JavaScript - работа с данными

Деструктуризация, работа с датой, JSON, spread оператор

Дата в JavaScript

Встроенный объект: Date. Он содержит дату и время, а также предоставляет методы управления ими. Его можно использовать для хранения времени создания/изменения, для измерения времени или просто для вывода текущей даты.

Объекты Date в JavaScript представляют момент времени в независимом от платформы формате. Объект Date содержит число миллисекунд, прошедших с полуночи (00:00:00 UTC) 1 января 1970 года (этот момент называют «эпохой Unix»).

Создание - new Date()

Для создания нового объекта Date нужно вызвать конструктор new Date()

Без аргументов – создать объект Date с текущими датой и временем

new Date(milliseconds) - Создать объект Date с временем, равным количеству миллисекунд (тысячная доля секунды), прошедших с 1 января 1970 года UTC+0.

new Date(datestring) - Если аргумент всего один, и это строка, то из неё «прочитывается» дата.

new Date(year, month, date, hours, minutes, seconds, ms) - Создать объект Date с заданными компонентами в местном часовом поясе. Обязательны только первые два аргумента.

- year должен состоять из четырех цифр. Для совместимости также принимаются 2 цифры и рассматриваются как 19xx, к примеру, 98 здесь это тоже самое, что и 1998, но настоятельно рекомендуется всегда использовать 4 цифры.
- month начинается с 0 (январь) по 11 (декабрь).
- Параметр date здесь представляет собой день месяца. Если параметр не задан, то принимается значение 1.
- Если параметры hours/minutes/seconds/ms отсутствуют, их значением становится 0.

```
let now = new Date();
alert( now ); // показывает текущие дату и время
```

```
// 0 соответствует 01.01.1970 UTC+0
let Jan01_1970 = new Date(0);
alert( Jan01_1970 );

// теперь добавим 24 часа и получим 02.01.1970 UTC+0
let Jan02_1970 = new Date(24 * 3600 * 1000);
alert( Jan02_1970 );
```

```
let date = new Date("2017-01-26");
alert(date);
// Время не указано, поэтому оно ставится в полночь по Гринвичу и
// меняется в соответствии с часовым поясом места выполнения кода
// Так что в результате можно получить
// Thu Jan 26 2017 11:00:00 GMT+1100 (восточно-австралийское время)
// или
// Wed Jan 25 2017 16:00:00 GMT-0800 (тихоокеанское время)
```

```
new Date(2011, 0, 1, 0, 0, 0, 0); // // 1 Jan 2011, 00:00:00
new Date(2011, 0, 1); // то же самое, так как часы и проч. равны 0
```

```
let date = new Date(2011, 0, 1, 2, 3, 4, 567);
alert( date ); // 1.01.2011, 02:03:04.567
```

Получение компонентов даты

Существуют методы получения года, месяца и т.д. из объекта Date:

- getFullYear() - Получить год (4 цифры)
- getMonth() - Получить месяц, от 0 до 11.
- getDate() - Получить день месяца, от 1 до 31, что несколько противоречит названию метода.
- getHours(), getMinutes(), getSeconds(), getMilliseconds()
  Получить, соответственно, часы, минуты, секунды или миллисекунды.
- getDay() - Вернуть день недели от 0 (воскресенье) до 6 (суббота).
- getTime() - Для заданной даты возвращает таймстамп – количество миллисекунд, прошедших с 1 января 1970 года UTC+0.

Все вышеперечисленные методы возвращают значения в соответствии с местным часовым поясом.
Однако существуют и их UTC-варианты, возвращающие день, месяц, год для временной зоны UTC+0: getUTCFullYear(), getUTCMonth(), getUTCDay(). Для их использования требуется после "get" подставить "UTC".

Никакого getYear(). Только getFullYear()
Многие интерпретаторы JavaScript реализуют нестандартный и устаревший метод getYear(), который порой возвращает год в виде двух цифр.

Установка компонентов даты

- setFullYear(year, [month], [date])
- setMonth(month, [date])
- setDate(date)
- setHours(hour, [min], [sec], [ms])
- setMinutes(min, [sec], [ms])
- setSeconds(sec, [ms])
- setMilliseconds(ms)
- setTime(milliseconds) (устанавливает дату в виде целого количества миллисекунд, прошедших с 01.01.1970 UTC)

У всех этих методов, кроме setTime(), есть UTC-вариант, например: setUTCHours().

Автоисправление даты

Автоисправление – это очень полезная особенность объектов Date. Можно устанавливать компоненты даты вне обычного диапазона значений, а объект сам себя исправит.

Предположим, нам требуется увеличить дату «28 февраля 2016» на два дня. В зависимости от того, високосный это год или нет, результатом будет «2 марта» или «1 марта». Нам об этом думать не нужно. Просто прибавляем два дня.

```
let date = new Date(2013, 0, 32); // 32 Jan 2013 ?!?
alert(date); // ...1st Feb 2013!
```

```
let date = new Date(2016, 1, 28);
date.setDate(date.getDate() + 2);

alert( date ); // 1 Mar 2016
```

```
let date = new Date();
date.setSeconds(date.getSeconds() + 70);

alert( date ); // выводит правильную дату
```

Преобразование к числу, разность дат

Если объект Date преобразовать в число, то получим таймстамп по аналогии с date.getTime():

Важный побочный эффект: даты можно вычитать, в результате получаем разность в миллисекундах.

```
let start = new Date(); // начинаем отсчёт времени

// выполняем некоторые действия
for (let i = 0; i < 100000; i++) {
  let doSomething = i * i * i;
}

let end = new Date(); // заканчиваем отсчёт времени

alert( `Цикл отработал за ${end - start} миллисекунд` );
```

```
let date = new Date();
alert(+date); // количество миллисекунд, то же самое, что date.getTime()
```

Date.now()

Если нужно просто измерить время, объект Date нам не нужен.

Существует особый метод Date.now(), возвращающий текущую метку времени.

Семантически он эквивалентен new Date().getTime(), однако метод не создаёт промежуточный объект Date. Так что этот способ работает быстрее и не нагружает сборщик мусора.

Данный метод используется из соображений удобства или когда важно быстродействие, например, при разработке игр на JavaScript или других специализированных приложений.

```
let start = Date.now(); // количество миллисекунд с 1 января 1970 года

// выполняем некоторые действия
for (let i = 0; i < 100000; i++) {
  let doSomething = i * i * i;
}

let end = Date.now(); // заканчиваем отсчёт времени

alert( `Цикл отработал за ${end - start} миллисекунд` ); // вычитаются числа
```

Разбор строки с датой

Метод Date.parse(str) считывает дату из строки.

Формат строки должен быть следующим: YYYY-MM-DDTHH:mm:ss.sssZ, где:

- YYYY-MM-DD – это дата: год-месяц-день.
- Символ "T" используется в качестве разделителя.
- HH:mm:ss.sss – время: часы, минуты, секунды и миллисекунды.
- Необязательная часть 'Z' обозначает часовой пояс в формате +-hh:mm. Если указать просто букву Z, то получим UTC+0.

Возможны и более короткие варианты, например, YYYY-MM-DD или YYYY-MM, или даже YYYY.

Вызов Date.parse(str) обрабатывает строку в заданном формате и возвращает таймстамп (количество миллисекунд с 1 января 1970 года UTC+0). Если формат неправильный, возвращается NaN.

```
let ms = Date.parse('2012-01-26T13:51:50.417-07:00');

alert(ms); // 1327611110417 (таймстамп)
```

```
let date = new Date( Date.parse('2012-01-26T13:51:50.417-07:00') );

alert(date);
```

Что такое JSON - Формат JSON, метод toJSON

JSON (JavaScript Object Notation) – это общий формат для представления значений и объектов. Его описание задокументировано в стандарте RFC 4627. Первоначально он был создан для JavaScript, но многие другие языки также имеют библиотеки, которые могут работать с ним. Таким образом, JSON легко использовать для обмена данными, когда клиент использует JavaScript, а сервер написан на Ruby/PHP/Java или любом другом языке.

Объект JSON содержит методы для разбора объектной нотации JavaScript (JavaScript Object Notation — сокращённо JSON) и преобразования значений в JSON. Его нельзя вызвать как функцию или сконструировать как объект, и кроме своих двух методов он не содержит никакой интересной функциональности.

JSON является синтаксисом для сериализации объектов, массивов, чисел, строк логических значений и значения null. Он основывается на синтаксисе JavaScript, однако всё же отличается от него: не каждый код на JavaScript является JSON, и не каждый JSON является кодом на JavaScript.

JavaScript предоставляет методы:

- JSON.stringify для преобразования объектов в JSON.
- JSON.parse для преобразования JSON обратно в объект.

JSON.stringify

Метод JSON.stringify(student) берёт объект и преобразует его в строку.

Полученная строка json называется JSON-форматированным или сериализованным объектом. Мы можем отправить его по сети или поместить в обычное хранилище данных.

JSON.stringify может быть применён и к примитивам.

JSON поддерживает следующие типы данных:

- Объекты { ... }
- Массивы [...]
- Примитивы:
- строки,
- числа,
- логические значения true/false,
- null.

```
// число в JSON остаётся числом
alert( JSON.stringify(1) ) // 1

// строка в JSON по-прежнему остаётся строкой, но в двойных кавычках
alert( JSON.stringify('test') ) // "test"

alert( JSON.stringify(true) ); // true

alert( JSON.stringify([1, 2, 3]) ); // [1,2,3]
```

```
let student = {
  name: 'John',
  age: 30,
  isAdmin: false,
  courses: ['html', 'css', 'js'],
  wife: null
};

let json = JSON.stringify(student);

alert(typeof json); // мы получили строку!

alert(json);
/* выведет объект в формате JSON:
{
  "name": "John",
  "age": 30,
  "isAdmin": false,
  "courses": ["html", "css", "js"],
  "wife": null
}
*/
```

JSON.parse

Чтобы декодировать JSON-строку, нам нужен другой метод с именем JSON.parse.

```
// строковый массив
let numbers = "[0, 1, 2, 3]";

numbers = JSON.parse(numbers);

alert( numbers[1] ); // 1
```

```
let value = JSON.parse(str[, reviver]);
```

Spread оператор “...”

Spread syntax позволяет расширить доступные для итерации элементы (например, массивы или строки) в местах

- для функций: где ожидаемое количество аргументов для вызовов функций равно нулю или больше нуля
- для элементов (литералов массива)
- для выражений объектов: в местах, где количество пар "ключ-значение" должно быть равно нулю или больше (для объектных литералов)

Синтаксис

Для вызовов функций:

```
myFunction(...iterableObj);
```

Для литералов массива или строк:

```
[...iterableObj, '4', 'five', 6];
```

Для литералов объекта (новое в ECMAScript 2018):

```
let objClone = { ...obj };
```

Остаточные параметры (rest parameters)

Синтаксис остаточных параметров функции позволяет представлять неограниченное множество аргументов в виде массива.

JS

```
function(a, b, ...theArgs) {
  // ...
}
```

```
function myFun(a, b, ...manyMoreArgs) {
  console.log("a", a);
  console.log("b", b);
  console.log("manyMoreArgs", manyMoreArgs);
}

myFun("один", "два", "три", "четыре", "пять", "шесть");

// a, один
// b, два
// manyMoreArgs, [три, четыре, пять, шесть]
```

Деструктуризация

В JavaScript есть две чаще всего используемые структуры данных – это Object и Array.

- Объекты позволяют нам создавать одну сущность, которая хранит элементы данных по ключам.
- Массивы позволяют нам собирать элементы данных в упорядоченный список.

Деструктурирующее присваивание – это специальный синтаксис, который позволяет нам «распаковать» массивы или объекты в несколько переменных, так как иногда они более удобны.

Деструктуризация массива

Теперь мы можем использовать переменные вместо элементов массива.

Отлично смотрится в сочетании со split или другими методами, возвращающими массив:

```
// у нас есть массив с именем и фамилией
let arr = ["Ilya", "Kantor"];

// деструктурирующее присваивание
// записывает firstName = arr[0]
// и surname = arr[1]
let [firstName, surname] = arr;

alert(firstName); // Ilya
alert(surname);  // Kantor
```

```
let [firstName, surname] = "Ilya Kantor".split(' ');
alert(firstName); // Ilya
alert(surname);  // Kantor
```

«Деструктуризация» не означает «разрушение».

«Деструктурирующее присваивание» не уничтожает массив. Оно вообще ничего не делает с правой частью присваивания, его задача – только скопировать нужные значения в переменные.

Это просто короткий вариант записи:

```
// let [firstName, surname] = arr;
let firstName = arr[0];
let surname = arr[1];
```

Пропускайте элементы, используя запятые

Нежелательные элементы массива также могут быть отброшены с помощью дополнительной запятой:

В примере второй элемент массива пропускается, а третий присваивается переменной title, оставшиеся элементы массива также пропускаются (так как для них нет переменных).

```
// второй элемент не нужен
let [firstName, , title] = ["Julius", "Caesar", "Consul", "of the Roman Republic"];

alert( title ); // Consul
```

Работает с любым перебираемым объектом с правой стороны

…На самом деле мы можем использовать любой перебираемый объект, не только массивы:

```
let [a, b, c] = "abc";
let [one, two, three] = new Set([1, 2, 3]);
```

Присваивайте чему угодно с левой стороны

Мы можем использовать что угодно «присваивающее» с левой стороны.

Например, можно присвоить свойству объекта:

```
let user = {};
[user.name, user.surname] = "Ilya Kantor".split(' ');

alert(user.name); // Ilya
alert(user.surname); // Kantor
```

Цикл с .entries()

В предыдущей главе мы видели метод Object.entries(obj).

Мы можем использовать его с деструктуризацией для цикличного перебора ключей и значений объекта:

```
let user = {
  name: "John",
  age: 30
};

// цикл по ключам и значениям
for (let [key, value] of Object.entries(user)) {
  alert(`${key}:${value}`); // name:John, затем age:30
}
```

```
let user = new Map();
user.set("name", "John");
user.set("age", "30");

// Map перебирает как пары [ключ, значение], что очень удобно для деструктурирования
for (let [key, value] of user) {
  alert(`${key}:${value}`); // name:John, затем age:30
}
```

Трюк обмена переменных

Существует хорошо известный трюк для обмена значений двух переменных с использованием деструктурирующего присваивания:

Здесь мы создаём временный массив из двух переменных и немедленно деструктурируем его в порядке замены.

Таким образом, мы можем поменять местами даже более двух переменных.

```
let guest = "Jane";
let admin = "Pete";

// Давайте поменяем местами значения: сделаем guest = "Pete", а admin = "Jane"
[guest, admin] = [admin, guest];

alert(`${guest} ${admin}`); // Pete Jane (успешно заменено!)
```

Значения по умолчанию

Если в массиве меньше значений, чем в присваивании, то ошибки не будет. Отсутствующие значения считаются неопределёнными:

```
let [firstName, surname] = [];

alert(firstName); // undefined
alert(surname); // undefined
```

Если мы хотим, чтобы значение «по умолчанию» заменило отсутствующее, мы можем указать его с помощью =:

```
// значения по умолчанию
let [name = "Guest", surname = "Anonymous"] = ["Julius"];

alert(name);    // Julius (из массива)
alert(surname); // Anonymous (значение по умолчанию)
```

Деструктуризация объекта

Деструктурирующее присваивание также работает с объектами.

У нас есть существующий объект с правой стороны, который мы хотим разделить на переменные. Левая сторона содержит «шаблон» для соответствующих свойств. В простом случае это список названий переменных в {...}.

```
let {var1, var2} = {var1:…, var2:…}
```

```
let options = {
  title: "Menu",
  width: 100,
  height: 200
};

let {title, width, height} = options;

alert(title);  // Menu
alert(width);  // 100
alert(height); // 200
```

Обратите внимание на let

В примерах выше переменные были объявлены в присваивании: let {…} = {…}. Конечно, мы могли бы использовать существующие переменные и не указывать let, но тут есть подвох.

Проблема в том, что JavaScript обрабатывает {...} в основном потоке кода (не внутри другого выражения) как блок кода. Такие блоки кода могут быть использованы для группировки операторов.

Так что здесь JavaScript считает, что видит блок кода, отсюда и ошибка. На самом-то деле у нас деструктуризация.

Чтобы показать JavaScript, что это не блок кода, мы можем заключить выражение в скобки (...):

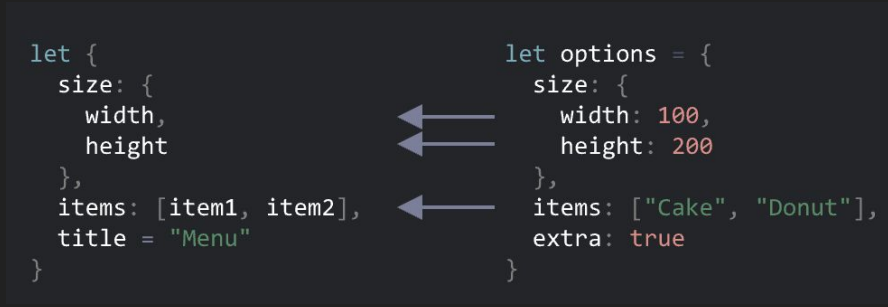
```
let title, width, height;

// ошибка будет в этой строке
{title, width, height} = {title: "Menu", width: 200, height: 100};
```

```
let title, width, height;

// сейчас всё работает
({title, width, height} = {title: "Menu", width: 200, height: 100});

alert( title ); // Menu
```

```
{
  // блок кода
  let message = "Hello";
  // ...
  alert( message );
}
```

Вложенная деструктуризация

Если объект или массив содержит другие вложенные объекты или массивы, то мы можем использовать более сложные шаблоны с левой стороны, чтобы извлечь более глубокие свойства.




```
let options = {
  size: {
    width: 100,
    height: 200
  },
  items: ["Cake", "Donut"],
  extra: true
};

// деструктуризация разбита на несколько строк для ясности
let {
  size: { // положим size сюда
    width,
    height
  },
  items: [item1, item2], // добавим элементы к items
  title = "Menu" // отсутствует в объекте (используется значение по умолчанию)
} = options;

alert(title);  // Menu
alert(width);  // 100
alert(height); // 200
alert(item1);  // Cake
alert(item2);  // Donut
```

Ресурсы

Работа с датой в JavaScript - ТЫК

Дата в JS MDN документация - ТЫК

Библиотека для работы с датой (DayJS) - ТЫК

Работа с JSON в JavaScript - ТЫК

JSON в JS MDN документация - ТЫК

Деструктуризация - ТЫК

Spread оператор MDN документация - ТЫК