Эта страница была переведена с английского языка силами сообщества. Вы тоже

Упорядоченные наборы данных

Данная глава знакомит читателя с массивами - коллекциями элементов, упорядоченных по индексу. Глава включает в себя описание массивов и массивоподобных структур, таких как `Array` и `TypedArray`.

`Array` объект

Массив представляется собой упорядоченный набор значений, к которому вы ссылаетесь по имени и индексу. Допустим, у вас есть массив с именем `emp`, содержащий имена сотрудников и упорядоченный по номеру сотрудников. Следовательно, `emp[1]` будет представлять собой имя сотрудника номер один, `emp[2]` — имя сотрудника номер два, и т.д.

Язык JavaScript не содержит явного типа данных "массив". Тем не менее, возможно использовать предопределённый объект `Array` и его методы для работы с массивами в создаваемых приложениях. Объект `Array` содержит методы для работы с массивами самыми различными способами, например, есть методы для объединения, переворачивания и сортировки. Объект содержит свойство для определения длины массива, а также свойства для работы с регулярными выражениями.

Создание массива

Следующие выражения создают одинаковые массивы:

JS

```
var arr = new Array(element0, element1, ..., elementN);
var arr = Array(element0, element1, ..., elementN);
var arr = [element0, element1, ..., elementN];
```

`element0, element1, ..., elementN` - список значений элементов массива. Если значения заданы, то эти значения будут являться элементами массива после его инициализации. Свойство `length` у массива будет равно количеству аргументов.

Синтаксис с использованием квадратных скобок называется "литерал массива" (array literal) или "инициализатор массива". Такая запись короче, чем другие способы создания массива, и, как правило, более предпочтительна. См. `Array literals`.

Для создания массива без элементов, но ненулевой длины, возможно использовать одно из следующих выражений:

JS

```
var arr = new Array(arrayLength);
var arr = Array(arrayLength);

// Точно такой же эффект
var arr = [];
arr.length = arrayLength;
```

> Примечание: в примере выше `arrayLength` должно иметь числовой тип `Number`. В противном случае будет создан массив с единственным элементом (указанное значение). Вызванная функция `arr.length` вернёт значение `arrayLength`, но на самом деле массив будет содержать пустые элементы (undefined). Использование цикла `for...in` для обработки значений массива не вернёт ни одного элемента.

Массивы могут быть присвоены свойству нового или уже существующего объекта, как показано ниже:

JS

```
var obj = {};
// ...
obj.prop = [element0, element1, ..., elementN];

// OR
var obj = {prop: [element0, element1, ...., elementN]}
```

Если вы хотите инициализировать массив одним элементом и этим элементом является число типа Number, то вы должны использовать квадратные скобки. Если вы создаёте массив с помощью Array (конструктора или функции), а единственным элементом этого массива будет число типа Number, то число это интерпретируется как длина массива (arrayLength), а не как элемент типа Number.

JS

```
var arr = [42]; // Создаёт массив  с одним элементом
var arr = Array(42); // Создаёт массив без элементов,
// но устанавливает длину массива arr.length в 42

// Это эквивалентно следующему
var arr = [];
arr.length = 42;
```

Вызов Array(N) выбросит RangeError, если N не целое значение, чья дробная часть не ноль. Следующий пример иллюстрирует это.

JS

```
var arr = Array(9.3); // RangeError: Invalid array length
```

Если ваш код нуждается в создании массива с одним элементом произвольного типа данных, то безопасней использовать литеральную запись. Или создайте пустой массив, а затем добавьте необходимый элемент.

Заполнение массива

Вы можете заполнить массив путём присвоения значений его элементам. Для примера:

JS

```
var emp = [];
emp[0] = "Casey Jones";
emp[1] = "Phil Lesh";
emp[2] = "August West";
```

> Примечание: Если вы используете нецелое значение в операторе [] обращения к элементу массива, то будет создано соответствующее свойство в объекте, представляющем массив, вместо элемента массива (так как массивы в JavaScript являются объектами).

JS

```
var arr = [];
arr[3.4] = "Oranges";
console.log(arr.length); // 0
console.log(arr.hasOwnProperty(3.4)); // true
```

Вы можете заполнить массив во время создания:

JS

```
var myArray = new Array("Hello", myVar, 3.14159);
var myArray = ["Mango", "Apple", "Orange"];
```

Работа с элементами массива

Вы можете ссылаться на элементы массива путём использования их порядковых номеров. Для примера, предположим, что вы определили следующий массив:

JS

```
var myArray = ["Wind", "Rain", "Fire"];
```

Затем вы сослались на первый элемент массива как myArray[0] и второй элемент массива как myArray[1]. Индексация элементов массива начинается с нуля.

> Примечание: оператор обращения к элементу массива (квадратные скобки []) также используется для доступа к свойствам массива (массивы также

являются объектами в JavaScript). Например:

```
JS
var arr = ["one", "two", "three"];
arr[2]; // three
arr["length"]; // Вернёт число 3, так как это свойство - длина массива
```

Понимание length

На уровне реализации, массивы в JavaScript хранят свои элементы как стандартные свойства объекта, используя индекс в качестве имени свойства. Специальное свойство `length` всегда возвращает индекс последнего элемента плюс один (в примере ниже, элемент 'Dusty' размещается под индексом 30, по этому cats.length возвращает 30 + 1). Особо следует запомнить, что в JavaScript массивы индексируются с нуля: отсчёт ведётся с 0, а не с 1. Из этого и следует, что свойство `length` всегда на единицу больше, чем наибольший индекс хранящийся в массиве:

```
JS
var cats = [];
cats[30] = ["Dusty"];
console.log(cats.length); // 31
```

Также, вы можете задавать значение для `length` . Установка значения меньшего, чем количество хранящихся в массиве элементов, обрезает массив с конца; установка `length` равным 0 очищает массив полностью:

```
JS
var cats = ["Dusty", "Misty", "Twiggy"];
console.log(cats.length); // 3

cats.length = 2;
console.log(cats); // выводит в консоль "Dusty,Misty" - элемент "Twiggy" был удалён

cats.length = 0;
console.log(cats); // выводит пустую строку; массив cats пуст

cats.length = 3;
console.log(cats); // выводит [undefined, undefined, undefined]
```

Перебор содержимого массивов

Очень распространённая задача - это перебор всех элементов массива и обработка каждого элемента некоторой операцией. Вот наипростейший способ сделать это:

JS

```
var colors = ["red", "green", "blue"];
for (var i = 0; i < colors.length; i++) {
  console.log(colors[i]);
}
```

Если вам заранее известно, что ни один элемент массива не будет расценён как `false` при приведении к boolean — например, каждый элемент массива является DOM (en-US) узлом, тогда вы можете блеснуть чуть более эффективным оборотом:

JS

```
var divs = document.getElementsByTagName("div");
for (var i = 0, div; (div = divs[i]); i++) {
  /* Обработать div некоторой операцией */
}
```

Подход в примере выше, позволяет избежать проверки длинны массива при каждой итерации, и лишь убеждается, что переменной `div` присвоен существующий текущий элемент массива при каждом прохождении цикла.

Метод `forEach()` предоставляет другой способ перебора элементов:

JS

```
var colors = ["red", "green", "blue"];
colors.forEach(function (color) {
  console.log(color);
});
```

Как вариант, вы можете сократить код программы, используя стрелочные функции из ES6:

JS

```
var colors = ["red", "green", "blue"];
colors.forEach((color) => console.log(color));
// red
// green
// blue
```

Функция, переданная в метод `forEach`, будет выполнена по одному разу для каждого элемента массива, при этом сам элемент массива будет передан как аргумент в эту функцию. Элементы, значения которым не присвоены, не обрабатываются `forEach` циклом.

Заметьте, что элементы, пропущенные при создании массива не обрабатываются методом `forEach`, однако, `undefined` элемент обрабатывается в том случае, когда он присвоен ячейке массива вручную:

JS

```
var array = ["first", "second", , "fourth"];

array.forEach(function (element) {
  console.log(element);
});
// first
// second
// fourth

if (array[2] === undefined) {
  console.log("array[2] is undefined"); // true
}

array = ["first", "second", undefined, "fourth"];

array.forEach(function (element) {
  console.log(element);
});
// first
// second
// undefined
// fourth
```

Так как в JavaScript элементы массива хранятся как обычные свойства объекта, использование `for...in` циклов для перебора элементов массива нежелательно, потому что будут обработаны не только элементы массива, но и все перечисляемые свойства массива.

Методы Array

Объект `Array` имеет следующие методы:

`concat()` объединяет два массива и возвращает новый массив.

```
var myArray = new Array("1", "2", "3");
myArray = myArray.concat("a", "b", "c");
// myArray = ["1", "2", "3", "a", "b", "c"]
```

`join(deliminator = ',')` объединяет элементы массива в текстовую строку.

```
var myArray = new Array("Wind", "Rain", "Fire");
var list = myArray.join(" - "); // list = "Wind - Rain - Fire"
```

`push()` добавляет один или несколько элементов в конец массива и возвращает результирующую длину.

```
var myArray = new Array("1", "2");
myArray.push("3"); // myArray =["1", "2", "3"]
```

`pop()` удаляет из массива последний элемент и возвращает его.

```
var myArray = new Array("1", "2", "3");
var last = myArray.pop();
// myArray =["1", "2"], last = "3"
```

`shift()` удаляет из массива первый элемент и возвращает его.

```
var myArray = new Array ("1", "2", "3");
var first = myArray.shift();
// myArray = ["2", "3"], first = "1"
```

`unshift()` добавляет один или несколько элементов в начало массива и возвращает его новую длину.

```
JS
var myArray = new Array("1", "2", "3");
myArray.unshift("4", "5");
// myArray becomes ["4", "5", "1", "2", "3"]
```

`slice(start_index, upto_index)` возвращает секцию массива как новый массив.

```
JS
var myArray = new Array("a", "b", "c", "d", "e");
myArray = myArray.slice(1, 4); // начиная с индекса 1 извлекаются элементы вплоть до индекса 3
// myArray = [ "b", "c", "d"]
```

`splice(index, count_to_remove, addElement1, addElement2, ...)` удаляет часть элементов из массива и (опционально) заменяет их. Возвращает удалённые элементы.

```
JS
var myArray = new Array("1", "2", "3", "4", "5");
myArray.splice(1, 3, "a", "b", "c", "d");
// myArray = ["1", "a", "b", "c", "d", "5"]
// Этот код, начиная с ячейки под индексом 1 (в которой находилось значение "2"),
// удаляет 3 элемента, и вставляет на их место
// элементы, переданные в качестве последующих параметров.
```

`reverse()` переставляет элементы массива в обратном порядке: первый элемент становится последним, а последний - первым.

```
JS
var myArray = new Array("1", "2", "3");
myArray.reverse();
// элементы переставлены myArray = [ "3", "2", "1" ]
```

`sort()` сортирует элементы массива.

JS

```
var myArray = new Array("Wind", "Rain", "Fire");
myArray.sort();
// массив отсортирован myArray = [ "Fire", "Rain", "Wind" ]
```

Метод `sort()` может принимать в качестве аргумента `callback`-функцию, которая определяет каким образом сравнивать элементы массива при сортировке. Функция сравнивает два значения, и возвращает одно из трёх значений (список вариантов значений смотрите после примера):

Пример. Следующий код сортирует массив по последнему символу в строке:

JS

```
var sortFn = function (a, b) {
  if (a[a.length - 1] < b[b.length - 1]) return -1;
  if (a[a.length - 1] > b[b.length - 1]) return 1;
  if (a[a.length - 1] == b[b.length - 1]) return 0;
};
myArray.sort(sortFn);
// массив отсортирован myArray = ["Wind","Fire","Rain"]
```

- если `a` меньше чем `b` в выбранной системе сравнения, возвращаем *-1 (или любое отрицательное число)*
- если `a` больше чем `b` в выбранной системе сравнения, возвращаем *1 (или любое положительное число)*
- если `a` и `b` считаются равными, возвращаем *0*.

`indexOf(searchElement[, fromIndex])` ищет в массиве элемент со значением `searchElement` и возвращает индекс первого совпадения.

JS

```
var a = ["a", "b", "a", "b", "a"];
console.log(a.indexOf("b")); // выводит 1
// Попробуем ещё раз, начиная с индекса последнего совпадения
console.log(a.indexOf("b", 2)); // выводит 3
console.log(a.indexOf("z")); // выводит -1, потому что 'z' не найдено
```

`lastIndexOf(searchElement[, fromIndex])` тоже самое, что и `indexOf`, но поиск ведётся в обратном порядке, с конца массива.

JS

```
var a = ["a", "b", "c", "d", "a", "b"];
console.log(a.lastIndexOf("b")); // выводит 5
// Попробуем ещё раз, начиная с индекса, предшествующего индексу последнего совпадения
console.log(a.lastIndexOf("b", 4)); // выводит 1
console.log(a.lastIndexOf("z")); // выводит -1
```

`forEach(callback[, thisObject])` выполняет `callback`-функцию по каждому элементу массива.

JS

```
var a = ["a", "b", "c"];
a.forEach(function (element) {
  console.log(element);
});
// выводит в консоль каждый элемент массива по порядку
```

`map(callback[, thisObject])` возвращает новый массив, содержащий результаты вызова `callback`-функции для каждого элемента исходного массива.

JS

```
var a1 = ["a", "b", "c"];
var a2 = a1.map(function (item) {
  return item.toUpperCase();
});
console.log(a2); // выводит A,B,C
```

`filter(callback[, thisObject])` возвращает новый массив, содержащий только те элементы исходного массива, для которых вызов `callback`-функции вернул true.

JS

```
var a1 = ["a", 10, "b", 20, "c", 30];
var a2 = a1.filter(function (item) {
  return typeof item == "number";
```

```
});
console.log(a2); // выводит 10,20,30
```

`every(callback[, thisObject])` возвращает true, если вызов `callback`-функции вернул true для всех элементов массива.

JS

```
function isNumber(value) {
  return typeof value == "number";
}
var a1 = [1, 2, 3];
console.log(a1.every(isNumber)); // выводит true
var a2 = [1, "2", 3];
console.log(a2.every(isNumber)); // выводит false
```

`some(callback[, thisObject])` возвращает true, если вызов `callback`-функции вернёт true хотя бы для одного элемента.

JS

```
function isNumber(value) {
  return typeof value == "number";
}
var a1 = [1, 2, 3];
console.log(a1.some(isNumber)); // выводит true
var a2 = [1, "2", 3];
console.log(a2.some(isNumber)); // выводит true
var a3 = ["1", "2", "3"];
console.log(a3.some(isNumber)); // выводит false
```

Те из методов выше, что принимают `callback`-функцию в качестве аргумента, известны как методы итерации (*iterative methods*), потому что определённым образом проходятся по всем элементам массива. Каждый из таких методов принимает второй, опциональный элемент, называемый `thisObject`. Если этот аргумент присутствует, то его значение присваивается ключевому слову `this` в теле `callback`-функции. Иначе, как и в любом другом случае вызова функции вне явного контекста, `this` будет ссылаться на глобальный объект (`window`).

В действительности `callback`-функция вызывается с тремя аргументами. Первый аргумент - текущий элемент массива, второй - индекс этого элемента, и третий - ссылка на сам массив. Однако, в JavaScript, функции игнорируют любые аргументы, которые не перечислены в списке аргументов. Таким образом, нет ничего страшного в использовании функции с одним аргументом, такой как `alert`.

`reduce(callback[, initialValue])` последовательно применяет `callback`-функцию `callback(firstValue, secondValue)` для того, чтобы свести все элементы массива к одному значению. В первый параметр функции передаётся предыдущий результат работы функции или первый элемент, а во второй - текущий элемент. Третьим параметром передаётся индекс текущего элемента.

```
JS
var a = [10, 20, 30];
var total = a.reduce(function (first, second, index) {
  return first + second;
}, 0);
console.log(total); // выводит 60
```

`reduceRight(callback[, initalvalue])` работает так же как и `reduce()`, но порядок обхода ведётся от конца к началу.

`Методы reduce и reduceRight` являются наименее очевидными методами объекта Array. Они должны использоваться в алгоритмах, которые рекурсивно совмещают два элемента массива, для сведения всей последовательности к одному значению.

Многомерные массивы

Массивы могут быть вложенными, то есть массив может содержать массивы в элементах. Используя эту возможность массивов JavaScript, можно построить многомерные массивы.

Следующий код создаёт двумерный массив:

```
JS
var a = new Array(4);
for (i = 0; i < 4; i++) {
```

```
  a[i] = new Array(4);
  for (j = 0; j < 4; j++) {
    a[i][j] = "[" + i + "," + j + "]";
  }
}
```

В этом примере создаётся массив со следующим содержимым:

```
Ряд 0: [0,0] [0,1] [0,2] [0,3]
Ряд 1: [1,0] [1,1] [1,2] [1,3]
Ряд 2: [2,0] [2,1] [2,2] [2,3]
Ряд 3: [3,0] [3,1] [3,2] [3,3]
```

Массивы и регулярные выражения

Когда массив является результатом вычислений регулярного выражения над строкой, он содержит свойства и элементы с информацией о совпадениях. Массив возвращается функциями `RegExp.exec()`, `String.match()` и `String.split()`. Подробнее о том, как использовать массивы с регулярными выражениями смотрите в Regular Expressions.

Работа с массивоподобными объектами

Некоторые объекты в JavaScript, такие как `NodeList`, возвращаемые методом `document.getElementsByTagName()`, или специальный объект `arguments`, доступный внутри функции, выглядят и ведут себя как обычные массивы, однако не имеет всех присущих массиву методов. Так например, объект `arguments` имеет свойство `length`, но не имеет метода `forEach()`.

Методы из прототипа Array, могут быть вызваны для массивоподобных объектов. Например:

```
function printArguments() {
  Array.prototype.forEach.call(arguments, function(item) {
    console.log(item);
  });
}
```

Также методы из прототипа Array могут быть применены и к строкам, потому как строки предоставляют доступ к своим символам сходным образом:

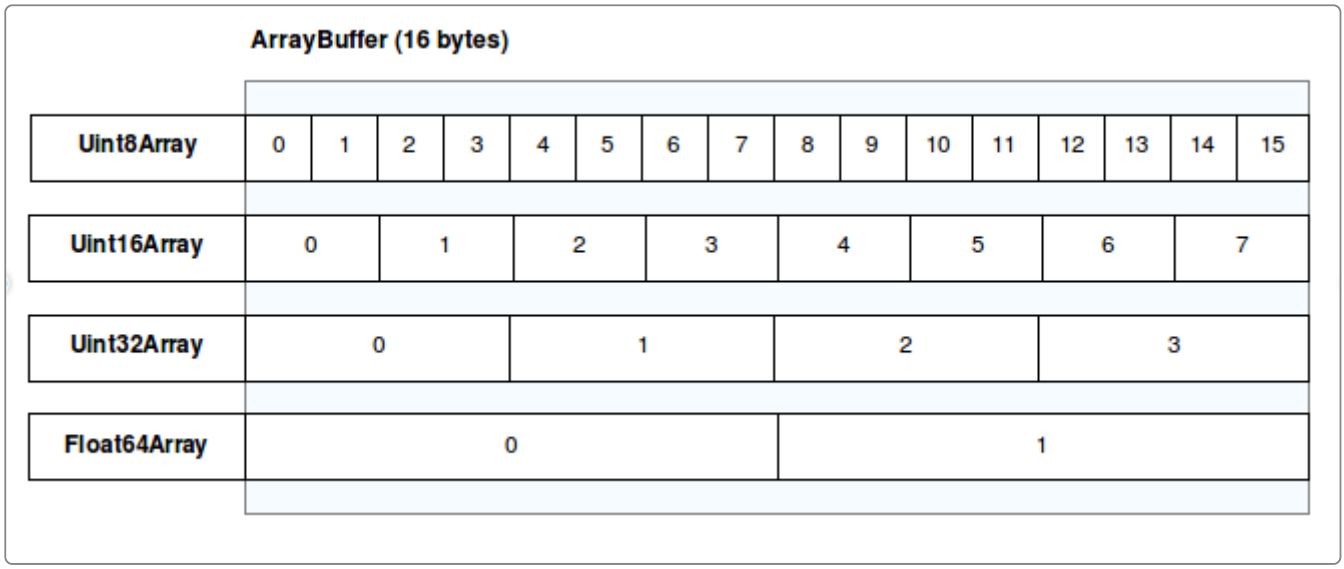
```
JS
Array.prototype.forEach.call("a string", function (chr) {
  console.log(chr);
});
```

Типизированные массивы

JavaScript typed arrays (типизированные массивы) являются массивоподобными объектами, которые предоставляют механизм доступа к сырым бинарным данным. Как вы уже знаете, `Array` массивы динамически растут, сокращаются и могут содержать значения любых типов JavaScript. Движки JavaScript производят оптимизации, благодаря чему, эти операции происходят быстро. Однако, веб приложения становятся все более мощными, добавляются возможности манипуляции со звуковыми и видеоданными, доступ к сырым данным `WebSockets`, и тому подобное. Становится очевидным, что возможность быстрой и эффективной работы с двоичными данными в JavaScript будет очень полезной. Для чего типизированные массивы и предназначены.

Буферы и представления: архитектура типизированных массивов

Для достижения максимальной гибкости и производительности, реализация типизированных массивов в JavaScript разделена на буферы и представления. Буфер (`ArrayBuffer`) это объект, представляющий из себя блок данных; он не имеет формата и не предоставляет возможности доступа к своему содержимому. Для доступа к памяти буфера вам нужно использовать представление. Представление являет собой контекст, имеющий тип данных, начальную позицию в буфере, и количество элементов — это позволяет представить данные в виде актуального типизированного массива.



ArrayBuffer

Объект `ArrayBuffer` это стандартный набор бинарных данных с фиксированной длиной. Вы не можете манипулировать содержимым `ArrayBuffer` напрямую. Вместо этого необходимо создать типизированное представление `DataView`, которое будет отображать буфер в определённом формате, и даст доступ на запись и чтение его содержимого.

Типизированные представления

Название типизированного представления массива говорит само за себя. Оно представляет массив в распространённых числовых форматах, таких как `Int8`, `Uint32`, `Float64` и так далее. Среди прочих существует специальное представление `Uint8ClampedArray`. Оно ограничивает значения интервалом от 0 до 255. Это полезно, например, при Обработке данных изображения в Canvas.

Type	Value Range	Size in bytes	Web IDL type
Int8Array	-128 to 127	1	byte
Uint8Array	0 to 255	1	octet
Uint8ClampedArray	0 to 255	1	octet
Int16Array	-32768 to 32767	2	short

Type	Value Range	Size in bytes	Web IDL type
Uint16Array	0 to 65535	2	unsigned short
Int32Array	-2147483648 to 2147483647	4	long
Uint32Array	0 to 4294967295	4	unsigned long
Float32Array	-3.4e38 to 3.4e38	4	unrestricted float
Float64Array	-1.8e308 to 1.8e308	8	unrestricted double
BigInt64Array	-2^{63} to $2^{63} - 1$	8	bigint
BigUint64Array	0 to $2^{64} - 1$	8	bigint

Для получения подробных сведений смотрите Типизированные массивы JavaScript и справочную документацию для TypedArray .