УДК 004.42

МНОГОПОТОЧНАЯ ОБРАБОТКА ИЗОБРАЖЕНИЙ С ИСПОЛЬЗОВАНИЕМ API WEB-WORKERS

Статья поступила в редакцию 01.03.2021, в окончательном варианте – 21.05.2021.

Бородин Олег Валерьевич, Волгоградский государственный технический университет, 400005, Российская Федерация, г. Волгоград, пр. им. Ленина, 28,
магистрант, e-mail: oleg.borodin.1998@mail.ru
Егунов Виталий Алексеевич, Волгоградский государственный технический университет, 400005, Российская Федерация, г. Волгоград, пр. им. Ленина, 28,
кандидат технических наук, доцент, e-mail: vegunov@mail.ru

В работе рассматривается WebWorkers API – средство, которое позволяет посредством языка JavaScript запускать пользовательские скрипты в фоновом потоке браузера. Использование WebWorkers API привносит в клиентскую веб-разработку многопоточность, которую изначально JavaScript не поддерживает. Рассматриваются проблемы асинхронного выполнения, с которыми можно столкнуться на клиентской стороне. Основное внимание уделяется одной из разновидностей воркеров – Dedicated Workers, их совместимости и применимости использования. Приводится в качестве примера несколько ситуаций, которые демонстрируют плюсы использования технологии. Рассматривается вычислительный эксперимент, в рамках которого осуществляется параллельная обработка нескольких изображений. Анализируется решение этой задачи как в многопоточной реализации, так и без использования WebWorkers API, анализируется полученное ускорение. Приводится обзор существующих решений, где в качестве альтернативы WebWorkers API приводится вариант асинхронного исполнения веб-приложений, в частности техника выполнения AJAX-запросов. Также в качестве некоторой альтернативы приводится использование Promise, которые используются для согласования синхронных и асинхронных действий приложения. Анализируются достоинства и недостатки технологии WebWorkers API. В конце работы подводятся итоги и выделяются реальные сценарии использования технологии.

Ключевые слова: JavaScript Multithreading, Web Development, Dedicated Workers, WebWorkers API, Image Processing, Loaded Background Calculations, JavaScript Parallelize

MULTI-THREADED FRONTEND IMAGE PROCESSING USING WEB-WORKERS API

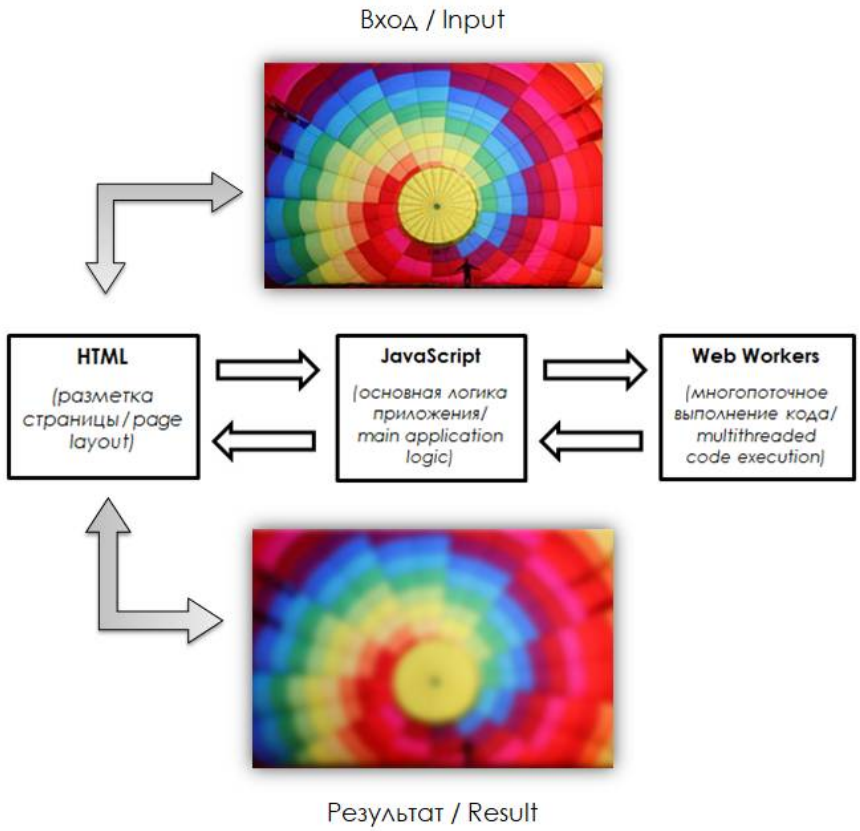
The article was received by the editorial board on 01.03.2021, in the final version – 21.05.2021.

Borodin Oleg V., Volgograd State Technical University, 28 Lenin Ave., Volgograd, 400005, Russian Federation,
master student, e-mail: oleg.borodin.1998@mail.ru
Egunov Vitaly A. Volgograd State Technical University, 28 Lenin Ave., Volgograd, 400005, Russian Federation
Cand. Sci. (Engineering), e-mail: vegunov@mail.ru

The paper considers the WebWorkers API, a tool that allows you to run user scripts in the background stream of the browser using the JavaScript language. Using the WebWorkers API brings multithreading to client web development that JavaScript does not support natively. The problems of asynchronous execution that can be encountered on the client side are considered. The main attention is paid to one of the types of workers – Dedicated Workers, their compatibility and applicability of use. As an example, there are several situations that demonstrate the advantages of using the technology. A computational experiment is considered in which several images are processed in parallel. The solution to this problem is analyzed both in a multithreaded implementation and without using the WebWorkers API, and the resulting acceleration is analyzed. An overview of existing solutions is provided, where, as an alternative to the WebWorkers API.

Keywords: JavaScript Multithreading, Web Development, Dedicated Workers, Web Workers API, Image Processing, Loaded Background Calculations, JavaScript Parallelize

Графическая аннотация (Graphical annotation)



Введение. Возможность использования параллельных вычислений, безусловно, является весомым технологическим аспектом, с помощью которого систему можно не только масштабировать для многопользовательского использования, но и задействовать все доступные вычислительные ресурсы.

Клиентская среда современных веб-приложений, как правило, представлена языком JavaScript, который в свою очередь не определяет параллельную модель выполнения – JavaScript выполняется в одном потоке. Это означает, что код на клиенте будет выполняться с использованием всего одного потока, независимо от возможностей клиентской машины. При определенных обстоятельствах это может накладывать на разработчиков существенные ограничения. И, конечно, идеологически клиент-серверная архитектура (рис. 1) не подразумевает выполнения сложных операций на клиенте – данные поступают в уже подготовленном виде, всё, что требуется сделать – отобразить их, возможно с некоторой предобработкой или дополнительной логикой, отрисовав соответствующие формы, таблицы и прочие элементы пользовательского интерфейса.

Рисунок 1 – Технология Клиент-сервер

Однако современные веб-приложения уже давно решают более сложные задачи, нежели 10 лет назад, а значит, и клиентская сторона стала обширнее и сложнее. В настоящее время фронтенд [1] представляет собой нечто большое, чем средство отображения форм и кнопок. Об этом могут сказать многочисленные графики популярности языков и выраженного интереса аудитории к ним. По данным Stack Overflow Developer Survey 2019 JavaScript 7 раз подряд стал самым популярным языком [2], что, безусловно, подчеркивает множество вариантов использования языка.

Также за последнее время выросли и размеры клиентских приложений. Об этом может сказать статья The Cost of Javascript Frameworks Тима Кадлеца, в которой проводится исследование зависимости приложений от использования комплексных фреймворков и библиотек, а также различных метрик, касающихся приложений и используемых системных ресурсов. «В состав проектов включается всё больше и больше JS-кода. По мере того, как организации движутся в сторону приложений, работающих на базе фреймворков и библиотек вроде React, Vue и прочих, мы делаем основную функциональность приложений очень сильно зависимой от JavaScript» [3]. Всё это говорит о том, что нагрузка на клиентскую сторону увеличилась.

В данной работе пойдет речь о веб-воркерах – относительно новой возможности языка JavaScript, которая позволяет выполнять вычисления в фоновом потоке браузера, не оказывая нагрузку на основной поток выполнения кода. Большинство браузеров исторически были однопоточными (сейчас эта ситуация уже изменилась), и большинство реализаций JavaScript создавалось для браузеров. Однако веб-воркеры не являются частью самого языка, они представляют собой возможность браузера, к которой можно получить доступ. То есть язык в данном случае выступает лишь в качестве интерфейса взаимодействия.

Обзор существующих решений. Как уже было сказано JavaScript – это однопоточный язык, а технология WEB WORKERS API достаточно свежая и уникальная. Поэтому конкретных аналогов данной технологии выделить сложно, но можно выделить подходы, которые применяются при распределении вычислений именно в среде JavaScript. Речь идёт об асинхронном выполнении кода.

В процессе работы браузеры выполняют различные задачи. Например, осуществляют обработку изображений, выполняют операции с локальными файлами, формируют сетевые запросы и ожидают ответа на них. Выполнение всех этих операций может занимать значительное время, соответственно в стеке запросов, куда помещаются данные задачи, формируются задержки («blocking»). В случае блокировки стека запросов, браузер не выполняет другой код, в том числе не обрабатывает запросы к пользовательскому интерфейсу. Для того чтобы избежать подобных ситуаций, браузеры используют асинхронную модель поведения [4], которая позволяет пользовательскому интерфейсу нормально функционировать, реагировать на команды пользователя.

Классическим примером использования асинхронных методов программирования является техника выполнение AJAX-запросов. В силу того, что заранее неизвестно, какое время займет ожидание ответа, запросы необходимо делать асинхронно. При этом в процессе ожидания клиент может выполнять код, который не относится к запросу. Ниже приведен пример использования такого запроса (предполагается использование библиотеки jQuery).

```
jQuery.ajax({
   url: 'https://some_site.ru',
   success: function(response) {
      // Код, выполняемый после получения ответа
   }
});
```

Однако даже при таком подходе все запросы обрабатываются WEB API браузера. Мы же хотим добиться возможности асинхронного выполнения произвольного кода. Например, код внутри функции обратного вызова, который интенсивно использует ресурсы компьютера:

```
let result = heavyCalculations();
```

Если функция heavyCalculations — произвольный код, блокирующий главный поток (например, долгий цикл), то при однопоточном подходе к JS-разработке нет никакого способа разблокировать интерфейс, браузер будет занят обработкой цикла. Существующие асинхронные функции лишь в какой-то мере смягчают ограничения, связанные с однопоточностью JS, например, в части выполнения асинхронных запросов.

Еще одним средством разгрузки главного потока можно считать функцию отложенного вызова setTimeout. Одним из широко распространенных приемов программирования является

разбиение сложных вычислений на фрагменты, которые выполняются в разных вызовах setTimeout. Данные фрагменты можно «распределить» по циклу событий, избежав блокировки выполнения основного сценария.

Рассмотрим простую функцию, которая вычисляет среднее значение для массива чисел методом обычного перебора в цикле for:

```
function average(numbers) {
  let len = numbers.length, sum = 0;
  if (len === 0) return 0;
  for (let i = 0; i < len; i++)
    sum += numbers[i];
  return sum / len;
}
```

Этот код можно переписать так, чтобы он имитировал асинхронное выполнение, разбив его на некоторые участки:

```
function averageAsync(numbers, callback) {
  let len = numbers.length,
    sum = 0;
  if (len === 0) {
    return 0;
  }
  function calculateSumAsync(i) {
    if (i < len) {
      setTimeout(function() {
        sum += numbers[i];
        calculateSumAsync(i + 1);
      }, 0);
    } else {
      callback(sum / len);
    }
  }
  calculateSumAsync(0);
}
```

В данном примере для планирования вычислений используется функция setTimeout. Это приводит к тому, что в цикле событий размещаются функции, выполняющие лишь часть общих вычислений, а между выполнением этих функций браузер может выполнять другие вычисления, в том числе и связанные с пользовательским интерфейсом

Первые операции представленной функции averageAsync ничем не отличаются от последовательной average. Происходит инициализация начальных переменных – длины и суммы, а также проверяется наличие элементов в массиве. Далее происходит объявление вспомогательной функции calculateSumAsync, которая в качестве аргумента получает индекс элемента, если индекс находится в пределах длины массива, мы выполняем вычисления и последующий рекурсивный вызов внутри анонимной функции, которая за счёт использования setTimeout выносится в цикл событий. Второй аргумент setTimeout – это время, на которое откладывается выполнение блока кода, выступающего в виде первого аргумента. Именно так и происходит распределение вычислений в основном потоке.

Говоря об асинхронных вычислениях в JavaScript, также следует упомянуть об использовании Promise [5], представляющих собой «обёртки» для функций обратного вызова (callback). Их можно использовать для упорядочивания синхронных и асинхронных действий. Например, используя данную возможность, можно переписать фрагмент кода с очень большой вложенностью setTimeout к более читаемому и удобному виду.

Однако использование Promise является лишь вариацией использования уже рассмотренной асинхронности. Поэтому, проанализировав имеющиеся возможности, можно установить, что веб-воркеры как клиентская технология уникальны в своем роде и предоставляет совершенно новые возможности. Поэтому любые тяжёлые операции, которые по какой-то причине не могут быть выполнены на сервере, необходимо однозначно выполнять с помощью воркеров.

Основные особенности WebWorkers API. Веб-воркеры – это потоки, принадлежащие браузеру, которые можно использовать для выполнения JS-кода без блокировки цикла событий [6]. Веб-воркеры позволяют выполнять длительные в вычислительном плане задачи без блокировки потока пользовательского интерфейса. Существующий воркер может отсылать сообщения JavaScript коду-создателю через обработчик событий, указанный этим кодом (и наоборот).

Существует три типа веб-воркеров:

- выделенные воркеры (dedicated workers);
- разделяемые воркеры (shared workers);
- сервис-воркеры (service workers).

Экземпляры выделенных веб-воркеров создаются главным процессом. Обмениваться данными с ними может только он. В данной работе основное внимание будет уделяться именно этим воркерам. Базовое взаимодействие с выделенным воркером представлено в следующем примере, реализованном в двух файлах. Файл script.js является основным и имеет следующее содержимое:

```
let worker = new Worker('webworker.js');
function test() {
   console.log('run test func');
   let data = 100;
   worker.postMessage({'key': data});
}
worker.addEventListener('message', function(event) {
   console.log(event.data);
}, false);
```

Второй файл webworker.js отвечает за вычисления на стороне воркера – именно этот код будет выполняться параллельно:

```
function print(n) { console.log(n) }
self.addEventListener('message', function(e) {
   let num = e.data.key;
   self.postMessage(print(num));
}, false);
```

Здесь определяется, как именно будут обрабатываться поступающие события. На стороне script.js отправляемые данные помещаются в объект по ключу key, соответственно по этому ключу они и будут обрабатываться в коде воркера.

Таким образом, использование данной технологии подразумевает под собой использование уже встроенного в браузер API, которое способно производить определенные операции на фоне основного потока. Схема стандартного сценария обмена данными с воркером представлена на рисунке 2.

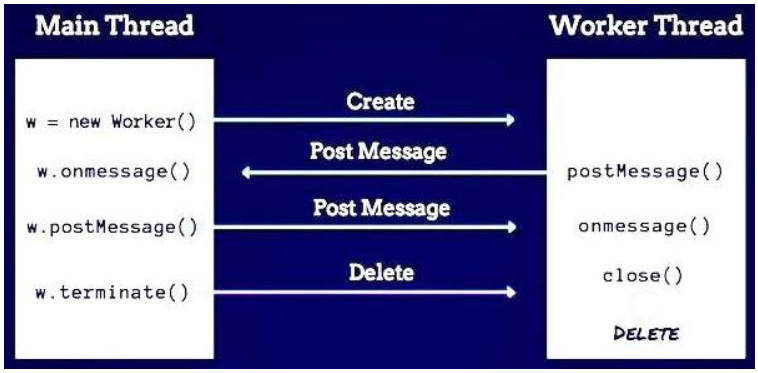


Рисунок 2 – Основной сценарий обмена данными с WebWorkers

Однако воркеры имеют свои ограничения – внутри воркер-скрипта доступен следующий набор полей [7]:

- объект navigator;
- объект location (только чтение);
- XMLHttpRequest;

- setTimeout()/clearTimeout() и setInterval()/clearInterval();
- кэш приложений;
- импорт внешних скриптов с использованием метода importScripts();
- создание других объектов Web Worker.

Но в свою очередь доступа у воркер-скрипта нет к следующим элементам:

- модель DOM (она не ориентирована на многопоточное исполнение);
- объект window;
- объект document;
- объект parent.

Пример использования нескольких воркеров для распараллеливания задачи обработки множества изображений. Воркеры позволяют продолжать исполнение логики основной страницы после запуска фоновых вычислений, таким образом приложение остается доступным для работы с другими функциями. При этом использование воркеров может позволить не только избежать полной занятости главного потока, но и значительно ускорить общий ход вычислений. Для этого необходимо использовать несколько воркеров в зависимости от размерности входных данных и сложности в их транспортировке.

Для того чтобы поработать сразу с несколькими воркерами, рассмотрим ситуацию обработки изображений с помощью JavaScript и применения алгоритма размытия, который работает по следующему принципу: рассматривается несколько пикселей вокруг каждого просматриваемого, а затем вычисляется и записывается среднее арифметическое каждой цветовой компоненты. То есть для каждого нового пикселя рассматриваются 4 пикселя в исходном изображении, чтобы смешать компоненты с нужными коэффициентами и получить сглаженный результат. Код данного алгоритма:

```
function blur(width, height, srcPixels, dstPixels) {
  let size = 5; let dstIdx = 0;
  for (let y = 0; y < height; y++) {
    for (let x = 0; x < width; x++) {
      let a = 0, r = 0, g = 0, b = 0, count = 0;
      for (let sy = y - size; sy <= y + size; sy++) {
        const yy = Math.min(height - 1, Math.max(0, sy));
        for (let sx = x - size; sx <= x + size; sx++) {
          const xx = Math.min(width - 1, Math.max(0, sx));
          let pix = srcPixels[yy * width + xx];
          r += pix & 0xff;
          g += (pix >> 8) & 0xff;
          b += (pix >> 16) & 0xff;
          a += (pix >> 24) & 0xff;
          count++;
        }
      }
      a = (a / count) & 0xff;
      r = (r / count) & 0xff;
      g = (g / count) & 0xff;
      b = (b / count) & 0xff;
      dstPixels[dstIdx++] = (a << 24) | (b << 16) | (g << 8) | r;
    }
  }
}
```

Данная функция получает на вход ширину и высоту изображения, а также ссылки на массивы пикселей в формате Uint32Array исходного изображения, которое необходимо преобразовать, и итогового изображения, в которое будет записан результат.

Стоит отметить, что в работе рассматривается подход обработки изображений с помощью нативного Canvas API, которое предоставляет средства для использования графики с помощью JavaScript и HTML-элемента <canvas> [8].

Тело самой страницы с рассматриваемым примером имеет следующий вид (там же можно встретить вышеупомянутый canvas):

```
<body>
  <h1>Without worker calculation</h1>
  <input onchange="onImageChange(event)" multiple="true" type="file" accept="image/png, image/jpeg">
  <br>
  <div class="images">
    <img alt="Initial image...">
    <canvas></canvas>
  </div>
  <script src="./script.js"></script>
</body>
```

Пользователь с помощью элемента input может отправлять изображения на клиент, а уже дальше с ними можно производить любые действия [9]. Если мы осуществим передачу изображения из файловой системы, то результат, согласно рассматриваемому сценарию, будет получен в виде двух изображений – исходного и обработанного (рис. 3).

Конечно же, сценарий, помимо алгоритма обработки изображения, предполагает ещё и различную служебную логику: работа с Canvas API, подсчёт времени выполнения вычислений с помощью performance() [10], предобработку данных и всевозможные выводы. В рамках рассмотрения этого примера заостряться внимание на этом не будет, поскольку здесь нас главным образом интересуют накладные вычисления.

Рисунок 3 – Полученный результат

Если речь идёт об обработке одного изображения, время исполнения кода приемлемо. Но, если представить себе более сложный алгоритм или увеличение размера и количества пользовательских изображений, возникают сложности, которые связаны, как минимум, с продолжением работы основной страницы. Для того чтобы детальнее это отследить, помимо подсчета времени исполнения всех вычислений на странице после старта обработки также запускается таймер, содержимое которого будет инкрементироваться каждую секунду. Выглядит это следующим образом:

```
tickCounter = 0;
t = setInterval(() => { tickCounter++ }, 1000);
```

Поскольку основной и единственный поток страницы будет занят вычислениями, инкремент счетчика таймера выполнен не будет. Чтобы убедиться в этом, а также удостовериться в эффективности использования WebWorkers, распараллелим задачу и посчитаем ускорение.

Алгоритм обработки изображения, а также вся служебная логика по обслуживанию воркера рассмотренная ранее, будут находиться в файле webworker.js, а файл script.js будет содержать в себе логику получения результата, а также обслуживания нескольких воркеров сразу:

```
let tickCounter = 0, t = null;
let startTime = 0, filesLength = 0, commonFileCounter = 0;
let workerCount = 4, workers = [];
for (let i = 0; i < workerCount; i++) {
  workers[i] = new Worker("webworker.js");
  workers[i].addEventListener("message", (e) => {
```

```
      const { blured, width, height } = e.data;
      console.log(i, " - worker");
      showResult(blured, width, height);
      commonFileCounter++;
      if (commonFileCounter === filesLength) {
        let endTime = performance.now();
        clearInterval(t);
        console.log("Time execution in milliseconds: ", (endTime - startTime).toFixed(2));
        console.log(tickCounter);
      }
    }, false);
}
```

Здесь происходит инициализация счётчика, который будет инкрементироваться каждую секунду при старте обработки изображения, таймера, точки начала отсчёта вычислений, количества файлов для распределения данных между потоками, счётчика обработанных файлов, числа создаваемых воркеров, а также структуры, в которой воркеры будут расположены. Далее в цикле воркеры инициализируются на основе файла webworker.js. При этом обработчик изображений, в котором начинаются все вычисления после загрузки картинок со стороны пользователя, принимает следующий вид:

```
function onImageChange(e) {
  startTime = performance.now();
  tickCounter = 0
  t = setInterval(() => { tickCounter++ }, 1000);
  filesLength = e.target.files.length;
  let files = [...e.target.files], i = 0;
  while (files.length) {
    const file = files.pop();
    processImage(file, i); i++;
    if (i === workerCount)
      i = 0;
  }
}
```

Здесь фиксируется время начала обработки всех изображений, осуществляется рассылка данных основным потоком. Рассылка производится последовательно по каждому воркеру, пока все файлы не будут отданы на обработку. Сам процесс отправки и подготовки находится в функции processImage, которая получает файл и индекс воркера, который потом используется внутри функции для отправки данных воркеру:

```
workers[workerIdx].postMessage({ name:filename, srcPixels:srcPixels, dstPixels: dstPixels, width: width, height: height });
```

Со своей стороны каждый воркер получает данные, выполняет их обработку, а затем отправляет обратно:

```
self.addEventListener("message", function (e) {
    console.log("Worker recieved file with name: ", e.data.name);
    const { width, height, srcPixels, dstPixels } = e.data;
    const blured = blur(width, height, srcPixels, dstPixels); // Uint32Array
    self.postMessage({ blured: blured, width: width, height: height });
  }, false);
```

Анализ результатов распараллеливания задачи обработки множества изображений. Запуск решения производился в Google Chrome Version 86.0.4240.198 (Official Build) (64-bit) на основе 4-ядерного процессора Intel® Core™ i5-7400 Processor, то есть в рамках задачи более 4-х потоков не рассматривалось.

В первую очередь были получены результаты выполнения кода для версии без использования WebWorkers API. Производилась обработка 10, 50, 100, 250 и 500 картинок. Затем эти же порции данных были отданы на выполнение уже в многопоточных вариантах. Помимо фиксирования времени

исполнения производился также подсчёт значения счетчика tickCounter, что позволяет сделать дополнительные выводы в пользу использования технологии. Ниже приведены таблицы и построенные по ним графики. Ниже представлены таблицы (табл. 1–5) и графики (рис. 4–5) на их основе.

Таблица 1 – Количество потоков (воркеров): 0, только основной поток

Количество картинок	Значение tickCounter	Время выполнения, мс (с)	Ускорение
10	2	4770,45 (4.8)	1
50	5	24471,69 (24)	1
100	6	56396,71 (56)	1
250	7	147514,02 (147)	1
500	10	275918,03 (276)	1

Таблица 2 – Количество потоков (воркеров): основной поток и 1 воркер

Количество картинок	Значение tickCounter	Время выполнения, мс (с)	Ускорение
10	4	4594,81 (4.6)	1,04
50	23	23506,81 (23)	1,04
100	51	53458,70 (53)	1,05
250	134	142626,70 (143)	1,03
500	240	254093,43 (254)	1,09

Таблица 3 – Количество потоков (воркеров): основной поток и 2

Количество картинок	Значение tickCounter	Время выполнения, мс (с)	Ускорение
10	2	2649,73 (2,7)	1,80
50	12	12763,82 (13)	1,92
100	26	28445,50 (28)	1,98
250	67	75792,44 (76)	1,95
500	122	139010,21 (139)	1,98

Таблица 4 – Количество потоков (воркеров): основной поток и 3

Количество картинок	Значение tickCounter	Время выполнения, мс (с)	Ускорение
10	2	2287,55 (2,3)	2,09
50	10	10400,57 (10)	2,35
100	20	22356,36 (22)	2,52
250	47	56703,73 (56)	2,60
500	83	102369,96 (102)	2,70

Таблица 5 – Количество потоков (воркеров): основной поток и 4

Количество картинок	Значение tickCounter	Время выполнения, мс (с)	Ускорение
10	2	2281,85 (2,3)	2,09
50	8	8219,14 (8,2)	2,98
100	15	17922,09 (18)	3,15
250	36	50886,75 (51)	2,90
500	65	90069,58 (90)	3,06

Полученнные результаты позволяют сделать вывод о том, что целесообразность использования нескольких воркеров возрастает с ростом объема входных данных. Уже на этапе с 50 картинками варианты с количеством воркеров больше одного позволяют получить прирост в скорости выполнения обработки изображений.

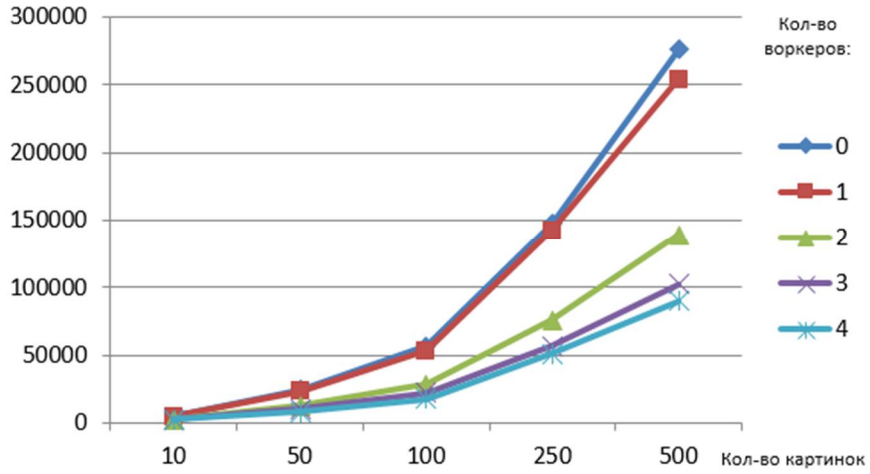


Рисунок 4 – Время выполнения вычислений (мс)

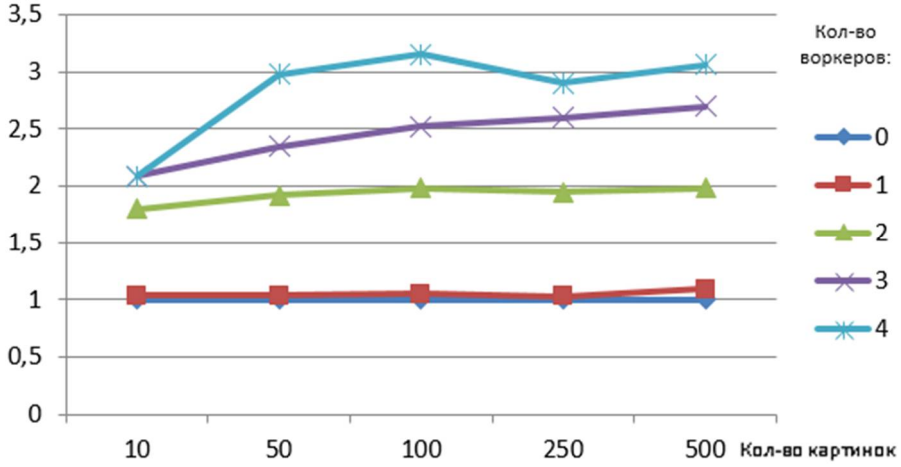


Рисунок 5 – Полученное ускорение

В случае вариантов, где вычисления выполняет либо только основной поток, либо основной поток и 1 воркер результаты практически идентичны – это связано с тем, что в обоих случаях сами вычисления производились только в одном потоке. Вариант с основным потоком и воркером при этом всё же получает небольшое ускорение, так как основной поток мог взять на себя всю работу с интерактивностью страницы, в том числе и ее отрисовку, в то время как алгоритмические расчеты выполнял воркер. Стандартное выполнение одним основным потоком оказывалось менее эффективным, поскольку вся работа в таком случае ложилась только на него. В итоге, пока происходили основные расчеты, пользователь терял доступ к приложению. Таким образом, только базируясь на результатах данного эксперимента, можно сделать вывод о том, что комбинация из одного воркера и основного потока уже приносит заметную оптимизацию даже в случае нагруженных сценариев с большим количеством входных данных. Ведь блокировки пользовательского интейрфейса в ходе вычислений не будет за счет вынесения их в отдельный поток. Это и было подробно рассмотрено на базовых примерах, которые наглядно показывают, что даже если решиние не удается разбить на несколько порций данных и разослать их независимо нскольким воркерам, существенная выгода всё равно есть. Она заключается в том, что основной поток не блокируется и пользователь может продолжать взаимодействовать с приложением, его интерактивность не теряется.

Также на примере кода с многопоточной обработкой картинок мы видим, что с помощью небольшого количества строк кода мы получаем понятное и масштабируемое решение. Инициализацией воркеров мы занимаемся самостоятельно, то есть их количество можно либо задать напрямую, либо вычислить на основе каких-то параметров, либо динамически менять в зависимости от специфики задачи. Более того, помимо масштабирования мы также получаем кроссплатформенность и кроссбраузерность технологии, в чем позволяют убедиться данные с ресурсов такого рода, как caniuse.com. Ознакомиться с данными о поддержке различных веб-браузеров можно на рисунке 6.

Рисунок 6 – Поддержка WebWorkers API различными браузерами

Данные цифры говорят о том, что технология не доступна для старых версий браузеров, поддержка которых официально уже прекращена. Таким образом, можно гарантировать безотказную работу решений на базе WebWorkers API на любой современной версии любого браузера.

Вышеупомянутое значение счетчика tickCounter тоже позволяет сделать определенные выводы. Инкремент этого значения производится основным скриптом каждую секунду, при этом известно общее время выполнения обработки того или иного количества картинок. В идеальном случае значение таймера должно быть равно времени исполнения в секундах – это будет означать, что ни одна из операций инкремента не была потеряна. Поскольку время выполнения сценариев из таблиц 1 и 2 практически идентичны – 276 секунд и 254 секунды (ускорение 1,09), сравним значения счетчиков именно для этих случаев. Для 276 секунд счетчик tickCounter вывел значение 10, для 254 секунд значение было равно 240. В случае работы без воркера грубый процент потерь полезных операций, которые мы могли бы совершить, составил примерно 96,4 %, для варианта с воркером – 5,5 %.

Если же мы будем анализировать другие варианты, где воркеров больше одного, результаты будут немного хуже, поскольку основному потоку приходилось также администрировать ресурсы и заниматься распределением данных, но так или иначе это окупается получаемым ускорением. Поэтому в случае с некоей реальной задачей, исходя из ее масштабов и аппаратных возможностей компьютера, необходимо самостоятельно решить, какие именно вычисления будут вынесены в отдельный поток, сколько потоков будет выделяться.

Выделяя большое количество потоков, мы не можем гарантировать, что задача обязательно ускорится, ведь как уже неоднократно было замечено, основной поток занимается администрированием ресурсов и транспортировкой данных. Также в конечном итоге ему будет необходимо разбирать очередь сообщений полученных в обратном направлении от воркеров. Взаимодействие основного потока с воркерами в общем виде схематично представлено на рисунке 7.

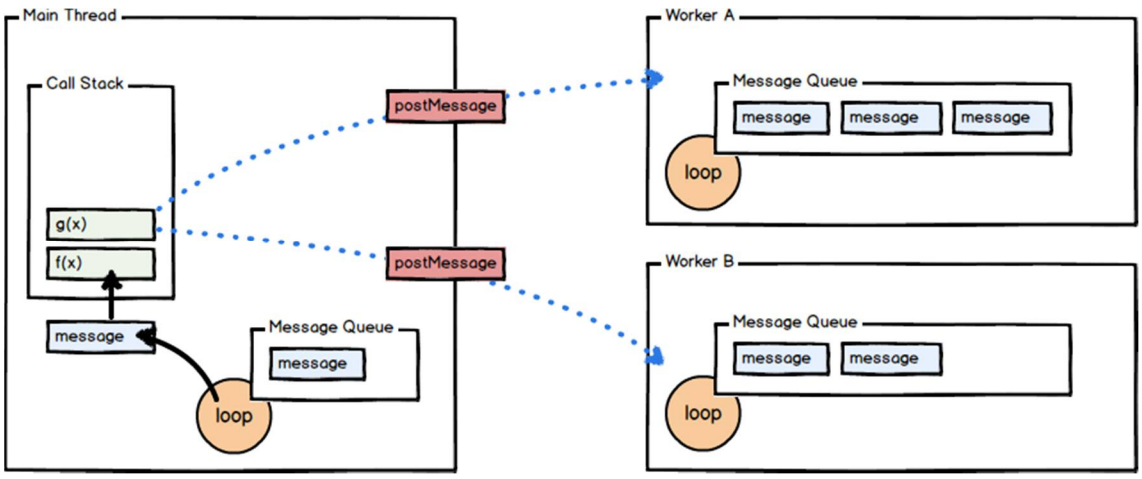


Рисунок 7 – Взаимодействие основного потока с выделяемыми воркерами

Теперь на основе полученных и проанализированных результатов следует выделить ряд ограничений, с которыми можно столкнуться в ходе распараллеливания данной задачи, а также рассмотрим достоинства применения технологии (табл. 6).

Таблица 6 – Достоинства и недостатки применения технологии

Достоинства	Недостатки
Поток пользовательского интерфейса свободен и может выполнять отрисовку и служебные задачи вовремя. Это, безусловно, повышает отзывчивость приложения и не вынуждает пользователя покидать страницу с мыслью о том, что процесс полностью «завис»	Накладные расходы на пересылку и сериализацию данных, невозможность передачи данных воркеру по ссылке, так как речь идёт о разных контекстах выполнения
Скорость выполнения вычислений увеличивается, но только для тех случаев, которые можно выполнять параллельно	Соответственно из первого пункта следует, что некоторые алгоритмы могут потребовать некоторого видоизменения для учёта всех тонкостей
Несмотря на некоторые ограничения, технология имеет хорошую поддержку и удобный интерфейс взаимодействия, кроссбраузерность и кроссплатформенность, не нуждается в настройке и установке	Также из первого пункта следует, что некоторые данные могут потребовать конвертацию из-за потери контекста и отсутствия доступа к некоторым нужным API

Выделим основные сценарии использования воркеров. На данный момент на основе полученных результатов, а также возможностей и ограничений технологии можно выделить несколько вариантов использования:

- работа с файлами;
- шифрование;
- предварительная загрузка данных;
- проверка правописания;
- рендеринг трёхмерных сцен.

Рассмотрим каждый немного подробнее.

Работа с файлами. На основе рассмотренного примера по работе с картинками можно сделать вывод, что при работе с другими типами файлов воркеры также могут быть актуальны. При всем бурном развитии web в целом и стандартов html в частности, работа с файлами практически никогда не менялась. Следует отметить, что с приходом HTML5 и связанных с ним API появилось гораздо больше возможностей для работы с файлами, чем когда-либо в предыдущих версиях браузеров.

Для WebWorker, например, доступен FileReaderSync – синхронный вариант FileReader. Его методы считывания read* не генерируют события, а возвращают результат, как это делают обычные функции. Используется это внутри веб-воркера, поскольку задержки в синхронных вызовах, которые возможны при чтении из файла, в веб-воркерах менее важны, они не влияют на работу страницы.

Можно предположить, что если в приложении планируется активная конвертация файлов на клиенте, например, в условиях медленного соединения или высокой серверной загрузки, веб-воркеры помогут взять всю сложную обработку на себя. Также в случае отправки большого файла на сервер по частям воркеры смогут помочь в обработке нескольких участков данных, не влияя на основную работу страницы.

Шифрование. Сквозное шифрование в сети становится актуальным из-за возрастающих требований к защите пользовательских приватных данных. Операции шифрования и дешифрования могут быть очень ресурсозатратными, особенно если сценарии использования приложения требуют частого обращения к большим объёмам данных. При этом для реализации данных операций достаточно базовых возможностей JS, доступ к объектам DOM в данном случае не требуется. Соответственно, процесс выполнения шифрования воркером не оказывает влияния на работу пользовательского интерфейса сайта.

Предварительная загрузка данных. Для уменьшения времени отклика приложения можно осуществлять предварительную загрузку и сохранение данных с использованием веб-воркеров. Эти данные можно использовать позднее, когда они понадобятся для вычислений. Если предварительную загрузку данных осуществлять средствами главного потока, это явно окажет негативное влияние на интерфейс приложения. В случае использования веб-воркеров подобного эффекта наблюдаться не будет.

Проверка правописания. Простые системы проверки правописания основаны на алгоритмах работы с файлом словаря, который содержит список правильно написанных слов. Данный файл загружается, на базе словаря формируется дерево поиска. Каждое слово, переданное для проверки, ищется в полученном дереве поиска. Если слово найти не удаётся, пользователю могут быть предоставлены альтернативные варианты его написания, которые получаются путём замены символов исходного слова. Полученные альтернативные варианты ищутся в дереве, система пытается оценить их корректность. Данные действия можно выполнять в веб-воркере. Это даст пользователю возможность работать с текстом без блокировки интерфейса при проверке слова и поиске альтернативных вариантов его написания.

Рендеринг трёхмерных сцен. Здесь речь, в частности, идёт о реализации метода трассировки лучей. Данный метод представляет собой технику рендеринга, основанной на отслеживании направления лучей света. Подобные алгоритмы используют интенсивные математические вычисления. Используя подобную технику, можно реализовать различные эффекты, например, отражение и преломление, можно добиться имитации внешнего вида различных материалов, и так далее. Описанные вычисления также могут быть вынесены в веб-воркер, чтобы избежать блокировки пользовательского интерфейса. Можно даже разделить процесс рендеринга изображения между несколькими воркерами (и, соответственно, между несколькими процессорными ядрами).

Развитие технологии. Как известно, open-source среда всегда будет предлагать какое-то решение для востребованной задачи. Так и в случае воркеров появилось несколько библиотек, которые базируются на данной технологии. В качестве примера можно привести библиотеки Parallel.js и Multithread.js. Обе эти библиотеки не представляют из себя нечто альтернативное, они содержат в себе готовые реализованные операции воркеров, доступные через свои интерфейсы, которые инкапсулируют работу по пересылке данных.

Parallel.js – простая библиотека для многоядерной обработки в Javascript. Она была разработана, чтобы в полной мере использовать все преимущества постоянно развивающегося API воркеров. Javascript, без сомнения, быстр, но ему не хватает возможностей параллельных вычислений, как у языков-аналогов из-за своей модели однопоточных вычислений. Parallel.js решает эту проблему, предоставляя высокоуровневый доступ к многоядерной обработке с использованием веб-воркеров. Ниже представлен небольшой пример с использованием Parallel.js в рекурсивном расчете чисел Фибоначчи.

```
let p = new Parallel([0, 1, 2, 3, 4, 5, 6]),
log = function () { console.log(arguments) };
function fib(n) {return n < 2 ? 1 : fib(n - 1) + fib(n - 2) };
p.map(fib).then(log)
```

Другая упомянутая библиотека Multithread.js также представляет собой простую оболочку, которая избавляет от необходимости выполнения операций, напрямую связанных с веб-воркерами и передаваемыми объектами. Библиотека гарантирует, что любой код со сложными вычислениями с ее помощью можно запустить асинхронно в отдельном потоке, не прерывая взаимодействия с пользователем. Пример работы с библиотекой представлен ниже

```
let num_threads = 2;
let MT = new Multithread(num_threads);
```

Здесь происходит инициализация библиотеки с явным указанием числа потоков, а затем запуск самих вычислений в формате: MT.process(function, callback)(arg1, arg2, ..., argN);

```
MT.process(
  function Recurse(m, n) {
    if(n>0) {
      return Recurse(m + 1, n--);
    } else {
      return m;
    }
  },
  function(r) {
    console.log(r);
  }
)(5, 2);
```

Multithread.js использует сериализацию JSON с помощью process(), что позволяет работать с многопоточными асинхронными функциями так же, как с обычными функциями JavaScript. Предоставляется также оптимизированная поддержка для типизированных данных, в частности Int32 и Float64 (32-битные целые числа со знаком и 64-битные числа с плавающей запятой соответственно).

Используя эти библиотеки, стоит помнить об ограничениях многопоточности в JavaScript. Все переменные, передаваемые функциям, должны быть JSON-сериализуемыми, то есть только массивы, объекты и базовые типы (Number, String, Boolean, null). Объекты и массивы, передаваемые в любую

поточную функцию, будут глубоко скопированы (переданы по значению, а не по ссылке). Кроме того, многопоточные функции не имеют доступа к DOM.

Заключение. В данной работе были рассмотрены веб-воркеры - сравнительно новая возможность, доступная веб-разработчикам в большинстве современных браузеров. Веб-воркеры позволяют выносить в отдельные потоки выполнение ресурсоёмких операций, что позволяет не нагружать главный поток, который может продолжать полноценно взаимодействовать с пользовательским интерфейсом. Были описаны конкретные случаи использования, подробно рассмотрена задача распараллеливания обработки множества изображений с помощью алгоритма размытия. На основе полученных результатов было посчитано ускорение для многопоточных случаев, построены графики. Также были упомянуты другие варианты использования технологии.

Библиографический список

1. What Is a Front-End Developer? – Frontend Masters // Front-End Developer Handbook 2018. – Режим доступа: https://frontendmasters.com/books/front-end-handbook/2018/what-is-a-FD.html/ (дата обращения: 10.11.2020).

2. Stack Overflow Developer Survey 2019 – Stack Overflow Insights // Stack Overflow. – Режим доступа: https://insights.stackoverflow.com/survey/2019 (дата обращения: 10.11.2020).

3. The Cost of Javascript Frameworks – Web Performance Consulting // TimKadlec. – Режим доступа: https://timkadlec.com/remembers/2020-04-21-the-cost-of-javascript-frameworks/ (дата обращения: 10.11.2020).

4. JavaScript Asynchronous // W3Schools. – Режим доступа: https://www.w3schools.com/js/js_asynchronous.asp (дата обращения: 12.11.2020).

5. Promise // JavaScript.info. – Режим доступа: https://javascript.info/promise-basics (дата обращения: 12.11.2020).

6. Background processing using web workers // Angular. – Режим доступа: https://angular.io/guide/web-worker (дата обращения: 08.11.2020).

7. Using Web Workers – Web APIs // MDN. – Режим доступа: developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers (дата обращения: 08.11.2020).

8. Canvas tutorial – Web APIs // MDN. – Режим доступа: https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API/Tutorial (дата обращения: 11.11.2020).

9. HTML5 – Developer guides // MDN. – Режим доступа: https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/HTML5 (дата обращения: 11.11.2020).

10. JavaScript performance – Learn web development // MDN. – Режим доступа: https://developer.mozilla.org/en-US/docs/Web/API/Performance (дата обращения: 12.11.2020).

References

1. What Is a Front-End Developer? – Frontend Masters. *Front-End Developer Handbook 2018.* Available at: https://frontendmasters.com/books/front-end-handbook/2018/what-is-a-FD.html/ (accessed 10.11.2020).

2. Stack Overflow Developer Survey 2019 – Stack Overflow Insights. *Stack Overflow.* Availablke at: https://insights.stackoverflow.com/survey/2019 (accessed 10.11.2020).

3. The Cost of Javascript Frameworks – Web Performance Consulting. *TimKadlec.* Available at: https://timkadlec.com/remembers/2020-04-21-the-cost-of-javascript-frameworks/ (accessed 10.11.2020).

4. JavaScript Asynchronous. *W3Schools.* Available at: https://www.w3schools.com/js/js_asynchronous.asp (accessed 12.11.2020).

5. Promise. *JavaScript.info.* Available at: https://javascript.info/promise-basics (accessed 12.11.2020).

6. Background processing using web workers. *Angular.* Available at: https://angular.io/guide/web-worker (accessed 08.11.2020).

7. Using Web Workers – Web APIs. *MDN.* Available at: developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers (accessed 08.11.2020).

8. Canvas tutorial – Web APIs. *MDN.* Available at: https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API/Tutorial (accessed 11.11.2020).

9. HTML5 – Developer guides. *MDN.* Available at: https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/HTML5 (accessed 11.11.2020).

10. JavaScript performance – Learn web development. *MDN.* Available at: https://developer.mozilla.org/en-US/docs/Web/API/Performance (accessed 12.11.2020).