РАЗРАБОТКА ВЕБ-САЙТОВ*, JAVASCRIPT*, БЛОГ КОМПАНИИ RUVDS.COM

Как работает JS: обзор движка, механизмов времени выполнения, стека вызовов

ПЕРЕВОД

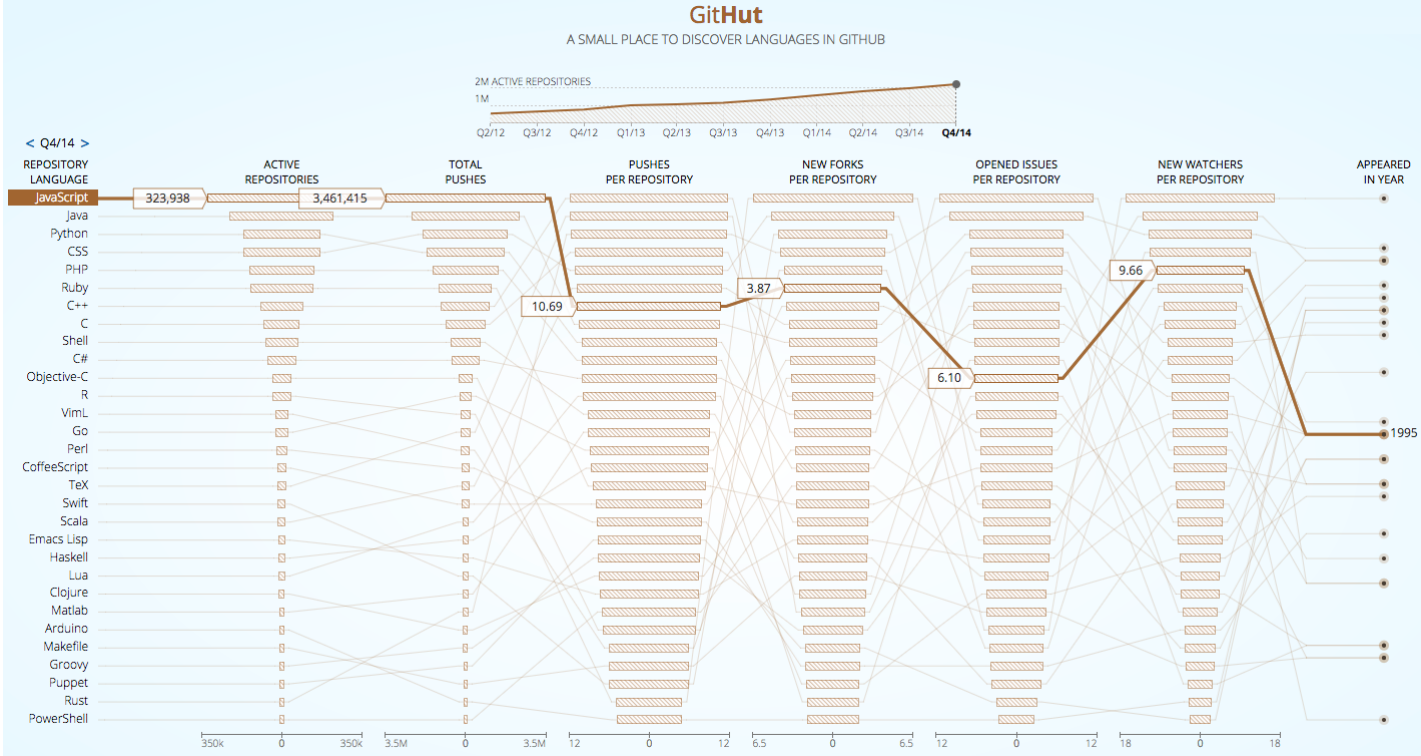
ru_vds 5 сентября 2017 в 12:02 57,9k

Оригинал: Alexander Zlatkov

→ Часть 2: Как работает JS: о внутреннем устройстве V8 и оптимизации кода
→ Часть 3: Как работает JS: управление памятью, четыре вида утечек памяти и борьба с ними

Популярность JavaScript растёт, его возможности используют на разных уровнях применяемых разработчиками стеков технологий и на множестве платформ. На JS делают фронтенд и бэкенд, пишут гибридные и встраиваемые приложения, а также многое другое.

Анализ статистики GitHub показывает, что по показателям активных репозиториев и push-запросов, JavaScript находится на первом месте, да и в других категориях он показывает довольно высокие позиции.



Статистические сведения по JavaScript с GitHub

С другой системой статистических сведений по GitHub можно ознакомиться здесь, она подтверждает то, что было сказано выше.

Если множество проектов плотно завязаны на JavaScript, значит, разработчикам необходимо как можно более эффективно использовать всё, что даёт им язык и его экосистема, стремясь, на пути разработки замечательных программ, к глубокому пониманию внутренних механизмов языка.

Как ни странно, существует множество разработчиков, которые регулярно пишут на JavaScript, но не знают, что происходит в его недрах. Пришло время это исправить: этот материал посвящён обзору JS-движка на примере V8, механизмов времени выполнения, и стека вызовов.

Обзор

Почти все слышали, в самых общих чертах, о JS-движке V8, и большинству разработчиков известно, что JavaScript — однопоточный язык, или то, что он использует очередь функций обратного вызова.

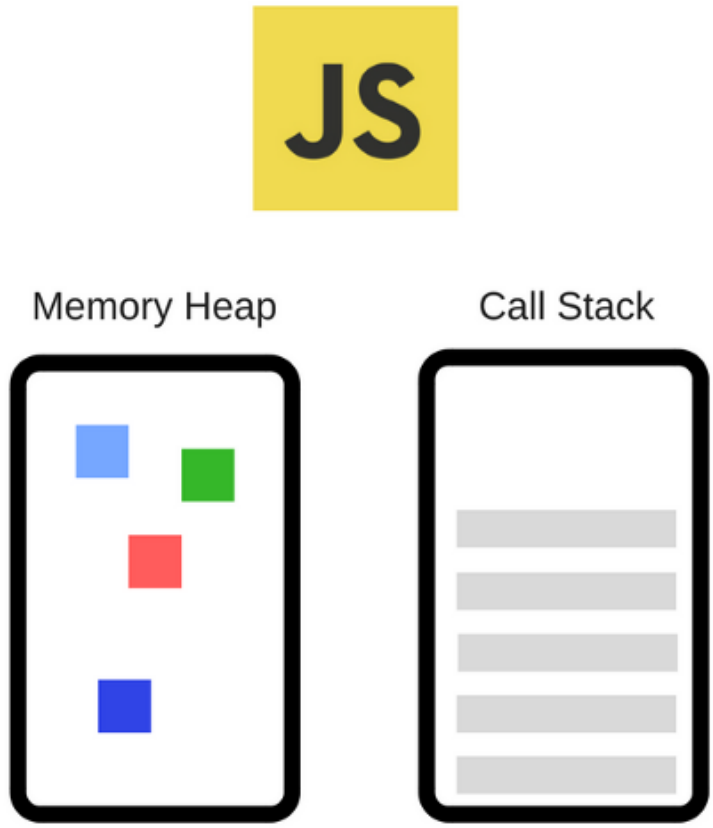
Здесь мы поговорим, на довольно высоком уровне, о выполнении JS-кода. Зная о том, что, на самом деле, происходит при выполнении JavaScript, вы сможете писать более качественные программы, которые выполняются без «подвисаний» и разумно используют имеющиеся API.

Если вы недавно начали писать на JavaScript, этот материал поможет вам понять, почему JS, в сравнении с другими языками, может показаться довольно-таки странным.

Если вы — опытный JS-разработчик, надеемся, этот материал поможет вам лучше понять, как на самом деле работает то, чем вы пользуетесь каждый день.

Движок JavaScript

V8 от Google — это широко известный JS-движок. Он используется, например, в браузере Chrome и в Node.js. Вот как его, очень упрощённо, можно представить:



Упрощённое представление движка V8

На нашей схеме движок представлен состоящим из двух основных компонентов:

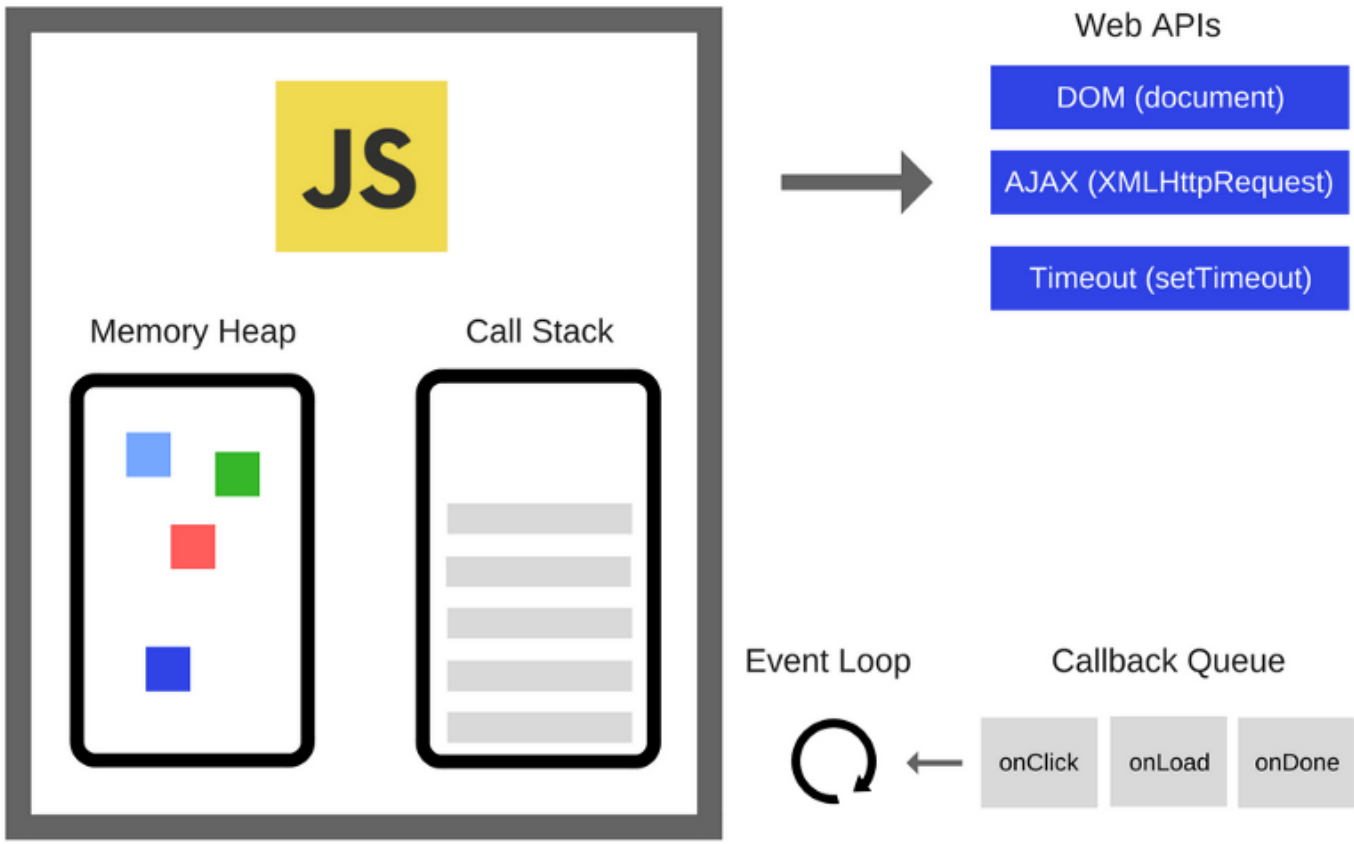
- Куча (Memory Heap) — то место, где происходит выделение памяти.
- Стек вызовов (Call Stack) — то место, куда в процессе выполнения кода попадают так называемые стековые кадры.

Механизмы времени выполнения

Если говорить о применении JavaScript в браузере, то здесь существуют API, например, что-то вроде функции `setTimeout`,

которые использует практически каждый JS-разработчик. Однако, эти API предоставляет не движок.

Откуда же они берутся? Оказывается, что реальность выглядит немного сложнее, чем может показаться на первый взгляд.



Движок, цикл событий, очередь функций обратного вызова и API, предоставляемые браузером

Итак, помимо движка у нас есть ещё очень много всего. Скажем — так называемые Web API, которые предоставляет нам браузер — средства для работы с DOM, инструменты для выполнения AJAX-запросов, нечто вроде функции `setTimeout`, и многое другое.

Стек вызовов

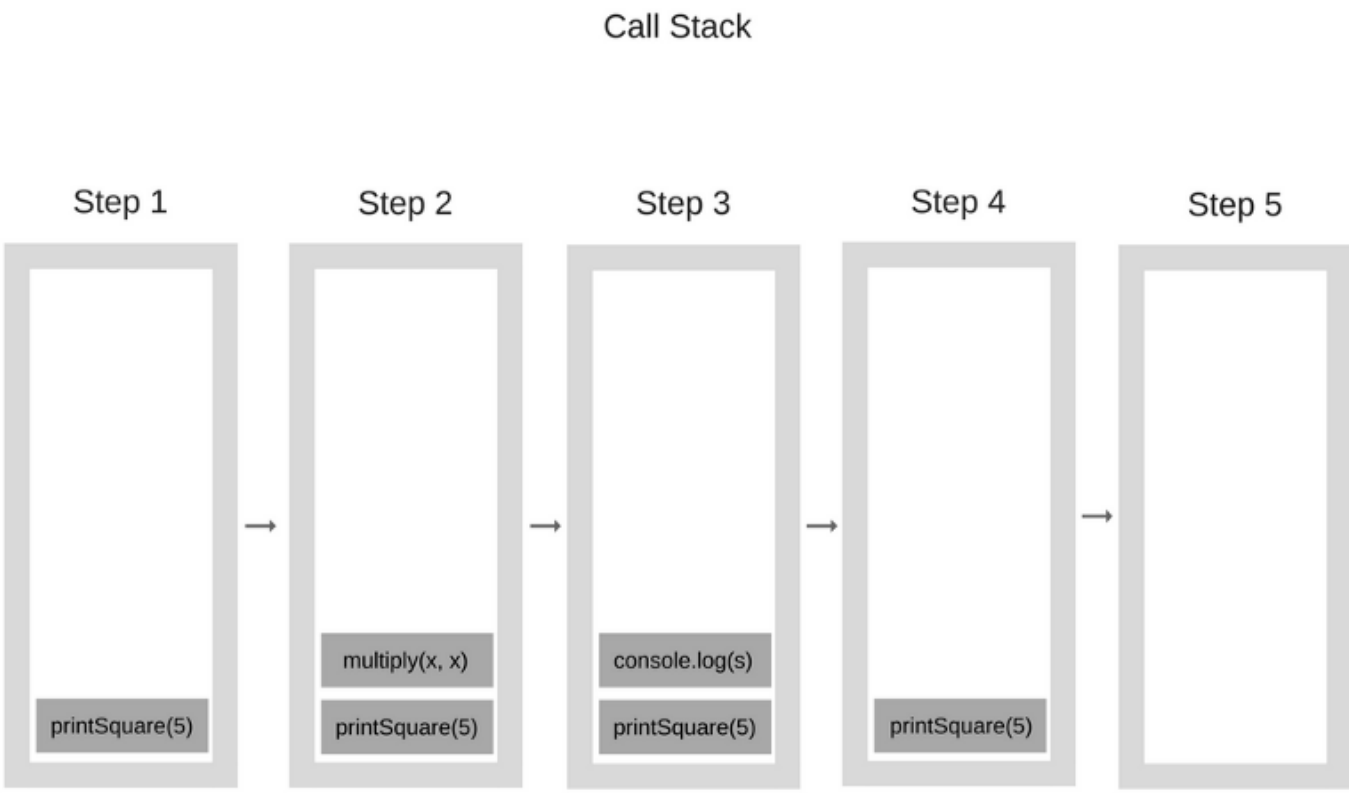
JavaScript — однопоточный язык программирования. Это означает, что у него один стек вызовов. Таким образом, в некий момент времени он может выполнять лишь какую-то одну задачу.

Стек вызовов — это структура данных, которая, говоря упрощённо, записывает сведения о месте в программе, где мы находимся. Если мы переходим в функцию, мы помещаем запись о ней в верхнюю часть стека. Когда мы из функции возвращаемся, мы вытаскиваем из стека самый верхний элемент и оказываемся там, откуда вызывали эту функцию. Это — всё, что умеет стек.

Рассмотрим пример. Взгляните на следующий код:

```
function multiply(x, y) {
    return x * y;
}
function printSquare(x) {
    var s = multiply(x, x);
    console.log(s);
}
printSquare(5);
```

Когда движок только начинает выполнять этот код, стек вызовов пуст. После этого происходит следующее:



Стек вызовов в ходе выполнения программы

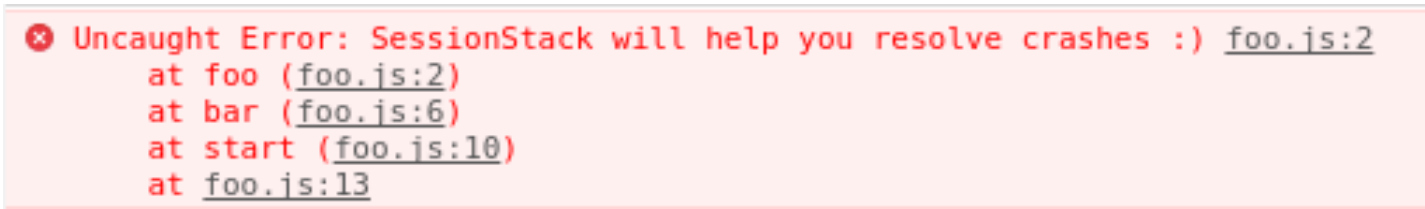
Каждая запись в стеке вызовов называется **стековым кадром**.

На механизме анализа стековых кадров основана информация о стеке вызовов, трассировка стека, выдаваемая при возникновении исключения. Трассировка стека представляет собой состояние стека в момент исключения. Взгляните на следующий код:

```
function foo() {
    throw new Error('SessionStack will help you resolve crashes
:)');
}
function bar() {
    foo();
}
```

```
function start() {
    bar();
}
start();
```

Если выполнить это в Chrome (предполагается, что код находится в файле `foo.js`), мы увидим следующие сведения о стеке:

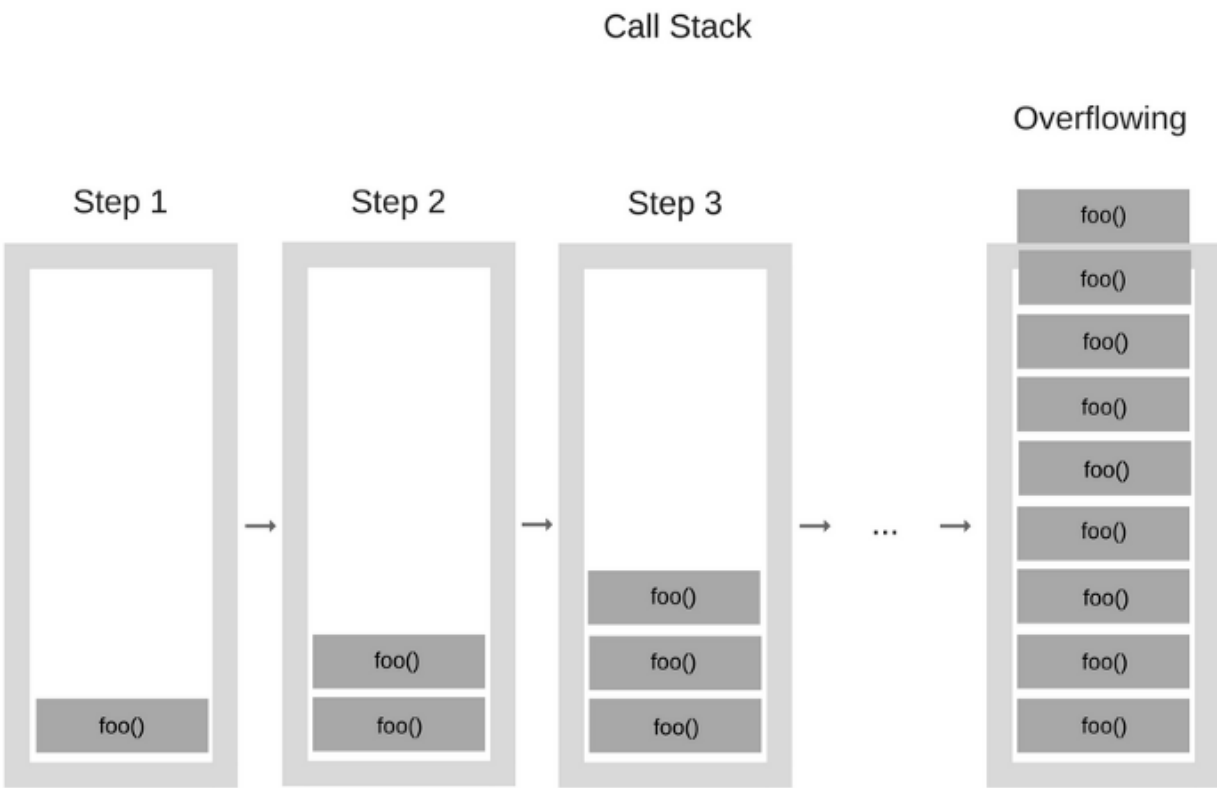


Трассировка стека после возникновения ошибки

Если будет достигнут максимальный размер стека, возникнет так называемое переполнение стека. Произойти такое может довольно просто, например, при необдуманном использовании рекурсии. Взгляните на этот фрагмент кода:

```
function foo() {
    foo();
}
foo();
```

Когда движок приступает к выполнению этого кода, всё начинается с вызова функции `foo`. Это — рекурсивная функция, которая не содержит условия прекращения рекурсии. Она бесконтрольно вызывает сама себя. В результате на каждом шаге выполнения в стек вызовов снова и снова добавляется информация об одной и той же функции. Выглядит это примерно так:



Переполнение стека

В определённый момент, однако, объём данных о вызовах функции превысит размер стека вызовов и браузер решит вмешаться, выдав ошибку:

```
▶Uncaught RangeError: Maximum call stack size exceeded
```

Превышение максимального размера стека вызовов

Модель выполнения кода в однопоточном режиме облегчает жизнь разработчика. Ему не нужно принимать во внимание сложные схемы взаимодействия программных механизмов, вроде возможности взаимной блокировки потоков, которые возникают в многопоточных окружениях.

Однако, и у исполнения кода в однопоточном режиме тоже есть определённые ограничения. Учитывая то, что у JavaScript имеется один стек вызовов, поговорим о том, что происходит, когда программа «тормозит».

Параллельное выполнение кода и цикл событий

Что происходит, когда в стеке вызовов имеется функция, на выполнение которой нужно очень много времени? Например, представьте, что вам надо выполнить какое-то сложно преобразование изображения с помощью JavaScript в браузере.

«А в чём тут проблема?», — спросите вы. Проблема заключается в том, что до тех пор, пока в стеке вызовов имеется выполняющаяся функция, браузер не может выполнять другие задачи — он оказывается заблокированным. Это означает, что браузер не может выводить ничего на экран, не может выполнять другой код. Он просто останавливается. Подобные эффекты, например, несовместимы с интерактивными интерфейсами.

Однако, это — не единственная проблема. Если браузер начинает заниматься обработкой тяжёлых задач, он может на достаточно долгое время перестать реагировать на какие-либо воздействия. Большинство браузеров в подобной ситуации выдают ошибку, спрашивая пользователя о том, хочет ли он завершить выполнение сценария и закрыть страницу.

Браузер предлагает завершить выполнение страницы

Пользователям подобные вещи точно не понравятся.

Итак, как же выполнять тяжёлые вычисления, не блокируя пользовательский интерфейс и не подвешивая браузер? Решение этой проблемы заключается в использовании асинхронных функций обратного вызова. Это — тема для отдельного разговора.

Итоги

Мы, в общих чертах, рассмотрели устройство JS-движка, механизмов времени выполнения и стека вызовов. Понимание изложенных здесь концепций позволяет улучшить качество кода.

Уважаемые читатели! Этот материал — первый в серии «How

JavaScript Works» из блога SessionStack. Уже опубликован второй — посвящённый особенностям V8 и техникам оптимизации кода. Как по-вашему, стоит ли его переводить?

Проголосовать:

+25

Поделиться:

Сохранить:

Комментарии (26)

Похожие публикации

Может ли в JavaScript конструкция (a==1 && a==2 && a==3) оказаться равной true? 95

ПЕРЕВОД

ru_vds • 25 января в 16:08

JavaScript и ужасы мутаций 48

ПЕРЕВОД

ru_vds • 19 января в 11:42

JavaScript: путь к ясности кода 11

ПЕРЕВОД

ru_vds • 15 ноября 2017 в 13:01

Популярное за сутки

Наташа — библиотека для извлечения структурированной информации из текстов на русском языке

14

alexkuku • вчера в 16:12

Unit-тестирование скриншотами: преодолеваем звуковой барьер. Расшифровка доклада

4

lahmatiy • вчера в 13:05

Люди не хотят чего-то действительно нового — они хотят привычное, но сделанное иначе

25

ПЕРЕВОД

Smileek • вчера в 10:32

Руководство по SEO JavaScript-сайтов. Часть 2. Проблемы, эксперименты и рекомендации

2

ПЕРЕВОД

ru_vds • вчера в 12:04

Как адаптировать игру на Unity под iPhone X к апрелю

0

P1CACHU • вчера в 16:13

Лучшее на Geektimes

Стивен Хокинг, автор «Краткой истории времени», умер на 77 году жизни — 33

HostingManager • вчера в 13:49

Обзор рынка моноколес 2018 — 70

lozga • вчера в 06:58

«Битва за Telegram»: 35 пользователей подали в суд на ФСБ — 40

alizar • вчера в 15:14

Стивен Хокинг и его работа — что дал ученый человечеству? — 8

marks • вчера в 14:46

Sunlike — светодиодный свет нового поколения — 17

AlexeyNadezhin • вчера в 20:32