует, чтобы последующие символы соответствовали шаблону *p*, но не включает их в найденное соответствие
(?!*p*)	Утверждение отрицательного просмотра вперед. Требует, чтобы последующие символы не соответствовали шаблону *p*

Нежадное повторение

/a+?b/ // "aaab"

Чередование, группирование и ссылки

/ab|cd|ef/ соответствует строке "ab" или строке "cd" или строке "ef"
/\d{3}|[a-z] {4}/ соответствует либо трем цифрам, либо четырем буквам нижнего регистра

В ES2018 синтаксис регулярных выражений был расширен, чтобы сделать возможными утверждения “просмотра назад” Они похожи на утверждения просмотра вперед, но обращаются к тексту перед текущей позицией совпадения.

Указывайте утверждение положительного просмотра назад с помощью (?<=..•) и утверждение отрицательного просмотра назад посредством (? < ! . . .) . Например, если вы работали с почтовыми адресами в США, то могли бы выполнять сопоставление с почтовым индексом из пяти цифр, но только когда он следует за двухбуквенным сокращением названия штата:

/(?<= [A-Z]{2})\d{5}/

И с помощью утверждения отрицательного просмотра назад вы можете вы полнять сопоставление со строкой цифр, которой не

предшествует символ валюты:

/(?<![\p{Currency_Symbol}\d.])\d+(\.\d+)?/u

Флаги

• g. Флаг у указывает, что регулярное выражение является "глобальным", т.е. мы намерены его использовать для нахождения всех совпадений внутри строки, а только первого совпадения. Флаг д не изменяет способ сопоставления с образцом, но, как будет показано позже, он изменяет поведение метода match () класса String и метода exec () класса RegEx во многих важных аспектах.
• i. Флаг i указывает, что сопоставление с образцом должно быть нечувствительным к регистру.
• m. Флаг т указывает, что сопоставление с образцом должно выполняться в "многострочном" режиме. Он говорит о том, что объект RegExp будет применяться с многострочными строками, а якоря ^ и $ должны соответствовать началу и концу строки плюс началу и концу индивидуальных линий внутри строки.
• s. Подобно т флаг з также полезен при работе с текстом, который включает символы новой строки. Обычно точка (.) в регулярном выражении соответствует любому символу кроме разделителя строк. Однако когда используется флаг s, точка будет соответствовать любому символу, в том числе разделителям строк. Флаг s был добавлен к JavaScript в ES2018 и по
состоянию на начало 2020 года он поддерживается в Node, Chrome, Edge и Safari, но не в Firefox.
• u. Флаг u означает Unicode и заставляет регулярное выражение соответствовать полным кодовым точкам Unicode, а не 16-битным значениям. Флаги был введен в ES6, и вы должны выработать привычку применять его во всех регулярных выражениях, если только не существует веская причина не поступать так. Если вы не используете этот флаг, тогда ваши объекты RegExp не будут нормально работать с текстом, содержащим эмотиконы и другие символы (включая многие китайские иероглифы), которые требуют более 16 бит. В отсутствие флага и символ . соответствует любому одному 16-битному значению UTF-16. Тем не менее, при наличии флага и символ . соответствует одной кодовой точке Unicode, в том числе содержащей свыше 16 бит. Установка флага и в объекте RegExp

также позволяет применять новую управляющую последовательность \u { ...) для символа Unicode и делает возможной запись \p {...} для классов символов Unicode.
• y. Флаг y указывает на то, что регулярное выражение является "липким" и должно совпадать в начале строки или на первом символе, который следует за предыдущим соответствием. При использовании с регулярным выражением, предназначенным для поиска одиночного совпадения, флаг y фактически трактует это регулярное выражение так, как если бы оно начиналось с ^ с целью его прикрепления к началу строки. Этот флаг более полезен с регулярными выражениями, которые применяются многократно для поиска всех совпадений внутри строки. В таком случае он инициирует специальное поведение метода match () класса String и метода exec () класса RegExp, чтобы обеспечить прикрепление каждого последующего совпадения к позиции в строке, в которой закончилось последнее совпадение.

___

Строковые методы для сопоставления с образцом

search()
replace()
match ()
matchAll ()
split ()

Свойства RegExp

test ()
exec ()

___

Дата и время

let now = new Date () ; / Текущее время
let epoch = new Date (0); / Полночь, 1 января 1970 года, гринвичское среднее время

Если вы хотите указать дату и время в скоординиро ванном всемирном времени (Universal Time Coordinated — UTC, оно же грин вичское среднее время, Greenwich Mean Time — GMT), тогда можете использо вать Date.UTC().

```
// Полночь в Англии, 1 января 2100 года
let century = new Date (Date. UTC (2100, 0, 1));
```

В случае вывода даты (скажем, посредством console.log (century)) по умолчанию она выводится в локальном часовом поясе. Если вы хотите отображать дату и время в UTC, тогда должны явно преобразовать ее в строку с помощью toUTCString () или toISOString ().

Отметки времени с высоким разрешением

```
performance.now()
```

Объект performance входит в состав более крупного API-интерфейса Performance, который не определено стандартом ECMAScript, но реализован веб-браузерами и Node. Чтобы использовать объект performance в Node, его потребуется импортировать:

```
const { performance } = require ("perf_hooks") ;
```

Как разработчик веб-приложений, вы должны быть в состоянии каким-то образом заново включать высокоточное измерение времени (скажем, путем установки privacy. reduceTimerPrecision в false в Firefox).

Форматирование и разбор строк с датами

• toString () . Этот метод применяет локальный часовой пояс, но не форматирует дату и время с учетом локали.
• toUTCString () . Этот метод использует часовой пояс UTC, но не форматирует дату и время с учетом локали.
• toISOString () . Этот метод выводит дату и время в формате год-месяц-день часы:минуты:секунды миллисекунды стандарта ISO-8601. Буква "" в выводе отделяет порцию даты от порции

времени. Время выражается в UTC, на что указывает буква "Z" в конце вывода.
• toLocaleString () . Этот метод применяет локальный часовой пояс и формат, соответствующий локали пользователя.
• toDateString () . Этот метод форматирует только порцию даты объекта Date, опуская время. Он использует локальный часовой пояс и не выполняет форматирование, соответствующее локали.
• toLocaleDateString () . Этот метод форматирует только дату. Он применяет локальный часовой пояс и формат, соответствующий локали.
• toTimestring () . Этот метод форматирует только время, опуская дату. Он использует локальный часовой пояс, но не выполняет форматирование времени, соответствующее локали.
• toLocaleTimeString () . Этот метод форматирует время в соответствии с локалью и применяет локальный часовой пояс.

Наконец, в дополнение к методам, преооразующим ооъект расе в строку имеется также статический метод Date.parse (), который принимает в своем аргументе строку, пытается разобрать ее как дату и время и возвращает отметку времени, представляющую эту дату. Метод Date.parse () способен разбирать такие же строки, как и конструктор Date (), и гарантированно может прово пить разбор вывода методов tolsoString (), toUrcstring () и toString ().

Классы ошибок

Кроме класса E r r o r в JavaScript определено несколько подклассов, которые применяются для сигнализации об ошибках специфических типов, устанавливаемых стандартом ECMAScript: EvalError, RangeError, ReferenceError, SyntaxError, TypeError и URIError

Сериализация и разбор данных в формате JS0N

Когда программе необходимо сохранять данные либо передавать их через сетевое подключение другой программе, она должна преобразовать структуры данных внутри памяти в строку байтов или символов, которые можно сохранить или передать и позже произвести разбор с целью восстановления первоначальных структур данных внутри памяти. Такой процесс преобразования структур данных в потоки байтов или символов

называется сериализацией (или маршализацией либо даже складированием).

Простейший способ сериализации данных в JavaScript предусматривает применение формата ISON. Аббревиатура ISON означает "JavaScript Obiect Notation" (представление объектов JavaScript) и, как вытекает из названия, формат JSON использует синтаксис литералов типа объектов и массивов JavaScript для преобразования структур данных, состоящих из объектов и массивов, в строки. JSON поддерживает элементарные числа и строки плюс значения true, false и null, а также массивы и объекты, построенные из таких элементарных значений. JSON не поддерживает другие типы JavaScript вроде Map, Set, RegExp, Date или типизированных массивов. Однако он оказался удивительно универсальным форматом данных и широко применяется с программами, не основанными на JavaScript.

```
let o = {s: "" , n: 0, a: [true, false, null]l;
let s = JSON.stringify(o); //s == '("s":"", "n": 0, "a": [true, false, null])'
let copy = JSON.parse (s); // copy == (s:"", n:0, a: [true, false, null])
```

Если мы пропустим часть, где сериализированные данные сохраняются или посылаются по сети, то можем использовать эту пару функций как несколько неэффективный способ создания глубокой копии объекта:

```
// Создать глубокую копию любого сериализируемого объекта или массива
function deepcopy (o) {
      return JSON.parse(JSON.stringify (o));
}

let o = ls: "test", n: 01;
JSON.stringify(o, null, 2)

// Указать, какие поля должны сериализироваться и в каком порядке
let text = JSON.stringify (address, ["city", "state", "country"]);

// Указать функцию замещения, которая опускается свойства со значениями RegExp
let json-JSON. stringify (o, (k, v) => v instanceof RegExp? undefined:
```

v) ;

___

API-интерфейс интернационализации

API-интерфейс интернационализации JavaScript состоит из трех классов **Intl.NumberFormat, Intl.DateTimeFormat, Intl.Collator,** которые позволяют форматировать числа (включая денежные суммы и процентные отношения), дату и время, а также сравнивать сроки способами, соответствующими локали.

Форматирование даты и времени

Вот несколько примеров:
let d = new Date ("2020-01-02T13:14:15Z"); //2 января 2020 года, 13:14:15 UTC

// Без параметров мы получаем базовый числовой формат даты
Intl. DateTimeFormat ("en-US") . format (d) // => "1/2/2020"
Intl. DateTimeFormat ("fr-FR") .format (d) // => "02/01/2020"

// Выводить день недели и месяц
let opts= { weekday: "long", month: "long", year: "numeric", day: "numeric" };
Intl. DateTimeFormat ("en-US", opts) .format (d) //=> "Thursday, January 2, 2020"
Intl. DateTimeFormat ("es-ES", opts) .format (d) //=>"juves, 2 de enero de 2020"

// Время в Нью-Йорке для франкоговорящих канадцев
opts= { hour: "numeric", minute: "2-digit", timeZone: "America/New York" }:
Intl.DateTimeFormat("fr-CA", opts).format (d) // => "8 h 14

___

API-интерфейс Console

• console. log () . Это самая известная консольная функция, которая пре. образует свои аргументы в строки и выводит их на консоль. Она включает пробелы между аргументами и переходит

на новую строку после вывода всех аргументов.
• console. debug (), console. info (), console. warn (), console. error (). Эти функции практически идентичны console. log (). В Node функция console.error () посылает свой вывод в поток stderr, а не stdout, но остальные функции являются псевдонимами console. log (). В браузерах выходные сообщения, генерируемые каждой функцией, могут предваряться значком, который указывает уровень их серьезности, и консоль разработчика может также позволять разработчикам фильтровать консольные сообщения по уровню серьезности.
• console. assert () . Если первый аргумент истинный (т.е. утверждение проходит), тогда эта функция ничего не делает. Но когда в первом аргументе передается false или другое ложное значение, то оставшиеся аргументы выводятся, как если бы они были переданы функции console.error () с префиксом "Assertion failed" (Отказ утверждения). Обратите внимание, что в отличие от типичных функций assert () в случае отказа утверждения функция console.assert () не генерирует исключение.
• console. clear () . Данная функция очищает консоль, когда это возможно. Она работает в браузерах и Node при отображении средой Node своего вывода в окне терминала. Тем не менее, если вывод Node перенаправляется в файл или канал, то вызов console.clear () ничего не делает.
• console. table () . Эта функция является необыкновенно мощным, но малоизвестным инструментом для выработки табличного вывода, который особенно удобен в программах Node, нуждающихся в выдаче вывода с подытоженными данными. Функция console.table () пытается отобразить свои аргументы в табличной форме (и если ей это не удается, то она отображает данные с использованием обычного форматирования console. log ()). Функция console.table () работает лучше всего, когда аргумент представляет собой относительно короткий массив объектов, и все объекты в массиве имеют один и тот же (относительно небольшой) набор свойств. В таком случае каждый объект в массиве форматируется в виде строки таблицы, причем каждое свойство становится столбцом таблицы. Можно также передать в необязательном втором аргументе массив имен свойств, чтобы указать желаемый набор столбцов. Если вместо массива объектов передан единственный объект, тогда на выходе будет таблица со столбцами для имен свойств и одной строкой для значений свойств. Если сами значения свойств являются

объектами, то имена их свойств станут столбами в таблице.

• console.trace () . Эта функция выводит свои аргументы подобно console. log () и вдобавок сопровождает вывод информацией трассировки стека. В Node вывод направляется в stderr, а не в stdout.
• console.count () . Функция console. count () принимает строковый аргумент и выводит эту строку вместе с количеством ее вызовов с данной строкой. Она может быть полезна, скажем, при отладке обработчика событий, если необходимо отслеживать, сколько раз обработчик событий запускался.
• console. countReset () . Эта функция принимает строковый аргумент и сбрасывает счетчик для данной строки.
• console.group () . Эта функция выводит свои аргументы на консоль, как если бы они были переданы функции console. log (), и затем устанавливает внутреннее состояние консоли, так что все последующие консольные сообщения (вплоть до вызова console.groupEnd ()) будут иметь отступы относительно текущего выведенного сообщения. Такой прием позволяет визуально выделять группу связанных сообщений с помощью отступов. Консоль разработчика в веб-браузерах обычно дает возможность сворачивать и разворачивать сгруппированные сообщения. Аргументы функции console.group (), как правило, применяются для снабжения группы поясняющим именем.
• console.groupCollapsed () . Эта функция работает подобно console.group (), но в веб-браузерах группа по умолчанию будет "свернутой", а содержащиеся в ней сообщения окажутся скрытыми до тех пор, пока пользователь не выполнит щелчок, чтобы развернуть группу. В Node данная функция является синонимом для console.group ().
• console.groupEnd () . Эта функция не принимает аргументов. Она не производит собственного вывода, но заканчивает отступы и группирование, обусловленные самым последним вызовом console.group () или console.groupCollapsed ().
• console. time () . Эта функция принимает единственный строковый аргумент, записывает время своего вызова с данной строкой и ничего не выводит. • console. timeLog () . Эта функция принимает строку в качестве своего первого аргумента. Если такая строка ранее передавалась методу console. time (), тогда функция выводит ее вместе со временем, которое прошло с момента вызова console. time () . Любые дополнительные

аргументы, указанные при вызове console. timeLog (), выводятся, как если бы они передавались функции console. log ().
• console. timeEnd () . Эта функция принимает единственный строковый аргумент. Если такая строка ранее передавалась методу console.time (), тогда она выводится вместе с истекшим временем. После вызова console. timeEnd () больше не разрешено вызывать console.timeLog () без предварительного вызова console. time ().

___

API-интерфейсы URL

let url = new URL ("https: //example.com: 8000/path/name?
q=term#fragment");
ur1.href // => "https://example.com: 8000/path/name?
q=term#fragment"
ur1.origin / => "https://example.com: 8000"
url.protocol
url.host
url.hostname
url.port
url.pathname
url.search // => "pa=tar"
url.hash

let ur1 = new URL ("ftp://admin: 1337!@ftp.example.com/");
url.href // ftp://admin: 1337!@ftp.example.com/"
url.origin
url.username // admin
url.password // 133710

let url = new URL ("https: //example.com/search");
url.search // => "»: пока запроса нет
url. searchParams. append ("q", "term"); // Добавить параметр поиска
url.search // => "?q=term"
url.searchParams.set ("g", "x") ; / / Изменить значение этого параметра
url.search 11 => "?g=x"
uri.searchParams.get ("g") // => "x": запросить значение параметра

url.searchParams.has ("g") => true: имеется параметр 9
url.searchParams.has ("p") / => false: параметр p отсутствует
url.searchParams.append ("opts", "l") ; / / Добавить еще один параметр поиска
url. search // => "?q=x&opts=1"
url. searchParams.append ("opts", "&") ; / / Добавить еще одно значение
// для того же самого имени
url.search // => "?g=x&opts=1&opts=826":
обратите внимание на отмену
url. searchParams. get ("opts") // => "1": первое значение
url.searchParams.getAll ("opts") // => ["l", "&"]: все значения
url. searchParams.sort (); / / Разместить параметры в алфавитном порядке
url. search // => "?opts=1&opts=826&g=x"
url. searchParams.set ("opts", "y"); / / Изменить параметр opts
url.search // => "?opts=y&q=x"
/ Объект searchParams итерируемый
[•..url.searchParams] / => ll"opts", "y"], ["g", "x"1)
uri.searchParams.delete ("opts"); / / Удалить параметр opts
url. search // => "?q=x"
url.href / => "https://example.com/search?q=x"

___

Унаследованные функции для работы с UR

encodeURI() и decodeURI()
encodeURIComponentO и decodeURIComponent ( )

___

Таймеры

setTimeout (() => { console. log ("Ready ..."); J, 1000);
setTimeout (() => { console.log ("set..."); }, 2000);
setTimeout (() => { console. log ("go!");), 3000);

// Раз в секунду: очистить консоль и вывести текущее время.
let clock = setInterval (l) => {
    console.clear ();

```
      console. log (new Date () . tolocaleTimeString () );
} , 1000);

// Спустя 10 секунд: прекратить повторение выполнения кода
выше.
setTimeout (() => { clearInterval (clock); , 10000);
```