MDN

> Эта страница была переведена с английского языка силами сообщества. Вы тоже можете внести свой вклад, присоединившись к русскоязычному сообществу MDN Web Docs.

Функции

Функции в JavaScript

Функции - ключевая концепция в JavaScript. Важнейшей особенностью языка является поддержка функции первого класса *(functions as first-class citizen)*. Любая функция это объект, и следовательно ею можно манипулировать как объектом, в частности:

- передавать как аргумент и возвращать в качестве результата при вызове других функций функций высшего порядка;
- создавать анонимно и присваивать в качестве значений переменных или свойств объектов.

Это определяет высокую выразительную мощность JavaScript и позволяет относить его к числу языков, реализующих функциональную парадигму программирования .

Функция в JavaScript специальный тип объектов, позволяющий формализовать средствами языка определённую логику поведения и обработки данных.

Для понимания работы функций необходимо (и достаточно?) иметь представление о следующих моментах:

- способы объявления
- способы вызова

- параметры и аргументы вызова (`arguments`)
- область данных (`Scope`) и замыкания (`Closures`)
- объект привязки (`this`)
- возвращаемое значение (`return`)
- исключения (`throw`)
- использование в качестве конструктора объектов
- сборщик мусора (`garbage collector`)

Объявление функций

Функции вида "function declaration statement"

Объявление функции (*function definition*, или *function declaration*, или *function statement*) состоит из ключевого слова `function` и следующих частей:

- Имя функции.
- Список параметров (принимаемых функцией) заключённых в круглые скобки `()` и разделённых запятыми.
- Инструкции, которые будут выполнены после вызова функции, заключают в фигурные скобки `{ }`.

Например, следующий код объявляет простую функцию с именем `square`:

JS

```
function square(number) {
  return number * number;
}
```

Функция `square` принимает один параметр, названный `number`. Состоит из одной инструкции, которая означает вернуть параметр этой функции (это `number`) умноженный на самого себя. Инструкция `return` указывает на значение, которые будет возвращено функцией.

JS

```
  return number * number;
```

Примитивные параметры (например, число) передаются функции значением; значение передаётся в функцию, но если функция меняет значение параметра, это изменение не отразится глобально или после вызова функции.

Если вы передадите объект как параметр (не примитив, например, массив или определяемые пользователем объекты), и функция изменит свойство переданного в неё объекта, это изменение будет видно и вне функции, как показано в следующем примере:

JS

```
function myFunc(theObject) {
  theObject.make = "Toyota";
}

var mycar = { make: "Honda", model: "Accord", year: 1998 };
var x, y;

x = mycar.make; // x получает значение "Honda"

myFunc(mycar);
y = mycar.make; // y получает значение "Toyota"
// (свойство было изменено функцией)
```

Функции вида "function definition expression"

Функция вида "function declaration statement" по синтаксису является инструкцией (*statement*), ещё функция может быть вида "function definition expression". Такая функция может быть анонимной (она не имеет имени). Например, функция square может быть вызвана так:

JS

```
var square = function (number) {
  return number * number;
};
var x = square(4); // x получает значение 16
```

Однако, имя может быть и присвоено для вызова самой себя внутри самой функции и для отладчика (*debugger*) для идентифицированные функции в стек-треках (*stack traces*; "trace" — "след" / "отпечаток").

JS

```
var factorial = function fac(n) {
  return n < 2 ? 1 : n * fac(n - 1);
};

console.log(factorial(3));
```

Функции вида "function definition expression" удобны, когда функция передаётся аргументом другой функции. Следующий пример показывает функцию `map`, которая должна получить функцию первым аргументом и массив вторым.

JS

```
function map(f, a) {
  var result = [], // Создаём новый массив
    i;
  for (i = 0; i != a.length; i++) result[i] = f(a[i]);
  return result;
}
```

В следующем коде наша функция принимает функцию, которая является function definition expression, и выполняет его для каждого элемента принятого массива вторым аргументом.

JS

```
function map(f, a) {
  var result = []; // Создаём новый массив
  var i; // Объявляем переменную
  for (i = 0; i != a.length; i++) result[i] = f(a[i]);
  return result;
}
var f = function (x) {
  return x * x * x;
};
var numbers = [0, 1, 2, 5, 10];
```

```
var cube = map(f, numbers);
console.log(cube);
```

Функция возвращает: [0, 1, 8, 125, 1000].

В JavaScript функция может быть объявлена с условием. Например, следующая функция будет присвоена переменной `myFunc` только, если `num` равно 0:

JS

```
var myFunc;
if (num === 0) {
  myFunc = function (theObject) {
    theObject.make = "Toyota";
  };
}
```

В дополнение к объявлениям функций, описанных здесь, вы также можете использовать конструктор `Function` для создания функций из строки во время выполнения (*runtime*), подобно `eval()`.

Метод — это функция, которая является свойством объекта. Узнать больше про объекты и методы можно по ссылке: Работа с объектами.

Вызовы функций

Объявление функции не выполняет её. Объявление функции просто называет функцию и указывает, что делать при вызове функции.

Вызов функции фактически выполняет указанные действия с указанными параметрами. Например, если вы определите функцию `square`, вы можете вызвать её следующим образом:

JS

```
square(5);
```

Эта инструкция вызывает функцию с аргументом 5. Функция вызывает свои инструкции и возвращает значение 25.

Функции могут быть в области видимости, когда они уже определены, но функции вида "function declaration statement" могут быть подняты (*поднятие* — *hoisting*), также как в этом примере:

JS

```
console.log(square(5));
/* ... */
function square(n) {
  return n * n;
}
```

Область видимости функции — функция, в котором она определена, или целая программа, если она объявлена по уровню выше.

> Примечание: Это работает только тогда, когда объявлении функции использует вышеупомянутый синтаксис (т.е. `function funcName(){}`). Код ниже не будет работать. Имеется в виду то, что поднятие функции работает только с function declaration и не работает с function expression.

JS

```
console.log(square); // square поднят со значением undefined.
console.log(square(5)); // TypeError: square is not a function
var square = function (n) {
  return n * n;
};
```

Аргументы функции не ограничиваются строками и числами. Вы можете передавать целые объекты в функцию. Функция `show_props()` (объявленная в Работа с объектами) является примером функции, принимающей объекты аргументом.

Функция может вызвать саму себя. Например, вот функция рекурсивного вычисления факториала:

JS

```
function factorial(n) {
  if (n === 0 || n === 1) return 1;
```

```
  else return n * factorial(n - 1);
}
```

Затем вы можете вычислить факториалы от одного до пяти следующим образом:

JS

```
var a, b, c, d, e;
a = factorial(1); // a получает значение 1
b = factorial(2); // b получает значение 2
c = factorial(3); // c получает значение 6
d = factorial(4); // d получает значение 24
e = factorial(5); // e получает значение 120
```

Есть другие способы вызвать функцию. Существуют частые случаи, когда функции необходимо вызывать динамически, или поменять номера аргументов функции, или необходимо вызвать функцию с привязкой к определённому контексту. Оказывается, что функции сами по себе являются объектами, и эти объекты в свою очередь имеют методы (посмотрите объект `Function`). Один из них это метод `apply()`, использование которого может достигнуть этой цели.

Область видимости функций

(function scope)

Переменные объявленные в функции не могут быть доступными где-нибудь вне этой функции, поэтому переменные (которые нужны именно для функции) объявляют только в scope функции. При этом функция имеет доступ ко всем переменным и функциям, объявленным внутри её scope. Другими словами функция объявленная в глобальном scope имеет доступ ко всем переменным в глобальном scope. Функция объявленная внутри другой функции ещё имеет доступ и ко всем переменным её родительской функции и другим переменным, к которым эта родительская функция имеет доступ.

JS

```
// Следующие переменные объявленны в глобальном scope
var num1 = 20,
  num2 = 3,
  name = "Chamahk";
```

```
// Эта функция объявленна в глобальном scope
function multiply() {
  return num1 * num2;
}

multiply(); // вернёт 60

// Пример вложенной функции
function getScore() {
  var num1 = 2,
    num2 = 3;

  function add() {
    return name + " scored " + (num1 + num2);
  }

  return add();
}

getScore(); // вернёт "Chamahk scored 5"
```

Scope и стек функции

(function stack)

Рекурсия

Функция может вызывать саму себя. Три способа такого вызова:

1. по имени функции
2. `arguments.callee`
3. по переменной, которая ссылается на функцию

Для примера рассмотрим следующие функцию:

JS

```
var foo = function bar() {
  // здесь будут выражения
};
```

Внутри функции (*function body*) все следующие вызовы эквивалентны:

1. `bar()`
2. `arguments.callee()`
3. `foo()`

Функция, которая вызывает саму себя, называется *рекурсивной функцией* (*recursive function*). Получается, что рекурсия аналогична циклу (*loop*). Оба вызывают некоторый код несколько раз, и оба требуют условия (чтобы избежать бесконечного цикла, вернее бесконечной рекурсии). Например, следующий цикл:

JS

```
var x = 0;
while (x < 10) {
  // "x < 10" — это условие для цикла
  // что-то делаем
  x++;
}
```

можно было изменить на рекурсивную функцию и вызовом этой функции:

JS

```
function loop(x) {
  if (x >= 10) {
    // "x >= 10" — это условие для конца выполнения (тоже самое, что "!(x < 10)")
    return;
  }
  // делать что-то
  loop(x + 1); // рекурсионный вызов
}
loop(0);
```

Однако некоторые алгоритмы не могут быть простыми повторяющимися циклами. Например, получение всех элементов структуры дерева (например, DOM (en-US)) проще всего реализуется использованием рекурсии:

JS

```
function walkTree(node) {
  if (node == null) return;
  // что-то делаем с элементами
  for (var i = 0; i < node.childNodes.length; i++) {
    walkTree(node.childNodes[i]);
  }
}
```

В сравнении с функцией `loop`, каждый рекурсивный вызов сам вызывает много рекурсивных вызовов.

Также возможно превращение некоторых рекурсивных алгоритмов в нерекурсивные, но часто их логика очень сложна, и для этого потребуется использование стека (*stack*). По факту рекурсия использует stack: function stack.

Поведение стека можно увидеть в следующем примере:

JS

```
function foo(i) {
  if (i < 0) return;
  console.log("begin: " + i);
  foo(i - 1);
  console.log("end: " + i);
}
foo(3);

// Output:

// begin: 3
// begin: 2
// begin: 1
// begin: 0
// end: 0
// end: 1
// end: 2
// end: 3
```

Вложенные функции (nested functions) и замыкания (closures)

Вы можете вложить одну функцию в другую. Вложенная функция (*nested function*; *inner*) приватная (*private*) и она помещена в другую функцию (*outer*). Так образуется *замыкание* (*closure*). Closure — это выражение (обычно функция), которое может иметь свободные переменные вместе со средой, которая связывает эти переменные (что "закрывает" (*"close"*) выражение).

Поскольку вложенная функция это closure, это означает, что вложенная функция может "унаследовать" (*inherit*) аргументы и переменные функции, в которую та вложена. Другими словами, вложенная функция содержит scope внешней (*"outer"*) функции.

Подведём итог:

- Вложенная функция имеет доступ ко всем инструкциям внешней функции.
- Вложенная функция формирует closure: она может использовать аргументы и переменные внешней функции, в то время как внешняя функция не может использовать аргументы и переменные вложенной функции.

Следующий пример показывает вложенную функцию:

JS

```
function addSquares(a, b) {
  function square(x) {
    return x * x;
  }
  return square(a) + square(b);
}
a = addSquares(2, 3); // возвращает 13
b = addSquares(3, 4); // возвращает 25
c = addSquares(4, 5); // возвращает 41
```

Поскольку вложенная функция формирует closure, вы можете вызвать внешнюю функцию и указать аргументы для обоих функций (для outer и innner).

JS

```
function outside(x) {
  function inside(y) {
    return x + y;
  }
  return inside;
}
fn_inside = outside(3); // Думайте об этом как: дайте мне функцию,
// которая добавляет 3 к любому введенному значению

result = fn_inside(5); // возвращает 8

result1 = outside(3)(5); // возвращает 8
```

Сохранение переменных

Обратите внимание, значение `x` сохранилось, когда возвращалось `inside`. Closure должно сохранять аргументы и переменные во всем scope. Поскольку каждый вызов предоставляет потенциально разные аргументы, создаётся новый closure для каждого вызова во вне. Память может быть очищена только тогда, когда `inside` уже возвратился и больше не доступен.

Это не отличается от хранения ссылок в других объектах, но часто менее очевидно, потому что не устанавливаются ссылки напрямую и нельзя посмотреть там.

Несколько уровней вложенности функций (Multiply-nested functions)

Функции можно вкладывать несколько раз, т.е. функция (A) хранит в себе функцию (B), которая хранит в себе функцию (C). Обе функции B и C формируют closures, так B имеет доступ к переменным и аргументам A, и C имеет такой же доступ к B. В добавок, поскольку C имеет такой доступ к B, который имеет такой же доступ к A, C ещё имеет такой же доступ к A. Таким образом closures может хранить в себе несколько scope; они рекурсивно хранят scope функций, содержащих его. Это называется *chaining* (*chain – цепь*; Почему названо "chaining" будет объяснено позже)

Рассмотрим следующий пример:

JS

```
function A(x) {
  function B(y) {
    function C(z) {
      console.log(x + y + z);
    }
    C(3);
  }
  B(2);
}
A(1); // в консоле выводится 6 (1 + 2 + 3)
```

В этом примере C имеет доступ к `y` функции `B` и к `x` функции `A`. Так получается, потому что:

1. Функция `B` формирует closure, включающее `A`, т.е. `B` имеет доступ к аргументам и переменным функции `A`.
2. Функция `C` формирует closure, включающее `B`.
3. Раз closure функции `B` включает `A`, то closure `C` тоже включает A, `C` имеет доступ к аргументам и переменным обоих функций `B` *и* `A`. Другими словами, `C` связывает *цепью* (*chain*) scopes функций `B` и `A` в таком порядке.

В обратном порядке, однако, это не верно. `A` не имеет доступ к переменным и аргументам `C`, потому что `A` не имеет такой доступ к `B`. Таким образом, `C` остаётся приватным только для `B`.

Конфликты имён (Name conflicts)

Когда два аргумента или переменных в scope у closure имеют одинаковые имена, происходит *конфликт имени* (*name conflict*). Более вложенный (*more inner*) scope имеет приоритет, так самый вложенный scope имеет наивысший приоритет, и наоборот. Это цепочка областей видимости (*scope chain*). Самым первым звеном является самый глубокий scope, и наоборот. Рассмотрим следующие:

JS

```
function outside() {
  var x = 5;
  function inside(x) {
    return x * 2;
```

```
  }
  return inside;
}

outside()(10); // возвращает 20 вместо 10
```

Конфликт имени произошёл в инструкции `return x * 2` между параметром `x` функции `inside` и переменной `x` функции `outside`. Scope chain здесь будет таким: {`inside ==> outside ==>` глобальный объект (*`global object`*)}. Следовательно `x` функции `inside` имеет больший приоритет по сравнению с `outside`, и нам вернулось 20 (= 10 * 2), а не 10 (= 5 * 2).

Замыкания

(Closures)

Closures это один из главных особенностей JavaScript. JavaScript разрешает вложенность функций и предоставляет вложенной функции полный доступ ко всем переменным и функциям, объявленным внутри внешней функции (и другим переменным и функции, к которым имеет доступ эта внешняя функция).

Однако, внешняя функция не имеет доступа к переменным и функциям, объявленным во внутренней функции. Это обеспечивает своего рода инкапсуляцию для переменных внутри вложенной функции.

Также, поскольку вложенная функция имеет доступ к scope внешней функции, переменные и функции, объявленные во внешней функции, будет продолжать существовать и после её выполнения для вложенной функции, если на них и на неё сохранился доступ (имеется ввиду, что переменные, объявленные во внешней функции, сохраняются, только если внутренняя функция обращается к ним).

Closure создаётся, когда вложенная функция как-то стала доступной в неком scope вне внешней функции.

JS

```
var pet = function (name) {
  // Внешняя функция объявила переменную "name"
```

```
  var getName = function () {
    return name; // Вложенная функция имеет доступ к "name" внешней функции
  };
  return getName; // Возвращаем вложенную функцию, тем самым сохраняя доступ
  // к ней для другого scope
};
myPet = pet("Vivie");

myPet(); // Возвращается "Vivie",
// т.к. даже после выполнения внешней функции
// name сохранился для вложенной функции
```

Более сложный пример представлен ниже. Объект с методами для манипуляции вложенной функции внешней функцией можно вернуть (*return*).

JS

```
var createPet = function (name) {
  var sex;

  return {
    setName: function (newName) {
      name = newName;
    },

    getName: function () {
      return name;
    },

    getSex: function () {
      return sex;
    },

    setSex: function (newSex) {
      if (
        typeof newSex === "string" &&
        (newSex.toLowerCase() === "male" || newSex.toLowerCase() === "female")
      ) {
        sex = newSex;
      }
    },
  };
};
```

```
var pet = createPet("Vivie");
pet.getName(); // Vivie

pet.setName("Oliver");
pet.setSex("male");
pet.getSex(); // male
pet.getName(); // Oliver
```

В коде выше переменная `name` внешней функции доступна для вложенной функции, и нет другого способа доступа к вложенным переменным кроме как через вложенную функцию. Вложенные переменные вложенной функции являются безопасными хранилищами для внешних аргументов и переменных. Они содержат "постоянные" и "инкапсулированные" данные для работы с ними вложенными функциями. Функции даже не должны присваиваться переменной или иметь имя.

JS

```
var getCode = (function () {
  var apiCode = "0]Eal(eh&2"; // Мы не хотим, чтобы данный код мог быть изменен кем-то извне...

  return function () {
    return apiCode;
  };
})();

getCode(); // Возвращает apiCode
```

Однако есть ряд подводных камней, которые следует учитывать при использовании замыканий. Если закрытая функция определяет переменную с тем же именем, что и имя переменной во внешней области, нет способа снова ссылаться на переменную во внешней области.

JS

```
var createPet = function (name) {
  // Внешняя функция определяет переменную с именем "name".
  return {
    setName: function (name) {
      // Внутренняя функция также определяет переменную с именем "name".
      name = name; // Как мы можем получить доступ к "name", определённой во внешней функции?
    },
```

```
  };
};
```

Использование объекта arguments

Объект arguments функции является псевдо-массивом. Внутри функции вы можете ссылаться к аргументам следующим образом:

JS

```
arguments[i];
```

где `i` — это порядковый номер аргумента, отсчитывающийся с 0. К первому аргументу, переданному функции, обращаются так `arguments[0]`. А получить количество всех аргументов — `arguments.length`.

С помощью объекта `arguments` Вы можете вызвать функцию, передавая в неё больше аргументов, чем формально объявили принять. Это очень полезно, если вы не знаете точно, сколько аргументов должна принять ваша функция. Вы можете использовать `arguments.length` для определения количества аргументов, переданных функции, а затем получить доступ к каждому аргументу, используя объект `arguments`.

Для примера рассмотрим функцию, которая конкатенирует несколько строк. Единственным формальным аргументом для функции будет строка, которая указывает символы, которые разделяют элементы для конкатенации. Функция определяется следующим образом:

JS

```
function myConcat(separator) {
  var result = "";
  var i;

  // iterate through arguments
  for (i = 1; i < arguments.length; i++) {
    result += arguments[i] + separator;
  }
  return result;
}
```

Вы можете передавать любое количество аргументов в эту функцию, и он конкатенирует каждый аргумент в одну строку.

JS

```
// возвращает "red, orange, blue, "
myConcat(", ", "red", "orange", "blue");

// возвращает "elephant; giraffe; lion; cheetah; "
myConcat("; ", "elephant", "giraffe", "lion", "cheetah");

// возвращает "sage. basil. oregano. pepper. parsley. "
myConcat(". ", "sage", "basil", "oregano", "pepper", "parsley");
```

> Примечание: `arguments` является псевдо-массивом, но не массивом. Это псевдо-массив, в котором есть пронумерованные индексы и свойство `length`. Однако он не обладает всеми методами массивов.

Рассмотрите объект `Function` в JavaScript-справочнике для большей информации.

Параметры функции

Начиная с ECMAScript 2015 появились два новых вида параметров: параметры по умолчанию (*default parameters*) и остаточные параметры (*rest parameters*).

Параметры по умолчанию (Default parameters)

В JavaScript параметры функции по умолчанию имеют значение `undefined`. Однако в некоторых ситуация может быть полезным поменять значение по умолчанию. В таких случаях default parameters могут быть весьма кстати.

В прошлом для этого было необходимо в теле функции проверять значения параметров на `undefined` и в положительном случае менять это значение на дефолтное (*default*). В следующем примере в случае, если при вызове не предоставили значение для `b`, то этим значением станет `undefined`, тогда результатом вычисления `a * b` в функции `multiply` будет `NaN`. Однако во второй строке мы поймаем это значение:

JS

```
function multiply(a, b) {
  b = typeof b !== "undefined" ? b : 1;

  return a * b;
}

multiply(5); // 5
```

С параметрами по умолчанию проверка наличия значения параметра в теле функции не нужна. Теперь вы можете просто указать значение по умолчанию для параметра `b` в объявлении функции:

JS

```
function multiply(a, b = 1) {
  return a * b;
}

multiply(5); // 5
```

Для более детального рассмотрения ознакомьтесь с параметрами по умолчанию.

Остаточные параметры (Rest parameters)

Остаточные параметры предоставляют нам массив неопределённых аргументов. В примере мы используем остаточные параметры, чтобы собрать аргументы с индексами со 2-го до последнего. Затем мы умножим каждый из них на значение первого аргумента. В этом примере используется стрелочная функция (*Arrow functions*), о которой будет рассказано в следующей секции.

JS

```
function multiply(multiplier, ...theArgs) {
  return theArgs.map((x) => multiplier * x);
}

var arr = multiply(2, 1, 2, 3);
console.log(arr); // [2, 4, 6]
```

Стрелочные функции

(Arrow functions)

Стрелочные функции — функции вида "arrow function expression" (неверно `fat arrow function`) — имеют укороченный синтаксис по сравнению с function expression и лексически связывает значение `this`. Стрелочные функции всегда анонимны. Посмотрите также пост блога hacks.mozilla.org "ES6 In Depth: Arrow functions ".

На введение стрелочных функций повлияли два фактора: более короткие функции и лексика `this`.

Более короткие функции

В некоторых функциональных паттернах приветствуется использование более коротких функций. Сравните:

JS

```
var a = ["Hydrogen", "Helium", "Lithium", "Beryllium"];

var a2 = a.map(function (s) {
  return s.length;
});

console.log(a2); // выводит [8, 6, 7, 9]

var a3 = a.map((s) => s.length);

console.log(a3); // выводит [8, 6, 7, 9]
```

Лексика `this`

До стрелочных функций каждая новая функция определяла своё значение `this` (новый объект в случае конструктора, undefined в strict mode, контекстный объект, если функция вызвана как метод объекта, и т.д.). Это оказалось раздражающим с точки зрения объектно-ориентированного стиля программирования.

JS

```
function Person() {
  // Конструктор Person() определяет `this` как самого себя.
```

```
  this.age = 0;

  setInterval(function growUp() {
    // Без strict mode функция growUp() определяет `this`
    // как global object, который отличается от `this`
    // определённого конструктором Person().
    this.age++;
  }, 1000);
}

var p = new Person();
```

В ECMAScript 3/5 эта проблема была исправлена путём присвоения значения `this` переменной, которую можно было бы замкнуть.

JS

```
function Person() {
  var self = this; // Некоторые выбирают `that` вместо `self`.
  // Выберите что-то одно и будьте последовательны.
  self.age = 0;

  setInterval(function growUp() {
    // Колбэк ссылается на переменную `self`,
    // значением которой является ожидаемый объект.
    self.age++;
  }, 1000);
}
```

Альтернативой может быть связанная функция (*bound function*), с которой можно правильно вручную определить значение `this` для функции `growUp()`.

В arrow function значением `this` является окружающий его контекст, так следующий код работает ожидаемо:

JS

```
function Person() {
  this.age = 0;

  setInterval(() => {
    this.age++; // |this| должным образом ссылается на объект Person
```

```
    }, 1000);
  }

  var p = new Person();
```

Далее

Подробное техническое описание функций в статье справочника Функции

Смотрите также Function (en-US) в Справочнике JavaScript для получения дополнительной информации по функции как объекту.

Внешние ресурсы:

- ECMAScript® 2015 Language Specification
- Учебник по Javascript - замыкания