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

Как работает JS: веб-воркеры и пять сценариев их использования

ПЕРЕВОД

ru_vds 7 февраля в 12:02 16,6k

Оригинал: Alexander Zlatkov

Публикуем перевод седьмой части часть серии материалов об особенностях работы различных механизмов JavaScript. Наша сегодняшняя тема — веб-воркеры. В частности, речь пойдёт о различных типах веб-воркеров, о том, как организована совместная работа тех частей, из которых они состоят, а также об их возможностях и об ограничениях, с которыми можно столкнуться в разных сценариях их использования. Здесь же будет показано 5 вариантов практического применения веб-воркеров.

Ограничения асинхронного программирования

Прежде чем мы начнём говорить о веб-воркерах, стоит вспомнить о том, что JavaScript — это однопоточный язык, однако, он поддерживает и возможности асинхронного выполнения кода.

Асинхронному программированию и вариантам его применения в JS-проектах был посвящён один из предыдущих материалов этой серии.

Асинхронное выполнение кода позволяет пользовательскому интерфейсу веб-приложений нормально функционировать, реагировать на команды пользователя. Система «планирует» нагрузку на цикл событий таким образом, чтобы в первую очередь выполнялись операции, связанные с пользовательским интерфейсом.

Хороший пример использования асинхронных методов программирования демонстрирует техника выполнения AJAX-запросов. Так как ожидание ответа способно занять много времени, запросы можно делать асинхронно, при этом, пока клиент ожидает ответа, может выполняться код, не относящийся к запросу.

```
// Предполагается использование jQuery
jQuery.ajax({
    url: 'https://api.example.com/endpoint',
    success: function(response) {
        // Код, который должен быть выполнен после получения ответа
    }
});
```

Такой подход, однако, демонстрирует следующую проблему: запросы обрабатываются WEB API браузера. Нас же интересует возможность асинхронного выполнения произвольного кода. Скажем, как быть, если код внутри функции обратного вызова интенсивно использует ресурсы процессора?

```
var result = performCPUIntensiveCalculation();
```

Если функция `performCPUIntensiveCalculation` — это не нечто вроде асинхронно выполняемого HTTP-запроса, а код, блокирующий главный поток (скажем, огромный и тяжёлый цикл `for`), то при однопоточном подходе к JS-разработке у нас нет способа освободить цикл событий и разблокировать интерфейс браузера. Как результат, пользователь не сможет с ним нормально работать.

Это означает, что асинхронные функции смягчают лишь небольшую часть ограничений, связанных с однопоточностью JS.

В некоторых случаях хорошего результата в деле разгрузки главного потока при выполнении ресурсоёмких операций можно достичь с помощью `setTimeout`. Например, если разбить сложные вычисления на фрагменты, выполняемые в разных вызовах `setTimeout`, их можно «распределить» по циклу событий, и таким образом, не блокировать пользовательский интерфейс.

Взглянем на простую функцию, которая вычисляет среднее значение для числового массива.

```
function average(numbers) {
    var len = numbers.length,
        sum = 0,
        i;

    if (len === 0) {
        return 0;
    }

    for (i = 0; i < len; i++) {
        sum += numbers[i];
    }

    return sum / len;
}
```

Этот код можно переписать так, чтобы он «эмулировал» асинхронное выполнение:

```
function averageAsync(numbers, callback) {
    var len = numbers.length,
        sum = 0;

    if (len === 0) {
        return 0;
    }

    function calculateSumAsync(i) {
        if (i < len) {
            // Поместим следующий вызов функции в цикл событий.
            setTimeout(function() {
                sum += numbers[i];
                calculateSumAsync(i + 1);
            }, 0);
        } else {
            // Так как достигнут конец массива, мы вызываем коллбэк
            callback(sum / len);
        }
    }
```

```
    calculateSumAsync(0);
}
```

При таком подходе мы используем функцию `setTimeout`, которая планирует выполнение вычислений. Это приводит к размещению в цикле событий функции, выполняющей следующую порцию вычислений, таким образом, что между сеансами выполнения этой функции достаточно времени для других вычислений, в том числе и для тех, которые связаны с пользовательским интерфейсом.

Веб-воркеры

HTML5 дал нам множество замечательных возможностей, среди которых можно отметить следующие:

- SSE (эту технологию мы рассматривали и сравнивали с протоколом WebSocket в одном из предыдущих материалов).
- Геолокация.
- Кэш приложения.
- Локальное хранилище.
- Технология Drag and Drop.
- Веб-воркеры.

Веб-воркеры — это потоки, принадлежащие браузеру, которые можно использовать для выполнения JS-кода без блокировки цикла событий.

Это поистине замечательная возможность. Система понятий JavaScript основана на идее однопоточного окружения, а теперь

перед нами технология, которая (частично) снимает это ограничение.

Веб-воркеры позволяют разработчику размещать задачи, для выполнения которых требуются длительные и сложные вычисления, интенсивно задействующие процессор, в фоновых потоках, без блокировки пользовательского интерфейса, что позволяет приложениям оперативно реагировать на воздействия пользователя. Более того, нам больше не нужны обходные пути, вроде рассмотренного выше трюка с `setTimeout` для того, чтобы найти приемлемый способ взаимодействия с циклом событий.

Вот простой пример, который демонстрирует разницу между сортировкой массива с помощью веб-воркера и без него.

Обзор веб-воркеров

Веб-воркеры позволяют выполнять тяжёлые в вычислительном плане и длительные задачи без блокировки потока пользовательского интерфейса. На самом деле, при их использовании вычисления выполняются параллельно. Перед нами настоящая многопоточность.

Возможно, вы вспомните о том, что JavaScript — это однопоточный язык программирования. Пожалуй, тут вы должны осознать, что JS — это язык, который не определяет модель потоков. Веб-воркеры не являются частью JavaScript. Они представляют собой возможность браузера, к которой можно получить доступ посредством JavaScript. Большинство браузеров исторически были

однопоточными (эта ситуация, конечно, изменилась), и большинство реализаций JavaScript создано для браузеров.

Веб-воркеры не реализованы в Node.js — там есть концепция «кластеров» или «дочерних процессов», а это уже немного другое.

Стоит отметить, что спецификация упоминает три типа веб-воркеров:

- Выделенные воркеры (Dedicated Workers)
- Разделяемые воркеры (Shared Workers)
- Сервис-воркеры (Service Workers)

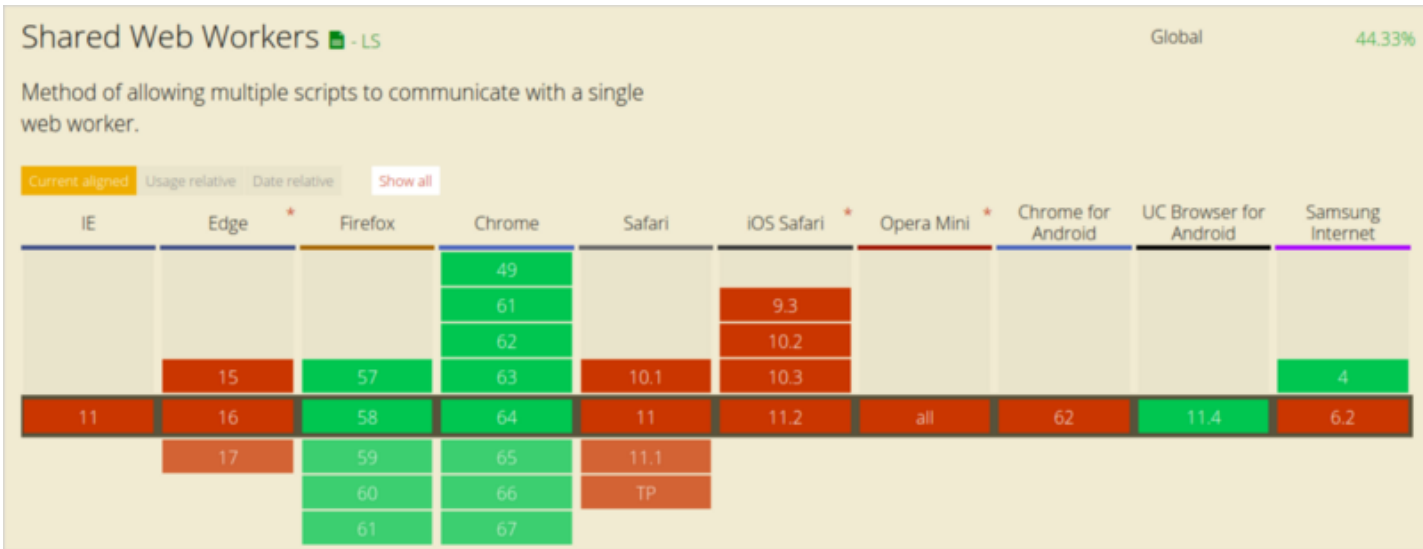
Выделенные воркеры

Экземпляры выделенных веб-воркеров создаются главным процессом. Обмениваться данными с ними может только он.

Поддержка выделенных воркеров в браузерах

Разделяемые воркеры

Доступ к разделяемому воркеру может получить любой процесс, имеющий тот же источник, что и воркер (например — разные вкладки браузера, `iframe`, и другие разделяемый воркеры).



Поддержка разделяемых воркеров в браузерах

Сервис-воркеры

Сервис-воркеры — это воркеры, управляемые событиями, зарегистрированные с использованием источника их происхождения и пути. Они могут контролировать веб-страницу, с которой связаны, перехватывая и модифицируя команды навигации и запросы ресурсов, и выполняя кэширование данных, которым можно очень точно управлять. Всё это даёт нам отличные средства управления поведением приложения в определённой ситуации (например, когда сеть недоступна).

Поддержка сервис-воркеров в браузерах

Надо отметить, что в этом материале мы будем заниматься выделенными воркерами, именно их мы будем иметь в виду, говоря о «веб-воркерах» или о «воркерах».

Как работают веб-воркеры

Веб-воркеры реализованы с использованием .js-файлов, которые включаются в страницу с применением асинхронного HTTP-запроса. Эти запросы полностью скрыты от разработчика благодаря Web Worker API.

Воркеры используют механизмы передачи сообщений, характерные для технологий, которые применяются для организации взаимодействия потоков, что позволяет организовать их параллельное выполнение. Они отлично подходят для того, чтобы выполнять тяжёлые вычислительные операции, не замедляя работу пользовательского интерфейса.

Веб-воркеры выполняются в изолированных потоках в браузере. Как результат, код, который они выполняют, должен быть включён в отдельный файл. Это важно запомнить.

Вот как создают веб-воркеры:

```
var worker = new Worker('task.js');
```

Если файл `task.js` существует и к нему есть доступ, браузер создаст новый поток, который асинхронно загрузит этот файл. После того, как загрузка будет завершена, начнётся выполнение кода воркера. Если при попытке загрузки файла браузер получит сообщение об ошибке 404, файл загружен не будет, при этом сообщения об ошибках не выводятся.

Для запуска только что созданного воркера нужно вызвать его метод `postMessage`:

```
worker.postMessage();
```

Обмен данными с веб-воркером

Для того чтобы страница, создавшая веб-воркер, могла взаимодействовать с ним, нужно использовать либо метод `postMessage`, либо широковещательный канал передачи данных, представленный объектом BroadcastChannel.

Метод postMessage

При вызове этого этого метода более новые браузеры поддерживают, в качестве первого параметра, объект JSON, а в более старых браузерах поддерживается лишь параметр типа `String`.

Посмотрим на пример того, как страница, создавшая воркер, может обмениваться с ним данными, используя JSON-объект. При передаче строки выглядит всё практически точно так же.

Вот часть HTML-страницы:

```
<button onclick="startComputation()">Start computation</button>

<script>
  function startComputation() {
    worker.postMessage({'cmd': 'average', 'data': [1, 2, 3, 4]});
  }

  var worker = new Worker('doWork.js');

  worker.addEventListener('message', function(e) {
    console.log(e.data);
  }, false);

</script>
```

Вот содержимое файла с кодом воркера:

```
self.addEventListener('message', function(e) {
  var data = e.data;
  switch (data.cmd) {
    case 'average':
      var result = calculateAverage(data); // Функция, вычисляющая
среднее значение числового массива
      self.postMessage(result);
```

```
      break;
    default:
      self.postMessage('Unknown command');
  }
}, false);
```

Когда нажимают на кнопку, на странице выполняется вызов метода `postMessage` воркера. Этот вызов передаёт воркеру JSON-объект с ключами `cmd` и `data` и соответствующими им значениями. Воркер обработает это сообщение посредством заданного в нём обработчика `message`.

Когда воркер получает сообщение и понимает, чего от него хотят, он будет выполнять вычисления самостоятельно, не блокируя цикл событий. То, чем занимается воркер, выглядит как стандартная JS-функция. Когда вычисления завершены, их результаты передаются главной странице.

В контексте воркера и `self`, и `this`, указывают на глобальное пространство имён для воркера.

Для того чтобы остановить воркер, можно воспользоваться одним из двух способов. Первый заключается в вызове с главной страницы метода `worker.terminate()`. Второй выполняется внутри воркера и реализуется командой `self.close()`.

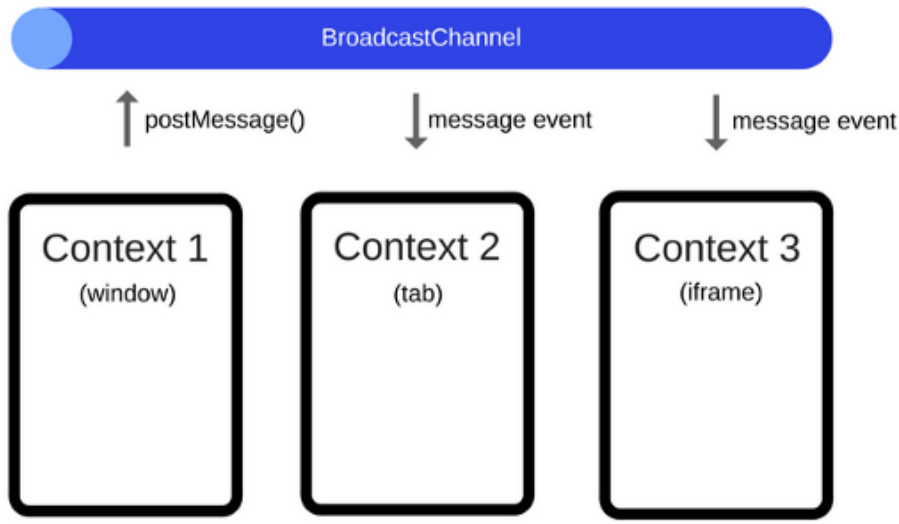
Широковещательный канал передачи данных

Объект BroadcastChannel представляет собой более универсальное API для передачи данных. Он позволяет

передавать сообщения, которые можно принять во всех контекстах, имеющих один и тот же источник. Все вкладки браузера, `iframe` или воркеры, относящиеся к одному источнику, могут передавать и принимать широковещательные сообщения:

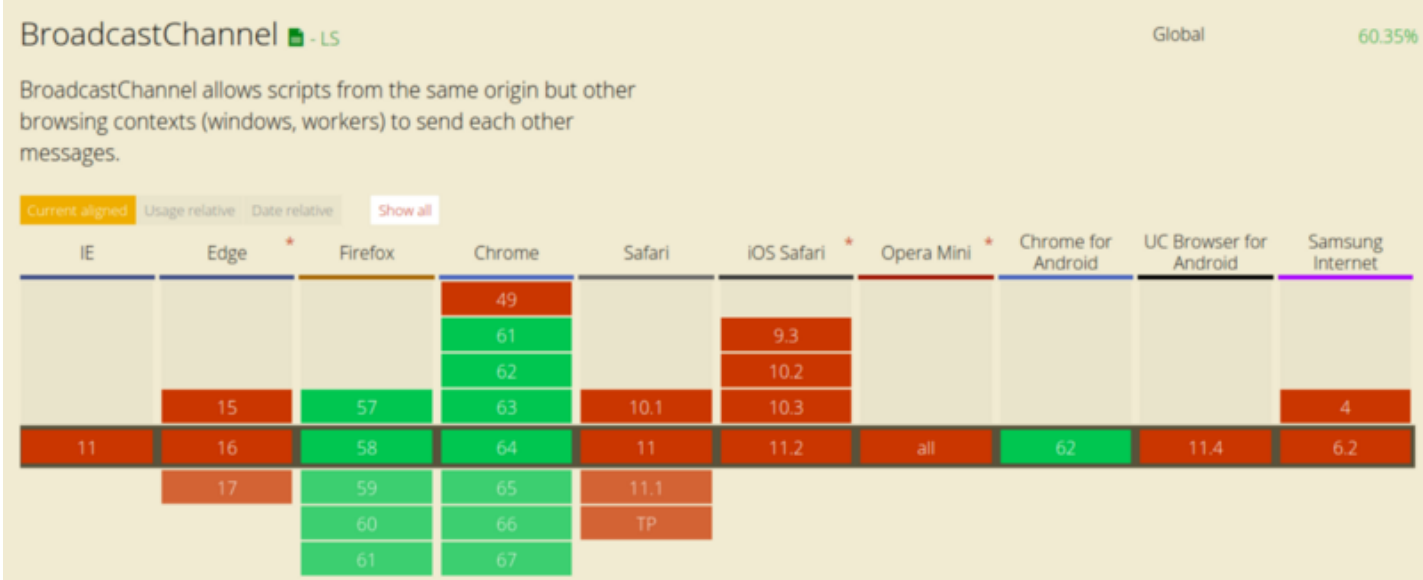
```
// Подключение к широковещательному каналу
var bc = new BroadcastChannel('test_channel');

// Пример отправки сообщения
bc.postMessage('This is a test message.');

// Пример простого обработчика событий, который
// выводит сообщения в консоль
bc.onmessage = function (e) {
  console.log(e.data);
}

// Отключение от канала
bc.close()
```

Вот как выглядит схема взаимодействия различных сущностей с использованием широковещательного канала обмена сообщениями:



Обмен данными с использованием широковещательного канала передачи сообщений

Однако тут стоит отметить, что объект `BroadcastChannel` пока имеет довольно ограниченную поддержку в браузерах.



Поддержка BroadcastChannel в браузерах

Способы отправки сообщений веб-воркерам

Есть два способа отправки сообщений веб-воркерам. Первый заключается в копировании данных, второй — в передаче данных от источника к приёмнику без их копирования. Рассмотрим эти способы работы с сообщениями:

- Копирование сообщения. Сообщение сериализуется, копируется, отправляется, а затем, на принимающей стороне, десериализуется. Страница и воркер не используют общий экземпляр сообщения, поэтому тут мы сталкиваемся с созданием копий данных в каждом сеансе отправки сообщений. Большинство браузеров реализуют эту возможность путём автоматического преобразования передаваемой информации в JSON на стороне передатчика и декодирования этих данных на стороне приёмника. Как можно ожидать, это добавляет значительную нагрузку на систему при отправке сообщений.

Чем больше сообщение — тем больше времени займёт его отправка.
- Передача сообщения. При таком подходе оказывается, что отправитель сообщения больше не может использовать сообщение после того, как оно отправлено. При этом передача сообщений выполняется практически мгновенно. Главная особенность этого метода заключается в том, что передать с его помощью можно только объект ArrayBuffer.

Возможности, доступные веб-воркерам

Веб-воркерам, из-за их многопоточной сущности, доступен лишь ограниченный набор возможностей JavaScript. Вот эти возможности:

- Объект `navigator`
- Объект `location` (только для чтения)
- `XMLHttpRequest`
- `setTimeout()/clearTimeout()` и `setInterval()/clearInterval()`
- Кэш приложения
- Импорт внешних скриптов с использованием `importScripts()`
- Создание других веб-воркеров

Ограничения веб-воркеров

К сожалению, у веб-воркеров нет доступа к некоторым весьма важным возможностям JavaScript. Среди них следующие:

- DOM (это не потокобезопасно)
- Объект `window`
- Объект `document`
- Объект `parent`

Всё это значит, что веб-воркеры не могут манипулировать DOM (и, таким образом, не могут прямо влиять на пользовательский интерфейс). Поначалу может показаться, что это значительно усложняет использование веб-воркеров, однако со временем, узнав о том, как правильно использовать веб-воркеры, вы начнёте воспринимать их как отдельные «вычислительные машины», в то время как то, что относится к работе с пользовательским интерфейсом, будет выполняться в коде страницы. Воркеры будут выполнять тяжёлые вычисления, и после того, как работа будет завершена, отправлять результаты на страницу, вызывающую их, код которой уже внесёт необходимые изменения в пользовательский интерфейс.

Обработка ошибок

Как и при работе с любым JS-кодом, в веб-воркерах нужно обрабатывать ошибки. Если ошибка возникает в процессе выполнения воркера, вызывается событие `ErrorEvent`. Объект ошибки содержит три полезных свойства, которые позволяют понять её суть:

- `filename` — имя файла, в котором содержится скрипт воркера, вызвавший ошибку.
- `lineno` — номер строки, в которой произошла ошибка.

- `message` — описание ошибки.

Вот пример кода для обработки ошибок в веб-воркере:

```
function onError(e) {
  console.log('Line: ' + e.lineno);
  console.log('In: ' + e.filename);
  console.log('Message: ' + e.message);
}

var worker = new Worker('workerWithError.js');
worker.addEventListener('error', onError, false);
worker.postMessage(); // Запустить воркер без сообщения.
```

Вот код воркера

```
self.addEventListener('message', function(e) {
  postMessage(x * 2); // Намеренная ошибка. 'x' не определено.
};
```

Тут вы можете видеть, как мы создали воркер и назначили ему обработчик события `error`.

Внутри воркера (второй фрагмент кода) мы намеренно вызываем исключение, умножая `x` на 2 в то время как `x` не определено в текущей области видимости. Исключение доходит до исходного скрипта и вызывается обработчик `onError`, выводящий сведения об ошибке.

Сценарии использования веб-воркеров

Мы рассказали о сильных и слабых сторонах веб-воркеров. Теперь

рассмотрим несколько сценариев их использования.

- Рендеринг трёхмерных сцен. В частности, речь идёт о реализации метода трассировки лучей — техники рендеринга, позволяющей создавать изображения путём отслеживания направления лучей света и определения цвета пикселей. Трассировка лучей использует интенсивные математические вычисления для моделирования особенностей распространения света. При таком подходе реализуются такие эффекты, как отражения и преломления, трассировка лучей позволяет добиться имитировать внешний вид различных материалов, и так далее. Вся эта вычислительная логика может быть вынесена в веб-воркер для того, чтобы она не блокировала поток пользовательского интерфейса. Можно сделать ещё интереснее, а именно, разделить рендеринг изображения между несколькими воркерами (и, соответственно, между несколькими процессорными ядрами). Вот простой пример реализации трассировки лучей с использованием веб-воркеров.
- Шифрование. Сквозное шифрование становится всё более популярным из за всё возрастающего внимания к регулированию распространения персональных и конфиденциальных данных. Операции шифрования могут быть достаточно продолжительными, особенно если возникает необходимость в частом шифровании больших объёмов данных. Это — весьма адекватный сценарий использования веб-воркера, так как тут не нужен доступ к объектам DOM или нечто подобное. Шифрование — это алгоритмы обработки информации, которым достаточно базовых возможностей JS.

Когда шифрование выполняется воркером, это не влияет на работу пользователя с интерфейсом сайта.

- Предварительная загрузка данных. Для того чтобы оптимизировать веб-сайт и улучшить впечатления пользователя от работы с ним, можно использовать веб-воркеры для заблаговременной загрузки и сохранения некоторых данных, которыми можно очень быстро воспользоваться тогда, когда позже в них возникнет необходимость. Веб-воркеры отлично подходят для подобного сценария использования, так как выполняемые ими операции не подействуют на интерфейс приложения, в отличие от предварительной загрузки данных, реализованной средствами главного потока.
- Прогрессивные веб-приложения. Такие приложения должны, даже при ненадёжном сетевом соединении, быстро загружаться. Это означает, что данные нужно хранить в браузере локально. Именно здесь в дело вступает IndexedDB или похожее API. В целом, речь идёт о необходимости обслуживания некоего хранилища данных на стороне клиента. Для того чтобы работать с этим хранилищем, не блокируя пользовательский интерфейс, работу надо организовать в веб-воркере. Тут надо отметить, что, в случае с IndexedDB, существует асинхронное API, которое позволяет не нагружать главный поток и без веб-воркеров, но раньше здесь было синхронное API (которое может появиться снова), которым нужно пользоваться только внутри веб-воркеров.
- Проверка правописания. Простая система проверки правописания работает так: программа считывает файл словаря со списком правильно написанных слов. Из словаря формируется дерево поиска, которое обеспечивает

эффективный поиск по тексту. Когда системе передают слово для проверки, она проверяет его наличие в дереве поиска. Если слово найти не удаётся, пользователю могут быть предоставлены альтернативные варианты этого слова, полученные путём замены символов исходного слова и поиска полученных слов в дереве на предмет проверки того, являются ли они, с точки зрения системы проверки, правильными. Всё это легко можно вынести в веб-воркер, что даст пользователю возможность работать с текстом, не испытывая проблем, связанных с блокировкой интерфейса при проверке слова и при поиске альтернативных вариантов его написания.

Итоги

В этом материале мы рассказали о веб-воркерах — сравнительно новой возможности, доступной веб-разработчикам в большинстве современных браузеров. Веб-воркеры позволяют выносить в отдельные потоки выполнение ресурсоёмких операций, что позволяет не нагружать главный поток, который может спокойно обрабатывать всё, что связано с пользовательским интерфейсом.

Предыдущие части цикла статей:

Часть 1: Как работает JS: обзор движка, механизмов времени выполнения, стека вызовов
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

Часть 4: Как работает JS: цикл событий, асинхронность и пять способов улучшения кода с помощью async / await
Часть 5: Как работает JS: WebSocket и HTTP/2+SSE. Что выбрать?
Часть 6: Как работает JS: особенности и сфера применения WebAssembly

Уважаемые читатели! Используете ли вы веб-воркеры в своих проектах?

Похожие публикации

Неявное преобразование типов в JavaScript. Сколько будет !+[]+[]+![]?

29

ПЕРЕВОД

ru_vds • 30 января в 14:13

JavaScript-прокси: и красиво, и полезно

13

ПЕРЕВОД

ru_vds • 29 января в 12:23

Машины состояний и разработка веб-приложений

8

ПЕРЕВОД

ru_vds • 18 января в 12:02

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

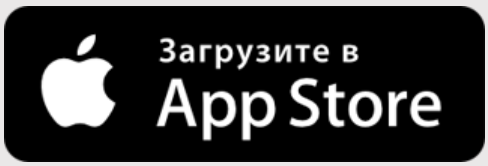
Стивен Хокинг и его работа — что дал ученый человечеству? 8

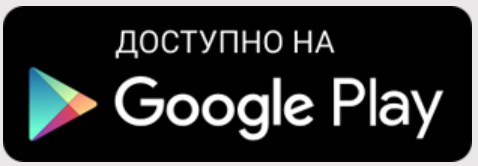
marks • вчера в 14:46

Sunlike — светодиодный свет нового поколения 17

AlexeyNadezhin • вчера в 20:32

Мобильное приложение





Полная версия