JSDG 🦏 | Functions | Функции | chapter 8

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

В этой главе рассматриваются функции JavaScript. Функции являются фун даментальными строительными блоками для программ на JavaScript и распространенным средством почти во всех языках программирования. Возможно, вы уже знакомы с концепцией функции, но под другим названием, таким как под программа или процедура.

Функция - это блок кода JavaScript, который определяется однажды, но может выполняться, или вызываться, любое количество раз. Функции JavaScript параметризированы: определение функции может включать список идентификаторов, известных как параметры, которые выступают в качестве локальных переменных для тела функции. При вызове функций для параметров предоставляются значения, или аргументы. Функции часто используют значения своих аргументов для вычисления возвращаемого значения, которое становится значением выражения вызова функции. В дополнение к аргументам каждый вызов имеет еще одно значение -- контекст вызова, который представляет собой значение ключевого слова this.

Если функция присваивается свойству объекта, тогда она называется методом данного объекта. Когда функция вызывается на объекте или через объект, то такой объект будет контекстом вызова или значением this для функции. Функции, предназначенные для инициализации вновь созданных объектов, называются конструкторами.

8.1. Определение функций

Самый простой способ определения функции JavaScript предусматривает применение ключевого слова function, которое можно использовать как обь-

явление или как выражение.

В версии ES6 устанавливается важный новый способ определения функций без ключевого слова function: "стрелочные функции" обладают чрезвычайно компактным синтаксисом и удобны, когда одна функция передается в виде аргумента другой функции. В последующих подразделах раскрываются три способа определения функций. Имейте в виду, что некоторые детали синтаксиса определения функций, касающиеся параметров функций, отложены до раздела 8.3.

В объектных литералах и определениях классов имеется удобный сокращенный синтаксис для определения методов. Он был описан в подразделе 6.10.5 и эквивалентен присваиванию выражения определения функции свойству обьекта с применением базового синтаксиса имя: значение объектных литералов.

В еще одном особом случае в объектных литералах можно использовать ключевые слова get и set для определения специальных методов получения и установки свойств. Такой синтаксис определения функций был раскрыт в подразделе 6.10.6.

Обратите внимание, что функции также можно определять с помощью конструктора Function (), который обсуждается в подразделе 8.7.7. Кроме того, в JavaScript определено несколько специализированных видов функций. Скажем, function* определяет генераторные функции (см. главу 12), a async function - асинхронные функции (см. главу 13).

Объявление функции на самом деле *объявляет* переменную и присваивает ей объект функции.

С выражениями функций рекомендуется использовать const, чтобы не преднамеренно не переписать свои функции, присваивая новые значения.

Существует важное отличие между определением функции f () с помощью объявления функции и присваиванием функции

переменной f после ее создания посредством выражения. Когда применяется форма объявления, объекты функций создаются до того, как содержащий их код начнет выполняться, а определения поднимаются, чтобы функции можно было вызывать в коде, который находится выше оператора определения. Однако это не так в случае функций, определенных как выражения: такие функции не существуют до тех пор, пока не будет фактически вычислено выражение, которое их определяет.

___

Стрелочные функции

В ES6 функции можно определять с использованием чрезвычайно компактного синтаксиса, который называется "стрелочными функциями". Этот синтаксис напоминает математическую запись и применяет "стрелку" => для отделения параметров функции от ее тела. Ключевое слово function не используется и поскольку стрелочные функции являются выражениями, а не операторами, то также нет необходимости в имени функции. Общая форма стрелочной функции выглядит как список разделенных запятыми параметров в круглых скобках, за которым следует стрелка => и затем тело функции в фигурных скобках:

const sum = (x, y) => (return x + y; };

const sum = (x, y) => x + y;

Кроме того, если стрелочная функция имеет в точности один параметр, то круглые скобки вокруг списка параметров можно опустить:

const polynomial = x => x*x + 2*x + 3;

Тем не менее, важно запомнить, что стрелочная функция без параметров должна быть записана с пустой парой круглых скобок:
const constantFunc = () => 42 ;

___

Условный вызов

Версия ES2020 позволяет вставлять ?. после выражения функции и перед открывающей круглой скобкой в вызове функции, чтобы вызывать функцию, только если она не пи11 или undefined. То есть выражение **f?. (×)** эквивалентно (при условии отсутствия побочных эффектов) такому выражению:

(f !== null && f !== undefined) ? f(x) : undefined // **f?. (×)**

Для вызова функций в нестрогом режиме контекстом вызова (значение this) будет глобальный объект. Тем не менее, в строгом режиме контекстом вызова является undefined. Обратите внимание, что функции, определенные с использованием стрелочного синтаксиса, ведут себя по-другому: они всегда наследуют значение this, которое действует там, где они определены.

В функциях, написанных для вызова как функций (не как методов), обычно вообще не применяется ключевое слово this. Однако this можно использовать для определения, действует ли строгий режим:

// Определение и вызов функции для выяснения, находимся ли мы в строгом режиме

const strict = (function () I return !this; }());

====

Рекурсивные функции и стек

Рекурсивная функция - это функция, подобная factorial () в начале главы, которая вызывает саму себя. Некоторые алгоритмы, например, работающие с древовидными структурами данных, могут быть особенно элегантно реализованы с помощью рекурсивных функций. Тем не менее, при написании рекурсивной функции важно учитывать ограничения, связанные с памятью.

Когда функция А вызывает функцию В, после чего функция В

вызывает функцию C, интерпретатор JavaScript должен отслеживать контексты выполнения для всех трех функций. Когда функция C завершается, интерпретатору необходимо знать, где возобновить выполнение функции B, а когда заканчивает работу функция B, то ему нужно знать, где возобновить выполнение функции A. Вы можете представлять себе такие контексты выполнения как стек. Когда одна функция вызывает другую функцию, новый контекст выполнения помещается в стек. После возврата из этой другой функции объект контекста выполнения первой функции извлекается из стека. Если функция вызывает саму себя рекурсивно 100 раз, тогда в стек будут помещены 100 объектов и затем те же 100 объектов извлечены. Такой стек вызовов занимает память. На современном оборудовании обычно нормально писать рекурсивные функции, которые вызывают сами себя сотни раз. Но если функция вызывает саму себя десять тысяч раз, то вполне вероятно, что она потерпит неудачу с выдачей сообщения об ошибке вроде Maximum call-stack size exceeded (Превышен максимальный размер стека вызовов).

====

Метод - это не более чем функция JavaScript, которая хранится в свойстве объекта. Имея функцию { и объект o, определить метод по имени т объекта o
можно следующим образом:

o.m = f:

В ES6 и последующих версиях еще один способ обхода проблемы предусматривает преобразование вложенной функции E в стрелочную функцию, которая надлежащим образом наследует значение this:

```
const f = () => {
      this === o // true, поскольку стрелочные функции наследуют this
```

Другой обходной путь заключается в вызове метода bind () вложенной функции с целью определения новой функции, которая неявно вызывается на указанном объекте:

```
const f = (function () {
      this === o //true, поскольку мы привязали эту функцию к
внешнему this
}.bind (this);
```

___

Вызов конструктора

```
o = new Object ();
```

Вызов конструктора создает новый пустой объект, который унаследован от объекта, заданного свойством prototype конструктора.

Функции конструкторов предназначены для инициализации объектов, и вновь созданный объект применяется в качестве контекста вызова, поэтому функция конструктора может ссылаться на него посредством ключевого слова this.

Косвенный вызов функции

JavaScript имеют методы. Два из них, ca1l () и apply (), вызывают функцию косвенно. Оба метода позволяют явно указывать значение this для вызова, т.е. вы можете вызвать любую функцию как метод любого объекта, даже если она в действительности не является методом этого объекта. Оба метода также позволяют указывать аргументы для вызова. Метод call () использует в качестве аргументов для функции собственный список аргументов, а метод apply () ожидает массива значений, которые должны применяться как аргументы. Методы call () и apply () подробно описаны в подразделе 8.7.4.

___

Неявный вызов функции

Существуют разнообразные языковые средства JavaScript, которые не похожи на вызовы функций, но служат причиной их вызова. Проявляйте особую осторожность, когда пишете функции, которые могут быть вызваны неявно, поскольку

ошибки, побочные эффекты и проблемы с производительностью в таких функциях труднее диагностировать и исправлять, чем в обыкновенных функциях, по той простой причине, что при инспектировании кода факты их вызова могут быть неочевидными.

Ниже перечислены языковые средства, которые способны приводить к неявному вызову функций.

• Если для объекта определены методы получения или установки, тогда запрашивание и установка значения его свойств могут вызывать упомянутые методы. Дополнительные сведения ищите в подразделе 6.10.6.

• Когда объект используется в строковом контексте (скажем, при конкатенации со строкой), вызывается его метод toString (). Аналогично, когда объект применяется в числовом контексте, вызывается его метод valueOf (). Детали ищите в подразделе 3.9.3.

• Когда вы проходите в цикле по элементам итерируемого объекта, может произойти несколько вызовов методов. В главе 12 объясняется, каким образом итераторы работают на уровне вызова функций, и демонстрируется, как писать такие методы, чтобы определять собственные итерируемые типы.

• Литерал тегированного шаблона представляет собой скрытый вызов функции. В разделе 14.5 будет показано, как писать функции, которые могут использоваться в сочетании со строками литералов шаблонов.

• Поведение прокси-объектов (описанных в разделе 14.7) полностью управляется функциями. Практически любое действие с одним из таких объектов станет причиной вызова какой-то функции.

Аргументы и параметры функций

function arraycopy({from, to = from, n= from.length, fromIndex=0, toIndex=0]) {

```
    let valuesToCopy = from.slice (fromIndex, fromIndex + n) ;
    to.splice(toIndex, 0, ...valuesToCopy) ;
    return to;
}
let a = [1,2,3,4,5], b = [9,8,7,6,5];
arraycopy({from: a, n: 3, to: b, toIndex: 4)) // => [9,8, 7, 6, 1, 2,3, 5]


// Эта функция ожидает аргумент типа массива.
// Первые два элемента массива распаковываются в параметры x и y.
// Любые оставшиеся элементы сохраняются в массиве coords.
// Любые аргументы после первого массива упаковываются в массив rest.

function f([x, y, ...coords], ...rest) I
    return [x + y, ...rest, ...coords];m// Примечание: здесь используется операция распространения
}
f((1, 2, 3, 41, 5, 6) //=> [3, 5, 6, 3, 4]
```

В ES2018 параметр остатка можно также использовать при деструктуризации объекта. Значением параметра остатка будет объект с любыми свойствами, которые не были деструктурированы. Объектные параметры остатка часто удобно применять с объектной операцией распространения, что тоже является нововведением ES2018:

```
// Умножить вектор (x, y} или (x, y, z) на скалярное значение,
// предохранив остальные свойства

function vectorMultiply ((x, Y, z=0, ...props), scalar) (
    return { x: x*scalar, y: y*scalar, z: z*scalar, ...props };
}
vectorMultiply({x: 1, y: 2, W: -1}, 2) // => Ix: 2, y: 4, z: 0, W: -1)
```

——

Определение собственных свойств функций

Функции в JavaScript представляют собой не элементарные значения, а специализированный вид объекта и потому могут

иметь свойства. Когда функции требуется **"статическая" переменная**, значение которой сохраняется между вызовами, часто удобно применять свойство самой функции.

Функции как пространства имен

```
(function () { // Функция chunkNamespace () , переписанная в виде неименованного выражения
    // фрагмент многократно используемого кода
)()); // Конец литерала типа функции и его вызов
```

Замыкания

Как и в большинстве современных языков программирования, в JavaScript используется *лексическая область видимости*. Это означает, что функции выполняются с применением области видимости переменных, которая действовала, когда они были определены, а не когда вызваны. Для реализации лексической области видимости внутреннее состояние объекта функции JavaScript должно включать не только код функции, но также ссылку на область видимости, в которой находится определение функции. Такое сочетание объекта функции и области видимости (набора привязок переменных), в которой распознаются переменные функции, в компьютерной литературе называется *замыканием*.

Следующая версия функции counter () представляет собой вариацию кода, который приводился в подразделе 6.10.6, но для закрытого состояния она использует замыкания вместо того, чтобы полагаться на обыкновенное свойство объекта:

```
//Аргумент функции n является закрытой переменной
function counter (n) {
    return {

        // Метод получения свойства возвращает и инкрементирует закрытую переменную счетчика
        get count () { return n++; } ,

        // Метод установки свойства не разрешает уменьшать
```

```
значение п
            set count (m) {
                  if (m > n) n = m;
                  else throw Error ("счетчик можно устанавливать
только в большее значение");
            }
      }
}

let c = counter (1000) ;
c. count // => 1000
c. count / => 1001
c.count = 2000;
c. count // => 2000
c.count = 2000; / !Error: счетчик можно устанавливать только в
большее значение


// Эта функция добавляет методы доступа для свойства с
указанным именем объекта о.
// Методы именуются как get<name> и set<name>.
// Если предоставляется функция предиката, тогда метод
установки применяет
// ее для проверки своего аргумента на предмет допустимости
перед его
// сохранением. Если предикат возвращает false, то метод
установки
// генерирует исключение.
// Необычной особенностью этой функции является то, что
значение свойства,
// которым манипулируют методы получения и установки, не
сохраняется в обьекте о.
// Взамен значение хранится только в локальной переменной в
данной функции.
// Методы получения и установки также определяются локально
внутри функции
// и потому имеют доступ к этой локальной переменной.
// Таким образом, значение закрыто по отношению к двум
методам доступа и его
// невозможно устанавливать или модифицировать иначе, чем
через метод установки.
```

```
function addPrivateProperty (o, name, predicate) (
    let value; // Значение свойства

    // Метод получения просто возвращает value
    o ['get$ (name]] = function () { return value: J;

    // Метод установки сохраняет значение или генерирует
    // исключение, если предикат отклоняет значение
    o['set$ {name]'] = function (v) {
        if (predicate && !predicate(v) ) I
            throw new TypeError ('set$ (name) : недопустимое
значение $(v)');
        } else {
            value = v;
        }
    };
}

// в следующем коде демонстрируется работа метода
addPrivateProperty ()
let o = {}; // Пустой объект

// Добавить методы доступа к свойству getName () и setName ().
// удостовериться, что разрешены только строковые значения.

addPrivateProperty (o, "Name", x => typeof x === "string");

o.setName ("Frank") ; // Установить значение свойства

o. getName()  // => "Frank"
o. setName(0) ; // !TypeError: попытка установки значения
неправильного типа
```

Важно помнить о том, что область видимости, ассоциированная с замыканием, является "динамической". Вложенные функции не создают закрытые копии области видимости или статические снимки привязок переменных. По существу проблема здесь в том, что переменные, объявленные с помощью var, определены повсюду в функции. Наш цикл for объявляет переменную цикла посредством var i, поэтому переменная i определена везде в

функции, а не имеет суженную область видимости внутри тела цикла. В коде демонстрируется распространенная категория ошибок в ES5 и предшествующих версиях, но введение переменных с блочной областью видимости в ES6 решает проблему. Если мы просто поменяем var на let или const, то проблема исчезнет. Поскольку let и const дают блочную область видимости, каждая итерация цикла определяет область видимости, независимую от областей видимости для всех остальных итераций, и каждая область видимости имеет собственную независимую привязку i.

—

Свойства, методы и конструктор функций

Поскольку функции — это объекты, как любой другой объект они могут иметь свойства и методы. Существует даже конструктор Function() для создания новых объектов функций.

Свойства

length

Предназначенное только для чтения свойство length функции указывает ее *арность* — количество параметров, объявленное в списке параметров функции, которое обычно соответствует количеству аргументов, ожидаемых функцией.

name

Предназначенное только для чтения свойство name функции указывает имя, которое использовалось при определении функции, если она была определена с именем либо имя переменной или свойства, которому было присвоено выражение неименованной функции, когда оно создавалось в первый раз. Свойство name главным образом полезно при написании отладочных сообщений и сообщений об ошибках.

prototype

Все функции кроме стрелочных имеют свойство prototype, которое ссылается на объект, известный как объект прототипа. Каждая функция имеет отличающийся объект прототипа. Когда функция применяется в качестве конструктора, вновь созданный объект наследует свойства от объекта

прототипа.

Методы

call(), apply ()

Вспомните, что стрелочные функции наследуют значение this контекста, где они определены. Это не удастся переопределить посредством методов call () и apply (). В случае вызова их на стрелочной функции первый аргумент фактически игнорируется.

```
let biggest = Math.max.apply(Math, arrayOfNumbers) ;
```

bind ()

Основная цель метода bind ( ) - привязка функции к объекту.

и toString(),

Как и все объекты JavaScript, функции имеют метод toString () . Спецификация ECMAScript требует, чтобы метод toString () возвращал строку, которая следует синтаксису оператора объявления функции. На практике большинство (но не все) реализаций метода toString () возвращают полный исходный код функции. Метод toString () встроенных функций обычно возвращает строку, которая включает что-то вроде "(native codel" * в качестве тела функции.

а также конструктор Function()

```
const f = new Function ("", "y", "return x*y;");
```

Последним аргументом должен быть текст тела функции;

—

Функциональное программирование

```
const map = function (a, ...args) ( return a.map (...args); );
const reduce = function (a, ...args) ( return a. reduce (...args); );
```

```
const sum = (x,y) => x+y;
const square = X => x*x;

let data = [1,1,3,5,5];
let mean = reduce (data, sum) /data. length;
let deviations = map (data, × => x-mean) ;
let stddev = Math.sart (reduce (map (deviations, square) , sum) /
(data. length-1));
stddev // => 2
```

——

Функции высшего порядка

Функция высшего порядка - это функция, которая оперирует функциями, принимая одну или большее количество функций в качестве аргументов и возвращая новую функцию. Вот пример:

```
// Эта функция высшего порядка возвращает новую функцию,
// которая передает свои аргументы f и возвращает логическое
// отрицание возвращаемого значения £

function not (f) {
      return function (..args) { / / Возвратить новую функцию,
            let result = f.apply (this, args) ; / / которая вызывает f

            return !result; //  и выполняет логическое отрицание ее
результата.
}

const even = x => x & 2 === 0; // функция для определения, четное
ли число
const odd = not (even) ; // Новая функция, которая делает
противоположное

[1,1,3,5,5]. every(odd) // => true: каждый элемент массива является
нечетным
```

Функция not () является функцией высшего порядка, т.к. она принимает аргумент типа функции и возвращает новую функцию.

В качестве другого примера рассмотрим приведенную ниже функцию mapper (). Она принимает аргумент типа функции и возвращает новую функцию, которая отображает один массив на другой, применяя переданную функцию. Функция mapper () использует определенную ранее функцию, и важно понимать, чем отличаются эти две функции:

```
// Возвращает функцию, которая ожидает аргумент типа массива
и применяет к каждому элементу, возвращая массив
возвращаемых значений.
// Сравните ее с определенной ранее функцией map () .

function mapper (f) {
      return a => map (a, f);
}

const increment = x => x+1;
const incrementAll = mapper (increment);

incrementAll ( [1,2,3]) // => [2,3,4]
```

Вот еще один, более универсальный пример функции высшего порядка, когорая принимает две функции, f и 9, а возвращает новую функцию, вычисляющую f (g ()):

```
// Возвращает новую функцию, которая вычисляет f(g(...)).
// Возвращаемая функция h передает все свои аргументы
функции 9,
// далее передает возвращаемое значение д функции f и затем
// возвращает возвращаемое значение £.
// Функции f и д вызываются с тем же значением this,
// с каким вызывалась h.

function compose (f, g) {
      return function (...args) {
            // Мы используем call для f, потому что передаем
одиночное значение,
            // и apply для g, поскольку передаем массив значений.

            return f.call (this, g.apply (this, args));
```

```
    };
}

const sum = (x,y) => x+y;
const square = x => x*x;
compose (square, sum) (2,3) // => 25; квадрат суммы
```

Функции partial () и memoize (), определяемые в подразделах ниже, являются двумя более важными функциями высшего порядка.

___

Функции с частичным применением

Метод bind () функции f (см. подраздел 8.7.5) возвращает новую функцию, которая вызывает £ в указанном контексте и с заданным набором аргументов. Мы говорим, что он привязывает функцию к объекту и частично применяет аргументы. Метод bind () частично применяет аргументы слева -- т.е. аргументы, передаваемые методу bind (), помещаются в начало списка аргументов, который передается исходной функции.

```
// Аргументы этой функции передаются слева
function partialleft (f, ...outerArgs)
    return function (...innerArgs) ( // Возвратить эту функцию
        let args = [...outerArgs, ...innerArgs]; // Построить СПИСОК аргументов
        return f.apply(this, args); // Затем вызвать с ним функцию f
    };
}



// Аргументы этой функции передаются справа
function partialRight (f, ...outerArgs) 1
    return function (...innerArgs) { // Возвратить эту функцию
        let args = [...innerArgs, ...outerArgs]; // Построить список аргументов
        return f.apply (this, args); / Затем вызвать с ним функцию f
    };
```

```
}

// Аргументы этой функции служат шаблоном. Неопределенные значения
// в списке аргументов заполняются значениями из внутреннего набора.
function partial (f, ...outerArgs) {
      return function (...innerArgs) {
            let args = [...outerArgs]; // Локальная копия шаблона внешних аргументов
            let innerIndex = 0;

            // Какой внутренний аргумент будет следующим
            // Проход в цикле по аргументам с заполнением неопределенных
            // значений значениями из внутренних аргументов
            for (let i = 0; i < args.length; it+)
                  if largs [i] === undefined) args [i) = innerArgs [innerIndex++];

                  // Присоединить любые оставшиеся внутренние аргументы
                  args.push (..innerArgs.slice (innerIndex));
                  return f.apply (this, args);
            };
}


// функция с тремя аргументами
const f = function (x,y, z) { return x * (y - z); }

// Обратите внимание на отличия этих трех частичных применений

partialleft(f, 2) (3,4) // => -2: Привязывает первый аргумент: 2 * (3 - 4)
partialRight(f, 2)(3,4) // => 6: Привязывает последний аргумент: 3 * (4 - 2)
partial (f, undefined, 2) (3,4) // => -6: Привязывает средний аргумент: 3 * (2 - 4)
```

Мемоизация

В подразделе 8.4.1 мы определили функцию вычисления факториала, которая кешировала свои ранее вычисленные результаты. В функциональном программировании такой вид кеширования называется мемоизацией (memoization). В следующем коде показана функция высшего порядка memoize (), которая принимает функцию в качестве аргумента и возвращает мемоизованную версию функции:

```
// Возвращает мемоизованную версию f. Работает, только если все
// аргументы f имеют отличающиеся строковые представления.

function memoize (f) {
    const cache = new Map () ; / / Кеш значений хранится в замыкании
    return function (...args) {
        // Создать строковую версию аргументов для использования в качестве ключа кеша
        let key = args. length + args. join ("+");

        if (cache.has (key)) {
            return cache.get (key);
        } else {
            let result = f.apply (this, args);
            cache. set (key, result);
            return result;
        }
    }
}
```