JavaScript Functions: scoping, module, cascade, closure, currying, .apply(), .call(), .bind()

Scoping

Область видимости (scope) переменной – это та часть программы, для которой эта переменная определена

В языке JavaScript имеется две области действия: *глобальная* и *локальная*.

Переменная, объявляемая вне функции, является *глобальной*, и ее значение доступно и может быть изменено в любом месте программы.

Переменная, объявляемая внутри определения функции, cоздается и уничтожается при каждом выполнении функции и не доступна никакому коду за пределами данной функции.

JavaScript *не поддерживает область видимости блока* (в которой пара фигурных скобок{. . .} определяет новую область

Локальная переменная может иметь такое же имя, как глобальная, но это совершенно разные переменные; изменение значения одной переменной не влияет на значение другой. Внутри функции, где объявлена локальная переменная, имеет значение только эта локальная версия.

В область видимости функции входит глобальный объект и переменные объемлющей функции, если она есть.

При вызове функции определяется ее область видимости: объект видимости ставится в начало цепочки видимости, затем в этот объект добавляется свойство arguments, затем параметры, а потом локальные переменнные

Лексическая область видимости - лексическая значит описательная т.е. область видимости на момент описания функции, а не на момент вызова.

В JavaScript переменные вычисляются так, как если бы они были объявлены в начале той области, в которой находятся. (Hoisting)

var aNumber = 100;

tweak();

function tweak(){

document.write(aNumber); // This prints "undefined", because aNumber is also defined locally below.

if (false)

{

var aNumber = 123;

}

}

Когда программа JavaScript выполняет какую-либо функцию, сначала она *находит все объявления переменных,* например, var someVariable. Она создает переменные с начальным значением *undefined*, а значение получает лишь *на этапе исполнения строки.*

Одно из первых дейсвий, твыполняемых интерпретатором JavaScript при запуске перед исполнением любого кода, – это создание *глобального объекта*, глобальные переменные – свойства глобального объекта.

В коде верхнего уровня (т. е. JavaScript коде, который не является частью функции) сослаться на глобальный объект можно посредством ключевого слова *this*

Этот глобальный объект имеет свойство *window*, ссылающееся на сам объект, которое можно использовать вместо ключевого слова this для ссылки на глобальный объект.

Объект window определяет базовые глобальные свойства, такие как parseInt и Math, а также глобальные клиентские свойства, такие как navigator и screen.

Локальные переменные - свойства объекта вызова (call object). Когда выполняется тело функции, аргументы и локальные переменные функции хранятся как свойства этого объекта.

Использование абсолютно отдельного объекта для локальных переменных позволяет JavaScript не допускать переписывания локальными переменными значений глобальных переменных с теми же именами

Объект вызова находится в начале цепочки областей видимости,

Module

*Преимущества* организации программ в модули: структуризация помогает незнакомым с кодом людям найти то, что им нужно, и помогает программистам хранить связанные друг с другом вещи в одном месте.

Ещё одна задача модулей – *изолировать* несвязанные между собой части кода так, как это делают интерфейсы объектов.

Основное назначение модуля — объявить функции так, чтобы ненужные подробности их реализаций были скрыты. В том числе: временные переменные, константы, вспомогательные мини-функции и т.п.

Оформление кода в модуль предусматривает следующие шаги:

1. Создаётся функция-обёртка, которая выполняется «на месте»:

|  |
| --- |
| (function() { |

|  |
| --- |
| ... |

|  |
| --- |
| })(); |

1. Внутрь этой функции пишутся локальные переменные и функции, которые пользователю модуля не нужны, но нужны самому модулю:

|  |  |
| --- | --- |
| 1 | (function() { |

|  |  |
| --- | --- |
| 2 | var count = 0; |

|  |  |
| --- | --- |
| 3 |  |

|  |  |
| --- | --- |
| 4 | function helper() { ... } |

|  |  |
| --- | --- |
| 5 | })(); |

Они будут доступны только изнутри.

3. Те функции, которые нужны пользователю, «экспортируются» во внешнюю область видимости.

Если функция одна — это можно сделать явным возвратом return:

|  |  |
| --- | --- |
| 1 | var func = (function() { |

|  |  |
| --- | --- |
| 2 | var count = 0; |

|  |  |
| --- | --- |
| 3 | function helper() { ... } |

|  |  |
| --- | --- |
| 5 | return function() { ... } |

|  |  |
| --- | --- |
| 6 | })(); |

…Если функций много — можно вернуть объект с ними:

|  |  |
| --- | --- |
| 01 | var MyLibrary = (function() {  return { |

|  |  |
| --- | --- |
| 06 | createMenu: function() { ... }, |

|  |  |
| --- | --- |
| 07 | createDialog: function() { ... } |

|  |  |
| --- | --- |
| 08 | }; |

|  |  |
| --- | --- |
| 10 | })(); |

|  |  |
| --- | --- |
| 12 | // использование |

|  |  |
| --- | --- |
| 13 | MyLibrary.createMenu(); |

Все функции модуля будут через замыкание иметь доступ к другим переменным и внутренним функциям. Но снаружи программист, использующий модуль, может обращаться напрямую только к тем, которые экспортированы.

Благодаря этому будут скрыты внутренние аспекты реализации, которые нужны только разработчику модуля.

(The general pattern of a module is a function that defines private variables and functions; creates privileged functions which, through closure, will have access to the private variables and functions; and that returns the privileged functions or stores them in an accessible place.)

Использование модуля может исключить использование глобальных переменных. Это способствует сокрытию, является очень эффективным средством инкапсуляции.

Сascade

Некоторые методы не возвращают никакого значения. Например, это характерно для методов, которые устанавливают или изменяют состояние объекта и при этом не возвращают ничего (лишь this). При работе с такими методами мы можем использовать cascade, т.е вызывать много методов объекта один за другим. Например:

str.replace("k", "R").toUpperCase().substr(0,4);

Зачем использовать каскад?

- не нужно создавать временные переменные;

var aUser= userController.findUser("test@test.com").

. formatName()

. createLayout()

. displayUser(layoutHTML);

Вместо

|  |
| --- |
| var aUser = userController.findUser("test@test.com"); |
| var formatUserName = aUser.formatName(); |
| var layoutHTML = formatUserName.createLayout(); |
| userController.displayUser(layoutHTML); |

- каждая строчка кода четко и лаконично описывает то, что она делает, особенно, когда имя каждого метода определяется с помощью глаголов;

- код становится более сопровождаемым, поскольку есть простые, специализированные методы;

- легче для чтения и понимания.

|  |  |
| --- | --- |
|  |  |

Closure

Замыкание позволяет связать какие-то данные (конкретное окружение) с функцией, которая работает с этими данными. Очевидна параллель с объектно-ориентированным программированием, где объект позволяет нам связать между собой набор данных (свойства объекта) с одним или несколькими методами.

Если вы объявляете функцию внутри другой функции, первая получает доступ к переменным и аргументам последней:

function outerFn(myArg) {  
    var myVar;  
    function innerFn() {       //имеет доступ к myVar и myArg    }  
}

При этом, такие переменные *продолжают существовать и остаются доступными внутренней функцией* даже после того, как внешняя функция, в которой они определены, была исполнена.

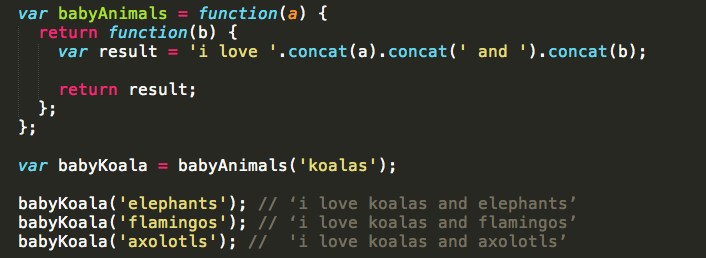
function createCounter() {  
    var numberOfCalls = 0;  
    return function() {  
  return ++numberOfCalls;  
    }  
 }  
var fn = createCounter();  
fn(); //1  
fn(); //2

При замыкании переменные внешней функции сохраняют свое значение между

вызовами. Вложенные функции замыкают на себя переменные и аргументы функции внутри которой они определены. Применение замыканий - функции, создающие функции(мета программирование). При замыкании вложенная функция не изменяется, но изменяется ее область видимости (точнее не ее, а функции-обертки ), так как параметры , с которыми вызывается функция- обертка могут меняться

Currying

Currying относится к процессу преобразования функции с возможностью вызова с разными параметрами к той же функции с меньшим числом параметров. Currying достигается за счет связывания некоторых аргументов во время первого вызова таким образом, что эти значения являются фиксированными для следующего вызова.



babyAnimals является каррирующей функцией, она предназначен для фиксации первого аргумента при первом вызове. При такой реализации параметр a будет равен "koalas" для всех вызовов функции babyKoala

Для реализации сurrying необходимо написать каррирующую функцию, которая принимает в качестве аргументов каррируемую функцию и значения, на которые необходимо заменить первые аргументы каррируемой функции.

В качестве каррируемой функции выступает функцию, складывающую два числа:

function sum(a, b) {

return a + b;

}

После того, как эта функция будет подвержена каррированию с заменой одного аргумента на некоторое значение, можно будет получить функцию, которая, например: инкрементирует (увеличивают на единицу), декрементирует (уменьшают на единицу) или прибавляют к своему уже единственному аргументу заданное заранее значение.

function curry(a) {

return function (b) {

// в этом вызове аргумент a заменён на переданное в функцию curry значение

return sum(a, b);

};

}

var inc = curry(1);

console.log(inc(5)); // 6

var dec = curry(-1);

console.log(dec(3)); // 2

var plusFive = curry(5);

console.log(plusFive(5)); // 10

Call

Метод **call()** вызывает функцию с указанным значением this и индивидуально предоставленными аргументами.

## Синтаксис

fun*.*call(thisArg[, arg1[,arg2[*, ...]]])*

### Параметры

*thisArg*

Значение this, предоставляемое для вызова функции *fun*.

this может не быть реальным значением, видимым этим методом: если метод является функцией в нестрогом режиме, значения [null](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/null" \o "Документация об этом ещё не написана; пожалуйста, поспособствуйте её написанию!) и [undefined](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/undefined" \o "Значение глобального свойства undefined представляет значение undefined. Это одно из примитивных значений JavaScript.) будут заменены глобальным объектом, а примитивные значения будут упакованы в объекты.

arg1, arg2, ...

Аргументы для объекта.

## Описание

C помощью этого метода можно присваивать различные объекты this при вызове существующей функции. This ссылается на текущий объект, вызвавший объект. С помощью call вы можно написать метод один раз, а затем наследовать его в других объектах, без необходимости переписывать метод для каждого нового объекта.

## Apply

Метод apply() вызывает функцию с указанным значением this и аргументами, предоставленными в виде массива (либо массивоподобного объекта).

***Примечание:****хотя синтаксис этой функции практически полностью идентичен функции*[*call()*](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Function/call)*, фундаментальное различие между ними заключается в том, что функция call()принимает список аргументов, в то время, как функция apply() принимает простой массив с аргументами.*

## Синтаксис

fun.apply(thisArg[, argsArray])

### Параметры

*thisArg*

Значение this, предоставляемое для вызова функции *fun*. this может не быть реальным значением, видимым этим методом: если метод является функцией в нестрогом режиме, значения [null](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/null" \o "Документация об этом ещё не написана; пожалуйста, поспособствуйте её написанию!) и [undefined](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/undefined" \o "Значение глобального свойства undefined представляет значение undefined. Это одно из примитивных значений JavaScript.) будут заменены глобальным объектом, а примитивные значения будут упакованы в объекты.

*argsArray*

Массив, определяющий аргументы, с которыми функция *fun* должна быть вызвана, либо [null](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/null" \o "Документация об этом ещё не написана; пожалуйста, поспособствуйте её написанию!) или [undefined](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/undefined" \o "Значение глобального свойства undefined представляет значение undefined. Это одно из примитивных значений JavaScript.), если в функцию не надо передавать аргументы.

Function.prototype.construct = function (aArgs) {

var oNew = Object.create(this.prototype);

this.apply(oNew, aArgs);

return oNew;

};

Bind

Метод **bind()** создаёт новую функцию, которая при вызове устанавливает в качестве контекста выполнения this предоставленное значение. В метод также передаётся набор аргументов, которые будут установлены перед переданными в привязанную функцию аргументами при её вызове.

## Синтаксис

fun.bind(thisArg[, arg1[, arg2[, ...]]])

### Параметры

*thisArg*

Значение, передаваемое в качестве this в целевую функцию при вызове привязаной функции. Значение игнорируется, если привязанная функция конструируется с помощью оператора [new](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/new" \o "Документация об этом ещё не написана; пожалуйста, поспособствуйте её написанию!).

*arg1, arg2, ...*

Аргументы целевой функции, передаваемые перед аргументами привязанной функции при вызове целевой функции.

## Описание

Метод bind() создаёт новую функцию (**привязанную функцию**) с тем же самым телом функции, что и у функции, для которой он вызывается, со значением this, привязанным к первому аргументу bind(), которое не может быть переопределено. Метод bind() также принимает ведущие аргументы по умолчанию, передаваемые в целевую функцию при вызове привязанной функции. Привязанная функция также может быть сконструирована с помощью оператора [new](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/new" \o "Документация об этом ещё не написана; пожалуйста, поспособствуйте её написанию!): это работает так, как если бы вместо неё конструировалась целевая функция. Предоставляемое значение this в этом случае игнорируется, хотя ведущие аргументы всё ещё передаются в эмулируемую функцию.

this.x = 9;

var module = {

x: 81,

getX: function() { return this.x; }

};

module.getX(); // 81

var getX = module.getX;

getX(); // 9, поскольку в этом случае this ссылается на глобальный объект

// создаём новую функцию с this, привязанным к module

var boundGetX = getX.bind(module);

boundGetX(); // 81

Links

<http://toddmotto.com/everything-you-wanted-to-know-about-javascript-scope/>

<http://ryanmorr.com/understanding-scope-and-context-in-javascript/>

<http://habrahabr.ru/post/239863/>

<https://msdn.microsoft.com/ru-ru/library/ie/bzt2dkta(v=vs.94).aspx>

<http://habrahabr.ru/post/243273/>

<http://learn.javascript.ru/closures-usage>

https://medium.com/@kbrainwave/currying-in-javascript-ce6da2d324fe

<http://true-coder.ru/javascript/karrirovanie-karring-v-javascript.html>

https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global\_Objects/Function/call