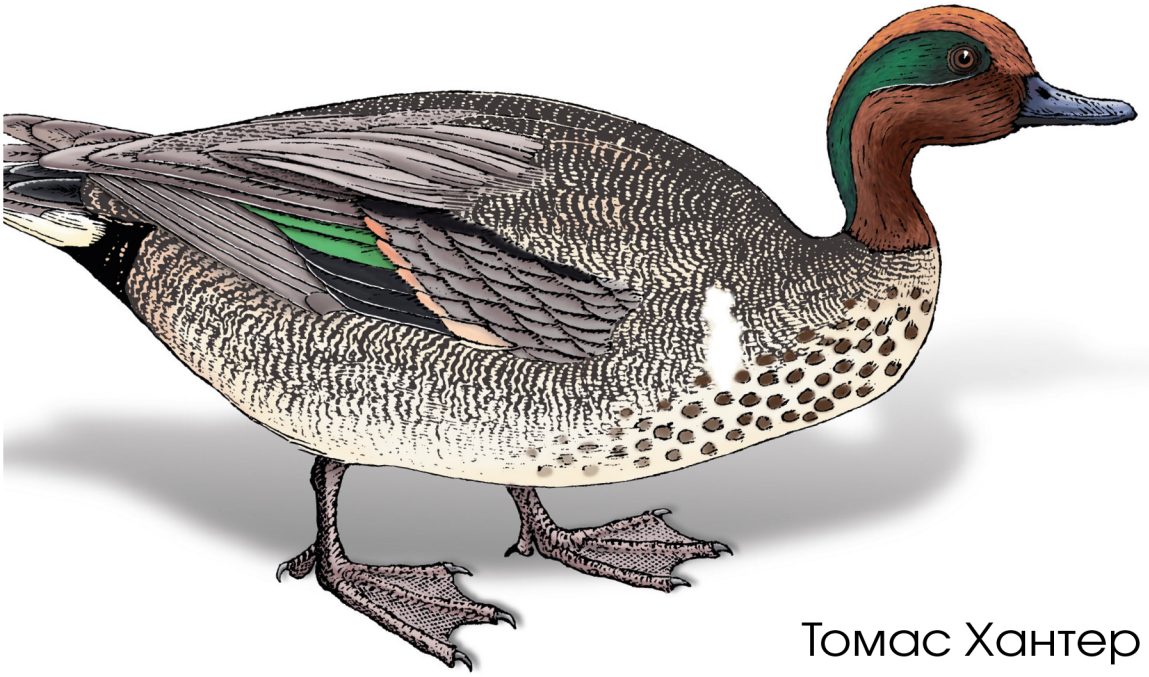
O'REILLY

Многопоточный JavaScript

Конкурентность за пределами цикла событий



Томас Хантер II
Брайан Инглиш

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО

Томас Хантер II, Брайан Инглиш

Многопоточный JavaScript

Multithreaded JavaScript

Concurrency Beyond the Event Loop

Thomas Hunter II and Bryan English

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Многопоточный JavaScript

Конкурентность за пределами цикла событий

Томас Хантер II, Брайан Инглиш

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО
Москва, 2022

УДК 004.42
ББК **32.972**
Х19

Хантер II Т., Инглиш Б.
Х19 Многопоточный JavaScript / пер. с англ. А. А. Слинкина. – М.: ДМК Пресс, 2022. – 188 с.: ил.

ISBN 978-5-93700-129-0

Цель данной книги – научить читателя нескольким аспектам написания многопоточных JavaScript-приложений. Прочитав книгу до конца, вы будете понимать различные API веб-исполнителей в браузерах, их сильные и слабые стороны и когда какой использовать. Также узнаете о модуле рабочих потоков в Node.js и сможете сравнить его API с тем, что имеется в браузере.

Издание предназначено в первую очередь разработчикам, уже знакомым с JavaScript, но мало знакомым с многопоточным программированием.

УДК 004.42
ББК 32.972

ISBN 978-1-098-10443-6 (англ.)

ISBN 978-5-93700-129-0 (рус.)

Посвящается Кэтлин и Ренье

Содержание

От издательства

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге, – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв на нашем сайте www.dmkpress.com, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com; при этом укажите название книги в теме письма.

Если вы являетесь экспертом в какой-либо области и заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг, мы будем очень благодарны, если вы сообщите о ней главному редактору по адресу dmkpress@gmail.com. Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и O'Reilly очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

Ссылку на подозрительные материалы можно прислать по адресу электронной почты dmkpress@gmail.com.

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

Вступительное слово

Книга, которую вы держите в руках, весьма любопытна. Это книга по JavaScript, открывающаяся примером, написанным на C, а речь в ней пойдет о многопоточности в языке, который явным образом объявлен однопоточным. В ней приведены интереснейшие примеры того, как и когда следует намеренно блокировать цикл событий, хотя эксперты на протяжении многих лет убеждали вас никогда так не поступать. А заканчивается она прекрасным списком причин, по которым не стоит использовать описанные механизмы, и подводных камней, подстерегающих на этом пути. Но я считаю, что эту книгу должен прочитать любой JavaScript-разработчик вне зависимости от того, где он предполагает развертывать и исполнять свой код.

Помогая компаниям создавать более эффективные и производительные приложения на JavaScript для Node.js, я часто должен был делать паузу и обсуждать с разработчиками распространенные заблуждения об этом языке программирования. Например, однажды мне встретился программист, имеющий большой опыт разработки на Java и .NET, который доказывал, что создание нового обещания в JavaScript очень напоминает создание потока в Java (это не так) и что обещания позволяют JavaScript выполнять код параллельно (не позволяют). В другой раз человек создал приложение для Node.js, которое запускало больше тысячи одновременных рабочих потоков, и никак не мог понять, почему не наблюдает ожидаемого повышения производительности при тестировании на машине, оснащенной всего восьмью процессорными ядрами. Урок понятен: многопоточность, конкурентность и параллелизм – все еще мало знакомые и трудные темы для очень большого процента JavaScript-разработчиков.

Борьба с этими заблуждениями и заставила меня (вместе с коллегой и членом технического руководящего комитета Node.js Маттео Коллина) создать семинар Broken Promises, посвященный основам асинхронного программирования на JavaScript, – мы учили команды разработчиков, как правильно рассуждать о порядке выполнения кода и хронометраже различных событий. Она же побудила меня заняться проектом с открытым исходным кодом Piscina (вместе с соразработчиком ядра Node.js Анной Хеннингсен), который предлагает учитывающую передовые практики реализацию модели пула потоков поверх рабочих потоков в Node.js. Но все это помогает решить лишь часть проблемы.

В этой книге Брайан и Томас квалифицированно описывают основы многопоточной разработки вообще и искусно иллюстрируют, как различные среды выполнения JavaScript, а именно браузеры и Node.js, допускают параллельные вычисления на языке, который не содержит никаких встроенных механизмов для этого. Поскольку ответственность за поддержку многопоточности возложена на среды выполнения, а между средами есть масса различий, браузеры и платформы типа Node.js реализуют многопоточность по-разному. И, хотя API похожи, рабочий поток в Node.js на деле совсем не

то, что веб-исполнитель в браузере. Поддержка разделяемых исполнителей, веб-исполнителей и сервисных исполнителей стала почти универсальной во всех браузерах, а рабочие потоки в Node.js существуют уже несколько лет, но все равно для JavaScript-разработчиков это относительно новые концепции. Но, где бы ни исполнялся ваш код, эта книга станет для вас источником прозрения и важной информации. Но самое главное – авторы точно объясняют, почему вообще следует «заморачиваться» многопоточностью в JavaScript-приложениях.

— Джеймс Снелл,
член технического руководящего комитета Node.js

Предисловие

Брайан и я (Томас) впервые встретились в Сан-Франциско на моем собеседовании при приеме на работу в филиал японской компании по разработке мобильных игр DeNA. Казалось, что большая часть руководства была готова отказать, но, после того как вечером в тот же день мы вдвоем заявились на неформальную встречу сообщества Node.js, Брайан убедил их сделать мне предложение.

Работая в DeNA, мы с Брайаном занимались написанием повторно используемых модулей для Node.js, чтобы группы разработчиков могли создавать игровые серверы из готовых компонентов, отвечающих целям игры. Мы всегда измеряли производительность, а обучение команд методам производительного программирования было частью нашей работы; наши серверы были объектами постоянного и пристального внимания разработчиков из индустрии, которая традиционно опиралась на C++.

Нам довелось работать вместе и в других местах. Одним из них стал небольшой стартап в области безопасности под названием Intrinsic, где в нашу задачу входило укрепление приложений для Node.js на таком всеобъемлющем и мелкоструктурном уровне, что я сомневаюсь, найдется ли в мире еще один подобный продукт. Оптимизация производительности также стояла там на одном из первых мест, поскольку заказчики не хотели терять ни грана пропускной способности. Мы потратили много часов на прогон тестов производительности, корпение над пламенными диаграммами и копание в потрохах Node.js. Если бы модуль рабочих потоков был доступен во всех версиях Node.js, поддержки которых требовал заказчик, то мы, безусловно, включили бы его в продукт.

Мы работали вместе и не по найму. Один из таких примеров – NodeSchool SF (https://nodeschool.io/sanfrancisco/), где мы бесплатно учили других использовать JavaScript и писать программы для Node.js. Мы также не раз выступали на одних и тех же конференциях и встречах по интересам.

Оба автора питают страсть к JavaScript и Node.js, а также к преподаванию этих предметов и развеиванию заблуждений. Осознав, как остро не хватает документации по созданию многопоточных JavaScript-приложений, мы поняли, что нужно делать. Эта книга родилась из нашего желания не только рассказывать другим о возможностях JavaScript, но также доказать, что платформы типа Node.js ничуть не хуже любых других с точки зрения создания высокопроизводительных служб, в полной мере задействующих доступное оборудование.

Для кого написана эта книга

Идеальный читатель этой книги – инженер, у которого за плечами несколько лет работы с JavaScript, но который, возможно, никогда не писал многопоточных приложений даже на таких традиционно поддерживающих многопоточ-

ность языках, как C++ или Java. Мы включили пример прикладного кода на C, как «всеобщем» многопоточном языке, но не предполагаем, что читатель знаком с ним или хотя бы понимает написанный на нем код.

Если у вас есть опыт работы с такими языками, отлично – и тогда эта книга поможет понять предлагаемый JavaScript эквивалент конструкциям, знакомым по другим языкам. С другой стороны, если вы писали код только на JavaScript, то и тогда эта книга для вас. Мы включили информацию разного уровня: ссылки на низкоуровневые API, высокоуровневые паттерны и множество технических деталей, заполняющих пространство между тем и другим.

Цели

Пожалуй, самая важная цель этой книги – принести сообществу благую весть о том, что на JavaScript можно писать многопоточные приложения. Традиционно считалось, что JavaScript-код занимает только одно ядро, и действительно в Twitter и на разных форумах полно постов такого толка. Назвав книгу «Многопоточный JavaScript», мы надеемся полностью развенчать миф о том, будто JavaScript-приложения ограничены единственным ядром.

На более конкретном уровне цель книги – научить читателя нескольким аспектам написания многопоточных JavaScript-приложений. Прочитав книгу до конца, вы будете понимать различные API веб-исполнителей в браузерах, их сильные и слабые стороны и когда какой использовать. Что до Node.js, вы узнаете о модуле рабочих потоков и сможете сравнить его API с тем, что имеется в браузере.

В книге рассматриваются два подхода к построению многопоточных приложений: с использованием передачи сообщений и разделяемой памяти. Дочитав книгу, вы будете знать, какие API используются для реализации того и другого, когда отдать предпочтение одному подходу, а когда другому и в каких ситуациях их можно сочетать. У вас даже будете шанс потренироваться в создании некоторых высокоуровневых паттернов на основе этих подходов.

Графические выделения

В книге применяются следующие графические выделения.

Курсив
: Новые термины, URL-адреса, адреса электронной почты, имена и расширения имен файлов.

`Моноширинный`
: Листинги программ, а также элементы кода в основном тексте: имена переменных и функций, базы данных, типы данных, переменные среды, предложения и ключевые слова языка.

Моноширинный полужирный
: Команды и иные строки, которые следует вводить буквально.

Моноширинный курсив
: Текст, вместо которого следует подставить значения, заданные пользователем или определяемые контекстом.


Так обозначается совет или рекомендация.


Так обозначается замечание общего характера.


Так обозначается предупреждение или предостережение.

О ПРИМЕРАХ КОДА

Дополнительные материалы (примеры кода, упражнения и т. д.) можно скачать по адресу https://github.com/MultithreadedJSBook/code-samples.

Если у вас возникнет технический вопрос или затруднение в использовании примеров кода, можете отправить сообщение на адрес bookquestions@oreilly.com.

Эта книга призвана помогать вам в работе. Поэтому вы можете использовать приведенный в ней код в собственных программах и в документации. Спрашивать у нас разрешение необязательно, если только вы не собираетесь воспроизводить значительную часть кода. Например, никто не возбраняет включить в свою программу несколько фрагментов кода из книги. Однако для продажи или распространения примеров из книг издательства O'Reilly разрешение требуется. Цитировать книгу и примеры в ответах на вопросы можно без ограничений. Но для включения значительных объемов кода в документацию по собственному продукту нужно получить разрешение.

Мы высоко ценим, хотя и не требуем, ссылки на наши издания. В ссылке обычно указываются название книги, имя автора, издательство и ISBN, например: «Multithreaded JavaScript by Thomas Hunter II and Bryan English (O'Reilly). Copyright 2022 Thomas Hunter II and Bryan English, 978-1-098-10443-6».

Если вы полагаете, что планируемое использование кода выходит за рамки изложенной выше лицензии, пожалуйста, обратитесь к нам по адресу permissions@oreilly.com.

Благодарности

Эта книга стала возможной благодаря подробным техническим рецензиям, написанным следующими лицами.

Анна Хеннингсен (@addaleax)
: В настоящее время работает в команде инструментов разработки MongoDB в Германии. Анна была одним из самых активных соразработчиков ядра

Node.js на протяжении последних пяти лет и принимала участие в реализации рабочих потоков для этой платформы. Она одержима страстью к Node.js и его сообществу.

Шу-ю Гуо (@_shu)

Шу работает над реализацией и стандартизацией JavaScript. Он входит в комитет TC39, является одним из редакторов спецификации ECMAScript и автором модели памяти. В настоящее время работает над движком Google V8, главным источником реализации и стандартов языковых средств JavaScript. До того работал в Mozilla и агентстве Bloomberg.

Фернандо Ларраньяга (@xabadu)

Фернандо – инженер и соразработчик ПО с открытым исходным кодом. Несколько лет возглавлял сообщества JavaScript и Node.js в Южной Америке и в США. В настоящее время занимает должность старшего инженера-программиста в компании Square и является организатором неформальных встреч NodeSchool SF. А на предыдущих местах работы в других крупных технологических компаниях, например Twilio и Groupon, он разрабатывал приложения Node.js уровня предприятия и занимался масштабированием веб-приложений, используемых миллионами пользователей, начиная с 2014 года.

Об авторах

Томас Хантер II участвовал в разработке десятков сервисов Node.js и работал в компании, занимающейся обеспечением безопасности Node.js. Он выступал на нескольких конференциях по Node.js и JavaScript, имеет сертификат JSNSD/JSNAD и является организатором NodeSchool SF.

Брайан Инглиш разрабатывает проекты с открытым исходным кодом на JavaScript и Rust, занимался крупными корпоративными системами, оснащением инструментальными средствами и безопасностью на уровне приложений. В настоящее время работает старшим инженером на проектах с открытым исходным кодом в компании Datadog. Использовал Node.js в рабочих и личных проектах чуть ли не сразу после его появления. Также является соразработчиком ядра Node.js и внес немалый вклад в различные аспекты Node.js, принимая участие в некоторых рабочих группах.

Об иллюстрации на обложке

На обложке книги изображен чирок-свистунок (Anas crecca). Эта птица семейства утиных часто встречается на болотах и в лесотундре северной Канады, но на зиму улетает гораздо южнее и расселяется по всей территории Северной Америки.

Селезень в брачном наряде выглядит серым с темной головой, желтоватой спиной и светлой полосой вдоль крыла. При ближайшем рассмотрении можно рассмотреть каштанового цвета голову с широкой блестящей темно-зеленой полосой, проходящей через глаз, и розоватую, с темными пятнышками грудь. Самки имеют светло-бурое оперение и очень похожи на самок обыкновенной кряквы. Чирок-свистунок – самая маленькая речная утка в Северной Америке.

Самцы издают чистый звонкий свист, а самки – негромкое гнусавое кряканье. Питаются преимущественно на мелководье и на неглубоких болотах, едят семена, корешки и листья водяных растений. На чирков охотятся люди, скунсы, лисы, еноты, вороны и сороки.

В настоящее время охранный статус чирков – «минимальный риск». Многие животные, изображенные на обложках книг издательства O'Reilly, находятся под угрозой, все они важны для мира. Иллюстрация выполнена Карен Монтгомери на основе черно-белой гравюры из книги «Птицы Британии».

Глава 1

Введение

Когда-то компьютеры были куда проще. Мы не хотим сказать, что их было легко использовать или что писать для них код не составляло труда, но концептуально возни с ними было гораздо меньше. Типичный ПК 80-х годов прошлого века имел один восьмиразрядный процессор и не бог весть сколько памяти. Как правило, в каждый момент времени могла работать только одна программа. Даже операционная система (в современной терминологии) не работала одновременно с программой, взаимодействующей с пользователем.

Но шло время, и люди возжелали запускать сразу несколько программ – так родилась многозадачность. Это дало возможность операционным системам исполнять одновременно несколько программ и переключаться между ними. Программы могли сами решать, когда следует уступить процессор операционной системе, чтобы та могла выполнить другую программу. Такой подход называется *кооперативной*, или *невытесняющей*, *многозадачностью*.

В системе с невытесняющей многозадачностью если какая-то программа по ошибке или намеренно не уступала процессор, то никакая другая программа не могла выполниться. Такие помехи работе других программ были нежелательны, поэтому в конечном итоге операционные системы перешли на *вытесняющую многозадачность*. В этой модели операционная система решала, какой программе и в какой момент выделить процессор, и использовала для этого собственные представления о справедливом планировании, не отдавая решения о переключении процессора самим программам. Сегодня почти во всех операционных системах используется именно этот подход, даже в многоядерных компьютерах, потому что обычно исполняемых программ больше, чем процессорных ядер.

Выполнение нескольких задач одновременно исключительно полезно и программистам, и пользователям. До изобретения потоков одна программа (т. е. один *процесс*) не могла выполнять сразу несколько задач. Если программисту все же нужны были конкурентные задачи, то он должен был либо разбить задачу на несколько меньших частей и самостоятельно планировать их выполнение внутри процесса, либо запускать отдельные задачи в разных процессах и организовать их взаимодействие.

Даже в наши дни в некоторых высокоуровневых языках для выполнения нескольких задач одновременно рекомендуется запускать дополнительные процессы. В таких языках, как Ruby и Python, имеется *глобальная блокировка*

интерпретатора (global interpreter lock – *GIL*), которая означает, что в каждый момент времени может работать только один поток. Хотя управление памятью при этом упрощается, сама идея многозадачности теряет привлекательность, и программисты предпочитают использовать несколько процессов.

До сравнительно недавнего времени JavaScript тоже был языком, в котором имелся единственный механизм многозадачности: разбить задачу на части и запланировать выполнение этих частей в будущем, а в случае Node.js – запускать дополнительные процессы. Обычно мы разбиваем код на асинхронные блоки, применяя обратные вызовы или обещания. Типичный фрагмент кода, написанный в таком духе, показан в примере 1.1, где операции организованы в виде обратных вызовов или `await`.

Пример 1.1 ❖ Два варианта написания типичного фрагмента асинхронного JavaScript-кода

```
readFile(filename, (data) => {
  doSomethingWithData(data, (modifiedData) => {
    writeFile(modifiedData, () => {
      console.log('done');
    });
  });
});

// или

const data = await readFile(filename);
const modifiedData = await doSomethingWithData(data);
await writeFile(filename);
console.log('done');
```

Сегодня во всех основных средах выполнения JavaScript имеется доступ к потокам, и, в отличие от Ruby и Python, нет никакой блокировки GIL, из-за которой они становятся практически бесполезными при выполнении счетных задач. Есть, правда, другие ограничения, например не использовать совместно один объект JavaScript в нескольких потоках (по крайней мере, не напрямую). Но все равно потоки полезны JavaScript-разработчикам, когда нужно огородить счетные задачи, отделив их от остальных. В браузере существуют также специальные потоки, которым доступна не такая функциональность, как у главного потока. Как все делается, мы подробно рассмотрим в последующих главах, но чтобы вы могли составить первоначальное представление, в примере 1.2 показано, как просто создать новый поток и обработать сообщение в браузере.

Пример 1.2 ❖ Запуск потока в браузере

```
const worker = new Worker('worker.js');
worker.postMessage('Hello, world');

// worker.js
self.onmessage = (msg) => console.log(msg.data);
```

Цель этой книги – исследовать и объяснить поток JavaScript как концепцию и элемент программирования. Вы узнаете, как их использовать и, что еще важнее, когда это делать. Не каждую проблему можно решить с помощью потоков. И даже не каждую счетную задачу следует решать с помощью потоков. Работа разработчиков ПО в том и заключается, чтобы оценивать проблемы и инструменты и выбирать наиболее подходящие решения. А наша цель – снабдить вас еще одним инструментом и достаточной информацией о том, как и когда применять его.

Что такое потоки?

Во всех современных операционных системах единицы исполнения вне ядра организованы в процессы и потоки. Разработчики могут использовать процессы и потоки и механизмы их взаимодействия, чтобы добавить в проект конкурентность. В системах с несколькими процессорами это также означает добавление параллелизма.

При выполнении любой программы, например Node.js или редактора кода, вы создаете процесс. Это означает, что код загружается в область памяти, выделенную этому и только этому процессу, и программа не сможет адресовать никакую другую область памяти, не запросив у ядра дополнительную память или отображение памяти. Если не создать дополнительных потоков или процессов, то в каждый момент времени можно будет выполнять только одну машинную *команду*, причем порядок команд предписан кодом программы. Можете считать, что команда – это единица кода, что-то вроде строки кода. (На самом деле команда обычно соответствует одной строке ассемблерного кода!)

Программа может запускать дополнительные процессы, имеющие собственную область памяти. Эти процессы не разделяют память между собой (если только она не была отображена с помощью специальных системных вызовов) и имеют свои собственные указатели команд, т. е. в одно и то же время выполняют разные команды. Если процессы выполняются на одном ядре, то процессор должен переключаться между процессами – временно приостанавливать выполнение одного и возобновлять выполнение другого.

Программа может также запускать потоки, а не полноценные процессы. Поток во многом напоминает процесс, но разделяет память с процессом, которому принадлежит. Процесс может создать много потоков, и у каждого будет свой указатель команд. Все сказанное о выполнении процессов относится и к потокам. Но, поскольку потоки разделяют память, им проще сообща использовать код и другие объекты. Поэтому для организации конкурентности потоки полезнее процессов, правда, ценой усложнения программирования, о чем мы будем говорить ниже в этой книге.

Типичный способ воспользоваться преимуществами потоков, например взять на себя математические операции, загружающие процессор, – создать дополнительный поток или пул потоков, освободив главный поток для взаимодействия с пользователями или другими программами и поручив ему

просто проверять в бесконечном цикле наличие запросов на новое взаимодействие. Многие классические веб-серверы, в частности Apache, именно так обрабатывают большое количество HTTP-запросов. В результате схема работы программы выглядит так, как на рис. 1.1. В этой модели данные HTTP-запроса передаются рабочему потоку для обработки, а когда ответ будет готов, он передается главному потоку, который возвращает его пользовательскому агенту.

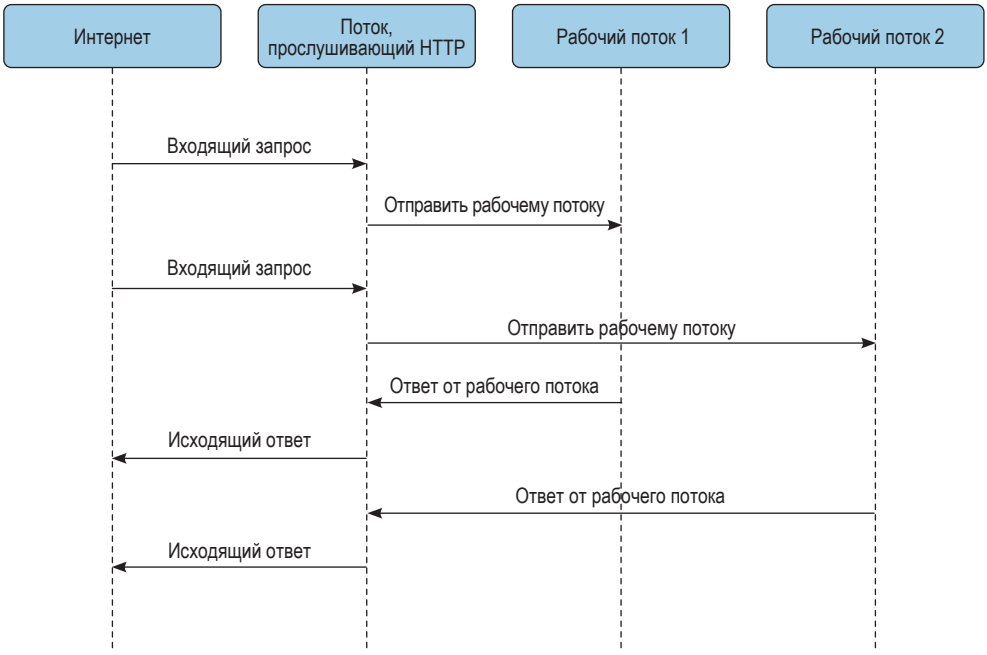


Рис. 1.1 ❖ Возможное использование рабочих потоков в HTTP-сервере

Чтобы от потоков была какая-то польза, они должны взаимодействовать. Это означает, например, что они должны уметь ждать возникновения событий в других потоках и получать от них данные. Как уже было сказано, потоки пользуются общей памятью, а с помощью дополнительных примитивов можно построить систему передачи сообщений между потоками. Часто такого рода конструкции доступны на уровне языка или платформы.

Конкурентность и параллелизм

Важно различать конкурентность и параллелизм, потому что они довольно часто возникают при многопоточном программировании. Эти термины тесно связаны, но в зависимости от обстоятельств могут означать похожие, но не вполне совпадающие вещи. Начнем с определений.

Конкурентность
Задачи перекрываются во времени.

Параллелизм
Задачи выполняются строго одновременно.

На первый взгляд кажется, что это одно и то же, но примите во внимание, что задачи могут быть разбиты на более мелкие части, которые чередуются во времени. В таком случае конкурентности можно добиться и без параллелизма, потому что интервалы времени, в течение которых задачи работают, могут перекрываться. Чтобы можно было говорить о параллельном выполнении задач, необходимо, чтобы они работали *строго одновременно*. В общем случае это означает, что они должны выполняться на разных процессорных ядрах в одно и то же время.

Рассмотрим рис. 1.2. Мы видим две задачи, работающие конкурентно и параллельно. В первом случае в каждый момент времени выполняется только одна задача, но на протяжении всего периода выполнение несколько раз переключается между задачами. Это значит, что интервалы работы двух задач перекрываются, что согласуется с определением конкурентности. Во втором случае обе задачи работают одновременно, т. е. параллельно. Поскольку интервалы их работы перекрываются, то они работают еще и конкурентно. Таким образом, из параллелизма следует конкурентность.

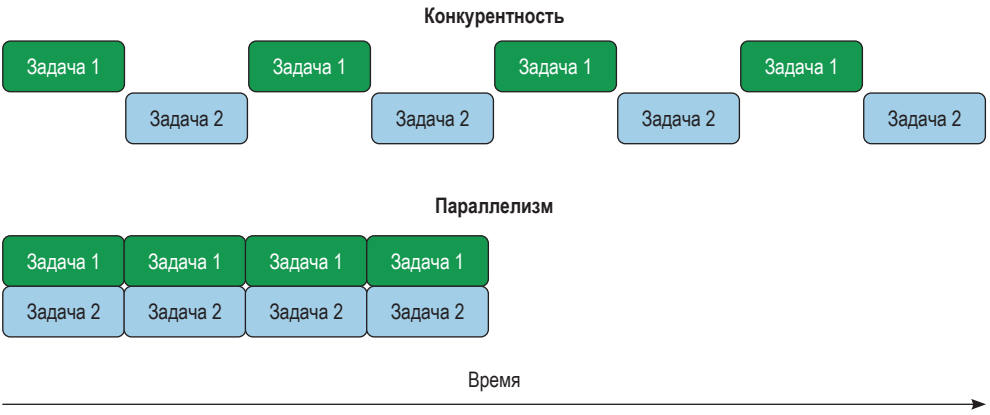


Рис. 1.2 ❖ Конкурентность и параллелизм

Потоки не обеспечивают параллелизм автоматически. Для этого необходимо, чтобы в системе было несколько процессорных ядер, а планировщик операционной системы решил запускать потоки на разных ядрах. Если в системе всего одно ядро или ядер меньше, чем потоков, то несколько потоков будут вынуждены работать на одном ядре конкурентно, а система будет переключать процессор между ними. Кроме того, в языках с блокировкой GIL, каковыми являются, в частности, Ruby и Python, потокам явно запрещено предоставлять параллелизм, потому что в среде может выполняться только одна команда в каждый момент времени.

Важно также принимать во внимание хронометраж, потому что обычно потоки включаются в программу ради повышения производительности. Если система допускает только конкурентность, потому что оснащена одним процессорным ядром или все ядра уже загружены другими задачами, то увеличение числа потоков может не принести желаемой выгоды. Более того, из-за накладных расходов на синхронизацию и контекстное переключение между потоками может даже оказаться, что программа стала работать медленнее. Всегда измеряйте производительность своего приложения в тех условиях, в которых оно предположительно будет работать. Только так вы сможете убедиться, дает ли многопоточная модель программирования какие-нибудь преимущества.

Однопоточный JavaScript

Исторически платформы, на которых работал JavaScript, вообще не поддерживали потоков, поэтому язык задумывался как однопоточный. Когда вы слышите от кого-то, что JavaScript однопоточный, говорящий имеет в виду именно это историческое наследие и стиль программирования, естественно тяготеющий к нему. Надо признать, что, вопреки названию книги, сам язык не содержит никаких встроенных средств для создания потоков. Это, впрочем, не должно вызывать удивления, потому что встроенных средств нет также для работы с сетью, устройствами, файловой системой или для выполнения системных вызовов. Даже такая важная функция, как `setTimeout()`, на самом деле не является принадлежностью JavaScript. Все это – посредством специальных API – предоставляет среда, в которую погружена виртуальная машина (ВМ), например Node.js или браузер.

Вместо использования потоков как примитива обеспечения конкурентности JavaScript-код чаще всего пишется в объектно-ориентированном стиле и исполняется в одном потоке. Возникающие события, например взаимодействие с пользователем или ввод–вывод, активируют выполнение функций, предварительно заданных как обработчики событий. Обычно эти функции называются *обратными вызовами*, и именно на них основано асинхронное программирование в Node.js и браузере. Даже при использовании обещаний или синтаксиса `async/await` в основе всего – глубоко или не очень глубоко внизу – лежат обратные вызовы. Важно понимать, что обратные вызовы не выполняются ни параллельно, ни наряду с каким-то другим кодом. Когда работает код обратного вызова, никакой другой код не работает. По-другому можно сказать, что в каждый момент времени активен только один стек вызовов.

Часто возникает ложное представление, будто операции выполняются параллельно, тогда как на самом деле они работают конкурентно. Например, допустим, что мы хотим открыть три файла, содержащих числа, *1.txt*, *2.txt* и *3.txt*, а затем сложить эти числа и напечатать результат. В Node.js можно написать код, показанный в примере 1.3.

Пример 1.3 ❖ Конкурентное чтение из файлов в Node.js

```
import fs from 'fs/promises';

async function getNum(filename) {
  return parseInt(await fs.readFile(filename, 'utf8'), 10);
}

try {
  const numberPromises = [1, 2, 3].map(i => getNum(`${i}.txt`));
  const numbers = await Promise.all(numberPromises);
  console.log(numbers[0] + numbers[1] + numbers[2]);
} catch (err) {
  console.error('Что-то не так:');
  console.error(err);
}
```

Чтобы выполнить этот код, сохраните его в файле *reader.js*. Проверьте, что текстовые файлы названы *1.txt*, *2.txt* и *3.txt* и что они содержат целые числа. Затем выполните команду `node reader.js`.

Мы воспользовались функцией `Promise.all()`, а значит, будем ждать, пока все три файла будут прочитаны и разобраны. При некотором воображении можно даже увидеть сходство с функцией `pthread_join()` из примера программы на C ниже в этой главе. Однако из того, что обещания создаются вместе и что мы ждем их совместного исполнения, еще не следует, что реализующий их код работает одновременно; можно лишь утверждать, что их временные интервалы перекрываются. Но указатель команд по-прежнему один, и в каждый момент времени выполняется только одна команда.

В отсутствие потоков JavaScript работает только с одной средой. Это означает, что существует один экземпляр ВМ, один указатель команд и один сборщик мусора. Говоря об указателе команд, мы имеем в виду, что интерпретатор JavaScript выполняет только одну команду в каждый момент времени. Но это не значит, что мы ограничены единственным глобальным объектом. И в браузере, и в Node.js в нашем распоряжении имеются области (realm) (https://262.ecma-international.org/11.0/#sec-code-realms).

Области можно рассматривать как экземпляры среды JavaScript, предоставляемые JavaScript-коду. Это значит, что каждая область получает свой глобальный объект со всеми его свойствами, например встроенным классом `Date`, объектом `Math` и пр. В Node.js глобальный объект называется `global`, а в браузерах `window`, но в современных версиях того и другого к нему можно также обращаться по имени `globalThis`.

В браузерах у каждого фрейма на веб-странице имеется область для всего содержащегося во фрейме JavaScript-кода. Поскольку у каждого фрейма своя копия `Object` и прочих примитивов внутри него, деревья наследования тоже разделены, поэтому оператор `instanceof` может работать не так, как вы ожидаете, если применяется к объектам из разных областей. Это показано в примере 1.4.

Пример 1.4 ❖ Объекты из внутреннего фрейма в браузере

```
const iframe = document.createElement('iframe');
document.body.appendChild(iframe);\
const FrameObject = iframe.contentWindow.Object; ❶

console.log(Object === FrameObject); ❷
console.log(new Object() instanceof FrameObject); ❸
console.log(FrameObject.name); ❹
```

❶ Глобальный объект внутри `iframe` доступен через свойство `contentWindow`.
❷ Здесь возвращается false, потому что `Object` внутри фрейма не совпадает с объектом из главного фрейма.
❸ `instanceof` возвращает `false`, как и следовало ожидать, поскольку это не один и тот же `Object`.
❹ Несмотря ни на что, конструкторы имеют одно и то же свойство `name`.

В Node.js области создаются функцией `vm.createContext()`, как показано в примере 1.5. В терминологии Node.js области называются контекстами (`Context`). Все правила и свойства, применимые к фреймам браузера, относятся и к контекстам, но в контексте отсутствует доступ к глобальным свойствам или еще чему-то, что может оказаться в области видимости файла Node.js. Если вам это нужно, то нужно вручную передать в контекст.

Пример 1.5 ❖ Объекты в новом контексте в Node.js

```
const vm = require('vm');
const ContextObject = vm.runInNewContext('Object'); ❶

console.log(Object === ContextObject); ❷
console.log(new Object() instanceof ContextObject); ❸
console.log(ContextObject.name); ❹
```

❶ Получить объекты из нового контекста можно с помощью функции `runInNewContext`.
❷ Возвращается false, потому что, как и во фреймах браузера, `Object` внутри контексте – не то же самое, что в главном контексте.
❸ Аналогично `instanceof` возвращает false.
❹ Как и раньше, у конструкторов имеется то же самое свойство `name`.

В любом случае важно помнить, что области пользуются одним и тем же указателем команд и что в каждый момент времени выполняется код только из одной области. Поэтому мы по-прежнему говорим об однопоточном выполнении.

Скрытые потоки

Хотя ваш JavaScript-код, может быть, и работает – по крайней мере по умолчанию – в однопоточной среде, это еще не значит, что процесс, исполняющий ваш код, однопоточный. На самом деле в нем может быть много потоков, обеспечивающих эффективное и ничем не омрачаемое выполнение вашего кода. Утверждение о том, что Node.js – однопоточный процесс, является распространенным заблуждением.

В современных движках JavaScript, например V8, отдельные потоки используются для сборки мусора и других операций, которые необязательно должны синхронизироваться с выполнением JavaScript-кода. Кроме того, сами платформенные среды выполнения могут использовать дополнительные потоки для представления каких-то служб.

В Node.js библиотека `libuv` предоставляет независимый от ОС интерфейс асинхронного ввода–вывода, а поскольку не все системные интерфейсы ввода–вывода асинхронны, он пользуется пулом рабочих потоков, чтобы избежать блокировки программного кода, когда тот обращается к блокирующим API, например API файловой системы. По умолчанию запускается четыре таких потока, но это число можно изменить с помощью переменной среды `UV_THREADPOOL_SIZE`; максимальное число потоков равно 1024.

В системах Linux дополнительные потоки можно увидеть, выполнив команду `top -H` для данного процесса. В примере 1.6 запущен простой веб-сервер Node.js, а затем его PID передан команде `top`. Видно, что имеется семь различных потоков V8 и `libuv`, включая тот, в котором работает JavaScript-код. Можете попробовать то же самое с собственными программами для Node.js и даже попытаться изменить переменную среды `UV_THREADPOOL_SIZE` и посмотреть, изменится ли число потоков.

Пример 1.6 ❖ Распечатка `top`, из которой видны потоки в процессе Node.js

```
$ top -H -p 81862
top - 14:18:49 up 1 day, 23:18, 1 user, load average: 0.59, 0.82, 0.83
Threads:  7 total,  0 running, 7  sleeping,  0 stopped,  0 zombie
%Cpu(s): 2.2 us, 0.0 sy, 0.0 ni, 97.8 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
MiB Mem : 15455.1 total, 2727.9 free, 5520.4 used, 7206.8 buff/cache
MiB Swap:  2048.0 total, 2048.0 free,    0.0 used.     8717.3 avail Mem

  PID  USER    PR  NI    VIRT    RES    SHR S  %CPU  %MEM    TIME+ COMMAND
81862 bengl    20   0  577084  29272  25064 S   0.0   0.2  0:00.03 node
81863 bengl    20   0  577084  29272  25064 S   0.0   0.2  0:00.00 node
81864 bengl    20   0  577084  29272  25064 S   0.0   0.2  0:00.00 node
81865 bengl    20   0  577084  29272  25064 S   0.0   0.2  0:00.00 node
81866 bengl    20   0  577084  29272  25064 S   0.0   0.2  0:00.00 node
81867 bengl    20   0  577084  29272  25064 S   0.0   0.2  0:00.00 node
81868 bengl    20   0  577084  29272  25064 S   0.0   0.2  0:00.00 node
```

Браузеры тоже выполняют многие задачи, например отрисовку объектной модели документа (DOM), в потоках, отличных от того, где работает JavaScript-код. Эксперимент с `top -H` покажет такую же горстку потоков, как в случае Node.js. Современные браузеры даже идут дальше, используя несколько процессов, чтобы благодаря изоляции повысить уровень безопасности.

Важно помнить об этих дополнительных потоках, планируя ресурсы для своего приложения. Никогда не следует предполагать, что раз JavaScript – однопоточный язык, то и в JavaScript-приложении будет использоваться только один поток. Например, в производственных приложениях Node.js измеряйте количество запущенных потоков и планируйте соответственно. Не забывайте, что многие дополнительные модули в экосистеме Node.js запускают еще и собственные потоки, так что это упражнение следует выполнять для каждого приложения.

Потоки на C: обогатитесь с помощью криптовалюты Happycoin

Понятно, что потоки не уникальная особенность JavaScript. Они уже давно укоренились на уровне операционной системы и не зависят от языка. Посмотрим, как могла бы выглядеть многопоточная программа, написанная на C. Язык C – очевидный выбор, потому что именно написанный на C интерфейс лежит в основе большинстве реализаций потоков в языках высокого уровня, даже если их семантика выглядит иначе. Рассмотрим следующий алгоритм доказательства выполнения работы для простой и практически бесполезной криптовалюты Happycoin.

1. Сгенерировать случайное 64-разрядное целое число без знака.
2. Определить, является ли это число счастливым.
3. Если нет, это не Happycoin.
4. Если оно не делится на 10 000, это не Happycoin.
5. В противном случае это Happycoin.

Число называется счастливым, если в цикле замены числа суммой квадратов его цифр встретилась либо 1, либо ранее возникавшее число. В «Википедии» (https://en.wikipedia.org/wiki/Happy_number) это понятие определено точно, а также отмечено, что число возникает повторно тогда и только тогда, когда встречается 4. Вы, наверное, обратили внимание, что наш алгоритм неэффективен, потому что проверять делимость на 10 000 можно было бы до проверки того, является ли число счастливым. Это сделано намеренно, потому что мы хотим продемонстрировать высокую нагрузку.

Напишем простую программу на C, которая прогоняет алгоритм доказательства выполнения работы 10 000 000 раз, печатая все найденные Happycoin'ы и их количество.

Команду `cc` в шагах компиляции можно заменить на `gcc` или `clang`, в зависимости от того, что есть в вашей системе. В большинстве систем `cc` – псевдоним `gcc` или `clang`, поэтому мы остановимся на этом имени.

Пользователям Windows придется еще немного потрудиться, чтобы этот пример заработал в Visual Studio, потому что в нем используются POSIX-потоки (в соответствии со стандартом Portable Operating System Interface), а не потоки Windows, которые имеют с ними мало общего. Чтобы упростить выполнение примера в Windows, рекомендуем воспользоваться подсистемой Windows для Linux, тогда вы получите совместимую с POSIX среду.

С одним главным потоком

Создайте файл *happycoin.c* в каталоге *ch1-c-threads/*. Мы будем постепенно заполнять его кодом на протяжении этого раздела. Для начала добавьте код, показанный в примере 1.7.

Пример 1.7 ❖ *ch1-c-threads/happycoin.c*

```
#include <inttypes.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

uint64_t random64(uint32_t * seed) {
  uint64_t result;
  uint8_t * result8 = (uint8_t *)&result; ❶
  for (size_t i = 0; i < sizeof(result); i++) {
    result8[i] = rand_r(seed);
  }
  return result;
}
```

❶ **В этой строке используются указатели, которые, возможно, вам незнакомы, если опыт вашей работы ограничивается в основном JavaScript. В двух словах здесь происходит следующее: `result8` – это массив восьми 8-разрядных целых без знака, хранящийся в памяти, отведенной под 64-разрядное целое без знака `result`.**

Мы добавили несколько директив `include`, которые дают доступ к таким удобным вещам, как типы, функции ввода–вывода, а также для работы со временем и случайными числами. Поскольку алгоритм требует генерирования случайных 64-разрядных целых без знака (типа `uint64_t`), нам нужно восемь случайных байтов, которые `random64()` получает от вызова функции `rand_r()` в цикле. Так как `rand_r()` требует ссылки на начальное значение (seed), мы передаем его функции `random64()`.

Теперь добавим вычисление счастливого числа.

Пример 1.8 ❖ *ch1-c-threads/happycoin.c*

```
uint64_t sum_digits_squared(uint64_t num) {
  uint64_t total = 0;
  while (num > 0) {
    uint64_t num_mod_base = num % 10;
    total += num_mod_base * num_mod_base;
    num = num / 10;
  }
  return total;
}

bool is_happy(uint64_t num) {
  while (num != 1 && num != 4) {
    num = sum_digits_squared(num);
  }
  return num == 1;
}

bool is_happycoin(uint64_t num) {
  return is_happy(num) && num % 10000 == 0;
}
```

Для нахождения суммы квадратов цифр мы в функции `sum_digits_squared` пользуемся оператором взятия остатка `%`, чтобы выделить одну цифру (слева направо), а затем прибавить ее квадрат к сумме. Затем в цикле вызывается функция `is_happy`, до тех пор пока не получим 1 или 4. 1 означает, что число счастливое, а 4 – что цикл бесконечный и мы никогда не встретим 1. Наконец, в функции `is_happycoin()` мы проверяем, является ли число счастливым и делится ли оно на 10 000.

Обернем все это функцией `main()`, как показано в примере 1.9.

Пример 1.9 ❖ *ch1-c-threads/happycoin.c*

```
int main() {
  uint32_t seed = time(NULL);
  int count = 0;
  for (int i = 1; i < 10000000; i++) {
    uint64_t random_num = random64(&seed);
    if (is_happycoin(random_num)) {
      printf("%" PRIu64 " ", random_num);
      count++;
    }
  }
  printf("\ncount %d\n", count);
  return 0;
}
```

Первым делом нужно инициализировать генератор случайных чисел. В качестве начального значения вполне подойдет текущее время, поэтому получим его с помощью функции `time()`. Затем выполним 10 000 000 итераций цикла, на каждой из которых сначала получаем случайное число от `random64()`, а затем проверяем, является ли оно Happycoin'ом. Если да, то увеличиваем счетчик и печатаем число. Странный синтаксис `PRIu64` в вызове `printf()` необходим для правильного форматирования 64-разрядных целых без знака. По завершении цикла печатаем счетчик и завершаем программу.

Для компиляции и запуска программы наберите следующие команды, находясь в каталоге *ch1-c-threads*.

```
$ cc -o happycoin happycoin.c
$ ./happycoin
```

В одной строке будет напечатан список найденных Happycoin'ов, а в другой – их количество. Выглядеть это может так:

```
11023541197304510000 ... [ еще 167 чисел ] ... 770541398378840000
count 169
```

Выполнение программы занимает нетривиальное время – примерно 2 с на стандартном компьютере. Это тот случай, когда потоки могут ускорить работу, потому что многократно повторяется однотипная математическая операция.

Давайте превратим эту программу в многопоточную.

С четырьмя рабочими потоками

Заведем четыре потока, каждый из которых будет выполнять четверть итераций цикла, в котором генерируется случайное число и проверяется, является ли оно Happycoin'ом.

В POSIX C для управления потоками предназначены функции из семейства `pthread_*`. Функция `pthread_create()` создает поток. Ей передается функция, которая будет исполняться в этом потоке. Затем продолжается выполнение главного потока. Программа может дождаться завершения потока, вызвав для него функцию `pthread_join()`. Функции, исполняемой в потоке, созданном `pthread_create()`, можно передать аргументы, а возвращенное ей значение получить от `pthread_join()`.

В нашей программе мы инкапсулируем генерирование Happycoin'ов в функции `get_happycoins()`, именно ее будем запускать в потоках. Мы создадим четыре потока и сразу же начнем ждать их завершения. Получив от потока результаты, сохраним их, чтобы в конце можно было напечатать общий итог. Для передачи результатов заведем простую структуру `happy_result`.

Скопируйте существующий файл *happycoin.c* и назовите его *happycoin-threads.c*. В новый файл добавьте код из примера 1.10, разместив его под последней директивой `#include`.

Пример 1.10 ❖ *ch1-c-threads/happycoin-threads.c*

```
#include <pthread.h>

struct happy_result {
  size_t count;
  uint64_t * nums;
};
```

В первой строке включается файл `pthread.h`, который дает доступ к нужным нам функциям для работы с потоками. Затем определяется структура `struct happy_result`, в которой будут возвращаться значения из работающей в потоке функции `get_happycoins()`: массив найденных Happycoin'ов, представленный указателем, и их количество.

Теперь удалите функцию `main()` целиком, потому что мы собираемся заменить ее. Сначала добавьте функцию `get_happycoins()` из примера 11.1 – это тот код, который будет выполняться в рабочих потоках.

Пример 1.11 ❖ *ch1-c-threads/happycoin-threads.c*

```
void * get_happycoins(void * arg) {
  int attempts = *(int *)arg; ❶
  int limit = attempts/10000;
  uint32_t seed = time(NULL);
  uint64_t * nums = malloc(limit * sizeof(uint64_t));
  struct happy_result * result = malloc(sizeof(struct happy_result));
  result->nums = nums;
  result->count = 0;
  for (int i = 1; i < attempts; i++) {
    if (result->count == limit) {
```

```
      break;
    }
    uint64_t random_num = random64(&seed);
    if (is_happycoin(random_num)) {
      result->nums[result->count++] = random_num;
    }
  }
  return (void *)result;
}
```

❶ **Эта странная конструкция – приведение типа указателя – означает «обращайся с этим указателем как с указателем на `int` и дай мне значение этого `int`».**

Обратите внимание, что эта функция принимает один аргумент типа `void *` и возвращает одно значение типа `void *`. Функцию с такой сигнатурой ожидает получить `pthread_create()`, так что у нас просто нет выбора. Это означает, что мы должны привести аргументы к ожидаемым типам. Мы хотим передать количество попыток, поэтому приводим аргумент к типу `int`. Затем инициализируем начальное значение, как в предыдущем примере, но теперь это делается в потоковой функции, так что каждый поток получает разные начальные значения.

Выделив достаточно памяти для массива и структуры `happy_result`, мы входим в такой же цикл, что в функции `main()` в однопоточной версии, но на этот раз помещаем результаты в структуру, а не печатаем их. По завершении цикла мы возвращаем указатель на структуру, предварительно приведя его к типу `void *` в соответствии с сигнатурой функции. Именно так информации передается главному потоку, который будет ее интерпретировать.

Мы продемонстрировали одно из важнейших свойств потоков, отличающих их от процессов, – использование общей памяти. Если бы мы использовали не потоки, а процессы и какой-то механизм *межпроцессного взаимодействия* (interprocess communication – IPC) для обратной передачи результатов, то не смогли бы просто передать адрес памяти главному процессу, потому что главный процесс не имеет доступа к памяти рабочего. Из-за виртуализации памяти этот адрес мог бы указывать в какое-то произвольное место главного процесса. Поэтому вместо указателя мы должны были бы передать главному процессу все значение целиком по каналу IPC, что повлекло бы за собой накладные расходы. Но, коль скоро мы используем потоки, а не процессы, можно передать просто указатель, потому что в главном потоке он будет указывать туда же, куда и в рабочем.

Но у разделяемой памяти есть свои недостатки. В нашем случае рабочий поток никак не использует память, после того как передал ее главному. Но так бывает не всегда. Чаще всего приходится аккуратно управлять доступом потоков к разделяемой памяти с помощью механизмов синхронизации, иначе результаты могут быть непредсказуемы. Как это работает в JavaScript, мы подробно рассмотрим в главах 4 и 5.

Теперь обернем все это функцией `main()`, как показано в примере 1.12.

Пример 1.12 ❖ *ch1-c-threads/happycoin-threads.c*

```
#define THREAD_COUNT 4

int main() {
```

```
  pthread_t thread [THREAD_COUNT];

  int attempts = 10000000/THREAD_COUNT;
  int count = 0;
  for (int i = 0; i < THREAD_COUNT; i++) {
    pthread_create(&thread[i], NULL, get_happycoins, &attempts);
  }
  for (int j = 0; j < THREAD_COUNT; j++) {
    struct happy_result * result;
    pthread_join(thread[j], (void **)&result);
    count += result->count;
    for (int k = 0; k < result->count; k++) {
      printf("%" PRIu64 " ", result->nums[k]);
    }
  }
  printf("\ncount %d\n", count);
  return 0;
}
```

Сначала мы объявляем все четыре потока в виде массива в стеке. Затем делим число итераций (10 000 000) на число потоков. Результаты мы будем передавать функции `get_happycoins()` в качестве аргумента, как показано в первом цикле, где `pthread_create()` создает потоки. В следующем цикле мы ждем завершения каждого потока с помощью `pthread_join()`. После этого можем напечатать результаты, собранные от всех потоков, и общее число Happycoin'ов – точно так же, как в однопоточной версии.

В этой программе есть утечка памяти. Одна из трудностей многопоточного программирования на C и некоторых других языках состоит в том, что очень легко запутаться, где и когда память выделяется и где и когда ее нужно освобождать. Попробуйте модифицировать код, так чтобы вся выделенная из кучи память гарантированно освобождалась перед выходом из программы.

Откомпилируйте и запустите новую версию программы, выполнив следующие команды из каталога *ch1-c-threads*.

```
$ cc -pthread -o happycoin-threads happycoin-threads.c
$ ./happycoin-threads
```

Результат будет выглядеть примерно так:

```
2466431682927540000 ... [ еще 154 числа ] ... 15764177621931310000
count 156
```

Вы, конечно, заметили, что результат похож на полученный ранее[1]. И наверное, обратили внимание, что программа работает немного быстрее. На стандартном компьютере она выполняется примерно 0.8 с. Это, безусловно, не в четыре раза быстрее из-за начальных накладных расходов в главном

[1] Счетчики в двух версиях различаются, но это несущественно, потому что счетчик зависит от того, сколько случайных чисел оказались Happycoin'ами. В любых двух прогонах счетчики будут различаться.

потоке, да и печать результатов тоже чего-то стоит. Можно было бы печатать результаты сразу после их получения в рабочем потоке, но тогда они могли бы наложиться друг на друга, потому что ничто не мешает двум потокам печатать в одно и то же место одновременно. Путем отправки результатов в главный поток мы координируем печать результатов, избегая мусора на экране.

Это иллюстрация главного преимущества и одного из недостатков многопоточного кода. С одной стороны, полезно разбивать счетные задачи на части, работающие параллельно. С другой стороны, необходимо гарантировать, что события надлежащим образом синхронизированы, иначе могут возникнуть загадочные ошибки. Добавляя потоки в программу на любом языке, нужно внимательно следить за их правильным использованием. И, как всегда при попытке ускорить программу, необходимо проводить измерения. Никому не нужна лишняя сложность, обусловленная многопоточностью, если это не приносит приложению никакой выгоды.

Любой язык программирования, поддерживающий потоки, предоставляет какие-то механизмы для создания и уничтожения потоков, передачи сообщений между ними и взаимодействия с данными, разделяемыми потоками. Эти механизмы могут выглядеть по-разному, потому что различаются языки и парадигмы, а равно и модели параллельного программирования. Познакомившись с тем, как выглядит многопоточная программа на языке низкого уровня C, перейдем к JavaScript. На поверхности здесь все устроено немного иначе, но, как мы увидим, принципы остаются теми же.

Глава 2

Браузеры

У JavaScript нет единой эталонной реализации, как у большинства языков программирования. Например, в случае Python вы, скорее всего, работаете с двоичной версией, собранной инженерами, сопровождающими язык. С другой стороны, у JavaScript много разных реализаций. К ним относятся движки в веб-браузерах: V8 в Chrome, SpiderMonkey в Firefox и JavaScriptCore в Safari. Движок V8 используется также в Node.js на сервере.

В основе всех этих реализаций лежит спецификация ECMAScript. Как следует из таблицы совместимости, с которой нам так часто приходится сверяться, не все движки реализуют JavaScript одинаково. Безусловно, производители браузеров стараются реализовать функциональность JavaScript единообразно, но ошибки все же случаются. На уровне языка есть кое-какие примитивы конкурентности, с которыми мы подробнее познакомимся в главах 4 и 5.

Но в каждую реализацию добавлены свои API, делающие JavaScript еще мощнее. В этой главе мы сосредоточимся исключительно на API многопоточности, которые предлагаются всеми современными браузерами, и самым доступным из них является веб-исполнитель (web worker).

Использовать рабочие потоки имеет смысл по многим причинам, но к браузерам в первую очередь относится тот факт, что за счет переноса сложных вычислений в отдельный поток главный поток может уделить больше внимания отрисовке пользовательского интерфейса (UI). Это делает работу пользователя более удобной и комфортной, чем при традиционном подходе.

Выделенные исполнители

Веб-исполнители позволяют запустить новую среду для выполнения JavaScript. Интерпретатор, запущенный таким образом, работает не в том же потоке, что запустивший его. Взаимодействие между двумя средами осуществляется путем *передачи сообщений*. Напомним, что по самой своей природе JavaScript – однопоточный язык. Веб-исполнители не противоречат этому, а обмен сообщениями производится путем активации функций, выполняемых в цикле событий.

Среда JavaScript может запустить более одного веб-исполнителя, да и сам веб-исполнитель может запускать дополнительных исполнителей. Однако если вы поймали себя за созданием обширных иерархий веб-исполнителей, то, вероятно, имеет смысл пересмотреть архитектуру приложения.

Есть несколько типов веб-исполнителей, самым простым является выделенный исполнитель.

Выделенный исполнитель Hello World

Изучить новую технологию проще всего, поработав с ней. Связь между страницей и исполнителем, которую мы собираемся построить, показана на рис. 2.1. В данном случае создается всего один исполнитель, но иерархия тоже возможна.

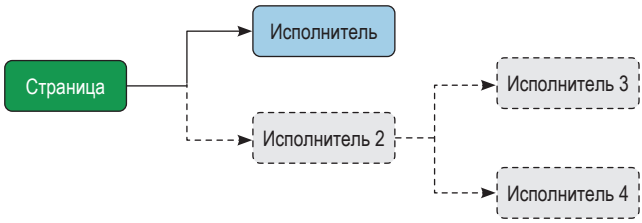


Рис. 2.1 ❖ Связь с выделенными исполнителями

Первым делом создайте каталог *ch2-web-workers/*. Там будут храниться все три файла, необходимых для этого проекта. Затем создайте в этом каталоге файл *index.html*. JavaScript, работающий в браузере, должен сначала загрузить веб-страницу, и этот файл будет ее основой. Поместите в него код, показанный на рис. 2.1.

Пример 2.1 ❖ *ch2-web-workers/index.html*

```
<html>
  <head>
    <title>Web Workers Hello World</title>
    <script src="main.js"></script>
  </head>
</html>
```

Как видите, файл проще не придумаешь. Мы всего лишь задаем заголовок страницы и загружаем один JavaScript-файл *main.js*. В остальных разделах этой главы будет примерно так же. Самая интересная часть – то, что находится внутри файла *main.js*.

Давайте создадим *main.js* прямо сейчас и добавим в него код из примера 2.2.

Пример 2.2 ❖ *ch2-web-workers/main.js*

```
console.log('привет от main.js');

const worker = new Worker('worker.js'); ❶

worker.onmessage = (msg) => { ❷
```

```
  console.log('получено сообщение от исполнителя', msg.data);
};

worker.postMessage('отправлено сообщение исполнителю'); ❸

console.log('конец main.js');
```

❶ Создание нового выделенного исполнителя.
❷ Присоединение обработчика сообщений к исполнителю.
❸ Исполнителю передается сообщение.

Первым делом мы вызываем функцию `console.log()`, чтобы было видно, в каком порядке выполняются файлы. Затем создается новый выделенный исполнитель, для чего вызывается `new Worker(`*`filename`*`)`. Будучи вызванным, движок JavaScript начинает загружать (или ищет в кеше) нужный файл, это делается в фоновом режиме.

Далее к исполнителю присоединяется обработчик события `message`. Для этого свойству `.onmessage` выделенного исполнителя присваивается функция. При получении сообщения эта функция будет вызвана. В качестве аргумента функции передается экземпляр `MessageEvent`. У него много свойств, но самое интересное из них `.data`. Оно представляет объект, возвращенный выделенным исполнителем.

Наконец, вызывается метод выделенного исполнителя `.postMessage()`. Именно так среда JavaScript, создавшая выделенный исполнитель, взаимодействует с ним. В данном случае исполнителю передается простая строка. Существуют ограничения на данные, которые можно передавать этому методу; дополнительные сведения см. в приложении «Алгоритм структурированного клонирования».

Покончив с главным JavaScript-файлом, перейдем к созданию файла, который будет выполняться выделенным исполнителем. Создайте файл *worker.js* и поместите в него код из примера 2.3.

Пример 2.3 ❖ *ch2-web-workers/worker.js*

```
console.log('привет от worker.js');

self.onmessage = (msg) => {
  console.log('сообщение от main', msg.data);

  postMessage('сообщение, отправленное исполнителем');
};
```

В этом файле определена одна глобальная функция `onmessage`. Функция `onmessage` выделенного исполнителя вызывается, когда вне этого исполнителя вызван метод `worker.postMessage()`. Присваивание можно было бы записать также в виде `onmessage =` или даже `var onmessage =`, но конструкции `const onmessage =` или `let onmessage =` или даже объявление `function onmessage` работать не будут. Идентификатор `self` является псевдонимом `globalThis` внутри веб-исполнителя, где хорошо знакомый идентификатор `window` недоступен.

Внутри функции `onmessage` сначала печатается сообщение, полученное извне. Затем вызывается глобальная функция `postMessage()`. Она принимает

аргумент, который затем передается вызывающей среде благодаря активации метода `onmessage()` выделенного исполнителя. Действуют те же правила передачи сообщения и клонирования объекта. Пока что мы передаем простую строку, как и раньше.

Для загрузки скрипта выделенного обработчика имеются дополнительные правила. Загружаемый файл должен принадлежать тому же источнику, из которого была загружена главная среда JavaScript. Кроме того, браузер запретит запускать выделенные исполнители, когда JavaScript работает по протоколу `file://`. Эта хитроумная фраза означает просто, что нельзя запустить приложение, дважды щелкнув по файлу *index.html*, а нужно запускать его на веб-сервере. Впрочем, если у вас установлена недавняя версия Node.js, то можно с помощью следующей команды локально запустить очень простой веб-сервер:

```
$ npx serve .
```

Эта команда запускает сервер, который обслуживает файлы из локальной файловой системы. Она также выводит URL-адрес, по которому этот сервер доступен. Обычно печатается следующий URL – в предположении, что порт свободен.

```
http://localhost:5000
```

Скопируйте напечатанный URL-адрес и откройте его в браузере. При первом открытии страницы вы, скорее всего, увидите белый экран. Но пугаться не стоит – ведь весь вывод идет на консоль веб-разработчика. Разные браузеры предоставляют доступ к консоли по-разному, но обычно достаточно щелкнуть правой кнопкой мыши в любой точке фона, а затем выбрать из контекстного меню пункт **Inspect** (Исследовать элемент) или нажать комбинацию клавиш **Ctrl**+**Shift**+**I** (или **Cmd**-**Shift**-**I**). Находясь в инспекторе, перейдите на вкладку **Console** и обновите страницу на случай, если консольные сообщения не были перехвачены. После всего этого вы должны увидеть сообщения, перечисленные в табл. 2.1.

Таблица 2.1. Пример вывода на консоль

Сообщение	Местоположение
привет от main.js	main.js 1:9
конец main.js	main.js 11:9
привет от worker.js	worker.js 1:9
сообщение от main, сообщение, отправленное исполнителем	worker.js 4:11
получено сообщение от исполнителя, сообщение, отправленное исполнителем	main.js 6:11

Этот вывод показывает, в каком порядке выполнялась отправка сообщений, хотя он и не вполне детерминирован. Сначала загружается *main.js* и печатается его сообщение. Создается и настраивается исполнитель, вызывается его метод `postMessage()`, и печатается сообщение, знаменующее окончание *main.js*. Затем начинает работать *worker.js* и вызывается его обработчик сообщений, который печатает сообщение. Затем обработчик вызывает `postMes-`

sage(), чтобы отправить сообщение *main.js*. Наконец, вызывается обработчик onmessage выделенного исполнителя в *main.js* и печатается последнее сообщение.

Продвинутое использование выделенного исполнителя

Познакомившись с основами выделенных исполнителей, мы готовы перейти к более сложной функциональности.

При работе с JavaScript без использования выделенных исполнителей весь загруженный код оказывается в одной области. Загрузка нового JavaScript-кода производится либо из скрипта с помощью тега <script>, либо в результате XHR-запроса и использования функции eval(), которой передается строка, представляющая код. В случае выделенных исполнителей в DOM нельзя включить тег <script>, потому что с исполнителем не связан никакой DOM.

Вместо этого можно использовать глобальную функцию importScripts(), доступную только внутри веб-исполнителей. Она принимает один или несколько аргументов, представляющих пути к загружаемым скриптам. Скрипты должны принадлежать тому же источнику, что и сама веб-страница. Скрипты загружаются синхронно, т. е. код, следующий за вызовом этой функции, будет выполняться только после загрузки скриптов.

Экземпляры Worker наследуют EventTarget и располагают общими методами для работы с событиями. Однако класс Worker предоставляет наиболее важные методы на уровне экземпляра. Ниже приведен список этих методов, с одними мы уже встречались, с другими еще нет.

`worker.postMessage(msg)`
: Отправляет исполнителю сообщение, которое обрабатывается в цикле событий до вызова функции self.onmessage, которой передается msg.

`worker.onmessage`
: Если настроен, то вызывается после вызова функции self.postMessage внутри исполнителя.

`worker.onerror`
: Если настроен, то вызывается, когда внутри исполнителя произошла ошибка. Получает один аргумент ErrorEvent, в котором заполнены свойства .colno, .lineno, .filename и .message. Ошибка распространяется вверх по стеку вызовов, если не был вызван метод err.preventDefault().

`worker.onmessageerror`
: Если настроен, то вызывается, когда исполнитель получает сообщение, которое не может десериализовать.

`worker.terminate()`
: Исполнитель немедленно завершается. Последующие вызовы worker.postMessage() ничего не делают и ничего не сообщают.

Внутри выделенного исполнителя глобальная переменная `self` – экземпляр класса `WorkerGlobalScope`. Самое заметное нововведение – функция `importScripts()` для включения новых JavaScript-файлов. Некоторые высокоуровневые коммуникационные API, например `XMLHttpRequest`, `WebSocket` и `fetch()`, доступны. Полезные функции, которые необязательно являются частью языка JavaScript, но предоставляются всеми основными движками, например `setTimeout()`, `setInterval()`, `atob()` и `btoa()`, также доступны. Доступны и оба API хранилищ данных, `localStorage` и `indexedDB`.

Что касается отсутствующих API, то стоит поэкспериментировать и понять, к чему вы имеете доступ. Вообще говоря, недоступны API для модификации глобального состояния веб-страницы. В главной области JavaScript глобальная переменная `location` доступна и является экземпляром класса `Location`. Внутри выделенного исполнителя `location` по-прежнему доступна, но является экземпляром класса `WorkerLocation` и работает немного иначе – прежде всего, отсутствует метод `.reload()`, перезагружающий страницу. Отсутствует также глобальная переменная `document`, предоставляющая API для доступа к DOM страницы.

Метод создания выделенного исполнителя может принимать необязательный второй аргумент, задающий параметры исполнителя:

```
const worker = new Worker(filename, options);
```

Аргумент `options` – объект, который может иметь следующие свойства.

`type`
: Либо `classic` (подразумевается по умолчанию и означает классический JavaScript-файл), либо `module`, обозначающий модуль ECMAScript (ESM).

`credentials`
: Значение определяет, передаются ли в запросе на получение файла исполнителя учетные данные HTTP. Оно может быть равно `omit` – опустить учетные данные, `same-origin` – отправить учетные данные (но только если источник тот же) или `include` – всегда отправлять учетные данные.

`name`
: Имя выделенного исполнителя, в основном для отладки. Значение доступно исполнителю в глобальной переменной `name`.

Разделяемые исполнители

Разделяемый исполнитель – еще один тип веб-исполнителя, отличающийся тем, что доступ к нему возможен из разных сред браузера, например окон (вкладок), внутренних фреймов (iframe) и даже из других веб-исполнителей. Кроме того, переменная `self` внутри такого исполнителя является экземпляром класса `SharedWorkerGlobalScope`. К разделяемому исполнителю может обращаться только JavaScript-код, загруженный из того же источника. Например, окно, работающее по адресу *http://localhost:5000*, не сможет обратиться к разделяемому исполнителю, скачанному с сайта *http://google.com:80*.

В настоящее время разделяемые исполнители отключены в Safari (https://bugs.webkit.org/show_bug.cgi?id=116359), и такое положение, похоже, сохраняется как минимум с 2013 года, что, без сомнения, нанесет урон внедрению технологии.

Прежде чем переходить к коду, остановимся на нескольких подвохах. Одна из особенностей разделяемых исполнителей, из-за которой рассуждать о них несколько затруднительно, – тот факт, что они необязательно присоединены к какому-то одному окну (среде). Конечно, первоначально такой исполнитель был запущен из конкретного окна, но затем мог обрести нескольких «владельцев». Это означает, что после закрытия первого окна разделяемый исполнитель продолжает существовать.

Поскольку разделяемый исполнитель не принадлежит конкретному окну, возникает интересный вопрос: куда выводит сообщения функция `console.log`? В версии Firefox v85 вывод направляется в то окно, которое запустило разделяемый исполнитель. Если вы откроете другое окно, то журнал все равно будет направляться в первое. Если закроете первое окно, то и журнал не увидите. Если откроете еще одно окно, то сохраненные записи журнала появятся в нем. С другой стороны, Chrome v87 вообще не отображает журналы разделяемых исполнителей. Имейте это в виду при отладке.

Отладка разделяемых исполнителей

И Firefox, и Chrome предлагают специальный способ отладки разделяемых исполнителей. В Firefox наберите в адресной строке *about:debugging*. Затем щелкните по ссылке **Этот Firefox** в левой колонке. Прокрутите страницу вниз до секции **Shared Worker'ы**, где будет приведен список скриптов разделяемых исполнителей. В нашем случае справа от элемента *sharedworker.js* появится кнопка **Исследовать**. В Chrome перейдите по адресу *chrome://inspect/#workers*, найдите элемент *shared-worker.js* и щелкните по ссылке **inspect** рядом с ним. В обоих браузерах вы попадете на специальную консоль, присоединенную к исполнителю.

Разделяемые исполнители можно использовать для хранения полупостоянного состояния, которое сохраняется при подключении других окон. Например, если Окно 1 просит разделяемого исполнителя записать значение, то Окно 2 может попросить того же исполнителя прочитать его. После обновления Окна 1 значение сохраняется. Но если закрыть или обновить последнее окно, использующее разделяемый исполнитель, то состояние будет потеряно и скрипт исполнителя будет выполнен заново.

JavaScript-файл разделяемого исполнителя сохраняется в кеше, пока его используют несколько окон; при обновлении страницы внесенные вами изменения могут и не прочитаться. Чтобы заставить браузер выполнить новый код, придется закрыть остальные его окна, а затем обновить оставшееся окно.

Памятуя об этих подводных камнях, перейдем к созданию простого приложения с разделяемыми исполнителями.

Разделяемый исполнитель Hello World

С разделяемым исполнителем ассоциируется «ключ», основанный на его положении в текущем источнике. Например, в этом примере мы будем работать с разделяемым исполнителем, находящимся по адресу вида *http://localhost:5000/shared-worker.js*. Неважно, из какого HTML-файла загружается исполнитель – */red.html, /blue.html* или даже */foo/index.html*, – экземпляр разделяемого исполнителя будет одним и тем же. Существует способ создать другие экземпляры разделяемого исполнителя с одним и тем же JavaScript-файлом, они рассмотрены в разделе «Продвинутое использование разделяемых исполнителей» ниже.

Связь между страницами и исполнителем показана на рис. 2.2.

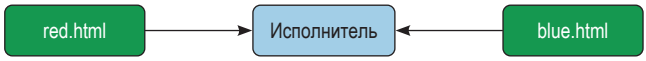


Рис. 2.2 ❖ Связь с разделяемым исполнителем

Пора уже создать какие-нибудь файлы. Для этого примера создайте каталог *ch2-shared-workers/*, все нужные нам файлы будут находиться в нем. Затем создайте HTML-файл, содержащий код из примера 2.4.

Пример 2.4 ❖ *ch2-shared-workers/red.html*

```
<html>
  <head>
    <title>Shared Workers Red</title>
    <script src="red.js"></script>
  </head>
</html>
```

Как и HTML-файл из предыдущего раздела, этот просто определяет заголовок страницы и загружает JavaScript-файл. Далее создайте еще один HTML-файл, как показано в примере 2.5.

Пример 2.5 ❖ *ch2-shared-workers/blue.html*

```
<html>
  <head>
    <title>Shared Workers Blue</title>
    <script src="blue.js"></script>
  </head>
</html>
```

В этом примере нам понадобятся два HTML-файла, представляющих разные JavaScript-среды из одного и того же источника. Вообще-то можно было бы использовать один HTML-файл в обоих окнах, но мы хотим предельно ясно подчеркнуть, что ни с HTML-файлами, ни с JavaScript-файлами *red/blue* не ассоциировано никакое состояние.

Вот теперь мы готовы создать первый JavaScript-файл, загружаемый непосредственно из HTML-файла. Создайте файл, как показано в примере 2.6.

Пример 2.6 ❖ *ch2-shared-workers/red.js*

```
console.log('red.js');

const worker = new SharedWorker('shared-worker.js'); ❶

worker.port.onmessage = (event) => { ❷
  console.log('EVENT', event.data);
};
```

❶ Создать разделяемый исполнитель.
❷ Обратите внимание на свойство `worker.port` для взаимодействия.

Это довольно простой JavaScript-файл. Здесь с помощью вызова `new SharedWorker()` создается экземпляр разделяемого исполнителя. Затем добавляется обработчик сообщений, отправляемых разделяемым исполнителем. Приходящие сообщения просто выводятся на консоль.

В отличие от экземпляров `Worker`, когда метод `.onmessage` вызывался напрямую, в экземплярах `SharedWorker` нужно использовать свойство `.port`.

Далее скопируйте только что созданный файл *red.js* в файл *blue.js*. Измените вызов `console.log()`, так чтобы он печатал *blue.js*, а больше ничего не меняйте.

Наконец, создайте файл *shared-worker.js*, как показано в примере 2.7. Именно здесь будет происходить все самое интересное.

Пример 2.7 ❖ *ch2-shared-workers/shared-worker.js*

```
const ID = Math.floor(Math.random() * 999999); ❶
console.log('shared-worker.js', ID);

const ports = new Set(); ❷

self.onconnect = (event) => { ❸
  const port = event.ports[0];
  ports.add(port);
  console.log('CONN', ID, ports.size);
  port.onmessage = (event) => { ❹
    console.log('MESSAGE', ID, event.data);

    for (let p of ports) { ❺
      p.postMessage([ID, event.data]);
    }
  };
};
```

❶ Случайный идентификатор для отладки.
❷ Список портов.
❸ Обработчик события подключения.
❹ Обратный вызов при получении нового сообщения.
❺ Сообщения переправляются в каждое окно.

Первым делом мы генерируем случайный идентификатор. Это значение будет печататься на консоли, а затем передаваться в вызывающую JavaScript-среду. В реальном приложении оно не особенно полезно, но позволяет понять, когда при работе с разделяемым исполнителем состояние сохраняется, а когда теряется.

Затем создается объект `Set` с именем `ports`[1]. В нем будет храниться список всех портов, доступных исполнителю. И свойство `worker.port`, доступное в окне, и переменная `port` в сервисном исполнителе являются экземплярами класса `MessagePort`.

Последнее, что происходит во внешней области видимости этого разделяемого исполнителя, – установка прослушивателя события `connect`. Эта функция вызывается всякий раз, как среда JavaScript создает экземпляр `SharedWorker`, ссылающийся на данный разделяемый исполнитель. При вызове прослушивателю передается экземпляр `MessageEvent` в качестве аргумента.

У события `connect` есть несколько свойств, самое важное из которых `ports`. Это массив, содержащий один элемент – ссылку на экземпляр класса `MessagePort`, который позволяет взаимодействовать с вызывающей средой JavaScript. Этот конкретный порт затем добавляется в множество `ports`.

Прослушиватель событий `message` также присоединяется к порту. Как и рассмотренный выше метод `onmessage` класса `Worker`, этот метод вызывается, когда одна из внешних сред JavaScript вызывает метод `.postMessage()`. Получив сообщение, наш обработчик печатает значение идентификатора и пришедшие данные.

Прослушиватель событий также отправляет сообщение обратно вызывающим средам. Для этого он перебирает элементы множества `ports`, вызывая для каждого метод `.postMessage()`. Поскольку этот метод принимает всего один аргумент, мы передаем ему массив, чтобы имитировать несколько аргументов. Первым элементом массива снова является идентификатор, а вторым – переданные в сообщении данные.

Если вам доводилось раньше работать с веб-сокетами в Node.js, то этот паттерн вам, наверное, знаком. В большинстве популярных пакетов WebSockets при установлении подключения генерируется событие, а к созданному подключению можно затем присоединить прослушиватель сообщений.

Теперь мы готовы протестировать приложение. Сначала выполните следующую команду, находясь в каталоге *ch2-shared-workers/*, а затем скопируйте напечатанный ей URL-адрес:

```
$ npx serve .
```

Как и раньше, был напечатан URL *http://localhost:5000*. Но на этот раз мы не будем открывать его непосредственно, а сначала откроем веб-инспектор в браузере, а только потом модифицированный URL-адрес.

Зайдите в браузер и откройте новую вкладку. Это может быть домашняя страница, пустая вкладка или что-нибудь еще, заданное по умолчанию. За-

---

[1] В версии Firefox v85 вне зависимости от того, сколько записей находится в множестве `ports`, вызов `console.log(ports)` печатает только одну запись. Чтобы во время отладки узнать размер, вызывайте `console.log(ports.size)`.

тем откройте веб-инспектор и перейдите на вкладку консоли. Скопируйте запомненный URL-адрес, добавив в конец */red.html*:

```
http://localhost:5000/red.html
```

Нажмите **Enter**, чтобы открыть страницу. Пакет `serve`, скорее всего, перенаправит ваш браузер с */red.html* на */red*, но это нормально.

После того как страница загрузится, вы должны увидеть на консоли сообщения, перечисленные в табл. 2.2. Если вы откроете инспектор после загрузки страницы, то, вероятно, не увидите никаких сообщений, но если обновить страницу, то сообщения должны появиться. Заметим, что на момент написания этой книги только Firefox показывал сообщения, сгенерированные в *shared-worker.js*.

Таблица 2.2. Вывод на консоль первого окна

Сообщение	Местоположение
`red.js`	`red.js:1:9`
`shared-worker.js 278794`	`shared-worker.js:2:9`
`CONN 278794 1`	`shared-worker.js:9:11`

Мы видим, что был выполнен файл *red.js*, что в этом конкретном экземпляре *sharedworker.js* был сгенерирован идентификатор 278794 и что в настоящий момент это – единственное окно, подключенное к данному разделяемому исполнителю.

Теперь откройте еще одно окно браузера. Как и раньше, сначала откройте окно веб-инспектора, перейдите на вкладку консоли, скопируйте базовый URL-адрес, полученный от команды `serve`, и добавьте в конец */blue.html*:

```
http://localhost:5000/blue.html
```

Нажмите **Enter**, чтобы открыть URL. После того как страница откроется, вы должны увидеть на консоли единственное сообщение, подтверждающее, что был выполнен файл *blue.js*. Пока что ничего интересного. Но вернитесь в окно, открытое на странице *red.html*. Вы увидите, что добавились новые записи, показанные в табл. 2.3.

Таблица 2.3. Вывод на консоль первого окна, продолжение

Сообщение	Местоположение
`CONN 278794 2`	`shared-worker.js:9:11`

Становится любопытно. Теперь в среде разделяемого исполнителя две ссылки на экземпляр `MessagePort`, указывающие на два разных окна. В то же время два окна ссылаются на экземпляры `MessagePort` для одного и того же разделяемого исполнителя.

Теперь мы готовы отправить сообщение разделяемому исполнителю из какого-нибудь окна. Перейдите в окно консоли и наберите такую команду:

```
worker.port.postMessage('hello, world');
```

Нажмите **Enter**, чтобы выполнить эту строку кода. Вы должны увидеть на первой консоли сообщение, сгенерированное внутри разделяемого исполнителя, на первой же консоли – сообщение от *red.js*, а на второй – сообщение от *blue.js*. Результат показан в табл. 2.4.

Таблица 2.4. Вывод на консоли первого и второго окна

Сообщение	Местоположение	Консоль
`MESSAGE 278794 hello, world`	`shared-worker.js:12:13`	1
`EVENT Array [ 278794, "hello, world" ]`	`red.js:6:11`	1
`EVENT Array [ 278794, "hello, world" ]`	`blue.js:6:11`	2

Итак, мы успешно отправили сообщение из среды JavaScript в одном окне в среду JavaScript в разделяемом исполнителе, а затем передали сообщение от исполнителя двум разным окнам.

Продвинутое использование разделяемого исполнителя

Разделяемые исполнители подчиняются тем же правилам клонирования объектов, описанным в приложении. Кроме того, как и выделенные исполнители, они имеют доступ к функции `importScripts()` для загрузки внешних JavaScript-файлов. Из версий Firefox v85 и Chrome v87 Firefox, пожалуй, более удобен для отладки разделяемых исполнителей, поскольку видно, что выводит `console.log()`.

Экземпляры разделяемого исполнителя имеют доступ к событию `connect`, которое можно обработать методом `self.onconnect()`. Бросается в глаза, особенно если вы знакомы с WebSockets, отсутствие события `disconnect` или `close`.

Если требуется создать коллекцию экземпляров `port`, как в примере кода выше, то очень легко напороться на утечку памяти. В нашем случае достаточно раз за разом обновлять одно из окон, и при каждом обновлении в множество будет добавляться новый элемент.

Это совсем нехорошо. Один из способов решить проблему – добавить в главные среды JavaScript (т. е. в *red.js* и *blue.js*) прослушиватель событий, который будет срабатывать при выгрузке страницы. Заставьте этот прослушиватель передавать специальное сообщение разделяемому исполнителю. Получив такое сообщение, разделяемый исполнитель должен будет удалить порт из списка. Вот как это делается.

```
// главный JavaScript-файл
window.addEventListener('beforeunload', () => {
  worker.port.postMessage('close');
});

// разделяемый исполнитель
port.onmessage = (event) => {
```

```
  if (event.data === 'close') {
    ports.delete(port);
    return;
  }
};
```

К сожалению, все равно возможны ситуации, когда порт «болтается». Если событие `beforeunload` не генерируется или если при его генерировании возникла ошибка или если страница «падает» непредвиденным образом, то ссылки на уже не используемые порты остаются в разделяемом исполнителе.

В более надежной системе следовало бы также предусмотреть периодический «прозвон» вызывающих сред со стороны разделяемого исполнителя путем отправки методом `port.postMessage()` специального сообщения, на которое вызывающая среда должна ответить. Тогда разделяемый исполнитель мог бы удалить экземпляры портов, если не получил ответа в течение определенного времени. Но даже этот подход несовершенен, поскольку в медленной среде JavaScript время отклика может быть велико. К счастью, попытка взаимодействия через порты, с которыми уже не связан никакой JavaScript-файл, не имеет нежелательных побочных эффектов.

Полностью конструктор класса `SharedWorker` выглядит следующим образом:

```
const worker = new SharedWorker(filename, nameOrOptions);
```

Сигнатура немного отличается от конструктора класса `Worker` прежде всего тем, что второй аргумент может содержать либо объект параметров, либо имя исполнителя. Как и в случае экземпляра `Worker`, имя доступно внутри исполнителя как `self.name`.

В этот момент у вас, наверное, возник вопрос, как это работает. Например, можно ли объявить разделяемый исполнитель с именем «red worker» в файле *red.js* и с именем «blue worker» в файле *blue.js*? В этом случае было бы создано два *разных* исполнителя, каждый со своей глобальной средой, своим значением идентификатора и заданным именем `self.name`.

Можно считать, что эти экземпляры разделяемых исполнителей «индексируются» не только своим URL-адресом, но и именем. Наверное, именно поэтому сигнатуры `Worker` и `SharedWorker` различаются, поскольку для последнего имя гораздо важнее.

Если не считать возможности указывать строку имени вместо параметров, то аргумент `options` для `SharedWorker` точно такой же, как для `Worker`.

В данном примере мы создали лишь по одному экземпляру `SharedWorker` для каждого окна, записав его в переменную `worker`, но ничто не мешает создать несколько таких экземпляров. На самом деле можно даже создать несколько разделяемых исполнителей, указывающих на один и тот же экземпляр, в предположении, что URL-адреса и имена совпадают. В таком случае принимать сообщения можно будет через свойства `.port` обоих экземпляров `SharedWorker`.

Эти экземпляры `SharedWorker`, безусловно, способны сохранять состояние между загрузками страницы. Собственно, мы это только что проделали, когда в переменной `ID` находилось уникальное число, а в переменной `ports` – спи-

сок портов. Это состояние сохраняется даже при обновлении страницы при условии, что одно окно остается открытым, например если бы мы обновили сначала страницу *blue.html*, а потом *red.html*. Однако оно будет потеряно, если обновить обе страницы одновременно, закрыть одну и обновить другую или закрыть обе страницы. В следующем разделе мы познакомимся с технологией, позволяющей сохранять состояние и исполнять код, даже когда все подключенные окна закрыты.

Сервисные исполнители

Сервисный исполнитель ведет себя как посредник, расположенный между одной или несколькими веб-страницами, работающими в браузере, и сервером. Будучи связан не с одной страницей, а, возможно, с несколькими, он больше похож на разделяемый, чем на выделенный исполнитель. Сервисные исполнители даже «индексируются» так же, как разделяемые. Но сервисный исполнитель может существовать и работать в фоновом режиме, даже когда страница уже закрыта. Поэтому выделенный исполнитель можно рассматривать как ассоциированный с одной страницей, разделяемый исполнитель – как ассоциированный с одной или более страницами, а сервисный исполнитель – как ассоциированный с нулем или более страницами. Но сервисный исполнитель не появляется из ниоткуда. Чтобы его создать, нужно сначала открыть страницу.

Сервисные исполнители предназначены главным образом для управления кешем сайта или одностраничного приложения. Чаще всего они вызываются, когда серверу отправляются запросы по сети, – тогда обработчик события внутри сервисного исполнителя перехватывает запрос. Предмет гордости сервисного исполнителя – умение возвращать кешированные активы, когда браузер отображает веб-страницу, но компьютер не имеет доступа к сети. Когда сервисный исполнитель получает запрос, он может поискать запрошенный ресурс в кеше, отправить запрос на сервер, чтобы получить некое подобие ресурса, или даже выполнить сложные вычисления и вернуть результат. Хотя последнее делает его похожим на другие веб-исполнители, не следует использовать сервисные исполнители только для того, чтобы разгрузить главный поток от вычислительной работы.

API сервисных исполнителей богаче, чем у других веб-исполнителей, хотя их основное назначение – вовсе не разгрузка главного потока. Сервисные исполнители достаточно сложны, чтобы посвятить им отдельную книгу. Но, поскольку цель этой книги – рассказать о многопоточных возможностях JavaScript, мы не станем рассматривать их во всех подробностях. Например, существует Push API специально для приема сообщений, отправленных сервером браузеру; его мы вообще касаться не будем.

Как и другие веб-исполнители, сервисный исполнитель не может обращаться к DOM. И блокирующих запросов он тоже делать не может. Например, запрещено задавать третий аргумент `XMLHttpRequest#open()` равным `false`, что заблокировало бы выполнение кода до прихода ответа или тайм-аута.

Браузеры разрешают работу сервисных исполнителей только на страницах, обслуживаемых по протоколу HTTPS. К счастью для нас, из этого правила есть одно важное исключение – `localhost` может загружать сервисные исполнители по протоколу HTTP, чтобы упростить разработку. Firefox запрещает сервисные исполнители в режиме приватного окна. Chrome, однако, разрешает сервисным исполнителям работать в режиме инкогнито. Впрочем, обмен сообщениями между окнами, работающими в нормальном режиме и в режиме инкогнито, все равно невозможен.

И в Firefox, и в Chrome в окне инспектора есть вкладка **Приложение**, в которой имеется секция сервисных исполнителей (**Service Workers**). В ней можно просматривать все сервисные исполнители, ассоциированные с текущей страницей, а также выполнять очень важное на этапе разработки действие: отменять их регистрацию, т. е. восстанавливать состояние браузера, имевшее место перед регистрацией исполнителя. К сожалению, ни одна из текущих версий браузеров не предоставляет способа сразу перейти в инспекторы JavaScript для работы с сервисными исполнителями.

Отладка сервисных исполнителей

Чтобы попасть на панели инспектора для экземпляров сервисных исполнителей, нужно отправиться в другое место. В Firefox введите в адресной строке адрес *about:debugging#/runtime/this-firefox*. Прокрутите до секции **Service workers**, там и будут видны все созданные в этой главе исполнители. В Chrome для доступа к сервисным исполнителям есть два разных экрана. Более полная страница находится по адресу *chrome://serviceworker-internals/*. Она содержит список сервисных исполнителей, их состояние и записи журнала. Другая страница – *chrome://inspect/#service-workers* – содержит гораздо меньше информации.

Итак, с некоторыми подводными камнями сервисных исполнителей мы познакомились, и можно приступить к созданию одного из них.

Сервисный исполнитель Hello World

В этом разделе мы напишем примитивный сервисный исполнитель, который перехватывает все HTTP-запросы, отправленные с простой веб-страницы. Большая часть запросов передается серверу без изменения. Но в ответ на запросы, адресованные одному конкретному ресурсу, возвращаются значения, вычисленные самим сервисным исполнителем. Большинство реальных серверных исполнителей вместо этого производят поиск в кеше, но наша-то цель – продемонстрировать их работу с точки зрения многопоточности.

Сначала снова создадим HTML-файл. Создайте новый каталог *ch2-service-workers/* и в нем файл, показанный в примере 2.8.

Пример 2.8 ❖ *ch2-service-workers/index.html*

```
<html>
  <head>
    <title>Service Workers Example</title>
```

```
    <script src="main.js"></script>
  </head>
</html>
```

Эта страница всего лишь загружает JavaScript-файл приложения, к которому мы и переходим. Создайте файл *main.js* и поместите в него код из примера 2.9.

Пример 2.9 ❖ *ch2-service-workers/main.js*

```
navigator.serviceWorker.register('/sw.js', { ❶
  scope: '/'
});

navigator.serviceWorker.oncontrollerchange = () => { ❷
  console.log('controller change');
};

async function makeRequest() { ❸
  const result = await fetch('/data.json');
  const payload = await result.json();
  console.log(payload);
}
```

❶ **Зарегистрировать сервер и определить область действия.**
❷ **Прослушивать событие `controllerchange`.**
❸ **Эта функция инициирует запрос.**

Вот теперь становится интереснее. Первым делом мы создаем сервисный исполнитель. В отличие от других веб-исполнителей здесь нет конструктора с ключевым словом `new`. Вместо этого создание исполнителя поручается объекту `navigator.serviceWorker`. Его методу `navigate` в первом аргументе передается путь к JavaScript-файлу, играющему роль сервисного исполнителя. Второй аргумент – необязательный конфигурационный объект с единственным свойством `scope`.

Область действия `scope` представляет каталог текущего источника такой, что запросы от любых HTML-страниц в этом каталоге проходят через сервисный исполнитель. По умолчанию `scope` совпадает с каталогом, из которого загружен сервисный исполнитель. В данном случае значение / берется относительно каталога, содержащего файл *index.html*, а поскольку *sw.js* в этом каталоге и находится, то область действия можно было бы опустить.

После того как сервисный исполнитель для страницы установлен, все исходящие HTTP-запросы будут проходить через него, в том числе и запросы к другим источникам. Поскольку область действия этой страницы совпадает с каталогом верхнего уровня, запросы к любой HTML-странице, открываемой на данном источнике, будут проходить через наш серверный исполнитель. Если бы `scope` был равен */foo*, то запрос к странице */bar.html* прошел бы мимо исполнителя, но на страницу */foo/baz.html* действие исполнителя распространялось бы.

Далее мы добавляем в объект `navigator.serviceWorker` прослушиватель события `controllerchange`. Когда он срабатывает, на консоль печатается сообще-

ние. Это отладочное сообщение говорит, что сервисный исполнитель перехватил запрос к странице, находящейся в его области действия.

Наконец, мы определяем функцию `makeRequest()`. Эта функция делает `GET`-запрос к файлу */data.json*, декодирует ответ, считая, что он представлен в формате JavaScript Object Notation (JSON), и печатает результат. Как вы, наверное, заметили, на эту функцию нет никаких ссылок. Позже мы вызовем ее с консоли, чтобы протестировать функциональность.

С этим файлом мы разобрались и можем перейти к самому сервисному исполнителю. Создайте файл *sw.js* и поместите в него код из примера 2.10.

Пример 2.10 ❖ *ch2-service-workers/sw.js*

```
let counter = 0;

self.oninstall = (event) => {
  console.log('установка сервисного исполнителя');
};

self.onactivate = (event) => {
  console.log('активация сервисного исполнителя');
  event.waitUntil(self.clients.claim()); ❶
};

self.onfetch = (event) => {
  console.log('fetch', event.request.url);

  if (event.request.url.endsWith('/data.json')) {
    counter++;
    event.respondWith( ❷
      new Response(JSON.stringify({counter}), {
        headers: {
          'Content-Type': 'application/json'
        }
      })
    );
    return;
  }

  // обычный HTTP-запрос
  event.respondWith(fetch(event.request)); ❸
};
```

❶ **Разрешить сервисному исполнителю контроль над страницей** *index.html*.
❷ **Перехватить запрос к файлу** */data.json*.
❸ **Все остальные URL-адреса обрабатывать как обычно.**

В этом файле мы первым делом инициализируем нулем глобальную переменную `counter`. После перехвата каждого запроса определенного типа это число будет увеличиваться на единицу. Это просто пример того, как можно доказать, что сервисный исполнитель работает; в реальном приложении никогда не следует сохранять состояние таким образом. Напротив, вы должны ожидать, что сервисные исполнители будут часто останавливаться и запус-

каться снова, причем предсказать, когда это произойдет, трудно, поскольку зависит от браузера.

Затем мы создаем обработчик события `install`, присваивая функцию свойству `self.oninstall`. Эта функция вызывается, когда данный обработчик впервые устанавливается в браузер. На этом этапе реальные приложения обычно выполняют инициализацию. Например, объект `self.caches` можно использовать для конфигурирования кешей, в которых хранятся результаты запросов. Но поскольку данное приложение ничего не инициализирует, то функция просто печатает сообщение и завершается.

Далее идет функция для обработки события `activate`. Это событие полезно, когда нужно произвести очистку в связи с появлением новых версий сервисного исполнителя. В реальном приложении его обработчик, скорее всего, будет очищать старые версии кешей.

В данном случае обработчик события `activate` вызывает метод `self.clients.claim()`. Это позволяет передать сервисному исполнителю контроль над страницей, которая его создала, т. е. страницей *index.html*. Если бы не эта строка, то страница не контролировалась бы сервисным исполнителем при первой загрузке. Однако после обновления *index.html* или ее загрузки в другую вкладку контроль был бы установлен.

Вызов `self.clients.claim()` возвращает обещание. Как это ни печально, обработчики событий, используемые в сервисных исполнителях, не являются асинхронными функциями, способными ждать обещаний с помощью `await`. Однако аргумент `event` – это объект, имеющий метод `.waitUntil()`, который умеет обращаться с обещаниями. После того как переданное этому методу обещание будет тем или иным образом разрешено, он позволит завершиться обработчикам `oninstall` и `onactivate` (а позднее и `onfetch`). Если не вызывать этот метод, как мы поступили в обработчике `oninstall`, то шаг будет считаться завершенным сразу после выхода из функции.

Последний обработчик события – функция `onfetch`. Он самый сложный и, к тому же, чаще всего вызывается на протяжении времени жизни сервисного исполнителя, а именно всякий раз, как производится запрос со страницы, находящейся под его контролем. Метод называется `onfetch`, чтобы намекнуть на его связь с функцией `fetch()` в браузере, хотя название следует признать не слишком удачным, потому что через этот метод проходит любой запрос. Например, если впоследствии на страницу будет добавлен тег `img`, то запрос изображения также приведет к вызову `onfetch`.

Эта функция сначала заносит в журнал сообщение о том, что и вправду работает, а заодно печатается запрашиваемый URL-адрес. Доступна и другая информация о запрошенном ресурсе, в частности заголовки и HTTP-метод. В реальном приложении эту информацию можно использовать, чтобы проверить, нет ли ресурса в кеше. Например, `GET`-запрос к ресурсу из текущего источника можно было бы обслужить из кеша, а если его там нет, то запросить с помощью функции `fetch()`, затем поместить в кеш и вернуть браузеру.

Этот простой пример всего лишь проверяет, заканчивается ли URL-адрес строкой */data.json*. Если нет, то тело `if` пропускается и выполняется последняя строка. Эта строка просто передает объект запроса (экземпляр класса

Request) методу fetch(), который возвращает обещание, а затем передает обещание методу event.respondWith(). Метод fetch() разрешает объект, который впоследствии будет представлять ответ, передаваемый браузеру. По существу, мы написали очень простой прокси HTTP.

Но вернемся к проверке наличия */data.json* в конце URL-адреса. Если она проходит, то выполняются более сложные действия. В этом случае переменная counter увеличивается на единицу и генерируется новый ответ (экземпляр класса Response) с нуля. Здесь мы конструируем JSON-строку, содержащую значение counter. Строка передается конструктору объекта Response в качестве первого аргумента, представляющего тело ответа. Второй аргумент содержит метаинформацию об ответе. В данном случае заголовок Content-Type устанавливается равным application/json, чтобы браузер интерпретировал полезную нагрузку как данные в формате JSON.

Создав все необходимые файлы, перейдите на консоли в каталог, где они находятся, и запустите веб-сервер следующей командой:

```
$ npx serve .
```

Снова скопируйте напечатанный URL-адрес, откройте новое окно браузера, в нем – окно инспектора и введите URL, чтобы зайти на страницу. На консоли должно появиться следующее сообщение (и, возможно, другие):

```
controller change                    main.js:6:11
```

Перейдите к списку установленных в браузер сервисных исполнителей, как было описано выше. В окне инспектора вы увидите сообщения:

```
установка сервисного исполнителя    sw.js:4:11
активация сервисного исполнителя    sw.js:8:11
```

Вернитесь в окно браузера. Находясь на вкладе консоли инспектора, выполните следующий код:

```
makeRequest();
```

При этом будет выполнена функция makeRequest(), которая отправляет GET-запрос для получения ресурса */data.json* из текущего источника. По завершении функции на консоли должно появиться сообщение Object { counter: 1 }. Это сообщение было сформировано сервисным исполнителем, а сам запрос так и не был отправлен веб-серверу. Переключившись на вкладку **Сеть** в инспекторе, вы увидите, как выглядит этот запрос для получения ресурса, нормальный во всех остальных отношениях. Щелкнув по запросу, вы увидите, что код состояния ответа равен 200, а заголовок Content-Type – application/json. С точки зрения веб-страницы, она выполнила самый обычный HTTP-запрос. Но вам-то лучше знать.

Вернитесь в секцию сервисных исполнителей в окне инспектора. Там должно появиться третье сообщение, содержащее детали запроса. На нашей машине оно выглядело так:

```
fetch http://localhost:5000/data.json sw.js:13:11
```

Итак, мы успешно перехватили HTTP-запрос, сделанный из среды JavaScript, выполнили вычисления в другой среде и вернули результат в главную среду. Как и в случае других веб-исполнителей, вычисление было произведено в отдельном потоке, работающем параллельно. Если бы сервисный исполнитель выполнял какие-то сложные и долгие вычисления, то веб-страница могла в ожидании ответа заниматься другими вещами.

В первом окне браузера вы, возможно, заметили ошибку при попытке загрузить файл *favicon.ico*. Наверное, у вас возник вопрос, почему на консоли сервисного исполнителя этот файл не упоминается. Дело в том, что в тот момент, когда окно было впервые открыто, оно еще не находилось под контролем сервисного исполнителя, поэтому запрос был отправлен прямо в сеть, минуя исполнителя. Отладка сервисных исполнителей – дело творческое, и это один из подводных камней, о которых следует помнить.

Написав работающий сервисный исполнитель, мы готовы рассказать о предлагаемых ими более продвинутых возможностях.

Продвинутые возможности сервисных исполнителей

Сервисные исполнители предназначены только для выполнения асинхронных операций. Поэтому API `localStorage`, который блокирует работу при чтении и записи, недоступен. Но асинхронный API `indexedDB` доступен. Кроме того, внутри сервисных исполнителей употреблять предложение `await` на верхнем уровне запрещено.

Если говорить о хранении состояния, то чаще всего вы будете использовать `self.caches` и `indexedDB`. Повторим еще раз, что хранить данные в глобальных переменных ненадежно. На самом деле при отладке сервисных исполнителей вы будете сталкиваться с тем, что они иногда останавливаются, и в этот момент нельзя перейти в окно инспектора. В браузерах есть кнопка, позволяющая снова запустить исполнитель и вернуться в инспектор. Но при такой остановке и перезапуске глобальное состояние стирается.

Браузер довольно агрессивно кеширует скрипты сервисных исполнителей. При перезагрузке страницы браузер может отправить запрос на получение скрипта, но если скрипт не изменился, то заменять его не станет. Chrome предлагает возможность активировать обновление скрипта при перезагрузке страницы; для этого перейдите на вкладку **Application** в инспекторе, нажмите **Service Workers**, а затем отметьте флажок **Update on reload** (Обновлять при перезагрузке).

Любой сервисный исполнитель несколько раз меняет состояние с момента запуска до момента, когда его можно использовать. Получить состояние внутри исполнителя позволяет свойство `self.serviceWorker.state`. Ниже приведен список последовательно принимаемых им значений.

parsed
: Это самое первое состояние сервисного исполнителя. В этот момент JavaScript-код файла разобран. Это внутреннее состояние, которое вы, скорее всего, никогда не встретите в своем приложении.

installing
: Установка началась, но еще не завершилась. В каждой версии исполнителя это случается один раз. Исполнитель пребывает в этом состоянии после вызова `oninstall` и до разрешения обещания `event.respondWith()`.

installed
: В этот момент установка завершена. Следующим будет вызван обработчик `onactivate`. Во время тестирования я обнаружил, что сервисные исполнители переходят от состояния `installing` к `activating` так быстро, что состояния `installed` мне ни разу ни удалось заметить.

activating
: Это состояние возникает, когда метод `onactivate` уже вызван, но обещание `event.respondWith()` еще не разрешено.

activated
: Активация завершена, и исполнитель готов к работе. Начиная с этого момента, события `fetch` будут перехватываться.

redundant
: В этот момент загружена более свежая версия скрипта, а предыдущая уже не нужна. Оно также может возникать, если входе загрузки скрипта исполнителя возникла ошибка, если сам скрипт содержит синтаксическую ошибку или если он возбудил исключение.

Подходя философски, можно сказать, что к сервисным исполнителям следует относиться как к форме прогрессивного улучшения. Это означает, что любая веб-страница, использующая их, должна вести себя как обычно, если сервисный исполнитель отсутствует. Это важно, потому что некоторые браузеры могут не поддерживать сервисных исполнителей, потому что на этапе установки может возникнуть ошибка или потому что озабоченные конфиденциальностью пользователи могут вообще отключать их. Иными словами, если ваша единственная цель – добавить в приложение средства многопоточности, то выбирайте какой-нибудь другой тип веб-исполнителей.

Глобальный объект `self`, используемый внутри сервисных исполнителей, – экземпляр класса `ServiceWorkerGlobalScope`. Функция `importScripts()`, доступная в других веб-исполнителях, доступна и в этой среде. Как и в других случаях, веб-исполнителю можно передавать и от него можно получать сообщения. Можно присвоить функцию обработчику `self.onmessage`. Например, ее можно использовать, чтобы приказать сервисному исполнителю сделать кеш недействительным. Передаваемые таким способом сообщения подчиняются всем правилам клонирования, описанным в приложении.

В процессе отладки сервисных исполнителей и запросов, посылаемых браузером, следует помнить о кешировании. Мало того что сервисный исполнитель может реализовывать кеши, управляемые вами программно, так еще и сам браузер занимается стандартным кешированием контента, поступающего из сети. Это означает, что запросы, отправляемые серверу из вашего сервисного исполнителя, не всегда доходят до сервера. Поэтому не забывайте о заголовках `Cache-Control` и `Expires` и задавайте их значения осознанно.

У сервисных исполнителей гораздо больше возможностей, чем мы смогли описать в этом разделе. Mozilla, компания-разработчик Firefox, взяла на себя труд организовать целый сайт с типичными стратегиями построения сервисных исполнителей. Его адрес *https://serviceworke.rs*, и мы рекомендуем заглянуть на него, когда соберетесь реализовать сервисные исполнители в своем следующем веб-приложении.

Сервисные и другие веб-исполнители, о которых мы рассказали, конечно, добавляют сложности. Но, к счастью для нас, существуют библиотеки и паттерны взаимодействия, которые позволяют немного упростить жизнь.

Междокументное взаимодействие

Существуют и другие способы применить многопоточное программирование на JavaScript в браузерах, не привлекая веб-исполнителей. Это можно сделать с помощью взаимодействия между различными браузерными контекстами – как полными страницами, так и IFRAME. Браузеры предоставляют API для этой цели.

Первый способ – встроить IFRAME в веб-страницу или создать всплывающее окно – существовал еще до появления веб-исполнителей. Родительское окно может получить ссылку на дочернее, а затем вызвать метод `.postMessage()` этой ссылки, чтобы отправить сообщение дочернему окну. Дочернее окно в это время прослушивает события `message` объекта `window`. Дочернее окно тоже может передавать сообщения родительскому. Этот паттерн, вероятно, и лег в основу интерфейсов веб-исполнителей.

Второй способ немного универсальнее. Он позволяет взаимодействовать не только со всплывающим окном и IFRAME, но и с любым окном, в которое загружена страница из того же источника. Мало того, он позволяет взаимодействовать и рабочим потокам. Для этого следует создать экземпляр класса `BroadcastChannel`, передав конструктору имя канала в первом аргументе. Созданный канал позволяет взаимодействовать в стиле pub/sub (публикация–подписка). У объекта есть метод `.postMessage()` и свойство `.onmessage`, которому можно присвоить функцию-обработчик. При отправке сообщения будет вызван обработчик во всех объектах, прослушивающих данный канал в различных средах. У объекта также имеется метод `.close()`, позволяющий отключить объект от канала.

Абстракции передачи сообщений

Все рассмотренные в этой главе веб-исполнители раскрывают интерфейс для передачи сообщений в другую среду JavaScript и для приема сообщений из другой среды. Это позволяет создавать приложения, которые могут исполнять JavaScript одновременно на нескольких ядрах.

Однако до сих пор мы имели дело только с простыми искусственными примерами, где передавали простые строки и вызывали простые функции. Когда дело доходит до построения крупных приложений, важно, чтобы и передаваемые сообщения, и код исполнителей допускал масштабирование. А упрощение интерфейса для работы с исполнителями будет способствовать уменьшению числа потенциальных ошибок.

Паттерн RPC

До сих пор мы передавали исполнителям только простые строки. Это, конечно, позволяет получить представление о возможностях веб-исполнителей, но для полноценного приложения маловато.

Предположим, к примеру, что имеется веб-исполнитель, который делает что-то одно, например вычисляет сумму квадратных корней из чисел от 1 до 1 000 000. Можно было бы просто вызвать метод исполнителя `postMessage()` без аргументов, затем выполнить длительную операцию в обработчике `onmessage` и отправить обратно сообщение с помощью функции исполнителя `postMessage()`. Но что, если исполнитель должен также вычислять последовательность чисел Фибоначчи? В таком случае можно было бы передать строку: `square_sum` для суммы корней и `fibonacci` для чисел Фибоначчи. А если нужны аргументы? Хорошо, передадим строку `square_sum|1000000`. А если у аргументов есть типы? Ладно, подойдет что-то вроде `square_sum|num:1000000`. В общем, вы видите, к чему все идет.

Паттерн RPC (Remote Procedure Call – удаленный вызов процедур) дает способ сериализовать функцию со всеми ее аргументами и передать удаленному узлу для выполнения. Строка `square_sum|num:1000000` – на самом деле форма RPC, которую мы изобрели заново. Возможно, эта строка в конечном итоге транслировалась бы в вызов `squareNum(1000000)`, как описано в разделе «Паттерн Диспетчер команд» ниже.

Есть еще одна сложность, о которой приложению нужно помнить. Если главный поток посылает веб-исполнителю по одному сообщению за раз, то, когда от исполнителя придет ответ, он будет знать, на какое сообщение. Но если веб-исполнителю одновременно посылается несколько сообщений, то сопоставить их с ответами не так просто. Например, представьте приложение, которое посылает веб-исполнителю два сообщения и получает два ответа.

```
worker.postMessage('square_sum|num:4');
worker.postMessage('fibonacci|num:33');
worker.onmessage = (result) => {
  // Какой результат к какому сообщению относится?
  // '3524578'
  // 4.1462643
};
```

По счастью, существует стандарт передачи сообщений, подсказанный паттерном RPC. Этот стандарт называется JSON-RPC (https://www.jsonrpc.org/), и реализовать его совсем несложно. Он определяет JSON-представления запроса и ответа в виде объектов «уведомлений». Так можно определить вызываемый метод и его аргументы в запросе, результат в ответе и механизм сопоставления запросов и ответов. Поддерживаются даже ошибки и пакеты запросов. Но в этом примере нам нужны будут только запрос и ответ.

Представление двух вызовов функций из примера выше в формате JSON-RPC могло бы выглядеть так:

```
// worker.postMessage
{"jsonrpc": "2.0", "method": "square_sum", "params": [4], "id": 1}
{"jsonrpc": "2.0", "method": "fibonacci", "params": [33], "id": 2}

// worker.onmessage
{"jsonrpc": "2.0", "result": "3524578", "id": 2}
{"jsonrpc": "2.0", "result": 4.1462643, "id": 1}
```

В этом случае налицо очевидная корреляция между ответами и соответствующими запросами.

JSON-RPC предполагает использование JSON для кодирования сериализованных сообщений, особенно при передаче их по сети. Поля `jsonrpc` каждой записи определяют версию JSON-RPC, которой отвечают сообщения; это очень важно в случае сетевых взаимодействий. Однако, поскольку веб-исполнители пользуются алгоритмом клонирования (см. приложение), который позволяет передавать JSON-совместимые объекты, приложение могло бы просто передавать объекты напрямую, не тратя ресурсы на сериализацию и десериализацию. Кроме того, поля `jsonrpc` не так важны в браузере, когда у вас имеется полный контроль над обеими сторонами коммуникационного канала.

Имея поля `id` для корреляции запросов и ответов, можно разобраться, что чему соответствует. Мы реализуем такое решение в разделе «Соберем все вместе» ниже. Но сначала нужно определить, какую функцию вызывать при получении сообщения.

Паттерн Диспетчер команд

Хотя паттерн RPC полезен для определения протоколов, он необязательно дает механизм для решения о том, по какому пути следовать на принимающей стороне. Эту проблему решает паттерн Диспетчер команд, который, получив сериализованную команду, находит подходящую функцию и выполняет ее, возможно, передавая аргументы.

Реализовать этот паттерн довольно просто, никакой магии не требуется. Прежде всего мы можем предположить, что имеются две переменные, содержащие необходимую информацию о методе или *команде*, которую должна выполнить программа. Первая переменная называется `method` и является строкой, вторая – `args` – является массивом значений, передаваемых методу. Будем предполагать, что обе они получены от слоя приложения, реализующего RPC.

Код, который в конечном итоге нужно выполнить, может находиться в других частях приложения. Например, функция вычисления суммы квадратных корней может быть взята из сторонней библиотеки, а код нахождения чисел Фибоначчи объявлен где-то в локальной области видимости. Но вне зависимости от того, где находится код, мы хотим иметь единый репозиторий, который отображает команды на соответствующий им код. Решить эту задачу можно несколькими способами, например воспользовавшись объектом `Map`, но, поскольку команды предполагаются статичными, достаточно и скромного объекта JavaScript.

Еще одна важная идея заключается в том, что выполнять можно только заранее определенные команды. Если вызывающая сторона захочет обратиться к несуществующему методу, то ей должна быть возвращена ошибка, причем так, чтобы не вызвать краха веб-исполнителя. И хотя аргументы можно передавать методу в виде массива, интерфейс получится гораздо симпатичнее, если развернуть массив в нормальные аргументы функции.

В примере 2.11 приведен пример реализации диспетчера команд, который вы можете использовать в собственных приложениях.

Пример 2.11 ❖ Пример диспетчера команд

```
const commands = { ❶
  square_sum(max) {
    let sum = 0;
    for (let i = 0; i < max; i++) sum += Math.sqrt(i);
    return sum;
  },
  fibonacci(limit) {
    let prev = 1n, next = 0n, swap;
    while (limit) {
  swap = prev; prev = prev + next;
  next = swap; limit--;
    }
    return String(next);
  }
};
function dispatch(method, args) {
  if (commands.hasOwnProperty(method)) { ❷
    return commands[method](...args); ❸
  }
  throw new TypeError(`Команда ${method} не определена!`);
}
```

❶ **Определение всех поддерживаемых команд.**
❷ **Проверить, существует ли команда.**
❸ **Распаковать массив аргументов и вызвать метод.**

Здесь определен объект `commands`, содержащий полный перечень команд, поддерживаемых диспетчером. В данном случае код команд находится прямо в списке, но вполне возможно и даже рекомендуется размещать его где-то в другом месте.

Функция `dispatch()` принимает два аргумента: имя метода и массив аргументов. Ее можно вызывать, когда веб-исполнитель получает RPC-сообщение, представляющее команду. Первым делом функция проверяет, существует ли метод, для чего вызывает функцию `commands.hasOwnProperty()`. Это гораздо безопаснее, чем пользоваться конструкциями `method in commands` и даже `commands[method]`, потому что нас не интересуют свойства, не относящиеся к командам, например `__proto__`.

Если команда существует, то переданный массив аргументов распаковывается: первый элемент становится первым аргументом и т. д. Затем вы-

зывается функция с этими аргументами, а результат вызова возвращается вызывающей стороне. Если же команды не существует, то возбуждается исключение `TypeError`.

Это и есть полная реализация простого диспетчера команд. Более продвинутые диспетчеры могут, например, проверять типы, т. е. убеждаться, что аргументы имеют определенный примитивный тип или предписанную форму, и в случае нарушения возбуждать исключения обобщенно, так чтобы методу команды не нужно было думать об этом.

Эти два паттерна определенно помогают писать приложения, но интерфейс можно упростить еще больше.

Соберем все вместе

В JavaScript-приложениях часто возникает необходимость выполнить работу вне сервиса. Например, иногда требуется обратиться к базе данных или отправить HTTP-запрос. В таком случае приходится дожидаться ответа. В идеале хотелось бы предоставить обратный вызов или рассматривать такое внешнее обращение, как обещание. И хотя интерфейс веб-исполнителя для обмена сообщениями не предусматривает выполнения такого рода вещей напрямую, мы, безусловно, можем сделать это самостоятельно.

Было бы также хорошо иметь более симметричный интерфейс внутри веб-исполнителя, например воспользовавшись асинхронной функцией, когда полученное в результате разрешения обещания значение автоматически отправляется вызывающей среде, без необходимости вручную вызывать `postMessage()`.

В этом разделе мы проделаем все это. Мы объединим паттерны RPC и Диспетчер команд и построим интерфейс, который сделает работу с веб-исполнителями похожей на работу с другими внешними библиотеками, к которым вы, наверное, привыкли. Мы будем использовать выделенный исполнитель, но все сказанное применимо также к разделяемому и сервисному исполнителю.

Прежде всего создайте новый каталог *ch2-patterns/*. Поместите в него HTML-файл *index.html*, содержащий код из примера 2.12.

Пример 2.12 ❖ *ch2-patterns/index.html*

```
<html>
  <head>
    <title>Worker Patterns</title>
    <script src="rpc-worker.js"></script>
    <script src="main.js"></script>
  </head>
</html>
```

На этот раз загружается два JavaScript-файла. Первый – новая библиотека, а второй – главный файл, к созданию которого мы сейчас приступим. Назовите файл *main.js* и скопируйте в него код из примера 2.13.

Пример 2.13 ❖ *ch2-patterns/main.js*

```
const worker = new RpcWorker('worker.js');

Promise.allSettled([
  worker.exec('square_sum', 1_000_000),
  worker.exec('fibonacci', 1_000),
  worker.exec('fake_method'),
  worker.exec('bad'),
]).then(([square_sum, fibonacci, fake, bad]) => {
  console.log('square sum', square_sum);
  console.log('fibonacci', fibonacci);
  console.log('fake', fake);
  console.log('bad', bad);
});
```

Этот файл представляет код приложения, в котором используются новые паттерны проектирования. Сначала создается экземпляр исполнителя, но не путем прямого вызова конструктора одного из классов веб-исполнителей. Вместо этого мы создаем экземпляр нового класса `RpcWorker`, который скоро определим.

Затем с помощью метода `worker.exec` вызываются четыре разных RPC-метода: `square_sum`, `fibonacci`, несуществующий метод `fake_method` и метод с ошибкой `bad`. Первый аргумент – имя метода, а все последующие – аргументы, передаваемые этому методу.

Метод `exec` возвращает обещание, которое разрешается успешно, если операция выполняется без ошибок, и отвергается в случае ошибки. А раз так, то все обещания обертываются одним вызовом `Promise.allSettled()`. В результате программа продолжит работу, когда будут разрешены все обещания – успешно или нет. После этого печатается результат каждой операции. В состав результата `allSettled()` входит массив объектов, имеющих строковое свойство `status`, а также одно из свойств `value` или `reason` в зависимости от того, закончилась операция успешно или неудачно.

Далее создайте файл *rpc-worker.js* и включите в него код из примера 2.14.

Пример 2.14 ❖ *ch2-patterns/rpc-worker.js (часть 1)*

```
class RpcWorker {
  constructor(path) {
    this.next_command_id = 0;
    this.in_flight_commands = new Map();
    this.worker = new Worker(path);
    this.worker.onmessage = this.onMessageHandler.bind(this);
  }
```

Эта часть файла начинается классом `RpcWorker` и определением его конструктора. Внутри конструктора инициализируется несколько свойств. Сначала свойству `next_command_id` присваивается значение 0. Оно будет использоваться для последовательного присваивания сообщениям идентификаторов в стиле JSON-RPC, которые нужны для сопоставления запросов и ответов.

Затем свойство `in_flight_commands` инициализируется пустым объектом `Map`. Это отображение будет содержать записи, индексированные идентификатором команды и содержащие функции разрешения и отмены обещания. Размер отображения растет, когда исполнителю параллельно отправляются сообщения, и уменьшается, когда возвращаются ответы.

Затем инициализируется выделенный исполнитель и присваивается свойству `worker`. Этот класс эффективно инкапсулирует экземпляр `Worker`. После этого обработчик `onmessage` исполнителя настраивается так, чтобы вызывался метод класса `onMessageHandler` (определенный в следующем фрагменте кода). Класс `RpcWorker` не расширяет `Worker`, потому что мы хотим не расширить функциональность веб-исполнителя, а создать совершенно новый интерфейс.

Добавьте в файл код из примера 2.15.

Пример 2.15 ❖ *ch2-patterns/rpc-worker.js (часть 2)*

```
onMessageHandler(msg) {
  const { result, error, id } = msg.data;
  const { resolve, reject } = this.in_flight_commands.get(id);
  this.in_flight_commands.delete(id);
  if (error) reject(error);
  else resolve(result);
}
```

Здесь определяется метод `onMessageHandler`, который вызывается, когда выделенный исполнитель отправляет сообщение. Предполагается, что сообщение в стиле JSON-RPC передается от веб-исполнителя вызывающей среде, поэтому первым делом из него извлекаются поля `result`, `error` и `id`.

Затем метод находит значение `id` в отображении `in_flight_commands`, извлекает из найденной записи функции разрешения и отмены обещания и удаляет запись из отображения. Если было получено значение `error`, то считается, что операция завершилась неудачно, и вызывается функция `reject()`, которой передается ошибка. В противном случае вызывается функция `resolve()`, которой передается результат операции. Заметим, что возбуждение false-подобных исключений не поддерживается.

Чтобы библиотеку можно было использовать в реальных приложениях, хорошо бы поддержать тайм-аут операций. Теоретически возможно, что исполнитель и обещание никогда не исполнит, и ошибку не вернет, а вызывающая среда хотела бы в таком случае отменить обещание и удалить данные из отображения. В противном случае в приложении может возникнуть утечка памяти.

Наконец, добавьте в файл последний фрагмент из примера 2.16.

Пример 2.16 ❖ *ch2-patterns/rpc-worker.js (часть 3)*

```
exec(method, ...args) {
  const id = ++this.next_command_id;
  let resolve, reject;
  const promise = new Promise((res, rej) => {
```

```
      resolve = res;
      reject = rej;
    });
    this.in_flight_commands.set(id, { resolve, reject });
    this.worker.postMessage({ method, params: args, id });
    return promise;
  }
}
```

В этом, последнем, фрагменте определен метод `exec()`, который вызывается, когда приложение хочет выполнить метод веб-исполнителя. Первым делом генерируется новый идентификатор `id`. Затем создается обещание, которое позже будет возвращено методом. Функции `reject` и `resolve` для этого обещания вынесены и помещены в запись отображения `in_flight_commands`, ассоциированную с данным значением `id`.

Затем сообщение отправляется исполнителю. Объект, передаваемый исполнителю, примерно соответствует форме, принятой в JSON-RPC. Он имеет свойство `method`, свойство `params`, являющееся массивом, содержащим остальные аргументы, и свойство `id`, сгенерированное для данного выполнения команды.

Это довольно распространенный паттерн, полезный, когда нужно сопоставить исходящие и входящие асинхронные сообщения. Быть может, вы сами реализовывали что-то подобное, например когда нужно было поместить сообщение в очередь для сети, а впоследствии получить его. Но, повторим еще раз, при этом потребляется дополнительная память.

С RPC-исполнителем мы разобрались, осталось создать последний файл. Назовите его *worker.js* и включите код из примера 2.17.

Пример 2.17 ❖ *ch2-patterns/worker.js*

```
const sleep = (ms) => new Promise((res) => setTimeout(res, ms)); ❶

function asyncOnMessageWrap(fn) { ❷
  return async function(msg) {
    postMessage(await fn(msg.data));
  }
}

const commands = {
  async square_sum(max) {
    await sleep(Math.random() * 100); ❸
    let sum = 0; for (let i = 0; i < max; i++) sum += Math.sqrt(i);
    return sum;
  },
  async fibonacci(limit) {
    await sleep(Math.random() * 100);
    let prev = 1n, next = 0n, swap;
    while (limit) { swap = prev; prev = prev + next; next = swap; limit--; }
    return String(next); ❹
  },
  async bad() {
```

```
    await sleep(Math.random() * 10);
    throw new Error('oh no');
  }
};

self.onmessage = asyncOnMessageWrap(async (rpc) => { ❺
  const { method, params, id } = rpc;

  if (commands.hasOwnProperty(method)) {
    try {
      const result = await commands[method](...params);
      return { id, result }; ❻
    } catch (err) {
      return { id, error: { code: -32000, message: err.message }};
    }
  } else {
    return { ❼
      id, error: {
        code: -32601,
        message: `method ${method} not found`
      }
    };
  }
});
```

❶ Искусственно замедлить методы.
❷ Простая обертка, превращающая `onmessage` в асинхронную функцию.
❸ Искусственная случайная задержка команд.
❹ Результат типа `BigInt` преобразуется в допустимую JSON-строку.
❺ Включается обертка `onmessage`.
❻ Разрешение успешно обработанного сообщения JSON-RPC.
❼ Отмена ошибочного сообщения JSON-RPC в случае, когда метода не существует.

В этом файле много интересного. Прежде всего функция `sleep` – это просто обещание, эквивалентное `setTimeout()`. Функция `asyncOnMessageWrap()` оборачивает переданную ей функцию, превращая ее в асинхронную, результат может быть присвоен обработчику `onmessage`. Это удобно, когда нужно извлечь свойство, содержащее данные, из входящего сообщения, передать его функции, дождаться результата, а затем передать результат `postMessage()`.

После этого на сцену возвращается объект `commands`. Но теперь мы добавили искусственные задержки, а функции сделали асинхронными. Это позволяет эмулировать медленные асинхронные методы.

Наконец, обработчику `onmessage` присваивается обернутая функция. Он принимает входящее сообщение JSON-RPC и извлекает из него свойства `method`, `params` и `id`. Как и раньше, проверяется, есть ли метод в коллекции команд. Если нет, то возвращается ошибка в стиле JSON-RPC. Значение `-32601` определено в JSON-RPC как код, означающий отсутствие метода. Если же команда существует, то выполняется ее метод, разрешенное значение обещания преобразуется в сообщение JSON-RPC об успехе и возвращается. Если команда возбуждает исключение, то возвращается код ошибки `-32000`, тоже определенный в JSON-RPC.

Создав файл, перейдите в браузер и откройте инспектор. Затем снова запустите веб-сервер, находясь в каталоге *ch2-patterns/*:

```
$ npx serve .
```

Вернитесь в браузер и скопируйте полученный URL-адрес. На странице вы не увидите ничего интересного, но на консоли должны появиться следующие сообщения:

```
square sum  { status: "fulfilled", value: 666666166.4588418 }
fibonacci   { status: "fulfilled", value: "4346655768..." }
fake        { status: "rejected", reason: { code: -32601,
              message: "method fake_method not found" } }
bad         { status: "rejected", reason: { code: -32000,
              message: "oh no" } }
```

Мы видим, что вызовы команд `square_sum` и `fibonacci` завершились успешно, а команда `fake_method` неудачно. Но важнее то, что, хотя методы разрешались не в том порядке, в котором вызывались, ответы – благодаря идентификаторам `id` – правильно сопоставлены запросам.

Глава 3

Node.js

Не считая браузеров, есть только одна достойная упоминания среда выполнения JavaScript – *Node.js*[1]. Начиналась она как платформа, популяризирующая однопоточную конкурентность на серверах с обратными вызовами в стиле продолжений, но со временем было приложено много усилий для превращения ее в универсальную программную платформу.

Многие задачи, решаемые программами для Node.js, не укладываются в категорию традиционного обслуживания веб-запросов или обработки сетевых подключений. Современные программы часто являются командными инструментами, составляющими системы сборки для JavaScript или их части. Для них характерен большой объем операций ввода–вывода, как и для серверов, но, кроме того, они активно занимаются обработкой данных.

Например, инструменты Babel (https://babeljs.io/) и TypeScript (https://www.typescriptlang.org/) транслируют код с одного языка (или версии языка) на другой. А инструменты типа Webpack (https://webpack.js.org/), Rollup (https://rollupjs.org/) и Parcel (https://parceljs.org/) упаковывают и минимизируют объем кода для распространения в системы, где время загрузки критично, например клиентскую часть приложения или бессерверную среду. Когда программа активно выполняет операции файлового ввода–вывода, ей приходится не менее активно обрабатывать данные, и обычно это делается синхронно. Но существуют ситуации, когда параллелизм может заметно ускорить работу.

Параллелизм полезен и на серверах, для чего Node.js первоначально и разрабатывался. Обработка данных может составлять значительную часть приложения. Например, *отрисовка на стороне сервера* (server side rendering – *SSR*) подразумевает большой объем операций со строками, когда исходные данные уже известны. Это один из многих примеров, когда имеет смысл включить в программу параллелизм. В разделе «Когда стоит использовать» главы 8 рассматривается ситуация, в которой параллелизм ускоряет отрисовку шаблона.

---

[1] Да, есть и другие не браузерные среды выполнения JavaScript, например Deno, но на момент написания книги Node.js завоевал такую популярность и долю рынка, что только о нем и имеет смысл говорить. Возможно, к моменту, когда вы будете читать книгу, ситуация изменится – и для мира JavaScript это будет прекрасно! Надеемся, что когда-нибудь выйдет и новое издание этой книги, где будет рассмотрена ваша любимая не браузерная среда выполнения JavaScript.

В настоящее время для распараллеливания кода в нашем распоряжении имеются рабочие потоки. Так было не всегда, но это не значит, что мы были ограничены только однопоточной конкурентностью.

Что было до потоков

Пока в Node.js не появились потоки, для того чтобы задействовать все имеющиеся процессорные ядра, нужно было использовать процессы. В главе 1 мы говорили, что при работе с процессами мы лишены некоторых преимуществ потоков. Но если разделение памяти не особенно важно (а так оно часто и бывает!), то процессы вполне способы решить стоящие перед нами задачи.

Рассмотрим рис. 1.1. В этом случае потоки отвечают на HTTP-запросы, передаваемые им главным потоком, который прослушивает порт. Хотя такая организация отлично подходит для обработки трафика на нескольких процессорных ядрах, того же результата можно добиться и с помощью процессов. Выглядеть это может, как показано на рис. 3.1.

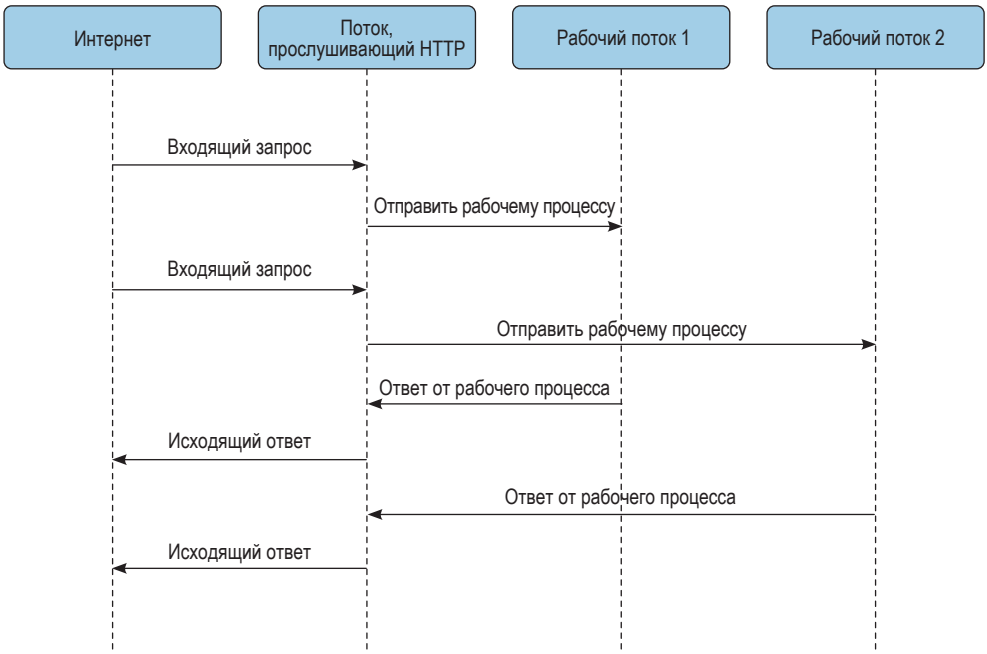


Рис. 3.1 ❖ Возможное использование процессов в HTTP-сервере

Хотя нечто подобное можно сделать с помощью API `child_process` в Node.js, лучше использовать модуль `cluster`, специально предназначенный для этой цели. Он распределяет трафик между несколькими рабочими процессами. Посмотрим, как он используется, на простом примере «Hello, World».

Код в примере 3.1 – стандартный HTTP-сервер в Node.js. Он просто отвечает на любой запрос фразой «Hello, World!», завершающейся символом новой строки.

Пример 3.1 ❖ Сервер «Hello, World» в Node.js

```
const http = require('http');

http.createServer((req, res) => {
  res.end('Hello, World!\n');
}).listen(3000);
```

Теперь добавим четыре процесса с помощью модуля `cluster`. Обычно при использовании `cluster` где-то в начале находится предложение `if`, в котором проверяется, является ли текущий процесс главным прослушивателем или рабочим. Если мы находимся в главном процессе, то должны запустить рабочие процессы. А в противном случае мы просто подготавливаем в каждом рабочем процессе обычный веб-сервер, как в примере выше. Выглядит это, как показано в примере 3.2.

Пример 3.2 ❖ Сервер «Hello, World» в Node.js с использованием `cluster`

```
const http = require('http');
const cluster = require('cluster'); ❶

if (cluster.isPrimary) { ❷
  cluster.fork(); ❸
  cluster.fork();
  cluster.fork();
  cluster.fork();
} else {
  http.createServer((req, res) => {
    res.end('Hello, World!\n');
  }).listen(3000); ❹
}
```

❶ Затребовать модуль `cluster`.
❷ Путь выполнения зависит от того, главный это процесс или рабочий.
❸ В главном процессе создать четыре рабочих процесса.
❹ В рабочих процессах создать веб-сервер и начать прослушивание, как в примере 3.1.

Вы, наверное, заметили, что мы создали веб-серверы, прослушивающие один и тот же порт в четырех разных процессах. На первый взгляд, это ошибка. Ведь попытка привязать сервер к уже используемому порту обычно заканчивается ошибкой. Но не волнуйтесь! На самом деле мы не прослушиваем один порт четыре раза. Node.js в модуле `cluster` вершит для нас волшебство.

Когда в кластере настраиваются рабочие процессы, любой вызов `listen()` на самом деле приводит к тому, что порт прослушивает главный, а не рабочий процесс. А установив соединение, главный процесс передает его рабочему посредством механизма IPC. В большинстве систем рабочие процессы перебираются циклически. То есть в этой несколько запутанной схеме только

кажется, что все рабочие процессы прослушивают один порт, тогда как в действительности порт прослушивает лишь один главный процесс.

Раньше свойство `isPrimary` объекта `cluster` называлось `isMaster`, и ради совместимости этот псевдоним все еще действует. Изменение было внесено в версию Node.js 16.0.0 исходя из соображений толерантности. Проект хочет сделать сообщество гостеприимным, а слова, которые исторически ассоциировались с рабством, мешают этому.

С процессами связаны дополнительные накладные расходы, которых нет у потоков, и, кроме того, мы не получаем разделяемой памяти, которая ускоряет обмен данными. Поэтому нам и нужен модуль `worker_threads`.

Модуль worker_threads

Поддержкой потоков в Node.js занимается встроенный модуль `worker_threads`. Он предоставляет интерфейс к потокам, во многом повторяющий тот, что имеется в браузерах для веб-исполнителей. Поскольку Node.js все-таки не браузер, API не совсем одинаковы, да и среда внутри рабочих потоков не совсем такая же, как внутри веб-исполнителей.

Вместо этого внутри рабочих потоков Node.js вы обнаружите обычный Node.js API, доступный с помощью предложения `require` или `import`, если вы используете ESM. Но по сравнению с главным потоком имеется несколько отличий в API:

- невозможно выйти из программы, вызвав `process.exit()`. Так вы выйдете только из потока;
- невозможно изменить рабочий каталог, вызвав `process.chdir()`. Эта функция даже и недоступна;
- невозможно обрабатывать сигналы с помощью `process.on()`.

Важно также иметь в виду, что пул рабочих потоков `libuv` разделяется между потоками. Вспомните раздел «Скрытые потоки» главы 1, где было сказано, что пул `libuv` по умолчанию состоит из четырех потоков и предназначен для организации неблокирующих интерфейсов к низкоуровневым блокирующим API. Если вам покажется, что этого мало (например, потому что программа очень интенсивно обращается к файловому вводу–выводу), и вы захотите добавить еще потоки с помощью модуля `worker_threads`, то может оказаться, что нагрузка не снизилась. Тогда, помимо различных видов кеширования и других приемов оптимизации, попробуйте увеличить переменную `UV_THREAD_POOL_SIZE`. Может также случиться, что это придется сделать даже при добавлении рабочих потоков JavaScript посредством модуля `worker_threads`, потому что они активно пользуются потоками из пула `libuv`.

Есть и другие подводные камни, так что настоятельно рекомендуем заглянуть в документацию по Node.js (https://nodejs.org/dist/latest/docs/api/worker_threads.html# worker_threads_class_worker), где описаны все отличия для конкретной версии Node.js.

Новый рабочий поток можно создать с помощью конструктора `Worker`, как в примере 3.3.

Пример 3.3 ❖ Запуск нового рабочего потока в Node.js

```
const { Worker } = require('worker_threads');

const worker = new Worker('/path/to/worker-file-name.js'); ❶
```

❶ **Здесь указывается имя файла – точки входа, который будет выполняться в рабочем потоке. Это аналог главной точки входа – файла, который указывается в параметре командной строки `node`.**

workerData

Создать рабочий поток недостаточно. С ним нужно еще и взаимодействовать! Конструктор класса `Worker` принимает второй аргумент, объект `options`, который в числе прочего позволяет задать данные, сразу передаваемые рабочему потоку. Содержимое свойства `workerData` этого объекта копируется в рабочий поток средствами, описанными в приложении. Внутри потока мы можем обратиться к клонированным данным с помощью свойства `workerData` модуля `worker_threads`. Как это работает, показано в примере 3.4.

Пример 3.4 ❖ Передача данных рабочему потоку посредством `workerData`

```
const {
  Worker,
  isMainThread,
  workerData
} = require('worker_threads');
const assert = require('assert');

if (isMainThread) { ❶
  const worker = new Worker(__filename, { workerData: { num: 42 } });
} else {
  assert.strictEqual(workerData.num, 42);
}
```

❶ **Вместо того чтобы заводить отдельный файл для рабочего потока, мы можем использовать текущий файл, передав имя `__filename`, и ветвиться в зависимости от `isMainThread`.**

Отметим, что свойства объекта `workerData` *клонируются*, а не разделяются между всеми потоками. В отличие от C, разделяемая память в JavaScript еще не означает, что все переменные видны. На самом деле любые изменения, произведенные в этом объекте, не будут видны из других потоков. У них свои объекты. При всем при том разделять память между потоками можно, для этого служит класс `SharedArrayBuffer`. Разделять данные можно через `workerData` или отправив их в порт `MessagePort`, рассматриваемый в следующем разделе. Сам же класс `SharedArrayBuffer` подробно описан в главе 4.

MessagePort

`MessagePort` — это одна сторона двустороннего потока данных. По умолчанию он предоставляется любому рабочему потоку для организации коммуника-

ционного канала с главным потоком. В рабочем потоке он доступен через свойство `parentPort` модуля `worker_threads`.

Чтобы отправить сообщение через порт, нужно вызвать метод `postMessage()` последнего. Первым аргументом является любой объект, который можно передать, как описано в приложении. Этот объект станет данными сообщения на другой стороне порта. Когда сообщение приходит в порт, генерируется событие `message`, а данные сообщения становятся первым аргументом функции-обработчика. В главном потоке событие и метод `postMessage()` являются атрибутами самого экземпляра рабочего потока, их не нужно получать от экземпляра `MessagePort`. В примере 3.5 показано, как сообщения, посылаемые главному потоку, эхом возвращаются рабочему потоку.

Пример 3.5 ❖ Двустороннее взаимодействие посредством объектов MessagePort по умолчанию

```
const {
  Worker,
  isMainThread,
  parentPort
} = require('worker_threads');

if (isMainThread) {
  const worker = new Worker(__filename);
  worker.on('message', msg => {
    worker.postMessage(msg);
  });
} else {
  parentPort.on('message', msg => {
    console.log('Мы получили сообщение от главного потока:', msg);
  });
  parentPort.postMessage('Hello, World!');
}
```

Можно также с помощью конструктора `MessageChannel` создать пару объектов `MessagePort`, соединенных друг с другом. Затем один из портов можно передать через существующий порт сообщений (например, предоставленный по умолчанию) или в составе `workerData`. Потребность в этом может возникнуть, когда ни один из участвующих во взаимодействии потоков не является главным или просто по организационным причинам. В примере 3.6 делается то же, что в предыдущем, только порты создаются с помощью `MessageChannel` и передаются посредством `workerData`.

Пример 3.6 ❖ Двустороннее взаимодействие через `MessagePort`, созданный с помощью `MessageChannel`

```
const {
  Worker,
  isMainThread,
  MessageChannel,
  workerData
} = require('worker_threads');
```

```
if (isMainThread) {
  const { port1, port2 } = new MessageChannel();
  const worker = new Worker(__filename, {
    workerData: {
      port: port2
    },
    transferList: [port2]
  });
  port1.on('message', msg => {
    port1.postMessage(msg);
  });
} else {
  const { port } = workerData;
  port.on('message', msg => {
    console.log('Мы получили сообщение от главного потока:', msg);
  });
  port.postMessage('Hello, World!');
}
```

Обратите внимание на параметр `transferList` при создании экземпляра `Worker`. Так передается владение объектами от одного потока другому. Это необходимо при отправке объектов `MessagePort`, `ArrayBuffer` или `FileHandle` с помощью `workerData` или `postMessage`. После того как объект передан, он уже не должен использоваться на передающей стороне.

В последних версиях Node.js доступны классы `ReadableStream` и `WritableStream` от группы Web Hypertext Application Technology Working Group (WHATWG). Узнать о них можно из документации по Node.js (https://nodejs.org/api/webstreams.html) и из примеров использования в некоторых API. Такие объекты можно передавать в составе `transferList` через `MessagePort`, чтобы организовать другой способ взаимодействия между потоками. Под капотом передача данных по ним реализована с помощью `MessagePort`.

И снова Happycoin

Познакомившись с тем, как запускаются потоки в Node.js и как устроено взаимодействие между ними, мы можем заново реализовать пример из раздела «Потоки на C: обогатитесь с помощью криптовалюты Happycoin» главы 1 – на сей раз в Node.js.

Напомним, что Happycoin – воображаемая криптовалюта с совершенно нелепым алгоритмом доказательства выполнения работы.

1. Сгенерировать случайное 64-разрядное целое число без знака.
2. Определить, является ли это число счастливым.
3. Если нет, это не Happycoin.
4. Если оно не делится на 10 000, это не Happycoin.
5. В противном случае это Happycoin.

Как и на C, сначала напишем однопоточную версию, а затем сделаем код многопоточным.

С одним главным потоком

Начнем с генерирования случайных чисел. Создайте файл *happycoin.js* в каталоге *ch3-happycoin/* и поместите в него код из примера 3.7.

Пример 3.7 ❖ *ch3-happycoin/happycoin.js*

```
const crypto = require('crypto');

const big64arr = new BigUint64Array(1)
function random64() {
  crypto.randomFillSync(big64arr);
  return big64arr[0];
}
```

Модуль `crypto` в Node.js содержит функции для генерирования криптографически стойких случайных чисел. Без него никак не обойтись, раз уж мы создаем криптовалюту! По счастью, работать с ним не так трудно, как в C.

Функция `randomFillSync` заполняет переданный массив типа `TypedArray` случайными данными. Поскольку нам нужно всего одно 64-разрядное целое без знака, можно использовать массив типа `BigUint64Array`. Этот частный случай `TypedArray` наряду с `BigInt64Array` – недавние добавления в JavaScript, ставшие возможными благодаря появлению нового типа `bigint` для хранения сколь угодно больших целых чисел. Взяв первый (и единственный) элемент заполненного массива, мы получаем искомое 64-разрядное целое без знака.

Теперь добавим вычисление счастливого числа. Включите в файл код из примера 3.8.

Пример 3.8 ❖ *ch3-happycoin/happycoin.js*

```
function sumDigitsSquared(num) {
  let total = 0n;
  while (num > 0) {
    const numModBase = num % 10n;
    total += numModBase ** 2n;
    num = num / 10n;
  }
  return total;
}

function isHappy(num) {
  while (num != 1n && num != 4n) {
    num = sumDigitsSquared(num);
  }
  return num === 1n;
}

function isHappycoin(num) {
  return isHappy(num) && num % 10000n === 0n;
}
```

Три функции, `sumDigitsSquared`, `isHappy` и `isHappycoin`, – результат прямой трансляции их прототипов на C, приведенных в главе 1. Если вы не знакомы с типом `bigint`, обратите внимание на суффикс `n` в конце всех числовых литералов. Он говорит JavaScript, что эти числа следует рассматривать как значения типа `bigint`, а не типа `number`. Это важно, так как оба типа поддерживают математические операторы `+`, `-`, `**` и др., поэтому не могут употребляться в одном выражении без явного преобразования типа. Например, выражение `1 + 1n` недопустимо, потому что мы пытаемся сложить число 1 типа `number` с числом 1 типа `bigint`.

И в конец файла добавьте цикл поиска и вывод количества найденных Happycoin'ов (пример 3.9).

Пример 3.9 ❖ *ch3-happycoin/happycoin.js*

```
let count = 0;
for (let i = 1; i < 10_000_000; i++) {
  const randomNum = random64();
  if (isHappycoin(randomNum)) {
    process.stdout.write(randomNum.toString() + ' ');
    count++;
  }
}

process.stdout.write('\ncount ' + count + '\n');
```

Код почти такой же, как ранее написанный на C. Цикл из 10 000 000 итераций, на каждой итерации получаем случайное число и проверяем, является ли оно Happycoin. Если да, печатаем. Отметим, что здесь мы не используем `console.log()`, потому что не хотим вставлять символ новой строки после каждого найденного числа. Вместо этого мы разделяем числа пробелами, поэтому пишем непосредственно в выходной поток. По завершении цикла нужно напечатать счетчик, поэтому добавляем перед ним символ новой строки, чтобы отделить от чисел.

Для запуска программы выполните следующую команду, находясь в каталоге *ch3-happycoin*:

```
$ node happycoin.js
```

Результат должен быть таким же, как в программе на C, т. е. выглядеть как-то так:

```
5503819098300300000 ... [ еще 125 чисел ] ... 5273033273820010000
count 127
```

Занимает это существенно больше времени, чем на C. На нашем стандартном компьютере с Node.js v16.0.0 программа работала 1 мин 45 с.

Для такой медленной работы есть много причин. При разработке и оптимизации приложений важно понимать, каковы источники накладных расходов. Да, вообще говоря, JavaScript часто «медленнее C», но такую огромную разницу только этим не объяснишь. Да, в следующем разделе мы улучшим

производительность, разбив работу на несколько потоков, но, как вы увидите, этого далеко не достаточно, чтобы сравниться с кодом на C.

И на этом замечании остановимся и посмотрим, что получится, если использовать `worker_threads` для разделения нагрузки на части.

С четырьмя потоками

Для добавления рабочих потоков возьмем за отправную точку уже написанный код. Скопируйте файл *happycoin.js* в *happycoin-threads.js*. Затем вставьте в начало файла код из примера 3.10.

Пример 3.10 ❖ *ch3-happycoin/happycoin-threads.js*

```
const {
  Worker,
  isMainThread,
  parentPort
} = require('worker_threads');
```

Нам понадобятся эти части из модуля `worker_threads`, поэтому загрузим их в самом начале с помощью `require`. Теперь замените кодом из примера 3.11 все, начиная с предложения `let count = 0;` и до конца файла.

Пример 3.11 ❖ *ch3-happycoin/happycoin-threads.js*

```
const THREAD_COUNT = 4;

if (isMainThread) {
  let inFlight = THREAD_COUNT;
  let count = 0;
  for (let i = 0; i < THREAD_COUNT; i++) {
    const worker = new Worker(__filename);
    worker.on('message', msg => {
      if (msg === 'done') {
        if (--inFlight === 0) {
          process.stdout.write('\ncount ' + count + '\n');
        }
      } else if (typeof msg === 'bigint') {
        process.stdout.write(msg.toString() + ' ');
        count++;
      }
    })
  }
} else {
  for (let i = 1; i < 10_000_000/THREAD_COUNT; i++) {
    const randomNum = random64();
    if (isHappycoin(randomNum)) {
      parentPort.postMessage(randomNum);
    }
  }
  parentPort.postMessage('done');
}
```

Здесь мы ветвимся в предложении `if`. Если выполняется главный поток, то мы запускаем четыре рабочих потока, указывая в качестве источника кода текущий файл. Напомним, что строка `__filename` содержит путь и имя текущего файла. Затем мы добавляем обработчик сообщений в созданный рабочий поток. Если сообщение содержит просто `done`, значит, поток завершил работу. Если при этом все потоки завершились, то мы выводим счетчик. Если сообщение содержит число, а точнее `bigint`, то мы предполагаем, что это Happycoin, печатаем его и увеличиваем счетчик на единицу, как в однопоточной программе.

Ветвь `else` выполняется, если мы находимся в одном из рабочих потоков. Здесь мы видим такой же цикл, как в однопоточном варианте, только теперь число итераций в четыре раза меньше, потому что работа распределена между четырьмя потоками. Кроме того, вместо того чтобы писать прямо в выходной поток, мы отправляем найденные Happycoin'ы главному потоку через переданный нам `MessagePort`, который называется `parentPort`. Обработчик этого сообщения в главном потоке уже подготовлен. По завершении цикла мы отправляем сообщение `done` в `parentPort`, чтобы главный поток знал, что в этом потоке Happycoin'ов уже не будет.

Можно было бы просто печатать найденные Happycoin'ы сразу, но, как и в примере на C, мы не хотим, чтобы вывод разных потоков перемешивался, поэтому нужна *синхронизация*. В главах 4 и 5 мы рассмотрим дополнительные методы синхронизации, а пока ограничимся тем, что передаем данные в главный поток через порт `parentPort` и поручаем ему заняться выводом.

Добавив потоки в программу, запустите ее следующей командой:

```
$ node happycoin-threads.js
```

Должно быть напечатано что-то в таком роде:

```
1724171919468655000 ... [ еще 137 чисел ] ... 17618203841507830000
count 139
```

Как и в случае программы на C, это код работает намного быстрее. При прогоне на том же компьютере с той же версией Node.js он занял примерно 33 с. Огромное улучшение по сравнение с однопоточным примером и еще один приз в копилку потоков!

Это не единственный способ разбить задачу на части с целью применения потоков. Например, можно было бы применить другие методы синхронизации, чтобы избежать передачи данных между потоками, или передавать сообщения пакетом. Всегда тестируйте и сравнивайте разные подходы, чтобы найти наиболее эффективное решение своей задачи.

Piscina – организация пула рабочих потоков

Многие типы рабочих нагрузок естественно тяготеют к использованию потоков. В Node.js рабочие нагрузки чаще всего сводятся к обработке HTTP-запросов. Если вы ловите себя на том, что программа выполняет много

математических операций или синхронной обработки данных, то имеет смысл вынести эту работу в один или несколько потоков. Это означает, что нужно будет передать потоку задачу и дождаться от него результата. По аналогии с тем, как работает многопоточный веб-сервер, имеет смысл организовать пул рабочих потоков, которому главный поток сможет отправлять задачи.

В этом разделе мы лишь мельком рассмотрим пулы потоков, взяв за основу приложение Happycoins и абстрагировав механизм управления пулом с помощью готового пакета. В разделе «Пул потоков» главы 6 эта тема будет рассмотрена более подробно, и мы создадим собственную реализацию с нуля.

Идея пулов ресурсов встречается не только в контексте потоков. Например, браузеры обычно создают пулы сокетов для подключения к веб-серверам, чтобы можно было мультиплексировать HTTP-запросы, необходимые для отрисовки страницы. Клиентские библиотеки для работы с базами данных часто делают нечто подобное для подключений к серверу БД.

Для Node.js имеется модуль *generic*-pool (https://www.npmjs.com/package/generic-pool) для работы с произвольными пулами ресурсов: подключений к базе данных, других сокетов, локальных кешей, потоков – да вообще всего, что может потребоваться в нескольких экземплярах, но с доступом по одному.

Для случая дискретных задач, отправляемых пулу рабочих потоков, в нашем распоряжении имеется модуль *piscina* (https://www.npmjs.com/package/piscina). Этот модуль инкапсулирует создание группы рабочих потоков и распределение задач между ними. Название модуля означает «пул» по-итальянски.

Использование модуля не вызывает трудностей. Мы создаем экземпляр класса `Piscina`, передавая ему имя файла `filename`, который будет исполняться в рабочем потоке. За кулисами создается пул рабочих потоков и очередь входящих задач. Для помещения задачи в очередь служит метод `.run()`, которому передаются все данные, необходимые для завершения задачи (отметим, что они будут клонированы так же, как в случае `postMessage()`). В ответ возвращается обещание, которое разрешается, когда задача выполнена рабочим потоком; при этом мы получаем доступ к результирующему значению. Из файла, подлежащего выполнению рабочим потоком, должна быть экспортирована функция, которая принимает все, что передано методу `.run()`, и возвращает конечное значение. Эта функция может быть асинхронной, поэтому в рабочем потоке можно выполнять асинхронные задачи, если это необходимо. Простой пример вычисления квадратных корней в рабочих потоках приведен в примере 3.12.

Пример 3.12 ❖ Вычисление квадратных корней с применением `piscina`

```
const Piscina = require('piscina');

if (!Piscina.isWorkerThread) { ❶
  const piscina = new Piscina({ filename: __filename }); ❷
  piscina.run(9).then(squareRootOfNine => { ❸
```

```
    console.log('Квадратный корень из девяти равен', squareRootOfNine);
  });
}

module.exports = num => Math.sqrt(num); ❹
```

❶ Как и модули `cluster` и `worker_threads`, `piscina` предоставляет булево свойство, позволяющее узнать, находимся мы в главном или рабочем потоке.
❷ Мы пользуемся той же техникой использования единого файла, как в примере Happycoin.
❸ Поскольку `.run()` возвращает обещание, мы можем вызывать его метод `.then()`.
❹ Экспортированная функция используется в рабочем потоке для выполнения полезной работы. В нашем случае это вычисление квадратного корня.

Хотя выполнять с помощью пула одну задачу тоже можно, нам интересно выполнять *много* задач. Допустим, что мы хотим вычислить квадратные корни из всех целых чисел, меньших миллиона. Для этого понадобится цикл с миллионом итераций. Кроме того, заменим печать утверждением о том, что получен числовой результат, поскольку печать создавала бы ненужный шум. Смотрите код в примере 3.13.

Пример 3.13 ❖ Вычисление 10 млн квадратных корней с применением `piscina`

```
const Piscina = require('piscina');
const assert = require('assert');

if (!Piscina.isWorkerThread) {
  const piscina = new Piscina({ filename: __filename });
  for (let i = 0; i < 10_000_000; i++) {
    piscina.run(i).then(squareRootOfI => {
      assert.ok(typeof squareRootOfI === 'number');
    });
  }
}

module.exports = num => Math.sqrt(num);
```

На первый взгляд, должно работать. Мы отправляем десять миллионов чисел для обработки пулом рабочих потоков. Однако, запустив эту программу, мы получим фатальную ошибку выделения памяти. В прогоне с Node.js v16.0.0 наблюдалась следующая ситуация.

```
FATAL ERROR: Reached heap limit Allocation failed
      - JavaScript heap out of memory
1: 0xb12b00 node::Abort() [node]
2: 0xa2fe25 node::FatalError(char const*, char const*) [node]
3: 0xcf8a9e v8::Utils::ReportOOMFailure(v8::internal::Isolate*,
   char const*, bool) [node]
4: 0xcf8e17 v8::internal::V8::FatalProcessOutOfMemory(v8::internal::Isolate*,
   char const*, bool) [node]
5: 0xee2d65 [node]
[ ... еще 13 строк не оособенно полезной трассы стека C++ ... ]
Aborted (core dumped)
```

И что же произошло? Оказывается, очередь задач не бесконечна. По умолчанию она неограниченно растет, пока не кончится память – вот как в этом случае. Чтобы избежать такого развития событий, нужно задать разумный предел. Модуль `piscina` позволяет задать максимальный размер очереди с помощью параметра `maxQueue` в конструкторе, который может быть равен любому положительному целому числу. Экспериментальным путем разработчики `piscina` выяснили, что идеальное значение `maxQueue` равно квадратному корню из числа рабочих потоков в пуле. Его можно задать, даже не зная реального значения, положив `maxQueue` равным `auto`.

Задав максимальный размер очереди, мы должны решить, как поступать, когда очередь заполнена. Есть два способа узнать, что очередь полна.

1. Сравнить значения `piscina.queueSize` и `piscina.options.maxQueue`. Если они равны, значит, очередь заполнена. Это можно делать до вызова `piscina.run()`, чтобы не пытаться вставить новую задачу в заполненную очередь. Это рекомендуемая практика.
2. Если метод `piscina.run()` вызван, когда очередь заполнена, то возвращенное обещание будет отменено с ошибкой, указывающей, что в очереди нет места. Это не идеальный выход, потому что в этот момент мы уже находимся в другом месте цикла событий, и, возможно, произошло много других попыток вставить в очередь.

Зная, что очередь заполнена, мы должны как-то узнать, когда можно будет помещать новые задачи. К счастью, пулы `piscina` генерируют событие `drain`, когда очередь пуста, и это самое подходящее время, чтобы начать добавлять новые задачи. В примере 3.14 все это собрано в асинхронную функцию, обертывающую цикл подачи задач.

Пример 3.14 ❖ Вычисление 10 млн квадратных корней с применением `piscina` без аварийного завершения

```
const Piscina = require('piscina');
const assert = require('assert');
const { once } = require('events');

if (!Piscina.isWorkerThread) {
  const piscina = new Piscina({
    filename: __filename,
    maxQueue: 'auto' ❶
  });
  (async () => { ❷
    for (let i = 0; i < 10_000_000; i++) {
      if (piscina.queueSize === piscina.options.maxQueue) { ❸
        await once(piscina, 'drain'); ❹
      }
      piscina.run(i).then(squareRootOfI => {
        assert.ok(typeof squareRootOfI === 'number');
      });
    }
  })();
}
```

```
module.exports = num => Math.sqrt(num);
```

❶ **Параметр `maxQueue` равен `auto`, что ограничивает размер очереди квадратом числа рабочих потоков в пуле `piscina`.**
❷ **Цикл `for` обернут асинхронным непосредственно вызываемым функциональным выражением (IIFE), чтобы внутри него можно было использовать `await`.**
❸ **Если это условие выполнено, то очередь заполнена**
❹ **В этом случае мы ждем события `drain`, перед тем как помещать в очередь новые задачи.**

При прогоне этой программы не возникает ошибки исчерпания памяти, как раньше. Она работает довольно долго, но в конце концов завершается корректно.

Как видим, легко попасть в ловушку, когда использование инструмента наиболее очевидным, на первый взгляд, способом оказывается не лучшим решением. При создании многопоточных приложений важно хорошо понимать, как работают инструменты типа `piscina`.

А теперь посмотрим, что произойдет при попытке использовать `piscina` для добычи Happycoin'ов.

Полный пул Happycoin'ов

Для производства Happycoin'ов с использованием `piscina` мы применим подход, немного отличающийся от первоначальной реализации с помощью `worker_threads`. Вместо того чтобы отправлять сообщение при каждом нахождении Happycoin'а, мы соберем их в пакет и отправим одним разом по завершении работы. Это сэкономит время на подготовке канала `MessageChannel` для отправки данных главному потоку. Побочный же эффект состоит в том, что результаты будут доставляться пакетами, а не по мере готовности. На главном потоке по-прежнему лежит обязанность запускать потоки и получать от них результаты.

Компромиссы

Программирование – это вообще сплошные компромиссы. И многопоточное программированиие не исключение. Компромиссы здесь на каждом шагу. Пожертвуешь удобством в одном месте – получишь выигрыш в производительности в другом. И наоборот. Иногда если сделать одну операцию чуть медленнее, то другая станет значительно быстрее.

Как всегда, нужно *измерять*. Вы можете сколь угодно напряженно размышлять над проблемой, но самый верный способ узнать, стоит ли компромисс свеч, – померить. Проверяйте свой код в разных условиях и смотрите, хорошо ли он себя ведет во всех заслуживающих внимания отношениях. А что это за отношения, зависит от конкретной задачи, вашей интерпретации этой задачи и потребностях заинтересованных сторон.

Помимо измерения, документируйте – это может сэкономить вам часы, дни, а то и недели бесплодной работы в будущем. Ну, обидно же потратить время на выработку компромисса, а через несколько месяцев забыть, почему было принято такое решение, и начать все сначала.

Для начала скопируйте файл *happycoin-threads.js* в *happycoinpiscina.js*. Мы возьмем за отправную точку уже написанный код на основе модуля `worker_threads`. Замените все предшествующее строке `require('crypto')` кодом из примера 3.15.

Пример 3.15 ❖ *ch3-happycoin/happycoin-piscina.js*

```
const Piscina = require('piscina');
```

С этим все просто. Но перейдем к более существенным изменениям. Замените все строки после объявления функции `isHappycoin()` кодом из примера 3.16.

Пример 3.16 ❖ *ch3-happycoin/happycoin-piscina.js*

```
const THREAD_COUNT = 4;

if (!Piscina.isWorkerThread) { ❶
  const piscina = new Piscina({
    filename: __filename, ❷
    minThreads: THREAD_COUNT, ❸
    maxThreads: THREAD_COUNT
  });
  let done = 0;
  let count = 0;
  for (let i = 0; i < THREAD_COUNT; i++) { ❹
    (async () => {
      const { total, happycoins } = await piscina.run(); ❺
      process.stdout.write(happycoins);
      count += total;
      if (++done === THREAD_COUNT) { ❻
        console.log('\ncount', count);
      }
    })();
  }
}
```

❶ Чтобы проверить, находимся ли мы в главном потоке, будем использовать свойство `isWorkerThread`.
❷ Создание рабочих потоков на основе текущего файла делается так же, как и раньше.
❸ Мы хотим ограничить число потоков четырьмя для честного сравнения с предыдущими примерами.
❹ Мы знаем, что потоков четыре, поэтому помещаем в очередь ровно четыре задачи. Каждая задача завершается после проверки своей доли случайных чисел на принадлежность к Happycoin'ам.
❺ Мы помещаем задачу в очередь в этом асинхронном IIFE, так что все они окажутся в очереди на одной итерации цикла событий. Но не волнуйтесь, память не кончится, как было раньше, потому что мы точно знаем, что потоков всего четыре, и помещаем в очередь ровно четыре задачи. Ниже мы увидим, что задача возвращает как выходную строку, так и количество найденных в потоке Happycoin'ов.
❻ Как и в предыдущих реализациях, мы проверяем, что все потоки завершили свои задачи, перед тем как выводить общее число найденных Happycoin'ов.

Добавьте код из примера 3.17, в котором экспортируется функция, выполняемая в рабочих потоках `piscina`.

Пример 3.17 ❖ *ch3-happycoin/happycoin-piscina.js*

```
module.exports = () => {
  let happycoins = '';
  let total = 0;
  for (let i = 0; i < 10_000_000/THREAD_COUNT; i++) { ❶
    const randomNum = random64();
    if (isHappycoin(randomNum)) {
      happycoins += randomNum.toString() + ' ';
      total++;
    }
  }
  return { total, happycoins }; ❷
}
```

❶ **Здесь мы видим обычный цикл поиска Happycoin'ов, но, как и в других распараллеленных примерах, все пространство поиска разделено на части по числу потоков.**

❷ **Мы должны передать строку, содержащую найденные Happycoin'ы и их общее число, в главный поток, для чего возвращаем значение из этой функции.**

Для выполнения этой программы нужно установить пакет `piscina`, если вы еще этого не сделали. Для этого нужно, находясь в каталоге *ch3-happycoin*, выполнить первые две из показанных ниже команд, которые подготавливают проект Node.js и добавляют зависимость `piscina`. Третья команда запускает вашу программу:

```
$ npm init -y
$ npm install piscina
$ node happycoin-piscina.js
```

Результат должен быть таким же, как в предыдущих примерах, с небольшим отличием. Если раньше Happycoin'ы появлялись один за другим, то теперь они печатаются почти все сразу, четырьмя большими группами. Это результат того, что мы возвращаем строки целиком, а не каждый Happycoin в отдельности. Время работы примерно такое же, как для *happycoin-threads.js*, потому что принцип тот же, мы всего лишь воспользовались уровнем абстракции, предоставляемым `piscina`.

Как видите, мы используем `piscina` не совсем обычным образом. Мы не передаем последовательность дискретных задач, что потребовало бы аккуратного управления очередью. Основная причина такого решения – производительность.

Если бы, например, мы завели в главном потоке цикл, выполняемый 10 млн раз, то программа работала бы так же медленно, как при синхронном выполнении в главном потоке. Мы могли бы *не* ждать ответа, а добавлять задачи в очереди в максимально возможном темпе, но накладные расходы на передачу 20 млн сообщений были бы куда выше, чем на передачу восьми сообщений.

При работе с первичными данными, например числами или потоками байтов, обычно быстрее передавать данные между потоками с помощью `SharedArrayBuffer`, о чем мы узнаем в следующей главе.

Глава 4

Разделяемая память

До сих пор мы рассказывали об API веб-исполнителей для браузеров в главе 2 и о модуле рабочих потоков для Node.js в главе 3. Это два мощных инструмента для организации конкурентности в JavaScript, которые позволяют исполнять код параллельно, что раньше в JavaScript было невозможно.

Однако рассмотренные выше способы взаимодействия между потоками ограничены. Код-то действительно можно исполнять параллельно, но при этом мы пользовались только API передачи сообщений, который опирается на хорошо знакомый цикл событий для обработки получения сообщений. В результате получается система, куда менее производительная, чем многопоточный код, написанный в разделе «Потоки на C: обогатитесь с помощью криптовалюты Happycoin» главы 2, где отдельные потоки могли обращаться к разделяемой памяти.

В этой главе мы рассмотрим два эффективных механизма, доступных JavaScript-приложениям: объект `Atomics` и класс `SharedArrayBuffer`. Они позволяют двум потокам сообща использовать память, не полагаясь на передачу сообщений. Но, прежде чем переходить к подробному техническому описанию этих механизмов, уместно будет привести вводный пример.

В неумелых руках описываемые здесь инструменты могут представлять опасность, поскольку вносят в приложение логические ошибки, проявляющиеся только на этапе эксплуатации. Но при правильном использовании они открывают перед приложением новые высоты и позволяют добиться ранее невиданной производительности от оборудования.

Введение в разделяемую память

В этом примере мы напишем очень простое приложение, в котором будут взаимодействовать два рабочих потока. Поначалу нам потребуется немного трафаретного кода с использованием `postMessage()` и `onmessage`, но последующие версии не будут опираться на эту функциональность.

Этот пример будет работать не только в Node.js, но и в браузере, хотя инициализация выглядит немного по-разному. Сейчас мы напишем код, работающий в браузере, и подробно объясним, что происходит. А потом, познакомившись с основными идеями, мы перепишем его для работы в Node.js.

Разделяемая память в браузере

Создайте каталог *ch4-webworkers/* для этого проекта. Поместите в него HTML-файл *index.html*, содержащий код из примера 4.1.

Пример 4.1 ❖ *ch4-web-workers/index.html*

```
<html>
  <head>
    <title>Shared Memory Hello World</title>
    <script src="main.js"></script>
  </head>
</html>
```

Теперь можно переходить к более сложной части приложения. Создайте файл *main.js* с кодом из примера 4.2.

Пример 4.2 ❖ *ch4-web-workers/main.js*

```
if (!crossOriginIsolated) { ❶
  throw new Error('Невозможно использовать SharedArrayBuffer');
}

const worker = new Worker('worker.js');

const buffer = new SharedArrayBuffer(1024); ❷
const view = new Uint8Array(buffer); ❸

console.log('сейчас', view[0]);

worker.postMessage(buffer);

setTimeout(() => {
  console.log('позже', view[0]);
  console.log('prop', buffer.foo); ❹
}, 500);
```

❶ Если `crossOriginIsolated` равно `true`, то использовать `SharedArrayBuffer` можно.
❷ Создать буфер размером 1 КБ.
❸ Создать представление буфера.
❹ Читать модифицированное свойство.

Этот файл похож на созданный ранее. В нем по-прежнему используется выделенный исполнитель. Но есть и дополнительные усложнения. Во-первых, это проверка глобальной переменной `crossOriginIsolated`, имеющейся в современных браузерах. Она говорит, в частности, разрешено ли исполняемому в данный момент JavaScript-коду создавать экземпляры класса `SharedArrayBuffer`.

Из соображений безопасности (связанных с атакой на уязвимость Spectre в современных процессорах) создавать объект `SharedArrayBuffer` разрешается не всегда. На самом деле несколько лет назад эта функциональность была полностью отключена в браузерах. Сейчас Chrome и Firefox поддерживают создание `SharedArrayBuffer`, но только при условии, что установлены допол-

нительные HTTP-заголовки. Node.js не налагает таких ограничений. Вот эти заголовки:

```
Cross-Origin-Opener-Policy: same-origin
Cross-Origin-Embedder-Policy: require-corp
```

Тестовый сервер, с которым вы работаете, устанавливает эти заголовки автоматически. Когда будете разрабатывать реальные приложения, в которых используются экземпляры `SharedArrayBuffer`, не забудьте их установить.

После создания выделенного исполнителя создается экземпляр `SharedArrayBuffer`. Его аргумент, в данном случае 1024, задает размер буфера в байтах. В отличие от массивов и объектов буферов, с которыми вы, возможно, знакомы, эти буферы не могут увеличиваться и уменьшаться после создания[1].

Также создается представление `view` для работы с буфером. Такие представления подробно обсуждаются в разделе «SharedArrayBuffer и типизированные массивы» ниже, а пока считайте, что это способ читать и записывать буфер.

Представление буфера позволяет читать из него, пользуясь синтаксисом индексирования массива. В данном случае мы читаем нулевой байт буфера, когда выводим на консоль байт `view[0]`. Затем экземпляр буфера передается исполнителю методом `worker.postMessage()`. И это единственное, что мы передаем в этой программе. Однако при желании можно было бы передать более сложный объект, одним из свойств которого является буфер. Хотя алгоритм, описанный в приложении, портит большинство сложных объектов, экземпляры `SharedArrayBuffer` являются исключением из этого правила.

Закончив инициализацию, скрипт планирует запуск функции через 500 мс. Она еще раз печатает нулевой байт буфера и пытается напечатать свойство буфера `.foo`. Заметим, что в этом файле не определен обработчик события `worker.onmessage`.

С главным файлом все, теперь можно перейти к исполнителю. Создайте файл *worker.js* и скопируйте в него код из примера 4.3.

Пример 4.3 ❖ *ch4-web-workers/worker.js*

```
self.onmessage = ({data: buffer}) => {
  buffer.foo = 42; ❶
  const view = new Uint8Array(buffer);
  view[0] = 2; ❷
  console.log('обновлено исполнителем');
};
```

❶ **Присваивание значения свойству объекта буфера.**
❷ **В элемент с индексом 0 записывается 2.**

Здесь мы присоединяем к событию `onmessage` обработчик, который будет вызван после выполнения метода `.postMessage()` в *main.js*. Обработчик извлекает из события буфер и первым делом присоединяет к нему свойство `.foo`. Затем

[1] В будущем это ограничение может быть снято, см. предложение «Буферные массивы, расширяемые на месте» по адресу *https://github.com/tc39/proposal-resizable-arraybuffer*.

он создает для буфера еще одно представление и производит через него обновление. После этого печатается сообщение, чтобы мы видели, что происходит.

Теперь все готово к запуску приложения. Откройте окно терминала и выполните показанную ниже команду. Она немного отличается от команд `serve`, которые мы выполняли раньше, потому что необходимо добавить заголовки для обеспечения безопасности:

```
$ npx MultithreadedJSBook/serve .
```

Как и раньше, откройте URL-адрес, напечатанный этой командой. Затем откройте веб-инспектор и перейдите на вкладку **Консоль**. Если там ничего нет, обновите страницу. Должны появиться сообщения, напечатанные приложением. Пример вывода показан в табл. 4.1.

Таблица 4.1. Пример вывода на консоль

Сообщение	Источник
сейчас 0	main.js:10:9
обновлено исполнителем	worker.js:5:11
позже 2	main.js:15:11
prop undefined	main.js:16:11

Первая напечатанная строка – начальное значение буфера, которое видит *main.js*. В данном случае оно равно `0`. Затем выполняется код в файле *worker.js*, хотя точный момент, когда это произойдет, заранее неизвестен. Спустя примерно полсекунды значение, видимое в *main.js*, печатается еще раз, и теперь оно равно `2`. Отметим, что между главным потоком, исполняющим *main.js*, и рабочим потоком, исполняющим `worker.js`, не передавалось никаких сообщений, кроме одного на этапе инициализации.

Это очень простой пример. Он, конечно, работает, но не дает представления о том, как на самом деле пишется многопоточный код. Нет никакой гарантии, что значение, измененное в *worker.js*, будет видно в *main.js*. Например, особо изощренный движок JavaScript мог бы трактовать это значение как константу, хотя надо еще постараться, чтобы найти браузер, который так себя ведет.

После печати буфера печатается также свойство `.foo`, которое равно `undefined`. Почему так произошло? Дело в том, что, хотя ссылка на ячейку памяти, в которой хранятся двоичные данные, находящиеся в буфере, разделяется между двумя средами JavaScript, сам объект не разделяется. Иначе было бы нарушено ограничение алгоритма структурированного клонирования, которое запрещает разделять ссылки на объекты между потоками.

Разделяемая память в Node.js

Эквивалентное приложение для Node.js в основных чертах похоже, но глобальный класс `Worker`, предоставляемый браузерами, отсутствует, а рабочие потоки не могут использовать событие `self.onmessage`. Вместо этого для до-

ступа к этой функциональности нужно импортировать модуль рабочих потоков. Поскольку Node.js не браузер, файл *index.html* для этой цели не годится.

Чтобы создать эквивалент для Node.js, нам понадобится всего два файла, которые можно создать в том же каталоге *ch4-web-workers/*. В файл *main-node.js* скопируйте код из примера 4.4.

Пример 4.4 ❖ *ch4-web-workers/main-node.js*

```
#!/usr/bin/env node

const { Worker } = require('worker_threads');
const worker = new Worker(__dirname + '/worker-node.js');

const buffer = new SharedArrayBuffer(1024);
const view = new Uint8Array(buffer);

console.log('сейчас', view[0]);

worker.postMessage(buffer);

setTimeout(() => {
  console.log('позже', view[0]);
  console.log('prop', buffer.foo);
  worker.unref();
}, 500);
```

Код немного отличается, но в основном кажется знакомым. Поскольку глобально класс `Worker` недоступен, мы получаем его из свойства `.Worker` загруженного модуля `worker_threads`. При создании экземпляра `worker` нужно задавать путь более явно, чем в браузерах. В данном случае был задан путь *./worker-node.js*, хотя браузеры довольствуются простым *worker.js*. В остальном главный JavaScript-файл для Node.js почти не отличается от своего эквивалента в браузере. Последний вызов `worker.unref()` нужен для того, чтобы рабочий поток не исполнял процесс вечно.

Далее создайте файл *worker-node.js*, который будет содержать код, эквивалентный веб-исполнителю в браузере. Включите в этот файл код из примера 4.5.

Пример 4.5 ❖ *ch4-web-workers/worker-node.js*

```
const { parentPort } = require('worker_threads');

parentPort.on('message', (buffer) => {
  buffer.foo = 42;
  const view = new Uint8Array(buffer);
  view[0] = 2;
  console.log('обновлено исполнителем');
});
```

Здесь событие `self.onmessage` недоступно рабочему потоку. Поэтому он снова загружает модуль `worker_threads` и использует его свойство `.parentPort`. Оно представляет соединение с портом из вызывающей среды JavaScript.

Можно присвоить обработчик `.onmessage` объекту `parentPort`, а можно вызывать метод `.on('message', cb)`. Если используются оба, то они будут вызваны в порядке использования. Функция обратного вызова для события `message` принимает переданный объект (в данном случае `buffer`) непосредственно в виде аргумента, а обработчик `onmessage` предоставляет экземпляр `MessageEvent` в свойстве `.data`, содержащем `buffer`. Какой подход предпочесть, дело вкуса.

Во всем остальном код для Node.js и браузера одинаковый. Доступны те же глобальные объекты, в частности `SharedArrayBuffer`, и работают они точно так же.

Теперь можно запустить программу командой

```
$ node main-node.js
```

Ее вывод должен быть таким же, как в табл. 4.1. Повторим, что алгоритм структурированного клонирования позволяет передавать экземпляры `SharedArrayBuffer`, но только двоичные данные в буфере, а не прямые ссылки на сам объект.

SharedArrayBuffer и типизированные массивы

Традиционно JavaScript не поддерживал взаимодействия с двоичными данными. Строки, конечно, были, но в действительности они абстрагировали истинный механизм хранения данных. Были также массивы, но они могут содержать значения любого типа и не подходят для представления двоичных буферов. В течение многих лет считалось, что это «достаточно хорошо», но только до появления Node.js, благодаря которому популярность выполнения JavaScript вне контекста веб-страницы резко возросла.

Среда исполнения Node.js в числе прочего может читать и записывать в файловую систему, передавать и получать данные из сети потоком и т. д. Такие взаимодействия не ограничиваются текстовыми ASCII-файлами, а могут включать обмен двоичными данными. Так появился на свет класс Node.js `Buffer`.

Вместе с расширением границ самого языка JavaScript расширялся и набор API, а также способности языка взаимодействовать с миром вне окна браузера. В конечном итоге появился объект `ArrayBuffer`, а вслед за ним и `SharedArrayBuffer`, которые теперь вошли в ядро языка. Если бы Node.js был создан сегодня, то, скорее всего, в нем не было бы собственной реализации `Buffer`.

Объекты типа `ArrayBuffer` и `SharedArrayBuffer` представляют буфер двоичных данных фиксированной длины, которую впоследствии изменить невозможно. Они похожи, но в этом разделе нас будет интересовать второй, потому что он позволяет разделять память между потоками. В двоичных данных, таких распространенных и полноправных во многих традиционных языках программирования, в частности C, легко запутаться, особенно если

разработчик привык к языкам более высокого уровня, например JavaScript.

На случай, если вы не знаете, *двоичной* называется система счисления по основанию 2, в которой есть всего две цифры: 0 и 1. Каждая цифра называется *битом*. *Десятичная* система, которой обычно пользуются люди, имеет основание 10, а цифрами в ней являются знаки от 0 до 9. Группа из 8 бит называется байтом, часто это наименьшая адресуемая единица памяти, поскольку с байтами иметь дело проще, чем с отдельными битами. Поэтому процессоры (и программисты) работают с байтами, а не с битами.

Байты часто представляются двумя *шестнадцатеричными* символами, т. е. цифрами в системе счисления по основанию 16: знаками 0–9 и A–F. При выводе в журнал экземпляра `ArrayBuffer` в Node.js данные в буфере отображаются в шестнадцатеричном виде.

Видя произвольный набор байтов, хранящихся на диске или в памяти компьютера, трудно сказать, что эти данные означают. Например, что может быть представлено значением `0x54` (префикс `0x` в JavaScript означает, что значение шестнадцатеричное)? Если это часть строки, то, наверное, представляет заглавную букву *T*. А если это целое число, то может означать десятичное 84. А быть может, что часть адреса памяти или пиксель изображения в формате JPEG или еще что-то. Контекст здесь очень важен. То же самое число в двоичной системе записывается как `0b01010100` (префикс `0b` означает, что значение двоичное).

Памятуя об этой неоднозначности, важно также отметить, что содержимое `ArrayBuffer` (и `SharedArrayBuffer`) не допускает прямой модификации. Вместо этого необходимо создать «представление» буфера. Кроме того, в отличие от других языков, которые могут предоставлять доступ к памяти, ранее занятой другими объектами, в JavaScript создаваемый буфер `ArrayBuffer` инициализируется нулями. Учитывая, что в буферах хранятся только числовые данные, следует признать, что это крайне примитивное средство хранения данных, но поверх него можно построить более сложные системы.

Классы `ArrayBuffer` и `SharedArrayBuffer` наследуют `Object` и обладают всеми его методами. Дополнительно у них есть еще два атрибута. Первый, свойство `.byteLength`, доступен только для чтения и представляет длину буфера в байтах. Второй, метод `.slice(begin, end)`, возвращает копию части буфера, определяемой заданным диапазоном. Начало `begin` включается в состав диапазона, а конец `end` исключается. Сравните этот метод, например, с методом `String#substr(begin, length)`, в котором второй параметр – длина. Если значение `begin` опущено, предполагается первый элемент, а если опущено значение `end`, то предполагается последний элемент. Отрицательные числа означают, что отсчет ведется от конца буфера.

Ниже приведены примеры простых операций с `ArrayBuffer`:

```
const ab = new ArrayBuffer(8);
const view = new Uint8Array(ab)
for (i = 0; i < 8; i++) view[i] = i;
console.log(view);
// Uint8Array(8) [
//   0, 1, 2, 3,
//   4, 5, 6, 7
```

```
// ]
ab.byteLength; // 8
ab.slice(); // 0, 1, 2, 3, 4, 5, 6, 7
ab.slice(4, 6); // 4, 5
ab.slice(-3, -2); // 5
```

В различных средах JavaScript содержимое `ArrayBuffer` отображается поразному. В Node.js отображается список шестнадцатеричных пар, как если бы данные просматривались как `Uint8Array`. В Chrome v88 отображается раскрывающийся объект в нескольких разных представлениях. В Firefox данные не отображаются, их необходимо сначала передать представлению.

Мы уже несколько раз употребляли термин *представление*, пора уже определить его. Из-за неоднозначности семантики двоичных данных нам необходимо представление, чтобы читать буфер и записывать в него. В JavaScript есть несколько таких представлений. Все они расширяют базовый класс `TypedArray`. Создать экземпляр самого этого класса невозможно, и как глобальный символ он недоступен, однако получить к нему доступ можно с помощью свойства `.prototype` имеющегося экземпляра дочернего класса.

В табл. 4.2 приведен перечень классов представлений, расширяющих `TypedArray`.

Таблица 4.2. Классы, расширяющие `TypedArray`

Класс	Байтов	Минимальное значение	Максимальное значение
`Int8Array`	1	-128	127
`UInt8Array`	1	0	255
`Uint8ClampedArray`	1	0	255
`Int16Array`	2	-32 768	32 767
`UInt16Array`	2	0	65 535
`Int32Array`	4	–2 147 483 648	2 147 483 647
`UInt32Array`	4	0	4 294 967 295
`Float32Array`	4	1.4012984643e-45	3.4028235e38
`Float64Array`	8	5e-324	1.7976931348623157e308
`BigInt64Array`	8	–9 223 372 036 854 775 808	9 223 372 036 854 775 807
`UBigInt64Array`	8	0	18 446 744 073 709 551 615

В столбце «Класс» приведено имя класса, экземпляры которого можно создавать. Эти классы глобальные и доступны в любом современном движке JavaScript. В столбце «Байтов» приведено количество байтов, занятых одним элементом представления. В столбцах «Минимальное значение» и «Максимальное значение» приведены границы диапазона элементов данного типа.

При создании любого из этих представлений конструктору передается экземпляр `ArrayBuffer`. Длина буфера в байтах должна быть кратна длине элемента данного представления. Например, объект `ArrayBuffer` длиной 6 байт можно передать конструктору `Int16Array` (длина элемента в байтах равна 2), поскольку будет представлено три элемента `Int16`. Но тот же буфер нельзя передать конструктору `Int32Array`, потому что он представлял бы только полтора элемента, что недопустимо.

Имена этих представлений могут быть знакомы читателям, работавшим на языках низкого уровня, например C или Rust.

Префикс `U` в именах половины этих классов означает «unsigned» (без знака), т. е. они позволяют представить только неотрицательные числа. Классы без префикса `U` могут представлять числа со знаком, т. е. положительные и отрицательные, но ценой сокращения максимального значения вдвое, поскольку первый бит зарезервирован под знак.

Ограничения на диапазон связаны с тем, какое число можно представить тем количеством битов, которое отведено под данный тип. Скажем, максимальное число типа `Uint8`, «целое без знака, представленное 8 битами», равно `0b11111111`, или десятичное 255.

В JavaScript нет целочисленного типа данных, а только тип `Number`, реализующий число с плавающей точкой по стандарту IEEE 754 (https://en.wikipedia.org/wiki/Floating-point_arithmetic), который эквивалентен типу `Float64`. Для всех остальных типов попытка записать число типа `Number` в представление требует преобразования типов.

При записи в массив типа `Float64Array` значение, как правило, не изменяется. Минимальная и максимальная границы диапазона совпадают соответственно с `Number.MIN_VALUE` и `Number.MAX_VALUE`. При записи в массив типа `Float32Array` уменьшаются не только границы диапазона, но и точность.

Для примера рассмотрим следующий код:

```
const buffer = new ArrayBuffer(16);

const view64 = new Float64Array(buffer);
view64[0] = 1.1234567890123456789; // байты 0 - 7
console.log(view64[0]); // 1.1234567890123457

const view32 = new Float32Array(buffer);
view32[2] = 1.1234567890123456789; // байты 8 - 11
console.log(view32[2]); // 1.1234568357467651
```

В этом случае точность представления числа типа `float64` – 15 десятичных знаков, а числа типа `float32` – только 6 десятичных знаков.

Этот код интересен еще в одном отношении. Мы имеем всего один экземпляр `ArrayBuffer`, он называется `buffer`, однако на него указывают два разных экземпляра `TypedArray`. В чем тут может быть странность? Подсказку дает рис. 4.1.

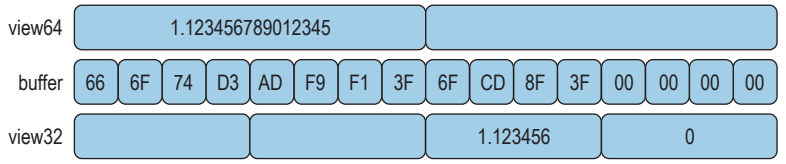


Рис. 4.1 ❖ Один `ArrayBuffer` и несколько представлений `TypedArray`

Как вы думаете, что будет возвращено при чтении `view64[1]`, `view32[0]` или `view32[1]`? В данном случае усеченные области памяти, предназначенные для

хранения данных одного типа, будут объединены или разбиты на части, чтобы представить данные другого типа. Возвращенные значения интерпретируются неправильно и бессмысленны, хотя должны быть детерминированы и согласованы.

Когда записываются числовые данные, выходящие за пределы поддерживаемого диапазона `TypedArray` для типов, отличных от типа с плавающей точкой, необходимо произвести преобразование для согласования с конечным типом. Сначала число должно быть преобразовано в целое, как если бы оно было передано функции `Math.trunc()`. Если число выходит за пределы допустимого диапазона, то берется остаток от деления на верхнюю границу, как при использовании оператора `%`. Ниже показано, что происходит с типом `Uint8Array` (это `TypedArray` с максимальным значением 255):

```
const buffer = new ArrayBuffer(8);
const view = new Uint8Array(buffer);
view[0] = 255;   view[1] = 256;
view[2] = 257;   view[3] = -1;
view[4] = 1.1;   view[5] = 1.999;
view[6] = -1.1;  view[7] = -1.9;
console.log(view);
```

В табл. 4.3 показано соответствие входных и выходных значений.

Таблица 4.3. Преобразования TypedArray

Вход	255	256	257	–1	1.1	1.999	–1.1	–1.9
Выход	255	0	1	255	1	1	255	255

Поведение для `Uint8ClampedArray` немного отличается. Отрицательное значение при записи преобразуется в `0`. Значение, большее 255, при записи преобразуется в 255. Нецелое значение передается `Math.round()`. Иногда в зависимости от ситуации имеет смысл использовать именно это представление.

Наконец, типы `BigInt64Array` и `BigUint64Array` заслуживают особого внимания. В отличие от других представлений `TypedArray`, которые работают с типом `Number`, эти два варианта работают с типом `BigInt` (литерал `1` имеет тип `Number`, а `1n` – тип `BigInt`). Это связано с тем, что числовые значения, представимые 64 байтами, выходят за пределы диапазона типа `Number` в JavaScript. Поэтому значения для этих представлений должны иметь тип `BigInt`, и извлекаемые значения тоже будут иметь тип `BigInt`.

Вообще говоря, использовать несколько представлений `TypedArray`, особенно разных размеров, для доступа к одному и тому же буферу опасно, и по возможности этого следует избегать. Может случиться, что при выполнении различных операций вы повредите данные. Между потоками можно передавать более одного объекта `SharedArrayBuffer`, так что если в программе нужны разные типы, то заведите для них отдельные буферы.

Познакомившись с основами `ArrayBuffer` и `SharedArrayBuffer`, мы готовы приступить к изучению более сложных API для работы с ними.

Атомарные методы манипулирования данными

С термином *атомарность* вы, наверное, уже встречались, особенно в контексте баз данных, где это первое слово в акрониме ACID (атомарность, согласованность, изолированность, долговечность). Говоря, что операция *атомарна*, имеют в виду, что, хотя она может состоять из более мелких шагов, гарантируется, что либо вся она будет выполнена целиком, либо не будет выполнена вовсе. Например, один запрос к базе данных выполняется атомарно, а три отдельных запроса – нет.

Но если эти три запроса обернуты транзакцией базы данных, то операция становится атомарной: либо все три запроса будут выполнены успешно, либо не будет выполнен ни один. Важно также, что операции выполняются в определенном порядке, – это существенно, когда они манипулируют одним и тем же состоянием или имеют еще какие-нибудь побочные эффекты, способные оказать взаимное влияние. *Изолированность* означает, что никакие другие операции не могут вклиниться в середине; например, если выполнена только часть операции, чтение невозможно.

Атомарные операции очень важны в информатике, особенно в распределенных вычислениях. Системы управления базами данных, к которым может подключаться много клиентов, обязаны поддерживать атомарные операции. Распределенные системы, включающие много взаимодействующих сетевых узлов, тоже должны поддерживать такие операции. Даже в одном компьютере, где доступ к данным осуществляется из нескольких потоков, атомарность необходима.

JavaScript предоставляет глобальный объект `Atomics` с несколькими статическими методами. Он устроен так же, как хорошо знакомый глобальный объект `Math`. В обоих случаях нельзя использовать оператор `new` для создания нового экземпляра, а все имеющиеся методы не хранят никакого состояния, т. е. не оказывают воздействия на сам глобальный объект. Вместо этого каждому методу передается ссылка на подлежащие модификации данные.

Далее в этом разделе мы рассмотрим все методы объекта `Atomics`, кроме трех. А оставшиеся три будут рассмотрены в разделе «Использование атомарных методов для координации» главы 5. За исключением `Atomics.isLockFree()`, все эти методы принимают экземпляр `TypedArray` в качестве первого аргумента и индекс элемента массива в качестве второго.

Atomics.add()

```
old = Atomics.add(typedArray, index, value)
```

Этот метод прибавляет значение `value` к значению элемента `typedArray` с индексом `index`. Возвращается старое значение. Неатомарная версия могла бы выглядеть так:

```
const old = typedArray[index];
typedArray[index] = old + value;
return old;
```

Atomics.and()

```
old = Atomics.and(typedArray, index, value)
```

Этот метод применяет поразрядное И к `value` и значению элемента `typedArray` с индексом `index`. Возвращается старое значение. Неатомарная версия могла бы выглядеть так:

```
const old = typedArray[index];
typedArray[index] = old & value;
return old;
```

Atomics.compareExchange()

```
old = Atomics.compareExchange(typedArray, index, oldExpectedValue, value)
```

Этот метод проверяет, содержит ли элемент `typedArray` с индексом `index` значение `oldExpectedValue`. Если да, то это значение заменяется на `value`. Если нет, то ничего не делается. Всегда возвращается старое значение, поэтому по результату сравнения `oldExpectedValue === old` можно сказать, имел ли место обмен. Неатомарная версия могла бы выглядеть так:

```
const old = typedArray[index];
if (old === oldExpectedValue) {
  typedArray[index] = value;
}
return old;
```

Atomics.exchange()

```
old = Atomics.exchange(typedArray, index, value)
```

Этот метод записывает в элемент `typedArray` с индексом `index` значение `value`. Возвращается старое значение. Неатомарная версия могла бы выглядеть так:

```
const old = typedArray[index];
typedArray[index] = value;
return old;
```

Atomics.isLockFree()

```
free = Atomics.isLockFree(size)
```

Этот метод возвращает `true`, если `size` совпадает с размером элемента в байтах (`BYTES_PER_ELEMENT`) в каком-нибудь подклассе `TypedArray` (обычно это значения 1, 2, 4, 8), и `false` в противном случае[1]. Если он возвращает `true`, то методы

[1] Если JavaScript исполняется на редком оборудовании, то этот метод может возвращать `false` для 1, 2 или 8. Но для `size = 4` он всегда возвращает `true`.

объекта Atomics будут работать очень быстро на оборудовании данной системы. Иначе приложение, возможно, предпочтет использовать ручные схемы блокировки, например те, что описаны в разделе «Мьютекс – простая блокировка» главы 6, особенно если производительность стоит на первом плане.

Atomics.load()

```
value = Atomics.load(typedArray, index)
```

Этот метод возвращает значение элемента typedArray с индексом index. Неатомарная версия могла бы выглядеть так:

```
const old = typedArray[index];
return old;
```

Atomics.or()

```
old = Atomics.or(typedArray, index, value)
```

Этот метод применяет поразрядное ИЛИ к value и значению элемента typedArray с индексом index. Возвращается старое значение. Неатомарная версия могла бы выглядеть так:

```
const old = typedArray[index];
typedArray[index] = old | value;
return old;
```

Atomics.store()

```
value = Atomics.store(typedArray, index, value)
```

Этот метод сохраняет значение value в элементе typedArray с индексом index. Возвращается переданное значение. Неатомарная версия могла бы выглядеть так:

```
typedArray[index] = value;
return value;
```

Atomics.sub()

```
old = Atomics.sub(typedArray, index, value)
```

Этот метод вычитает значение value из значения элемента typedArray с индексом index. Возвращается старое значение. Неатомарная версия могла бы выглядеть так:

```
const old = typedArray[index];
typedArray[index] = old - value;
return old;
```

Atomics.xor()

```
old = Atomics.xor(typedArray, index, value)
```

Этот метод применяет поразрядное ИСКЛЮЧАЮЩЕЕ ИЛИ к `value` и значению элемента `typedArray` с индексом `index`. Возвращается старое значение. Неатомарная версия могла бы выглядеть так:

```
const old = typedArray[index];
typedArray[index] = old ^ value;
return old;
```

Несколько замечаний об атомарности

Гарантируется, что все методы, рассмотренные в предыдущем разделе, выполняются атомарно. Например, рассмотрим метод `Atomics.compareExchange()`. Он принимает значение `oldExpectedValue` и новое значение `value` и заменяет старое новым, только если `oldExpectedValue` равно `value`. Хотя для записи этой операции на JavaScript потребовалось бы несколько предложений, гарантируется, что она будет выполнена целиком.

Для иллюстрации предположим, что имеется массив `typedArray` типа `Uint8Array` и его нулевой элемент равен 7. Предположим далее, что к этому массиву обращается несколько потоков и каждый выполняет вариант следующей строки кода:

```
let old1 = Atomics.compareExchange(typedArray, 0, 7, 1); // Поток #1
let old2 = Atomics.compareExchange(typedArray, 0, 7, 2); // Поток #2
```

При этом ни порядок вызова методов, ни время их выполнения не детерминированы. Они даже могли бы быть вызваны строго одновременно! Однако благодаря гарантиям атомарности объекта `Atomics` мы можем быть уверены, что ровно один поток вернет начальное значение 7, а другой – измененное значение 1 или 2. Хронометраж операций мог бы выглядеть, как показано на рис. 4.2, где `CEX(oldExpectedValue, value)` – сокращенная запись `Atomics.compareExchange()`.

С другой стороны, если используется неатомарный аналог `compareExchange()`, например прямое чтение и запись `typedArray[0]`, то может случиться, что программа случайно затрет значение. В этом случае оба потока читают текущее значение примерно в одно и то же время, оба видят, что оно равно ожидаемому значению, и оба записывают новое. Ниже еще раз приведена снабженная комментариями неатомарная версия `compareExchange()`:

```
const old = typedArray[0]; // GET()
if (old === oldExpectedValue) {
  typedArray[0] = value;   // SET(value)
}
```

Этот код выполняет несколько операций над разделяемыми данными: в строке, где данные читаются (она помечена `GET()`), и позже в строке, где

данные записываются (`SET(value)`). Чтобы этот код работал правильно, необходимо, чтобы во время его работы никакие другие потоки не могли ни читать, ни изменять значение. То есть нужна гарантия, что к разделяемым ресурсам в так называемой *критической секции* имеет монопольный доступ только один поток.

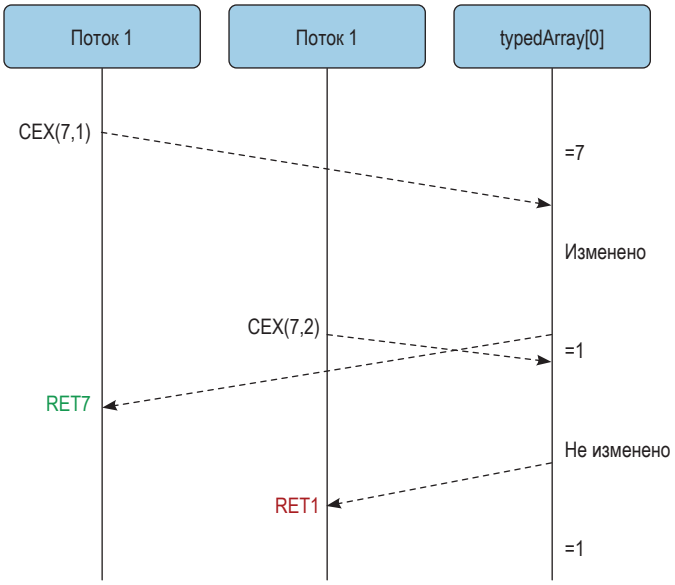


Рис. 4.2 ❖ Атомарное выполнение `Atomics.compareExchange()`

На рис. 4.3 показано, как мог бы выполняться этот код в отсутствие гарантий монопольного доступа.

В этом случае оба потока думают, что успешно записали значение, тогда как в реальности сохранено только значение, записанное вторым потоком. Такой тип ошибок, когда два потока или более конкурируют друг с другом за выполнение некоторого действия, называется *состоянием гонки*[1]. Самое неприятное в таких ошибках – то, что их очень трудно воспроизвести, а проявляться они могут только на производственном сервере, а не на ноутбуке, где ведется разработка.

Чтобы пользоваться преимуществами атомарности операций объекта `Atomics` при взаимодействии с буферным массивом, необходимо не смешивать вызовы атомарных методов с прямым доступом к элементам массива. Если один поток пользуется методом `compareExchange()`, а другой читает и записывает тот же самый элемент напрямую, то все механизмы обеспечения безопасности сводятся на нет, и поведение приложения оказывается недетерминированным. По существу, атомарные методы ставят неявную блокировку, чтобы гарантировать корректность операций.

[1] Может случиться, что после компиляции ошибочной программы порядок ее выполнения будет таким, что результат окажется совершенно неожиданным, необъяснимым с помощью приведенной выше диаграммы чередующихся шагов.

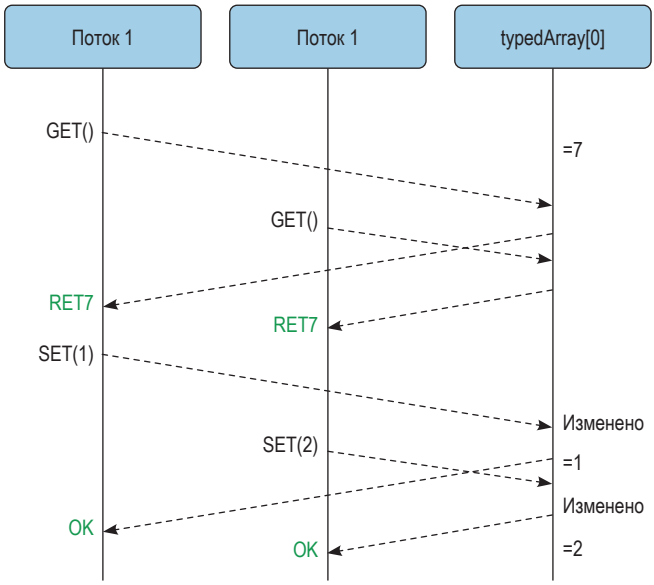


Рис. 4.3 ❖ Неатомарное выполнение `Atomics.compareExchange()`

Как это ни печально, не все желаемые операции с разделяемой памятью можно выполнить с помощью метода `Atomics`. В таком случае необходимы ручные механизмы блокировки, позволяющие читать и записывать данные, не опасаясь вмешательства со стороны других потоков. Эта идея рассматривается в разделе «Мьютекс: простая блокировка» главы 6.

Возвращаемые значения игнорируют преобразования типов

С методами `Atomics` связана одна проблема: они ничего не знают о преобразовании типов, которому будет подвергнут данный `TypedArray`, поэтому возвращают значение *до* преобразования. Рассмотрим, к примеру, следующую ситуацию, когда сохраняется значение, большее, чем позволяет данное представление:

```
const buffer = new SharedArrayBuffer(1);
const view = new Uint8Array(buffer);
const ret = Atomics.store(view, 0, 999);
console.log(ret); // 999
console.log(view[0]); // 231
```

Здесь создается буфер и представление `Uint8Array` для него. Затем вызывается метод `Atomics.store()` для записи значения `999` через представление. Метод `Atomics.store()` возвращает то значение, которое ему было передано, т. е. `999`, хотя в буфер на самом деле было записано значение `231` (поскольку 999 больше максимально допустимого для этого представления значения 255). Помните об этом ограничении при разработке своих приложений. Чтобы чувствовать себя в безопасности, проектируйте приложение, так чтобы оно не зависело от преобразований типов, и записывайте только значения, принадлежащие допустимому диапазону.

Сериализация данных

Буферы – вещь чрезвычайно мощная. Но тот факт, что они умеют работать только с числами, иногда вызывает затруднения. Бывает, что с помощью буфера хочется представить нечисловые данные. В таком случае необходимо как-то сериализовать данные перед записью в буфер и десериализовать после чтения из буфера.

В зависимости от типа данных для сериализации используются различные средства. Некоторые из них работают в разных ситуациях, но для всех характерны компромиссы в отношении размера требуемой памяти и производительности сериализации.

Булевы значения

Булевы значения легко представить, потому что они занимают всего один бит, а бит меньше, чем байт. Поэтому мы можем создать одно из самых узких представлений, например `Uint8Array`, указать на `ArrayBuffer`, задав длину в байтах 1, – и готово. Разумеется, интересно здесь то, что в одном байте можно сохранить до восьми булевых значений. На самом деле если вы работаете с большим количеством булевых значений, то хранение их в буфере может оказаться более выгодным, чем использование средств самого движка JavaScript, потому что в последнем случае с каждым булевым значением связаны метаданные, т. е. дополнительные накладные расходы.

На рис. 4.4 приведен список булевых значений, представленный одним байтом.

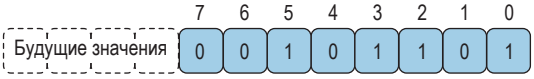


Рис. 4.4 ❖ Хранение булевых значений в байте

При хранении данных в отдельных битах, как в данном случае, лучше начинать с младшего бита, т. е. самого правого, с номером 0, а затем, если нужно сохранить дополнительные значения, перемещаться к более старшим. Причина проста: по мере увеличения числа булевых значений растет и размер буфера, а уже занятые биты при этом не должны затираться. Хотя сам буфер не может расти динамически, новые версии приложения могут создавать буферы большего размера.

Если буфер, в котором хранятся булевы значения, сегодня занимает 1 байт, а завтра 2 байта, то благодаря использованию младших битов первыми десятичное представление данных останется прежним – 0 или 1. А если бы мы начали со старших битов, то сегодня значение было бы равно 0 или 128, а завтра – 0 или 32768. Это могло бы стать причиной проблем, если мы где-то сохраняем значения в одной версии, а затем используем их в другой.

Ниже показано, как сохранять и извлекать булевы значения, хранящиеся в `ArrayBuffer`:

```
const buffer = new ArrayBuffer(1);
const view = new Uint8Array(buffer);
function setBool(slot, value) {
  view[0] = (view[0] & ~(1 << slot)) | ((value|0) << slot);
}
function getBool(slot) {
  return !((view[0] & (1 << slot)) === 0);
}
```

Здесь создается однобайтовый буфер (`0b00000000` в двоичной записи), а затем представление этого буфера. Чтобы установить значение младшего бита `ArrayBuffer` равным `true`, следует вызвать функцию `setBool(0, true)`. Чтобы установить значение второго справа байта равным `false`, вызываем функцию `setBool(1, false)`. Чтобы извлечь значение, хранящееся в третьем справа бите, пишем `getBool(2)`.

Функция `setBool()` принимает булево значение `value` и преобразует его в целое число (`value|0` преобразует `false` в 0, а `true` в 1). Затем она «сдвигает значение влево», добавляя справа нули – их количество зависит от позиции `slot`, в которой нужно сохранить значение (`0b1<<0` так и остается равным `0b1`, `0b1<<1` равно `0b10` и т. д.). Затем она сдвигает число 1 влево на `slot` позиций (т. е. получается `0b1000`, если `slot` равно 3), инвертирует биты (с помощью операции `~`) и получает новое значение, применяя операцию И (`&`) к существующему значению и результату предыдущей операции (`view[0] & ~(1 << slot)`). Наконец, оба значения, измененное старое и новое, объединяются с помощью операции ИЛИ (`|`), и результат присваивается элементу `view[0]`. Смысл всего этого заключается в том, чтобы прочитать текущие биты, заменить интересующий бит и записать результат обратно.

Функция `getBool()` сдвигает число 1 влево на `slot` позиций, а затем применяет к результату и текущему значению операцию `&`. Сдвинутое значение (справа от знака `&`) содержит одну единицу и семь нулей. В результате применения к нему и текущему значению операции И получается число, представляющее значение в позиции `slot`; оно отлично от нуля, если в этой позиции находилась единица, и равно нулю в противном случае. Мы сравниваем это значение с нулем (`===0`) и берем логическое отрицание результата (`!`). В итоге возвращается булево значение бита в позиции `slot`.

У этого кода есть кое-какие недостатки и использовать его в реальной программе не стоит. Например, он не работает с буферами, содержащими более одного байта, т. е. попытка прочитать или записать значение в позицию, большую 7, приводит к неопределенному поведению. Производственная версия должна учитывать размер буфера и проверять выход за границы, но это мы оставляем в качестве упражнения для читателя.

Строки

Кодировать строки не так просто, как может показаться на первый взгляд. Сразу хочется предположить, что каждый символ строки можно представить одним байтом и что свойства строки `.length` достаточно для выбора размера буфера, в котором она будет сохранена. Иногда это и вправду работает,

особенно для простых строк, но при работе с более сложными данными неизбежны ошибки.

С простыми строками это проходит, потому что данные в них представлены в кодировке ASCII, предполагающей размещение одного символа в одном байте. Собственно, в языке C тип для представления одного байта данных так и называется `char`.

Существует много способов кодирования символов с помощью строк. Кодировка ASCII использует для представления всего диапазона символов один байт, но в мире, где много разных культур, языков и картинок-эмодзи, такое представление абсолютно невозможно. Поэтому используются кодировки с переменным числом байтов для представления одного символа. На внутреннем уровне движки JavaScript применяют различные форматы кодирования для представления строк в зависимости от ситуации, но вся эта сложность от приложений скрыта. Один из возможных внутренних форматов, UTF-16, использует для кодирования одного символа 2 или 4 байта, а для некоторых эмодзи даже 14 байт. В более широко распространенном формате UTF-8 для кодирования символов используется от 1 до 4 байт, причем он обладает обратной совместимостью с ASCII.

В следующем примере показано, что бывает, когда байты строки перебираются по одному до длины `.length` и результирующие символы помещаются в экземпляр `Uint8Array`:

```
// Осторожно: антипаттерн!
function stringToArrayBuffer(str) {
  const buffer = new ArrayBuffer(str.length);
  const view = new Uint8Array(buffer);
  for (let i = 0; i < str.length; i++) {
    view[i] = str.charCodeAt(i);
  }
  return view;
}

stringToArrayBuffer('foo'); // Uint8Array(3) [ 102, 111, 111 ]
stringToArrayBuffer('€'); // Uint8Array(1) [ 172 ]
```

В данном случае с сохранением простой строки `foo` проблем не возникает. Однако символ `€` представлен значением 8364, которое больше максимально допустимого (255), поддерживаемого классом `Uint8Array`, поэтому он был усечен до 172. После преобразования этого числа обратно в символ получается неверное значение.

В современных версиях JavaScript имеется API для кодирования и декодирования строк прямо в экземплярах `ArrayBuffer`. Этот API предоставляется глобальными классами `TextEncoder` и `TextDecoder`, имеющимися как в браузерах, так и в Node.js. В них применяется кодировка UTF-8, поскольку она распространена повсеместно.

В примере ниже показано, как безопасно представлять строки в кодировке UTF-8 с помощью этого API:

```
const enc = new TextEncoder();
enc.encode('foo'); // Uint8Array(3) [ 102, 111, 111 ]
enc.encode('€');   // Uint8Array(3) [ 226, 130, 172 ]
```

А вот как такие значения декодируются:

```
const ab = new ArrayBuffer(3);
const view = new Uint8Array(ab);
view[0] = 226; view[1] = 130; view[2] = 172;
const dec = new TextDecoder();
dec.decode(view); // '€'
dec.decode(ab);   // '€'
```

Отметим, что метод `TextDecoder#decode()` работает как с представлением `Uint8Array`, так и с самим экземпляром `ArrayBuffer`. Поэтому он удобен для декодирования данных, поступивших из сети, без предварительного обертывания их представлением.

Объекты

Учитывая, что JavaScript уже умеет представлять объекты в виде строк в формате JSON, у нас есть возможность взять объект, который мы хотим разделить между двумя потоками, сериализовать его в виде JSON-строки и записать эту строку в буферный массив с помощью API `TextEncoder`, который мы видели в предыдущем разделе. Вот код, который это и делает:

```
const enc = new TextEncoder();
return enc.encode(JSON.stringify(obj));
```

Метод `JSON.stringify(obj)` принимает объект JavaScript и преобразует его в строковое представление. При этом выходная строка сильно избыточна. Если нужно снизить размер полезной нагрузки до минимума, то можно воспользоваться форматом MessagePack (https://msgpack.org/), который уменьшает размер сериализованного объекта, представляя его метаданные в двоичном виде. Из-за этого MessagePack может не подойти в случаях, когда необходим чистый текст, например в почтовых сообщениях, но когда между частями приложения можно передавать двоичные буферы, это как раз то, что надо. Соответствующий npm-пакет `msgpack5` совместим с браузером и Node.js.

Впрочем, потери производительности при взаимодействии потоков обычно связаны не с размером передаваемой полезной нагрузки, а с расходами на ее сериализацию и десериализацию. Поэтому предпочтительнее передавать между потоками более простые представления данных. И даже когда приходится передавать объекты, может оказаться, что алгоритм структурированного клонирования в сочетании с методами `.onmessage` и `.postMessage` работает быстрее и безопаснее, чем сериализация объектов с последующей записью их в буфер.

Если вы пишете приложение, которое сериализует и десериализует объекты и помещает их в `SharedArrayBuffer`, то имеет смысл хотя бы частично переосмыслить архитектуру приложения. Почти всегда будет эффективнее сериализовать передаваемые объекты с использованием типов низкого уровня и передавать их в таком виде.

Глава 5

Дополнительные способы работы с разделяемой памятью

В главе 4 мы говорили об использовании объекта `SharedArrayBuffer` для непосредственного чтения и записи разделяемых данных из разных потоков. Но это рискованное дело, поскольку один поток может затереть данные, записанные другим. Впрочем, благодаря объекту `Atomics` мы можем выполнять очень простые операции с данными, не опасаясь их повреждения.

Хотя простые операции, поддерживаемые `Atomics`, удобны, часто возникает необходимость работать с данными более сложным образом. Например, допустим, что мы сериализовали данные, как описано в разделе «Сериализация данных» главы 4, получили строку длиной 1 КБ и хотим записать ее в экземпляр `SharedArrayBuffer`. Увы, ни один из методов `Atomics` не позволит атомарно записать значение целиком.

В этой главе рассматривается дополнительная функциональность, позволяющая координировать чтение и запись разделяемых данных из разных потоков в ситуациях, когда методов `Atomics` недостаточно.

Атомарные методы координации

Эти методы немного отличаются от рассмотренных в предыдущей главе. Именно те методы, что рассматривались ранее, работают с типизированными массивами любого вида и применимы к экземплярам `SharedArrayBuffer` и `ArrayBuffer`. Те же методы, что мы рассмотрим ниже, работают только с экземплярами `Int32Array` и `BigInt64Array` и имеют смысл только при использовании вместе с экземплярами `SharedArrayBuffer`.

Попытавшись применить их к неподдерживаемому типу `TypedArray`, вы получите одну из следующих ошибок:

```
# Firefox v88
Uncaught TypeError: invalid array type for the operation
```

```
# Chrome v90 / Node.js v16
Uncaught TypeError: [object Int8Array] is not an int32 or BigInt64 typed array.
```

Ничто не рождается на пустом месте, и эти методы спроектированы по образцу *фьютексов* в ядре Linux; слово futex означает «fast userspace mutex» (быстрый мьютекс в пространстве пользователя). А слово mutex (*мьютекс*) в свою очередь образовано от «mutual exclusion» (взаимное исключение) и означает, что один поток выполнения получает монопольный доступ к некоторым данным. Мьютекс можно также называть *блокировкой*, понимая под этим, что один поток блокирует доступ к данным, делает свое дело, а затем разблокирует доступ и позволяет другим потокам обратиться к данным. Фьютекс включает две простые операции: «ждать» (wait) и «разбудить» (wake).

Atomics.wait()

```
status = Atomics.wait(typedArray, index, value, timeout = Infinity)
```

Этот метод сначала проверяет, содержит ли элемент массива `typedArray` с индексом `index` значение, равное `value`. Если нет, то возвращается значение `not-equal`. Если же значения равны, то выполнение потока приостанавливается на `timeout` миллисекунд. Если за это время ничего не произойдет, то функция возвращает значение `timed-out`. Если же другой поток вызовет функцию `Atomics.notify()` для элемента с тем же индексом, то функция вернет значение `ok`. Все сказанное сведено в табл. 5.1.

Таблица 5.1. Значения, возвращаемые `Atomics.wait()`

Значение	Семантика
`not-equal`	Переданное значение не равно значению элемента массива
`timed-out`	Другой поток не вызвал `Atomics.notify()` в отведенное время
`ok`	Другой поток вызвал `Atomics.notify()` в отведенное время

Возникает вопрос, почему этот метод не возбуждает исключения в первых двух случаях, а продолжает работать. Поскольку главная цель многопоточного программирования – повышение производительности, легко понять, что эти методы `Atomics` вызываются на *критических путях*, т. е. там, где приложение проводит большую часть времени. В JavaScript создание объектов `Error` и генерирование трасс вызовов обходится дороже, чем возврат простой строки, поэтому такой подход обеспечивает довольно высокую производительность. Другая причина заключается в том, что возврат `not-equal` – это и не ошибка вовсе, а просто означает, что то, чего мы ждем, уже произошло.

Такая блокировка поначалу шокирует. Идея блокировать весь поток кажется чрезмерно экстравагантной, и во многих случаях так оно и есть. Другой пример действия, вызывающего блокировку всего потока JavaScript, – вызов функции `alert()` в браузере. В этом случае браузер открывает диалоговое окно, и больше ничего – даже фоновые задачи, пользующиеся циклом событий – работать не может, пока это окно не будет закрыто. Метод `Atomics.wait()` замораживает поток точно так же.

Это поведение настолько радикально, что «главному» потоку – потоку по умолчанию, который всегда существует при выполнении JavaScript и не совпадает с веб-исполнителями – не разрешено вызывать этот метод, по крайней мере, в браузере. Причина в том, что блокировка главного потока так мешает пользователям, что авторы API не пожелали допускать это. Если вы все-таки попробуете вызвать этот метод в главном потоке, то получите одну из следующих ошибок:

```
# Firefox
Uncaught TypeError: waiting is not allowed on this thread

# Chrome v90
Uncaught TypeError: Atomics.wait cannot be called in this context
```

Node.js, впрочем, разрешает вызывать `Atomics.wait()` в главном потоке. Поскольку у Node.js нет пользовательского интерфейса, это необязательно плохо. Более того, даже полезно в скриптах, где допустим вызов `fs.readFileSync()`.

Если вы из тех JavaScript-разработчиков, которым доводилось работать в компании, занимающейся созданием приложений для мобильных устройств или настольных компьютеров, то, наверное, слышали, как коллеги говорят о «разгрузке главного потока» или «блокировке главного потока». Эти заботы, которые традиционно доставались разработчикам приложений на языках низкого уровня, будут все больше докучать и нам, пишущим на JavaScript, по мере развития языка. Применительно к браузерам эта проблема часто называется *scroll jank* (тормозной скроллинг) и означает, что процессор слишком занят, чтобы отрисовывать пользовательский интерфейс во время скроллинга.

Atomics.notify()

```
awaken = Atomics.notify(typedArray, index, count = Infinity)
```

Метод `Atomics.notify()`[1] пытается разбудить потоки, вызвавшие `Atomics.wait()` для того же массива `typedArray` и индекса `index`. Если какие-то потоки в данный момент заморожены, то они проснутся. Одновременно может быть заморожено несколько потоков, ждущих уведомления. Аргумент `count` говорит, сколько из них следует разбудить. По умолчанию `count` равен `Infinity`, т. е. будут разбужены все потоки. Но если, например, ждут четыре потока, а `count` равен трем, то будут разбужены все, кроме одного. В разделе «Хронометраж и недетерминированность» ниже описывается, в каком порядке пробуждаются потоки.

Возвращается количество разбуженных потоков. Если передан экземпляр `TypedArray`, указывающий на неразделяемый экземпляр `ArrayBuffer`, то возвращается 0. Если оказалось, что в момент вызова не было ни одного жду-

[1] `Atomics.notify()` первоначально хотели назвать `Atomics.wake()`, как во фьютексах Linux, но позже его переименовали, чтобы предотвратить путаницу из-за похожего написания «wake» и «wait».

щего потока, то также возвращается 0. Поскольку этот метод не блокирует поток, его можно вызывать из главного потока JavaScript.

Atomics.waitAsync()

```
promise = Atomics.waitAsync(typedArray, index, value, timeout = Infinity)
```

Эта версия `Atomics.wait()`, основанная на обещании, – недавнее добавление в семейство `Atomics`. На момент написания книги она была доступна в Node.js v16 и Chrome v87, но еще не вошла в Firefox и Safari.

Этот неблокирующий и менее производительный вариант `Atomics.wait()` возвращает обещание, результатом разрешения которого является состояние операции ожидания. Из-за потери части производительности (разрешение обещания требует больших накладных расходов, чем приостановка потока и возврат строки) он не так полезен на критическом пути алгоритма, выполняющего большой объем вычислений. С другой стороны, он может быть полезен в ситуациях, когда изменение состояния блокировки – более удобный механизм отправки сигнала другому потоку, чем передача сообщения с помощью `postMessage()`. Поскольку этот метод не блокирует поток, его можно вызывать из главного потока приложения.

Один из побудительных мотивов для добавления этого метода – желание, чтобы коду, откомпилированному с помощью программы Emscripten (рассматриваемой в разделе «Компиляция программ на C в WebAssembly с помощью Emscripten»), в котором используются потоки, было разрешено выполняться в главном, а не только в рабочих потоках.

Хронометраж и недетерминированность

Корректное приложение обычно должно вести себя детерминировано. Функция `Atomics.notify()` принимает аргумент `count` – число подлежащих пробуждению потоков. Очевидный вопрос – какие именно потоки пробуждаются и в каком порядке?

Пример недетерминированности

Потоки пробуждаются в порядке *FIFO* (первым пришел, первым ушел), т. е. первый поток, вызвавший `Atomics.wait()`, будет разбужен первым, второй вызвавший поток – вторым и т. д. Но удостовериться в этом трудно, потому что сообщения, печатаемые разными исполнителями на консоли, необязательно отображаются на экране именно в том порядке, в котором были выведены. В идеале приложение следует писать так, чтобы оно продолжало работать вне зависимости от порядка пробуждения потоков.

Для проверки можете создать новое приложение. Сначала создайте каталог *ch5-notify-order/*, а в нем – файл *index.html*, содержащий код из примера 5.1.

Пример 5.1 ❖ *ch5-notify-order/index.html*

```
<html>
  <head>
    <title>Shared Memory for Coordination</title>
    <script src="main.js"></script>
  </head>
</html>
```

Затем создайте файл *main.js* и скопируйте в него код из примера 5.2.

Пример 5.2 ❖ *ch5-notify-order/main.js*

```
if (!crossOriginIsolated) throw new Error('Cannot use SharedArrayBuffer');

const buffer = new SharedArrayBuffer(4);
const view = new Int32Array(buffer);

for (let i = 0; i < 4; i++) { ❶
  const worker = new Worker('worker.js');
  worker.postMessage({buffer, name: i});
}

setTimeout(() => {
  Atomics.notify(view, 0, 3); ❷
}, 500); ❸
```

❶ **Создается четыре выделенных исполнителя.**
❷ **Отправляется уведомление, связанное с позицией 0 разделяемого буфера.**
❸ **Уведомление отправляется через полсекунды.**

Здесь создается 4-байтовый буфер – наименьший, способный поддержать представление `Int32Array`. Затем в цикле `for` создается четыре выделенных исполнителя. Каждый исполнитель немедленно вызывает метод `postMessage()`, передавая буфер и идентификатор потока. В итоге получается пять потоков: главный и четыре исполнителя с именами 0, 1, 2, 3.

JavaScript создает потоки, а движок за кулисами аккумулирует ресурсы, выделяет память и т. д. Сколько времени это займет, заранее сказать нельзя, и это печально. Например, мы не можем утверждать, что на завершение подготовительной работы уйдет 100 мс. В действительности это во многом зависит от машины – сколько на ней процессорных ядер, сильно ли она загружена и т. п. Но, к счастью для нас, вызовы `postMessage()` ставятся в очередь; движок JavaScript вызовет функцию исполнителя `onmessage`, как только будет готов.

Создав рабочие потоки, главный поток ждет полсекунды (500 мс) с помощью `setTimeout` и затем вызывает `Atomics.notify()`. Что будет, если задать слишком маленький тайм-аут, скажем 10 мс? Или если вообще вызвать `notify()` без тайм-аута? В таком случае потоки еще не инициализированы, они еще не успели вызвать `Atomics.wait()`, и `notify()` немедленно вернет значение 0. А что, если тайм-аут слишком велик? Тогда приложение будет работать мучительно медленно или истечет тайм-аут, заданный при вызове `Atomics.wait()`.

На ноутбуке Томаса порог готовности составляет где-то 120 мс. В этот момент одни потоки готовы, а другие еще нет. Если тайм-аут равен 100 мс, то обычно не готов ни один поток, а если 180 мс, то обычно готовы все потоки. Но программисты не любят слова *обычно*. Трудно точно сказать, когда поток будет готов. Часто эта проблема возникает при запуске приложения, а не в процессе его жизненного цикла.

Чтобы завершить приложение, создайте файл *worker.js* и скопируйте в него код из примера 5.3.

Пример 5.3 ❖ *ch5-notify-order/worker.js*

```
self.onmessage = ({data: {buffer, name}}) => {
  const view = new Int32Array(buffer);
  console.log(`Исполнитель ${name} запущен`);
  const result = Atomics.wait(view, 0, 0, 1000); ❶
  console.log(`Исполнитель ${name} разбужен с кодом ${result}`);
};
```

❶ **Ждать не более 1 с у элемента 0 буфера в предположении, что начальное значение равно 0.**

Исполнитель принимает разделяемый буфер и имя рабочего потока, сохраняет эти значения и печатает сообщение о том, что поток инициализирован. Затем он вызывает метод `Atomics.wait()` для нулевого элемента буфера. Предполагается, что начальное значение этого элемента равно 0 (так оно и есть, потому что мы его не изменяли). При вызове метода задан тайм-аут 1 с (1000 мс). Наконец, по завершении метода на терминал выводится возвращенное значение.

Создав все необходимые файлы, перейдите в окно терминала и запустите еще один веб-сервер, чтобы видеть результаты. Для этого выполните команду

```
$ npx MultithreadedJSBook/serve .
```

Как обычно, введите в браузере напечатанный в ответ URL-адрес и откройте консоль. Если на ней ничего нет, обновите страницу. В табл. 5.2 приведен результат тестового прогона.

Таблица 5.2. Пример недетерминированного вывода

Сообщение	Место
Исполнитель 1 запущен	worker.js:4:11
Исполнитель 0 запущен	worker.js:4:11
Исполнитель 3 запущен	worker.js:4:11
Исполнитель 2 запущен	worker.js:4:11
Исполнитель 0 разбужен с кодом ok	worker.js:7:11
Исполнитель 3 разбужен с кодом ok	worker.js:7:11
Исполнитель 1 разбужен с кодом ok	worker.js:7:11
Исполнитель 2 разбужен с кодом timed-out	worker.js:7:11

На вашей машине картина, скорее всего, будет другая. Даже просто обновив страницу, можно получить другой результат. А может быть, после каждо-

го обновления будет одно и то же. Но в идеале исполнитель, напечатавший «запущен» последним, будет разбужен с кодом «timed-out».

Этот результат может поначалу вызвать недоумение. Выше мы говорили о порядке FIFO, а тут числа напечатаны не в порядке от 0 до 3. Причина в том, что следование номеров зависит не от порядка создания потоков (0, 1, 2, 3), а от порядка выполнения `Atomics.wait()` (в данном случае – 1, 0, 3, 2). Но все равно смущает порядок сообщений «разбужен» (0, 3, 1, 2). Вероятно, это связано с состоянием гонки в движке JavaScript, возникающим, когда разные потоки печатают сообщения почти в одно и то же время.

Напечатанные сообщения не сразу отображаются на экране, иначе они наползали бы друг на друга, и все закончилось бы мешаниной пикселей. Поэтому движок ставит подлежащие печати сообщения в очередь, а некий внутренний механизм браузера, скрытый от разработчиков, определяет, в каком порядке они извлекаются из очереди и выводятся на экран. Поэтому порядок сообщений в обоих наборах может не совпадать. Но можно достоверно утверждать, что сообщение о тайм-ауте должно исходить от потока, запущенного последним. Так оно в данном случае и есть.

Определение готовности потока

Описанный эксперимент ставит вопрос: как приложение может детерминировано узнать, что поток инициализирован и готов принять задачу?

Простой способ – вызывать `postMessage()` из рабочих потоков где-то в обработчике `onmessage()`, отправляя сообщение родительскому потоку. Это работает, потому что коль скоро обработчик `onmessage()` вызван, значит, рабочий поток завершил инициализацию и теперь исполняет JavaScript-код.

Ниже приведен пример. Скопируйте созданный ранее каталог *ch5-notify-order/*, создав новый каталог *ch5-notify-when-ready/*. Файл *index.html* в нем будет прежним, а оба JavaScript-файла нужно изменить. Скопируйте в файл *main.js* код из примера 5.4.

Пример 5.4 ❖ *ch5-notify-when-ready/main.js*

```
if (!crossOriginIsolated) throw new Error('Cannot use SharedArrayBuffer');

const buffer = new SharedArrayBuffer(4);
const view = new Int32Array(buffer);
const now = Date.now();
let count = 4;

for (let i = 0; i < 4; i++) { ❶
  const worker = new Worker('worker.js');
  worker.postMessage({buffer, name: i}); ❷
  worker.onmessage = () => {
    console.log(`Готов; id=${i},count=${--count},time=${Date.now()-now}мс`);
    if (count === 0) { ❸
      Atomics.notify(view, 0);
    }
  };
}
```

❶ Создать четыре исполнителя.
❷ Сразу же отправить каждому исполнителю сообщение.
❸ Уведомить нулевой элемент, когда все исполнители ответят.

Скрипт модифицирован, так что `Atomics.notify()` вызывается, после того как все четыре исполнителя отправили сообщения главному потоку. Когда пришло сообщение от четвертого, последнего, исполнителя, посылается уведомление. Это позволяет приложению начать содержательную работу, как только оно будет к этому готово, и, вероятно, сэкономить сотни миллисекунд в лучшем случае и избежать ошибки в худшем (когда код исполняется на очень медленном одноядерном компьютере).

Вызов `Atomics.notify()` также был изменен – теперь он пробуждает все потоки, а не только три, и для тайм-аута установлено значение по умолчанию `Infinity`. Это сделано для того, чтобы показать, что каждый поток получает сообщение вовремя.

Теперь измените файл *worker.js*, скопировав в него код из примера 5.5.

Пример 5.5 ❖ *ch5-notify-when-ready/worker.js*

```
self.onmessage = ({data: {buffer, name}}) => {
  postMessage('ready'); ❶
  const view = new Int32Array(buffer);
  console.log(`Исполнитель ${name} запущен`);
  const result = Atomics.wait(view, 0, 0); ❷
  console.log(`Исполнитель ${name} разбужен с кодом ${result}`);
};
```

❶ Отправить сообщение родительскому потоку, сигнализируя о готовности.
❷ Ждать уведомления у нулевого элемента.

На этот раз обработчик `onmessage` сразу вызывает `postMessage()`, чтобы отправить сообщение родителю. Вскоре после этого начинается ожидание. Строго говоря, если родительский поток каким-то образом сумеет получить сообщение до того, как будет вызван метод `Atomics.wait()`, приложение будет работать неправильно. Но код полагается на тот факт, что передача сообщения занимает гораздо больше времени, чем выполнение нескольких строк синхронного JavaScript-кода.

Помните, что вызов `Atomics.wait()` приостанавливает поток. Это значит, что вызывать `postMessage()` после него нельзя.

При выполнении этого кода печатается еще три элемента: имя потока, обратный счетчик (всегда в порядке 3, 2, 1, 0) и время, прошедшее от момента запуска скрипта до готовности потока. Запустите сервер той же командой, что и раньше, и откройте выведенный ей URL-адрес в браузере. В табл. 5.3 показаны результаты, напечатанные в нескольких прогонах.

В данном случае на 16-разрядном ноутбуке Firefox, похоже, инициализирует рабочие потоки в четыре раз медленнее, чем Chrome. Кроме того, порядок запуска потоков в Firefox более случайный. При каждом обновлении страницы порядок запуска потоков в Firefox изменяется, а в Chrome нет. Это наводит на мысль, что в движке V8, используемом в Chrome, создание новых сред JavaScript или инициализация API браузера оптимизированы лучше, чем в движке SpiderMonkey, которым пользуется Firefox.

Таблица 5.3. Хронометраж запуска потоков

Firefox v88	Chrome v90
T1, 86 мс	T0, 21 мс
T0, 99 мс	T1, 24 мс
T2, 101 мс	T2, 26 мс
T3, 108 мс	T3, 29 мс

Обязательно протестируйте этот код в нескольких браузерах и сравните результаты. Также нужно иметь в виду, что время инициализации потоков с большой вероятностью зависит от числа имеющихся процессорных ядер. Чтобы было интереснее, измените значение 4 переменной `count` и счетчика цикла `for` на большее и посмотрите, что получится. После его увеличения до 128 время инициализации потоков в обоих браузерах резко подскочило. Кроме того, при этом стал нарушаться порядок запуска потоков в Chrome. Вообще, если потоков слишком много, то производительность падает, и этот феномен подробно изучен в разделе «Недостаточное число ядер» главы 8.

Пример приложения: игра «Жизнь» Конвея

Итак, мы познакомились с методами `Atomics.wait()` и `Atomics.notify()`, и самое время рассмотреть конкретный пример. В качестве такового мы возьмем игру «Жизнь» Конвея, которая естественно приводит к параллельному программированию. Смысл «игры» – моделирование увеличения и уменьшения популяции. «Мир» эволюционирует на сетке, каждая клетка которой может находиться в одном из двух состояний: жива или мертва. Моделирование производится итеративно, и на каждой итерации для каждой клетки выполняется следующий алгоритм.

1. Если клетка жива:
   a) если число живых соседей равно 2 или 3, то клетка остается живой;
   b) если число живых соседей равно 0 или 1, то клетка умирает (от одиночества);
   c) если число живых соседей равно 4 или более, то клетка умирает (от перенаселенности).
2. Если клетка мертва:
   a) если число живых соседей равно 3, то в клетке зарождается жизнь (размножение);
   b) в любом другом случае клетка остается мертвой.

Говоря о «живых соседях», мы имеем в виду клетки, отстоящие от данной на одну единицу, в том числе по диагонали, и их состояние на предыдущей итерации. Эти правила можно сформулировать проще.

1. Если живо ровно 3 соседа, новое состояние клетки – живая (вне зависимости от предыдущего состояния).
2. Если клетка живая и у нее ровно 2 живых соседа, то она остается живой.
3. Во всех остальных случаях новое состояние клетки – мертвая.

Для нашей реализации примем следующие предположения.

- Сетка квадратная. Это небольшое упрощение позволяет ограничиться всего одним измерением.
- Сетка свернута в тор. Это означает, что на краю, когда соседняя клетка оказывается за пределами сетки, мы рассматриваем клетку на противоположном краю.

Мы напишем код для браузера, потому что он предоставляет холст, на котором удобно рисовать состояние игры «Жизнь». Впрочем, сравнительно несложно адаптировать пример к другим средам, располагающим какими-то средствами отрисовки изображений. В Node.js можно было бы даже рисовать на экране терминала, пользуясь управляющими кодами ANSI.

Однопоточная игра «Жизнь»

Для начала создадим класс `Grid`, который содержит массив, представляющий мир в игре «Жизнь», и реализует все итерации. Мы сделаем его код безразличным к фронтальному интерфейсу и так, чтобы его можно было использовать без изменений в многопоточном примере. Для моделирования игры «Жизнь» нам понадобится многомерный массив клеток. Можно было бы использовать для этой цели массив массивов, но, чтобы упростить последующий код, мы будем хранить все клетки в одномерном массиве (на самом деле в `Uint8Array`), так что состояние клетки с координатами `x` и `y` будет находиться в элементе `cells[size * x + y]`. Нам понадобятся два таких массива – для текущего и предыдущего состояния. Чтобы еще больше упростить код, мы будем хранить их последовательно в одном объекте `ArrayBuffer`.

Создайте каталог *ch5-game-of-life/*, а в нем файл *gol.js*, в который скопируйте код из примера.

Пример 5.6 ❖ *ch5-game-of-life/gol.js (часть 1)*

```
class Grid {
  constructor(size, buffer, paint = () => {}) {
    const sizeSquared = size * size;
    this.buffer = buffer;
    this.size = size;
    this.cells = new Uint8Array(this.buffer, 0, sizeSquared);
    this.nextCells = new Uint8Array(this.buffer, sizeSquared, sizeSquared);
    this.paint = paint;
  }
```

Это конструктор класса `Grid`. Он принимает длину стороны квадрата в параметре `size`, объект `ArrayBuffer`, названный `buffer`, и функцию `paint`, которая понадобится нам позже. Далее мы создаем массивы `cells` и `nextCells`, являющиеся экземплярами `Uint8Array`, и располагаем их один за другим в буфере `buffer`.

Далее мы можем добавить метод чтения клетки, который понадобится при реализации итераций. Добавьте в файл код из примера 5.7.

Пример 5.7 ❖ *ch5-game-of-life/gol.js (часть 2)*

```
  getCell(x, y) {
    const size = this.size;
    const sizeM1 = size - 1;
    x = x < 0 ? sizeM1 : x > sizeM1 ? 0 : x;
    y = y < 0 ? sizeM1 : y > sizeM1 ? 0 : y;
    return this.cells[size * x + y];
}
```

Для получения клетки с заданными координатами необходимо нормализовать индексы. Напомним, что наша сетка свернута в тор. В процессе нормализации мы смотрим, выходит ли запрошенная клетка на один шаг за пределы сетки, и если да, берем клетку с противоположного края.

Теперь добавьте код, выполняемый на каждой итерации, из примера 5.8.

Пример 5.8 ❖ *ch5-game-of-life/gol.js (часть 3)*

```
  static NEIGHBORS = [ ❶
    [-1, -1], [-1, 0], [-1, 1], [0, -1], [0, 1], [1, -1], [1, 0], [1, 1]
  ];

  iterate(minX, minY, maxX, maxY) { ❷
    const size = this.size;

    for (let x = minX; x < maxX; x++) {
      for (let y = minY; y < maxY; y++) {
        const cell = this.cells[size * x + y];
        let alive = 0;
        for (const [i, j] of Grid.NEIGHBORS) {
          alive += this.getCell(x + i, y + j);
        }
        const newCell = alive === 3 || (cell && alive === 2) ? 1 : 0;
        this.nextCells[size * x + y] = newCell;
        this.paint(newCell, x, y);
      }
    }

    const cells = this.nextCells;
    this.nextCells = this.cells;
    this.cells = cells;
    }
  }
```

❶ **Множество координат соседей используется для поиска соседних клеток в восьми направлениях. Этот массив будет нужен при обработке каждой клетки.**

❷ **Метод `iterate()` принимает обрабатываемый диапазон, описываемый минимальными (включаются) и максимальными (не включаются) значениями X и Y. В однопоточном примере он всегда будет равен `(0, 0, size, size)`, но его параметрическое задание упростит разбиение диапазона в многопоточной версии, где задание границ X и Y позволит выделить участки, обрабатываемые каждым потоком.**

Мы в цикле обходим все клетки сетки и для каждой получаем число живых соседей. Число `1` представляет живые клетки, число `0` – мертвые, поэтому, чтобы подсчитать число живых соседей, нужно эти числа сложить. Зная

число соседей, мы можем применить упрощенный алгоритм игры «Жизнь». Сохраняем новое состояние клетки в массиве `nextCells`, а затем передаем новое состояние и координаты функции `paint` для визуализации. После этого меняем местами массивы `cells` и `nextCells` для следующей итерации. Таким образом, на каждой итерации `cells` содержит результат предыдущей итерации, а `newCells` – результат текущей итерации.

До этого момента код не отличался от того, что будет использован в многопоточной версии. Теперь мы можем создать экземпляр класса `Grid` и привязать к пользовательскому интерфейсу. Добавьте в файл код из примера 5.9.

Пример 5.9 ❖ *ch5-game-of-life/gol.js (часть 4)*

```
const BLACK = 0xFF000000; ❶
const WHITE = 0xFFFFFFFF;
const SIZE = 1000;

const iterationCounter = document.getElementById('iteration'); ❷
const gridCanvas = document.getElementById('gridcanvas');
gridCanvas.height = SIZE;
gridCanvas.width = SIZE;
const ctx = gridCanvas.getContext('2d');
const data = ctx.createImageData(SIZE, SIZE); ❸
const buf = new Uint32Array(data.data.buffer);

function paint(cell, x, y) { ❹
  buf[SIZE * x + y] = cell ? BLACK : WHITE;
}

const grid = new Grid(SIZE, new ArrayBuffer(2 * SIZE * SIZE), paint); ❺
for (let x = 0; x < SIZE; x++) { ❻
  for (let y = 0; y < SIZE; y++) {
    const cell = Math.random() < 0.5 ? 0 : 1;
    grid.cells[SIZE * x + y] = cell;
    paint(cell, x, y);
  }
}

ctx.putImageData(data, 0, 0); ❼
```

❶ **Задаем константы, описывающие черные и белые пиксели на экране, и размер (длину стороны квадрата) сетки. Можете поэкспериментировать с заданием размера, чтобы посмотреть, как развивается игра «Жизнь» на разных сетках.**

❷ **Берем число итераций и холст из HTML-кода (который напишем позже). Задаем ширину и высоту холста равными `SIZE` и получаем двумерный контекст, на котором будем работать.**

❸ **Мы используем экземпляр класса `ImageData` для прямой модификации пикселей на холсте в соответствии с данными в массиве `Uint32Array`.**

❹ **Это та самая функция `paint()`, которая используется для инициализации сетки и на каждой итерации, чтобы модифицировать буфер, стоящий за объектом `ImageData`. Если клетка живая, то она рисуется черным цветом, иначе белым.**

❺ **Создаем экземпляр сетки, передавая размер, буфер `ArrayBuffer`, достаточно большой для хранения `cells` и `nextCells`, и нашу функцию `paint()`.**

❻ **Для инициализации сетки обходим в цикле все клетки и каждой случайным образом присваиваем состояние: живая или мертвая. И тут же передаем результат функции `paint()`, чтобы та обновила изображение.**

❼ **После модификации `ImageData` мы должны перенести изображение на холст. После этого инициализация завершена.**

Наконец-то мы готовы начать итерации. Добавьте код из примера 5.10.

Пример 5.10 ❖ *ch5-game-of-life/gol.js (часть 5)*

```
let iteration = 0;
function iterate(...args) {
  grid.iterate(...args);
  ctx.putImageData(data, 0, 0);
  iterationCounter.innerHTML = ++iteration;
  window.requestAnimationFrame(() => iterate(...args));
}

iterate(0, 0, SIZE, SIZE);
```

Каждая итерация начинается с вызова метода `grid.iterate()`, который модифицирует клетки. Отметим, что он вызывает функцию `paint()` для каждой клетки, поэтому после возврата из него данные изображения готовы, и нам нужно только перенести их на холст, вызвав `putImageData()`. Затем мы увеличиваем счетчик итераций и планируем следующую итерацию в обратном вызове `requestAnimationFrame()`. И напоследок запускаем программу вызовом `iterate()`.

JavaScript-код написан, осталось написать объемлющий HTML-код. К счастью, он совсем коротенький. Добавьте код из примера 5.11 в файл *gol.html* в том же каталоге, а затем откройте этот файл в браузере.

Пример 5.11 ❖ *ch5-game-of-life/gol.html*

```
<h3>Iteration: <span id="iteration">0</span></h3>
<canvas id="gridcanvas"></canvas>
<script src="gol.js"></script>
```

На экране должно появиться изображение сетки 1000×1000, на которой развивается игра «Жизнь» Конвея. Итерации выполняются с максимально возможной скоростью. Выглядит это так, как показано на рис. 5.1. Если компьютер слабый, то возможны небольшие задержки, а вообще-то, анимация должна быть плавной. Для обхода всех клеток и выполнения вычислений требуется мощный процессор. Чтобы немного ускорить работу, воспользуемся дополнительными процессорными ядрами, взяв на вооружение потоки.

Многопоточная игра «Жизнь»

В многопоточной версии игры «Жизнь» можно повторно использовать значительную часть уже написанного кода. В частности, не меняется ни HTML-файл, ни класс `Grid`. Мы запустим несколько рабочих потоков и один дополнительный, который будет заниматься координацией и модифицировать данные изображения. Этот дополнительный поток нужен, потому что использовать метод `Atomics.wait()` в главном потоке браузера запрещено. Мы

воспользуемся классом SharedArrayBuffer вместо ArrayBuffer. Для координации потоков нам понадобится 8 байт, по четыре для синхронизации в каждом направлении, так как Atomics.wait() требует по крайней мере экземпляра Int32Array. Поскольку поток-координатор будет также генерировать данные изображения, нам потребуется достаточно разделяемой памяти для их хранения. Для сетки с длиной стороны SIZE буфер SharedArrayBuffer будет устроен, как показано в табл. 5.4.

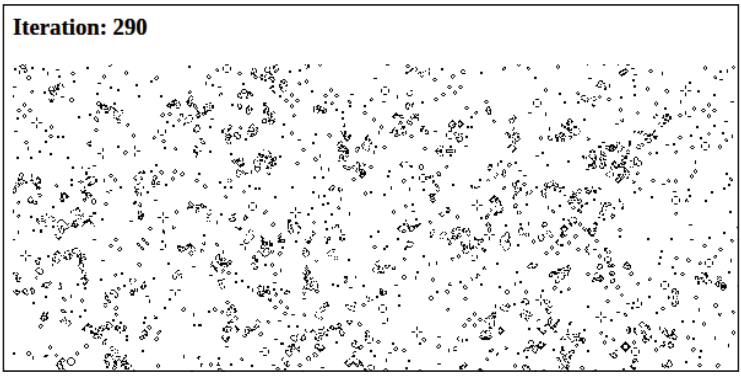


Рис. 5.1 ❖ Игра «Жизнь» Конвея после 290 итераций

Таблица 5.4. Структура памяти для четырех рабочих потоков

Назначение	Число байтов
cells (или nextCells)	SIZE * SIZE
cells (или nextCells)	SIZE * SIZE
Данные изображения	4 * SIZE * SIZE
Ожидание рабочего потока	4
Ожидание потока-координатора	4

Скопируйте *.html* и *.js*-файлы из предыдущего примера и назовите их соответственно *thread-gol.html* и *thread-gol.js*. Измените *thread-gol.html*, так чтобы он ссылался на новый JavaScript-файл.

Удалите все после определения класса Grid. Дальше нужно будет определить некоторые константы. Добавьте код из примера 5.12 в файл *thread-gol.js*.

Пример 5.12 ❖ *ch5-game-of-life/thread-gol.js (часть 1)*

```
const BLACK = 0xFF000000;
const WHITE = 0xFFFFFFFF;
const SIZE = 1000;
const THREADS = 5; // должно быть делителем SIZE
const imageOffset = 2 * SIZE * SIZE
const syncOffset = imageOffset + 4 * SIZE * SIZE;
const isMainThread = !!self.window;
```

Константы BLACK, WHITE и SIZE имеют тот же смысл, что и в однопоточной версии. Константа THREADS может быть любым делителем SIZE, она равна чис-

лу потоков, производящих вычисления в игре «Жизнь». Мы разобьем сетку на участки, которые могут обрабатываться отдельными потоками. Можете поэкспериментировать со значениями THREADS и SIZE, только не забывайте, что THREADS должно быть делителем SIZE. Нам понадобятся смещения данных изображения и байтов синхронизации от начала буфера, поэтому заведем константы и для них тоже. Наконец, мы собираемся исполнять один и тот же файл в главном и рабочих потоках, поэтому нужно знать, находимся мы в главном потоке или нет.

Далее напишем код главного потока. Добавьте в файл содержимое примера 5.13.

Пример 5.13 ❖ *ch5-game-of-life/thread-gol.js (часть 2)*

```
if (isMainThread) {
  const gridCanvas = document.getElementById('gridcanvas');
  gridCanvas.height = SIZE;
  gridCanvas.width = SIZE;
  const ctx = gridCanvas.getContext('2d');
  const iterationCounter = document.getElementById('iteration');

  const sharedMemory = new SharedArrayBuffer( ❶
    syncOffset + // data + imageData
    THREADS * 4 // для синхронизации
  );
  const imageData = new ImageData(SIZE, SIZE);
  const cells = new Uint8Array(sharedMemory, 0, imageOffset);
  const sharedImageBuf = new Uint32Array(sharedMemory, imageOffset);
  const sharedImageBuf8 =
    new Uint8ClampedArray(sharedMemory, imageOffset, 4 * SIZE * SIZE);
  for (let x = 0; x < SIZE; x++) {
    for (let y = 0; y < SIZE; y++) {
      // клетка живая с вероятностью 50%
      const cell = Math.random() < 0.5 ? 0 : 1;
      cells[SIZE * x + y] = cell;
      sharedImageBuf[SIZE * x + y] = cell ? BLACK : WHITE;
    }
  }

  imageData.data.set(sharedImageBuf8);
  ctx.putImageData(imageData, 0, 0);
```

❶ **SharedArrayBuffer продолжается еще на 20 байт после syncOffset, потому что нам нужно 4 байта для синхронизации каждого из пяти потоков.**

Первая часть примерно такая же, как в однопоточной версии. Мы просто читаем элементы DOM и устанавливаем размер сетки. Затем создается экземпляр SharedArrayBuffer, который мы назвали sharedMemory, и его представления для доступа к массиву cells (который мы скоро заполним) и к данным изображения. Для данных изображения мы будем использовать Uint32Array и Uint8ClampedArray соответственно для модификации и присваивания экземпляру ImageData.

После этого мы случайным образом инициализируем сетку и попутно модифицируем данные изображения и копируем эти данные в контекст холста. На этом подготовка начального состояния сетки завершается. Теперь можно запускать рабочие потоки. Добавьте в файл код из примера 5.14.

Пример 5.14 ❖ *ch5-game-of-life/thread-gol.js (часть 3)*

```
const chunkSize = SIZE / THREADS;
for (let i = 0; i < THREADS; i++) {
  const worker = new Worker('thread-gol.js', { name: `gol-worker-${i}` });
  worker.postMessage({
    range: [0, chunkSize * i, SIZE, chunkSize * (i + 1)],
    sharedMemory,
    i
  });
}

const coordWorker = new Worker('thread-gol.js',
                               { name: 'gol-coordination' });
coordWorker.postMessage({ coord: true, sharedMemory });

let iteration = 0;
coordWorker.addEventListener('message', () => {
  imageData.data.set(sharedImageBuf8);
  ctx.putImageData(imageData, 0, 0);
  iterationCounter.innerHTML = ++iteration;
  window.requestAnimationFrame(() => coordWorker.postMessage({}));
});
```

В цикле создаются рабочие потоки. Каждому мы для отладки присваиваем уникальное имя и передаем сообщение, которое говорит, за какой участок сетки (описываемый координатами углов `minX`, `minY`, `maxX` и `maxY`) он отвечает. В том же сообщении мы передаем ему ссылку на разделяемую память `sharedMemory`. Затем добавляется поток-координатор, ему также передается `sharedMemory` и признак, извещающий, что он является координатором.

Главный поток будет общаться только с координатором. Мы настраиваем его таким образом, чтобы он отправлял сообщение всякий раз, как получает сообщение, но только после того, как выберет данные изображения из `SharedMemory`, произведет необходимые обновления UI и запросит кадр анимации.

Весь остальной код работает в других потоках. Добавьте код из примера 5.15.

Пример 5.15 ❖ *ch5-game-of-life/thread-gol.js (часть 4)*

```
} else {
  let sharedMemory;
  let sync;
  let sharedImageBuf;
  let cells;
  let nextCells;

  self.addEventListener('message', initListener);
```

```
function initListener(msg) {
  const opts = msg.data;
  sharedMemory = opts.sharedMemory;
  sync = new Int32Array(sharedMemory, syncOffset);
  self.removeEventListener('message', initListener);
  if (opts.coord) {
    self.addEventListener('message', runCoord);
    cells = new Uint8Array(sharedMemory);
    nextCells = new Uint8Array(sharedMemory, SIZE * SIZE);
    sharedImageBuf = new Uint32Array(sharedMemory, imageOffset);
    runCoord();
  } else {
    runWorker(opts);
  }
}
```

Здесь выполняется ветвь `else` предложения, проверяющего переменную `isMainThread`, т. е. мы находимся в рабочем потоке или в потоке-координаторе. Объявляется несколько переменных, а затем добавляется начальный прослушиватель события `message`. Неважно, что это – координатор или рабочий поток, – нужно присвоить значения переменным `sharedMemory` и `sync`, что мы и делаем в прослушивателе. Затем удаляем начальный прослушиватель, поскольку больше он не понадобится. Рабочие потоки вообще не используют механизм передачи сообщений, а у координатора будет другой прослушиватель, как мы скоро увидим.

Если инициализируется поток-координатор, то мы добавляем новый прослушиватель `message` – функцию `runCoord`, которую определим позже. Затем мы получаем ссылки на `cells` и `nextCells`, потому что в координаторе нужно отслеживать положение независимо от того, что происходит в экземплярах `Grid` в рабочих потоках. Поскольку изображение генерируется координатором, нам нужна и ссылка на него тоже. Далее выполняется первая итерация `runCoord`. Если инициализируется рабочий поток, то мы просто передаем параметры (описание порученного ему участка сетки) функции `runWorker()`.

Теперь определим функцию `runWorker()`. Добавьте код из примера 5.16.

Пример 5.16 ❖ *ch5-game-of-life/thread-gol.js (часть 5)*

```
function runWorker({ range, i }) {
  const grid = new Grid(SIZE, sharedMemory);
  while (true) {
    Atomics.wait(sync, i, 0);
    grid.iterate(...range);
    Atomics.store(sync, i, 0);
    Atomics.notify(sync, i);
  }
}
```

Только рабочим потокам нужен экземпляр класса `Grid`, поэтому мы сначала создаем его, передавая конструктору ссылку `sharedMemory` на буфер данных. Это работает, потому что мы решили, что в начале `sharedMemory` будут расположены массивы `cells` и `nextCells`, как и в однопоточной версии.

Затем запускается бесконечный цикл. В нем выполняются следующие действия.

1. Вызывается метод `Atomics.wait()` для `i`-го элемента массива `sync`. В потоке-координаторе мы позаботимся о соответствующем вызове `Atomics.notify()`, чтобы этот поток мог продолжить выполнение. Здесь мы ждем координатора, потому что в противном случае могли бы начать изменять данные и обменивать ссылки на `cells` и `nextCells` еще до того, как остальные потоки будут готовы и данные дойдут до главного потока браузера.
   Затем выполняется одна итерация обработки экземпляра `Grid`. Напомним, что мы обрабатываем только ту часть сетки, о которой координатор сообщил нам с помощью свойства `range`.
2. По завершении итерации мы уведомляем главный поток о том, что выполнили задачу. Для этого `i`-й элемент массива `sync` устанавливается в 1 методом `Atomics.store()`, после чего мы пробуждаем ждущий у этого элемента поток методом `Atomics.notify()`. Переход от 0 к 1 служит сигналом о необходимости выполнить некоторую работу, а обратный переход к 0 – сигналом о том, что работа выполнена.

Мы используем `Atomics.wait()`, чтобы приостановить выполнение потока-координатора на время, пока рабочие потоки модифицируют данные, а затем приостановить рабочие потоки, пока координатор занимается своим делом. Потоки по обе стороны вызывают метод `Atomics.notify()`, чтобы разбудить другой поток, после чего сразу переходят в состояние ожидания уведомления от другого потока. Поскольку мы применяем атомарные операции для модификации данных и управления после модификации, гарантируется, что все обращения к данным *последовательно согласованы*. В программе, где выполнение потоков чередуется, взаимоблокировка не может возникнуть, поскольку выполнение передается только от координатора к рабочим потокам и обратно. Рабочие потоки никогда не обращаются к одному и тому же участку памяти, поэтому можно не беспокоиться, что они повредят совместно обрабатываемые данные.

Рабочие потоки могут работать бесконечно. Мы можем не беспокоиться насчет этого бесконечного цикла, потому что он возобновляется только после возврата из `Atomics.wait()`, для чего необходимо, чтобы какой-то другой поток вызвал `Atomics.notify()` для того же элемента массива.

Обернем код функцией `runCoord()`, которая запускается в ответ на сообщение от главного потока браузера после сообщения о завершении инициализации. Добавьте в файл код из примера 5.17.

Пример 5.17 ❖ *ch5-game-of-life/thread-gol.js (часть 6)*

```
function runCoord() {
  for (let i = 0; i < THREADS; i++) {
    Atomics.store(sync, i, 1);
    Atomics.notify(sync, i);
  }
  for (let i = 0; i < THREADS; i++) {
```

```
      Atomics.wait(sync, i, 1);
    }
    const oldCells = cells;
    cells = nextCells;
    nextCells = oldCells;
    for (let x = 0; x < SIZE; x++) {
      for (let y = 0; y < SIZE; y++) {
        sharedImageBuf[SIZE * x + y] = cells[SIZE * x + y] ? BLACK : WHITE;
      }
    }
    self.postMessage({});
  }
}
```

Первым делом поток-координатор уведомляет каждый рабочий поток `i` посредством соответствующего ему элемента массива `sync`, т. е. пробуждает их для выполнения итерации. Закончив работу, каждый поток уведомляет координатора посредством того же элемента, поэтому мы ждем у этих элементов. Именно потому, что каждый из этих вызовов `Atomics.wait()` блокирует выполнение потока, нам и нужен поток-координатор, так как блокировать главный поток запрещено.

Затем мы обмениваем ссылки на `cells` и `nextCells`. Рабочие потоки уже проделали это для себя внутри метода `iterate()`, и мы поступаем по аналогии. Теперь мы готовы обойти массив `cells` и преобразовать значения клеток в пиксели изображения. И напоследок отправляем главному потоку браузера сообщение о том, что данные можно отображать в пользовательском интерфейсе. Координатору нечего делать до получения сообщения, а получив его, он снова вызывает `runCoord`. Этот метод завершает концептуальный цикл, начатый в примере 5.14.

Все сделано! Напомним, что для использования `SharedArrayBuffer` необходимо, чтобы сервер установил определенные заголовки. Поэтому, находясь в каталоге *ch5-game-of-life*, выполните команду:

```
$ npx MultithreadedJSBook/serve .
```

Затем допишите */thread-gol.html* в конец напечатанного URL-адреса и наслаждайтесь многопоточной игрой «Жизнь». Поскольку мы не изменяли код UI, все должно выглядеть так же, как в однопоточной версии на рис. 5.1. Единственно, что отличается, – производительность. Переходы между итерациями должны быть более быстрыми и плавными. Да-да, вам не показалось! Мы перенесли вычисление состояния клеток и рисование пикселей в отдельные потоки, поэтому у главного потока остается больше времени на плавную анимацию, а итерации завершаются быстрее, т. к. для параллельной работы задействовано больше процессорных ядер.

Но самое главное – мы избежали накладных расходов на передачу сообщений между потоками, поскольку использовали метод `Atomics.notify()`, чтобы дать потокам знать, что они могут продолжить работу после приостановки себя методом `Atomics.wait()`.

Атомарные операции и события

В сердце JavaScript лежит цикл событий, который позволяет языку создавать новые стеки вызовов и обрабатывать события. Он существовал с самого начала и JavaScript-программисты всегда полагались на него. Это справедливо и в браузере, где вы, возможно, использовали jQuery для прослушивания события `click` в DOM, и на сервере, где, вы, возможно, пользовались сервером Fastify, который ждет входящего запроса на установление TCP-соединения.

А теперь встречайте новенького: метод `Atomics.wait()` и разделяемую память. Этот паттерн позволяет приостанавливать выполнение приложения, а значит, и всего цикла событий. Поэтому нельзя ожидать, что не возникнет никаких проблем после включения в приложение элементов многопоточности. Чтобы приложение и дальше работало правильно, нужно соблюдать некоторые ограничения.

Одно из таких ограничений уже встречалось нам в браузере: главный поток приложения не должен вызывать `Atomics.wait()`. Да, в простом скрипте Node.js это делать можно, но в крупном приложении лучше воздержаться. Например, предположим, что главный поток Node.js обрабатывает входящие HTTP-запросы или сигналы операционной системы. Что будет, если цикл событий остановится, потому что поток перешел в состояние ожидания? В примере 5.18 приведен пример такой программы.

Пример 5.18 ❖ *ch5-node-block/main.js*

```
#!/usr/bin/env node

const http = require('http');

const view = new Int32Array(new SharedArrayBuffer(4));
setInterval(() => Atomics.wait(view, 0, 0, 1900), 2000); ❶

const server = http.createServer((req, res) => {
  res.end('Hello World');
});

server.listen(1337, (err, addr) => {
  if (err) throw err;
  console.log('http://localhost:1337/');
});
```

❶ **Раз в две секунды приложение приостанавливается на 1.9 с.**

Если хотите, создайте каталог для этого файла и запустите сервер, выполнив следующую команду:

```
$ node main.js
```

Пока программа работает, несколько раз выполните в окне терминала показанную ниже команду со случайными интервалами между запусками:

```
$ time curl http://localhost:1337
```

Это приложение первым делом создает HTTP-сервер и начинает ждать запросов. Затем каждые две секунды оно вызывает метод `Atomics.wait()`, настроенный так, что приложение приостанавливается на 1.9 с для имитации долгих пауз. Запускаемой вами команде `curl` предшествует команда `time`, которая показывает, сколько времени работала следующая за ней команда. Результат будет случайным числом в диапазоне от 0 до 1.9 с – огромное время для обработки веб-запроса. Даже если уменьшать время задержки, приближая его к нулю, все равно будет присутствовать небольшое «спотыкание», распространяющееся на все входящие запросы. Если бы браузеры разрешали использовать `Atomics.wait()` в главном потоке, то такое спотыкание вы наблюдали бы сегодня на посещаемых сайтах.

Остается еще один вопрос: какие ограничения должны действовать для дополнительных потоков, запускаемых приложением, учитывая, что в каждом из них имеется свой цикл событий?

Мы рекомендуем заранее определить главную цель каждого потока. Поток может быть либо расчетным, в котором много вызовов `Atomics`, либо событийно-обрабатывающим, в котором количество вызовов `Atomics` минимально. При таком подходе вы можете получить рабочий поток в полном смысле слова, т. е. он постоянно выполняет сложные вычисления и записывает результаты в разделяемый массив. Должен также существовать главный поток, взаимодействие с которым осуществляется преимущественно с помощью передачи сообщений; он занимается работой, связанной с циклом событий. И, быть может, имеет смысл завести простые потоки-посредники, которые вызывают `Atomics.wait()` в ожидании завершения работы другими потоками, а затем `postMessage()` для отправки полученных данных главному потоку для их обработки на гораздо более высоком уровне.

Подведем итоги всему сказанному в этом разделе:

- не используйте `Atomics.wait()` в главном потоке;
- при проектировании программы выделяйте расчетные потоки с большим числом вызовов `Atomics` и событийно-обрабатывающие потоки;
- подумайте о создании потоков-посредников, которые ждут других потоков и отправляют сообщения по мере необходимости.

Это очень общие рекомендации по проектированию приложений. Но иногда имеет смысл применить более конкретные паттерны для достижения желаемой цели. В главе 6 как раз и описаны некоторые из таких паттернов.

Глава 6

Паттерны многопоточного программирования

JavaScript API, касающиеся многопоточности, сами по себе довольно примитивны. В главе 4 мы видели, что класс `SharedArrayBuffer` предназначен для хранения двоичного представления данных. А в главе, посвященной объекту `Atomics`, было показано, что он предоставляет примитивные методы для координации потоков или атомарной модификации горстки байтов.

Глядя на такой абстрактный низкоуровневый API, трудно составить полную картину и понять, для чего такие API можно реально использовать. В этой главе мы объясним, как из них можно сделать нечто, действительно полезное приложению.

Мы включили в эту главу популярные паттерны проектирования для реализации многопоточной функциональности. Эти паттерны корнями уходят в те времена, когда и JavaScript еще не было. И хотя рабочие примеры можно найти в разных формах, в том числе в учебниках по C++, перевод их на JavaScript не всегда очевиден.

Изучив эти паттерны, вы станете намного лучше понимать, какую выгоду ваши приложения могут получить от многопоточности.

Пул потоков

Пул потоков – очень популярный паттерн, который в той или иной форме используется в большинстве многопоточных приложений. По существу, это набор однородных рабочих потоков, каждый из которых может выполнять счетные задачи, необходимые приложению. Это несколько отличается от ранее рассмотренных подходов, когда использовался один рабочий поток или их конечное множество. Например, библиотека `libuv`, на которую опирается Node.js, предоставляет пул, по умолчанию состоящий из четырех потоков, для выполнения низкоуровневых операций ввода–вывода.

Этот паттерн может показаться похожим на распределенные системы, с которыми вы, возможно, работали в прошлом. Например, на платформах

оркестрации контейнеров обычно имеется набор машин, каждая из которых может выполнять контейнерные приложения. В такой системе у каждой машины могут быть отличные от других характеристики, например разные операционные системы, разный объем памяти или количество и мощность процессоров. Это позволяет оркестратору назначать каждой машине пункты на основе ресурсов и приложений, а затем потреблять эти пункты. С другой стороны, пул потоков гораздо проще, потому что все потоки могут выполнять одну и ту же работу и их возможности одинаковы, так как все работают на одной и той же машине.

Первый вопрос, возникающий при создании пула, – сколько в нем должно быть потоков?

Размер пула

Существует два типа программ: работающие в фоновом режиме, как системные процессы-демоны, которые в идеале не должны потреблять много ресурсов, и работающие в приоритетном режиме. Последние гораздо лучше знакомы пользователям, это, например, персональные приложения или веб-сервер. Браузерные приложения обычно запускаются в приоритетном режиме, тогда как приложения для Node.js могут работать и в фоне – хотя Node.js чаще всего используется для построения серверов, которые зачастую являются единственным процессом в контейнере. В любом случае чаще всего JavaScript-приложения пишутся так, чтобы в какой-то момент времени находиться в центре внимания, поэтому все вычисления, необходимые для достижения цели программы, должны выполняться как можно быстрее.

Чтобы выполнить группу команд максимально быстро, имеет смысл разбить ее на части и выполнять их параллельно. Чтобы процессор использовался максимально эффективно, приложение должно задействовать все имеющиеся ядра и загружать их по возможности равномерно. Поэтому количество доступных ядер должно быть определяющим фактором при выборе числа потоков – или исполнителей – в приложении.

Обычно размер пула потоков не изменяется динамически во время работы приложения, поскольку для выбора того или иного числа потоков имеется причина, и эта причина неизменна. Именно поэтому мы работаем с пулом потоков фиксированного размера, который динамически задается на этапе инициализации приложения.

Ниже показан идиоматический подход к определению числа потоков, доступных JavaScript-приложению, работающему в браузере или Node.js:

```
// браузер
cores = navigator.hardwareConcurrency;

// Node.js
cores = require('os').cpus().length;
```

Следует помнить, что в большинстве операционных систем нет прямой связи между потоком и процессорным ядром. Например, если четырехпоточ-

ное приложение исполняется четырехъядерным процессором, то это не значит, что первое ядро всегда будет исполнять первый поток, второе – второй и т. д. На самом деле операционная система постоянно перебрасывает задачи с одного ядра на другое и иногда прерывает работающее приложение, чтобы дать возможность поработать другому приложению. В современной ОС обычно работают сотни фоновых процессов, которые периодически требуют внимания. Это означает, что одно ядро будет исполнять работу не одного, а нескольких потоков.

С каждым переключением процессора между программами – или потоками одной программы – связаны небольшие накладные расходы. Из-за этого возможна потеря производительности, если число потоков значительно больше числа ядер. Может случиться, что из-за контекстных переключений приложение даже станет работать медленнее, поэтому нужно по возможности уменьшать число потоков, требующих внимания со стороны ОС. Но если потоков слишком мало, то приложению потребуется больше времени, чтобы справиться со своей задачей, поэтому и пользователь будет недоволен, и оборудование будет недозагружено.

Следует также помнить, что если приложение создает пул с четырьмя потоками, то на самом деле число потоков в программе не меньше пяти, потому что нужно учитывать и главный поток приложения. Кроме того, есть еще фоновые потоки, например пул потоков `libuv`, поток сборки мусора, если движок JavaScript его использует, поток отрисовки обрамления (chrome) браузера и т. д. Все они также влияют на производительность приложения.

Характеристики самого приложения тоже играют роль при определении идеального размера пула потоков. Быть может, вы пишете программу для добычи криптовалюты, которая 99.9 % времени проводит в рабочих потоках и почти не выполняет ввода–вывода в главном потоке? В таком случае задавать размер пула равным числу ядер правильно. Или же ваше приложение предоставляет услуги видеостриминга и перекодирования, а значит, для него характерен большой объем вычислений и ввода–вывода? Тогда лучше, чтобы размер пула был равен числу потоков минус 2. Для нахождения идеального размера нужны тесты производительности, но в качестве отправной точки имеет смысл взять число ядер минус единица, а потом настроить размер точнее.

Определив, сколько потоков должно быть в пуле, нужно решить, как распределить между ними работу.

Стратегии диспетчеризации

Поскольку цель пула потоков – максимизировать объем работы, выполняемой параллельно, очевидно, что ни один рабочий поток не должен получать слишком много работы и не должно быть простаивающих потоков. Наивный подход – собрать подлежащие выполнению задачи, затем передать их пулу, как только число готовых задач сравняется с числом рабочих потоков, и продолжить работу, когда все задачи завершатся. Однако не гарантируется, что

всем задачам для завершения нужно одно и то же время. Быть может, одни работают очень быстро, всего несколько миллисекунд, а другим требуется несколько секунд, а то и больше. Поэтому необходимо более надежное решение.

Существует несколько стратегий диспетчеризации задач между рабочими потоками из пула. Можно провести параллель со стратегиями, применяемыми обратными прокси-серверами для раздачи задач обсуживающим серверам. Перечислим наиболее употребительные стратегии.

Циклическая
: Очередная задача отдается следующему рабочему потоку, причем за последним потоком следует первый. Так что если размер пула равен трем, то первая задача будет отдана Исполнителю 1, вторая – Исполнителю 2, третья – Исполнителю 3, следующая – снова Исполнителю 1 и т. д. Преимущество этой стратегии в том, что все потоки получают одинаковое число задач, а недостаток в том, что если частота сложных задач кратна количеству потоков (например, каждая шестая задача выполняется долго), то работа будет распределяться неравномерно. В обратном прокси-сервере HAProxy эта стратегия называется *циклической* (round robin).

Случайная
: Каждой задаче назначается случайный рабочий поток из пула. Такую стратегию проще всего реализовать, потому что не нужно хранить никакой информации о состоянии, но некоторые рабочие потоки могут получать слишком много работы, тогда как другие иногда будут простаивать.

Наименее занятый
: Запоминается количество задач, отданных каждому исполнителю, и очередная задача отдается наименее занятому исполнителю. Можно даже реализовать стратегию так, что у каждого исполнителя в каждый момент времени будет только одна задача. Если имеется более одного рабочего потока с наименьшим объемом работы, то исполнитель выбирается случайно. Это, пожалуй, самый надежный подход, особенно в ситуации, когда все задачи потребляют процессор одинаково, но его труднее всего реализовать. Если некоторые задачи требуют меньше ресурсов, например вызывают `setTimeout()`, то возможно неравномерное распределение нагрузки между исполнителями. В сервере HAProxy эта стратегия называется *leastconn*.

Другие стратегии, применяемые обратными прокси-серверами и имеющие неочевидную реализацию, тоже могут быть включены в ваши приложения. Например, в HAProxy имеется стратегия балансировки нагрузки `source`, которая принимает хеш IP-адреса клиента и с его помощью отдает запросы от этого клиента одному и тому же обслуживающему серверу. Аналог этой стратегии может быть полезен в случаях, когда рабочие потоки хранят в памяти кеш данных и маршрутизация задач одному и тому же потоку может увеличить коэффициент попадания в кеш. Но обобщить такой подход немного труднее.

В зависимости от природы приложения одни стратегии могут оказаться производительнее других. Еще раз повторим, что при измерении производительности приложения тестирование – ваш лучший друг.

Пример реализации

В этом примере мы повторно воспользуемся файлами из каталога *ch2-patterns/*, созданного в разделе «Соберем все вместе» главы 2, но для краткости код обработки ошибок убран, а вся программа сделана совместимой с Node.js. Создайте новый каталог *ch6-thread-pool/* для файлов из этого раздела.

Первым делом мы создадим файл *main.js*. Он будет играть роль точки входа в приложение. В предыдущей версии этого кода мы просто вызывали `Promise.allSettled()`, чтобы добавить задачи в пул, но это не так интересно, потому что все задачи добавляются одновременно. Новое приложение будет веб-сервером, и при поступлении каждого запроса будет создаваться новая задача. При таком подходе возможно, что предыдущие задачи уже завершились к моменту опроса пула, и это приводит к более интересному поведению, характерному для реальных приложений.

Скопируйте код из примера 6.1 в файл *main.js*.

Пример 6.1 ❖ *ch6-thread-pool/main.js*

```
#!/usr/bin/env node
const http = require('http');
const RpcWorkerPool = require('./rpc-worker.js');
const worker = new RpcWorkerPool('./worker.js',
  Number(process.env.THREADS), ❶
  process.env.STRATEGY); ❷

const server = http.createServer(async (req, res) => {
  const value = Math.floor(Math.random() * 100_000_000);
  const sum = await worker.exec('square_sum', value);
  res.end(JSON.stringify({ sum, value }));
});

server.listen(1337, (err) => {
  if (err) throw err;
  console.log('http://localhost:1337/');
});
```

❶ **Переменная среды `THREADS` управляет размером пула.**
❷ **Переменная среды `STRATEGY` задает стратегию диспетчеризации.**

В этом приложении используется две переменные среды, чтобы было проще экспериментировать. Первая, `THREADS`, задает число потоков в пуле. Вторая, `STRATEGY`, определяет стратегию диспетчеризации. В остальных отношениях сервер не представляет собой ничего особенного, поскольку он основан на встроенном модуле `http`. Сервер прослушивает порт 1337, и любой запрос, вне зависимости от пути, активирует обработчик. В ответ на любой запрос

вызывается команда `square_sum`, определенная в коде исполнителей, которой передается случайное число от 0 до 100 млн.

Далее создайте файл *worker.js* и скопируйте в него код из примера 6.2.

Пример 6.2 ❖ *ch6-thread-pool/worker.js*

```
const { parentPort } = require('worker_threads');

function asyncOnMessageWrap(fn) {
  return async function(msg) {
    parentPort.postMessage(await fn(msg));
  }
}

const commands = {
  async square_sum(max) {
    await new Promise((res) => setTimeout(res, 100));
    let sum = 0; for (let i = 0; i < max; i++) sum += Math.sqrt(i);
    return sum;
  }
};

parentPort.on('message', asyncOnMessageWrap(async ({method, params, id}) =>
  ({
    result: await commands[method](...params), id
  })
));
```

Этот файл тоже не особенно интересен, поскольку является всего лишь упрощенной версией ранее созданного файла *worker.js*. Вся обработка ошибок удалена, чтобы сделать код короче (если хотите, можете вернуть ее). Кроме того, код сделан совместимым с Node.js API. Мы оставили в этом примере только одну команду, `square_sum`.

Следующий файл – *rpc-worker.js*. Он довольно большой, поэтому мы разобьем его на части. Сначала добавьте код из примера 6.3.

Пример 6.3 ❖ *ch6-thread-pool/rpc-worker.js (часть 1)*

```
const { Worker } = require('worker_threads');
const CORES = require('os').cpus().length;
const STRATEGIES = new Set([ 'roundrobin', 'random', 'leastbusy' ]);

module.exports = class RpcWorkerPool {
  constructor(path, size = 0, strategy = 'roundrobin') {
    if (size === 0) this.size = CORES; ❶
    else if (size < 0) this.size = Math.max(CORES + size, 1);
    else this.size = size;

    if (!STRATEGIES.has(strategy)) throw new TypeError('invalid strategy');
    this.strategy = strategy; ❷
    this.rr_index = -1;

    this.next_command_id = 0;
```

```
    this.workers = []; ❸
    for (let i = 0; i < this.size; i++) {
      const worker = new Worker(path);
      this.workers.push({ worker, in_flight_commands: new Map() }); ❹
      worker.on('message', (msg) => {
        this.onMessageHandler(msg, i);
      });
    }
  }
```

❶ Размер пула потоков настраивается.
❷ Проверяется и запоминается стратегия.
❸ Создается не один исполнитель, а целый массив.
❹ `in_flight_commands` теперь хранится для каждого исполнителя.

В начале этого файла загружается модуль `worker_threads` для создания потоков и модуль `os` для получения числа доступных процессорных ядер. Затем определяется и экспортируется класс `RpcWorkerPool`. Далее следует конструктор класса. Он принимает три аргумента: путь к файлу исполнителя, размер пула и стратегию диспетчеризации.

Размер пула допускает настройку и задается вызывающей стороной. Если параметр положителен, то интерпретируется как размер пула. Подразумеваемое по умолчанию значение 0 означает, что размер пула равен числу процессорных ядер. Наконец, если параметр отрицателен, то его абсолютная величина вычитается из количества ядер и результат становится размером пула. Так, на 8-ядерной машине параметр –2 означает, что размер пула будет равен 6.

Стратегия может быть равна `roundrobin` (по умолчанию), `random` или `leastbusy`. Переданное конструктору значение проверяется. Свойство `rr_index` используется как индекс в циклической стратегии, оно увеличивается на 1 для каждой задачи и возвращается к 0 по достижении максимального номера исполнителя.

Свойство `next_command_id` по-прежнему является глобальным, т. е. вне зависимости от того, обрабатываются команды одним рабочим потоком или разными, первая команда получит идентификатор `1`, а следующая `2`.

Наконец, свойство класса `workers` – это массив исполнителей, оно заменило прежнее свойство `worker`. Код его обработки по большому счету не изменился, но список `in_flight_commands` теперь является локальным для исполнителя, а методу `onMessageHandler()` передается дополнительный аргумент – ИД исполнителя. Это нужно, потому что впоследствии, когда главному потоку будет посылаться сообщение, понадобится найти конкретный поток.

Добавьте в файл код из примера 6.4.

Пример 6.4 ❖ *ch6-thread-pool/rpc-worker.js (часть 2)*

```
onMessageHandler(msg, worker_id) {
  const worker = this.workers[worker_id];
  const { result, error, id } = msg;
  const { resolve, reject } = worker.in_flight_commands.get(id);
  worker.in_flight_commands.delete(id);
```

```
    if (error) reject(error);
    else resolve(result);
}
```

В этой части файла определяется метод `onMessageHandler()`, который вызывается, когда исполнитель отправляет сообщение главному потоку. Он почти такой же, как раньше, только принимает дополнительный аргумент, `worker_id`, который нужен, чтобы найти исполнителя, отправившего сообщение. Найденный исполнитель обрабатывает разрешение или отмену обещания и удаляет запись из списка ожидающих команд.

Добавьте в файл код из примера 6.5.

Пример 6.5 ❖ *ch6-thread-pool/rpc-worker.js (часть 3)*

```
  exec(method, ...args) {
    const id = ++this.next_command_id;
    let resolve, reject;
    const promise = new Promise((res, rej) =>
                                { resolve = res; reject = rej; });
    const worker = this.getWorker(); ❶
    worker.in_flight_commands.set(id, { resolve, reject });
worker.worker.postMessage({ method, params: args, id });
    return promise;
  }
```

❶ **Ищется подходящий исполнитель.**

В этой части файла определяется метод `exec()`, который приложение вызывает, когда хочет выполнить команду в одном из исполнителей. Он также почти не изменился, но теперь вызывается метод `getWorker()`, который ищет подходящего для обработки следующей команды исполнителя, а не ограничивается единственным исполнителем, как раньше. Этот метод определен в следующем разделе.

Добавьте в файл последнюю часть – код из примера 6.6.

Пример 6.6 ❖ *ch6-thread-pool/rpc-worker.js (часть 4)*

```
  getWorker() {
    let id;
    if (this.strategy === 'random') {
      id = Math.floor(Math.random() * this.size);
    } else if (this.strategy === 'roundrobin') {
      this.rr_index++;
      if (this.rr_index >= this.size) this.rr_index = 0;
      id = this.rr_index;
    } else if (this.strategy === 'leastbusy') {
      let min = Infinity;
      for (let i = 0; i < this.size; i++) {
        let worker = this.workers[i];
        if (worker.in_flight_commands.size < min) {
          min = worker.in_flight_commands.size;
          id = i;
```

```
        }
      }
    }
    console.log('Выбран исполнитель:', id);
    return this.workers[id];
  }
};
```

В этой части определяется последний метод, `getWorker()`. Он учитывает стратегию выбора следующего исполнителя, заданную при создании экземпляра класса. Функция в основном состоит из большого предложения `if`, ветви которого соответствуют стратегиям.

Для первой стратегии, `random`, дополнительная информация о состоянии не нужна, что делает ее самой простой. Нужно только случайным образом выбрать один из потоков в пуле.

Вторая стратегия, `roundrobin`, чуть сложнее. В ней используется свойство класса `rr_index`: оно увеличивается на 1, и возвращается исполнитель с новым индексом. Если индекс превысил число исполнителей, то он сбрасывается в 0.

Последняя стратегия, `leastbusy`, самая сложная. Мы перебираем всех исполнителей и для каждого смотрим, сколько команд ожидает выполнения – их количество равно размеру отображения `in_flight_commands`. Выбирается исполнитель с наименьшим числом команд. Заметим, что эта реализация выбирает первого исполнителя с наименьшим числом ожидающих команд; в частности, при первом вызове она всегда выбирает исполнителя 0. Более справедливая реализация могла бы найти всех исполнителей с наименьшим числом команд (одинаковым) и случайным образом выбрать один из них. Идентификатор выбранного исполнителя записывается в журнал, чтобы мы знали, что происходит.

Итак, приложение написано, и можно его запустить. Откройте два окна терминала и перейдите в каталог *ch6-thread-pool/* в первом из них. А этом окне выполните следующую команду:

```
$ THREADS=3 STRATEGY=leastbusy node main.js
```

Она запускает процесс с тремя исполнителями в пуле потоков и стратегией `leastbusy`.

Во втором окне терминала выполните такую команду:

```
$ npx autocannon -c 5 -a 20 http://localhost:1337
```

Это команда `autocannon` – npm-пакет для тестирования производительности. Но в данном случае мы не тестируем производительность, а просто выполняем серию запросов. Команда открывает сразу пять соединений и отправляет 20 запросов. Так мы моделируем параллельное выполнение 5 запросов, а по мере их завершения отправляем последующие 15. Это близко к нагрузке на реальный веб-сервер.

Поскольку приложение использует стратегию `leastbusy`, а код написан так, что выбирается первый поток с наименьшим числом команд, первые пять

запросов должны обрабатываться, как в циклической стратегии. При размере пула 3 сразу после запуска приложения у каждого исполнителя 0 задач. Поэтому будет выбран Исполнитель 0. Для второго запроса у первого исполнителя уже есть задача, а у второго и третьего нет, поэтому выбирается второй. И затем третий. При получении четвертого запроса все три исполнителя имеют по одной задаче, поэтому снова выбирается первый.

После распределения первых пяти задач остальные назначения исполнителей по существу случайны, так как для завершения каждой команды требуется случайное время.

Остановите сервер нажатием **Ctrl**+**C** и снова запустите его со стратегией `roundrobin`:

```
$ THREADS=3 STRATEGY=roundrobin node main.js
```

На втором терминале запустите ту же команду `autocannon`, что и выше. Теперь задачи должны всегда исполняться в порядке 0, 1, 2, 0 и т. д.

Наконец, снова остановите сервер, нажав **Ctrl**+**C**, и запустите со стратегией `random`:

```
$ THREADS=3 STRATEGY=random node main.js
```

В последний раз запустите команду `autocannon` и посмотрите на результаты. Теперь они должны быть совершенно случайными. Если один и тот же исполнитель будет выбран несколько раз подряд, то, скорее всего, он перегружен.

В табл. 6.1 приведены выборочные результаты этого эксперимента. Каждый столбец соответствует запросу, а число – идентификатор исполнителя, выбранного для обслуживания запроса.

Таблица 6.1. Выборочные результаты работы пула потоков с разными стратегиями

Стратегия	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10
Наименее занятый	0	1	2	0	1	0	1	2	1	0
Циклическая	0	1	2	0	1	2	0	1	2	0
Случайная	2	0	1	1	0	0	0	1	1	0

В данном случае при выборе случайной стратегии Исполнитель 2 выбирался всего один раз.

Мьютекс: простая блокировка

Взаимно исключающая блокировка, или *мьютекс*, – это механизм управления доступом к разделяемым данным. Он гарантирует, что в каждый момент времени работать с ресурсом может не более одной задачи. Под задачей здесь понимается конкурентная задача любого вида, но чаще всего мьютексы используются, чтобы избежать состояния гонки нескольких потоков. Задача *захватывает* блокировку, чтобы выполнить код, обращающийся к разделяемым данным, а после этого *освобождает* блокировку. Участок кода между

захватом и освобождением называется *критической секцией*. Если одна задача пытается захватить блокировку, уже захваченную другой, то она будет заблокирована, пока другая задача не освободит блокировку.

Сразу может быть непонятно, зачем нужно использовать мьютекс, если у нас уже есть объект `Atomics` с его атомарными операциями. Безусловно, использовать атомарные операции для чтения и изменения данных эффективнее, потому что другие операции блокируются на меньшее время, разве не так? Но бывает так, что программе необходимо, чтобы данные никем не модифицировались на протяжении нескольких операций. Иначе говоря, единицы атомарности, предлагаемые атомарными операциями, слишком мелкие. Например, допустим, что нужно прочитать два целых числа из разных мест разделяемой памяти, сложить их и записать результат в третье место. Если между двумя операциями чтения значения будут изменены, то сумма отразит противоречивые представления о слагаемых, что впоследствии может привести к логической ошибке в программе.

Рассмотрим пример программы, которая инициализирует буфер целыми числами, после чего выполняет простые операции с ними в разных потоках. Каждый поток будет выбирать значение в позиции с уникальным для потока индексом и значение в позиции с разделяемым индексом, перемножать их и записывать в позицию с разделяемым индексом. Затем мы прочитаем число в позиции с разделяемым индексом и проверим, равно ли оно произведению ранее прочитанных сомножителей. В промежутке между двумя чтениями будет выполняться цикл активного ожидания, моделирующий работу, которая занимает некоторое время.

Создайте каталог *ch6-mutex*, а в нем файл *thread_product.js*, в который скопируйте код из примера 6.7.

Пример 6.7 ❖ *ch6-mutex/thread-product.js*

```
const {
  Worker, isMainThread, workerData
  } = require('worker_threads');
  const assert = require('assert');

  if (isMainThread) {
    const shared = new SharedArrayBuffer(4 * 4); ❶
    const sharedInts = new Int32Array(shared);
    sharedInts.set([2, 3, 5, 7]);
    for (let i = 0; i < 3; i++) {
      new Worker(__filename, { workerData: { i, shared } });
    }
  } else {
    const { i, shared } = workerData;
    const sharedInts = new Int32Array(shared);
    const a = Atomics.load(sharedInts, i);
    for (let j = 0; j < 1_000_000; j++) {}
      const b = Atomics.load(sharedInts, 3);
    Atomics.store(sharedInts, 3, a * b);
    assert.strictEqual(Atomics.load(sharedInts, 3), a * b); ❷
}
```

❶ **Мы будем использовать три потока и массив `Int32Array` для хранения данных, надо сделать его достаточно большим для размещения трех 32-разрядных целых чисел плюс четвертое, которое будет использоваться в качестве разделяемого результата.**
❷ **Здесь проверяется результат. В реальном приложении здесь, вероятно, не было бы никакой проверки, но мы хотим смоделировать зависимость последующих действий программы от результата.**

Запустите программу командой

```
$ node thread-product.js
```

Возможно, в первом прогоне или даже в нескольких прогонах все будет работать правильно, но не останавливайтесь на этом и продолжайте. А быть может, утверждение окажется не выполненным сразу же. Так или иначе, в пределах первых 20 попыток вы, скорее всего, увидите, что утверждение не выполнено. Хотя мы и используем атомарные операции, но между ними может вклиниться изменение. Это классический пример состояния гонки. Все потоки читают и пишут конкурентно (хотя и не параллельно, так как сами операции атомарны), поэтому результаты недетерминированы.

Для решения проблемы мы реализуем класс `Mutex`, пользуясь примитивами, предоставляемыми объектом `Atomics`. Будем вызывать `Atomics.wait()`, чтобы дождаться захвата блокировки, и `Atomics.notify()`, чтобы уведомить другие потоки об освобождении блокировки. И будем использовать `Atomics.compareExchange()`, чтобы обменять состояние блокирован / не блокирован, и понять, нужно ли дожидаться блокировки. Создайте в том же каталоге файл *mutex.js* и скопируйте в него код из примера 6.8

Пример 6.8 ❖ *ch6-mutex/mutex.js (часть 1)*

```
const UNLOCKED = 0;
const LOCKED = 1;

const {
  compareExchange, wait, notify
} = Atomics;

class Mutex {
  constructor(shared, index) {
    this.shared = shared;
    this.index = index;
}
```

Здесь мы определили состояния `LOCKED` и `UNLOCKED` как 1 и 0 соответственно. Можно было бы выбрать любые значения, допускающие хранение в массиве типа `TypedArray`, переданном конструктору класса `Mutex`, но 1 и 0 проще воспринимать как булевы. Конструктор принимает два значения и записывает их в свойства: массив `TypedArray`, с которым мы будем работать, и индекс позиции в этом массиве, которая будет играть роль состояния блокировки. Теперь все готово к реализации метода `acquire()`, в котором используется деструктурированный `Atomics`. Добавьте код этого метода из примера 6.9.

Пример 6.9 ❖ *ch6-mutex/mutex.js (часть 2)*

```
acquire() {
  if (compareExchange(this.shared, this.index, UNLOCKED, LOCKED) === UNLOCKED) {
    return;
  }
  wait(this.shared, this.index, LOCKED);
  this.acquire();
}
```

Для захвата блокировки мы пытаемся обменять состояние `UNLOCKED` с `LOCKED` в позиции массива-мьютекса, пользуясь методом `Atomics.compareExchange()`. Если обмен состоялся успешно, то больше ничего делать не надо – мы захватили мьютекс и можем вернуть управление. В противном случае нужно дождаться разблокировки – в данном случае уведомления о том, что значение элемента поменялось с `LOCKED` на какое-то другое. Затем мы делаем еще одну попытку захватить мьютекс. Это делается рекурсивно, чтобы подчеркнуть идею «повторной попытки», но можно было бы сделать и в цикле. Во второй раз все должно получиться, потому что мы как раз и ждали освобождения блокировки, но между `wait()` и `compareExchange()` значение могло измениться, поэтому на всякий случай нужно проверить еще раз. В реальной программе можно было бы добавить тайм-аут в `wait()` и ограничить число попыток.

Во многих производственных реализациях мьютексов, помимо состояний «блокирован» и «разблокирован», есть еще состояние «блокирован и есть конфликт». *Конфликт* возникает, когда один поток пытается захватить блокировку, удерживаемую другим потоком. Отслеживая это состояние, мьютекс может избежать лишних обращений к `notify()` и улучшить производительность.

Семафоры

Элемент разделяемого массива, который мы используем для представления состояния – блокирован или разблокирован, – это тривиальный пример *семафора*. Семафорами называются переменные, служащие для передачи информации о состоянии между потоками. Они показывают, сколько потоков используют ресурс. В случае мьютекса таких потоков может быть 0 или 1, но в других ситуациях возможно большее число.

Теперь рассмотрим освобождение блокировки. Добавьте метод `release()` из примера 6.10.

Пример 6.10 ❖ *ch6-mutex/mutex.js (часть 3)*

```
release() {
if (compareExchange(this.shared, this.index, LOCKED, UNLOCKED) !== LOCKED) {
    throw new Error('was not acquired');
  }
  notify(this.shared, this.index, 1);
}
```

Здесь метод `Atomics.compareExchange()` снова используется для обмена состояний, как и в случае захвата блокировки. Но на этот раз мы хотим удостовериться, что исходное состояние действительно `LOCKED`, потому что не следует освобождать блокировку, которую мы не захватывали. Если все хорошо, то осталось только вызвать `notify()` и тем самым позволить ожидающему потоку (если таковой имеется) захватить блокировку. Мы задаем счетчик в `notify()` равным 1, потому что нет нужды будить более одного спящего потока, так как удерживать блокировку все равно может только один поток.

Того, что мы сделали, уже достаточно для создания работоспособного мьютекса. Однако довольно легко захватить мьютекс и забыть освободить его или еще как-то организовать незапланированную критическую секцию. Во многих случаях критическая секции четко определена и известна заранее. А раз так, то имеет смысл добавить в класс `Mutex` вспомогательный метод для обертывания критических секций. С этой целью добавьте метод `exec()` из примера 6.11.

Пример 6.11 ❖ *ch6-mutex/mutex.js (часть 4)*

```
  exec(fn) {
    this.acquire();
    try {
      return fn();
    } finally {
      this.release();
    }
  }
}

module.exports = Mutex;
```

Здесь мы всего лишь вызываем переданную функцию и возвращаем ее значение, но до и после вызова ставим соответственно `acquire()` и `release()`. В результате получается, что переданная функция содержит весь код критической секции. Отметим, что функция вызывается в блоке `try`, а вызов `release()` находится в соответствующем блоке `finally`. Это сделано на тот случай, если функция возбудит исключение, – ведь даже и тогда блокировка должна быть освобождена. На этом разработка класса `Mutex` завершена, и мы можем продемонстрировать его использование на примере.

Скопируйте файл *thread-product.js* в *thread-productmutex.js*. В этом файле загрузите файл *mutex.js* в предложении `require` и присвойте результат `const`-переменной `Mutex`. Добавьте еще 4 байта в `SharedArrayBuffer` (например, `new SharedArrayBuffer(4 * 5)`), которые понадобятся нашей блокировке, и замените все содержимое ветви `else` кодом из примера 6.12.

Пример 6.12 ❖ *ch6-mutex/thread-product-mutex.js*

```
const { i, shared } = workerData;
const sharedInts = new Int32Array(shared);
const mutex = new Mutex(sharedInts, 4); ❶
mutex.exec(() => {
```

```
  const a = sharedInts[i]; ❷
  for (let j = 0; j < 1_000_000; j++) {}
  const b = sharedInts[3];
  sharedInts[3] = a * b;
  assert.strictEqual(sharedInts[3], a * b);
});
```

❶ **До этой строки код такой же, как в программе без мьютекса. А здесь мы инициализируем мьютекс, используя пятый элемента массива `Int32Array` в качестве данных блокировки.**

❷ **Попав в функцию, переданную `exec()`, мы оказываемся в критической секции, защищенной блокировкой. Это значит, что для чтения или изменения массива атомарные операции не нужны. Мы можем работать так же, как с любым другим типизированным массивом.**

Мьютекс не только дает возможность обращаться к массиву как обычно, но и гарантирует, что никакой другой поток не сможет модифицировать данные, пока мы с ними работаем. Поэтому утверждение `assert` всегда выполняется. Попробуйте! Запустите пример показанной ниже командой. Можете повторить ее десять, сто, тысячу раз – и никогда не увидите сообщения об ошибке утверждения, как в версии, где использовались только атомарные операции:

```
$ node thread-product-mutex.js
```

Мьютексы – это прямолинейное средство для блокировки доступа к ресурсу. Они позволяют выполнять код в критической секции, не опасаясь вмешательства со стороны других потоков. Это пример того, как, комбинируя атомарные операции, можно создавать новые элементы для многопоточного программирования. В следующем разделе мы на практике продемонстрируем, как использовать новый элемент.

Потоковая обработка данных с помощью кольцевых буферов

Во многих приложениях необходимо обрабатывать потоки данных. Например, HTTP-запросы и ответы обычно представлены в API последовательностями байтов, подаваемых программе по мере получения из сети. В сетевых приложениях размер блоков данных ограничен размером пакета. В приложениях файловых систем на размер блоков данных накладывают ограничения размеры буферов в ядре. Даже если выводить данные в сеть или в файл, вообще не думая о потоках, ядро разобьет данные на части перед отправкой получателю.

Потоковая обработка также встречается в пользовательских приложениях как способ передачи больших объемов данных между единицами организации вычислений, например процессами или потоками. Но даже если отдельных вычислителей нет, все равно иногда возникает желание или необходимость сохранить данные в каком-то буфере перед обработкой. Тут-то и приходят на помощь *кольцевые*, или *циклические*, *буферы*.

Кольцевой буфер – это реализация FIFO-очереди (первым пришел, первым ушел). Он состоит из массива данных в памяти и двух индексов в нем. Важ-

но, что в целях эффективности данные после добавления в очередь никогда не перемещаются. Вместо этого при добавлении и удалении из очереди мы только изменяем индексы. Работа с массивом организована так, будто один его конец соединен с другим, образуя кольцо. Это означает, что по достижении конца массива индекс возвращается в его начало.

В реальном мире аналогом может служить колесо заказов, часто встречающееся в американских закусочных. Оно размещено между частью ресторана, обращенной к посетителям, и кухней. Официанты записывают заказы на листке бумаги и кладут их на колесо. Повара выполняют заказы в том же порядке, так чтобы посетителям не приходилось ждать еды слишком долго. Это ограниченная[1] FIFO-очередь – как и наши кольцевые буферы!

Для реализации кольцевого буфера нам понадобятся два индекса, `head` (голова) и `tail` (хвост). Индекс `head` указывает на следующую позицию очереди, куда будут записаны данные, а индекс `tail` – на следующую позицию, из которой данные будут прочитаны. При записи или чтении данных мы соответственно увеличиваем индекс `head` или `tail` на количество элементов записанных или прочитанных данных по модулю размера буфера.

На рис. 6.1 показано, как работает 16-байтовый кольцевой буфер. На первом рисунке буфер содержит 4 байта, начиная с байта 0 (на него указывает `Tail`) и заканчивая байтом 3 (`Head` указывает на следующий за ним байт 4). После добавления в буфер еще четырех байтов указатель `Head` сдвигается вперед до байта 8, как показано на втором рисунке. И на последнем рисунке показано, что после чтения первых четырех байтов указатель `Tail` сдвинулся к байту 4.

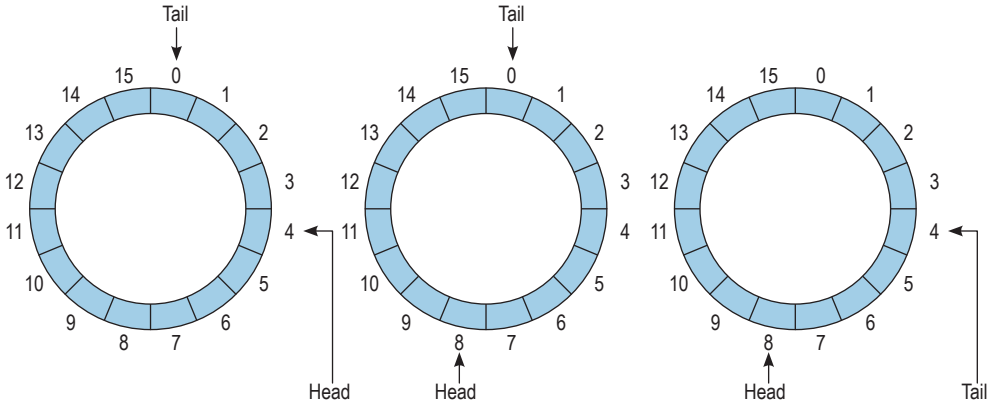


Рис. 6.1 ❖ Запись данных сдвигает вперед `Head`, а чтение сдвигает вперед `Tail`

[1] На практике заказов может быть гораздо больше, чем секторов на колесе заказов. Рестораны часто решают эту проблему, кладя в один сектор несколько заказов и соблюдая заранее оговоренный порядок исполнения заказов в одном секторе. Наши кольцевые буферы не позволяют поместить более одного элемента данных в позицию массива, поэтому такой трюк мы не применяем. Вместо этого в системе может быть реализован способ указать, что очередь заполнена и в данный момент помещать в нее дополнительные данные нельзя. Как вы скоро увидите, именно так мы и поступили.

Давайте реализуем кольцевой буфер. На первом этапе забудем о потоках и упростим себе жизнь – будем хранить `head`, `tail`, а также текущую длину очереди `length` в `TypedArray`. Можно было бы попробовать использовать в качестве длины разность между `head` и `tail`, но тогда возникнет неоднозначность, потому что в случае, когда `head` и `tail` одинаковы, непонятно, пуста очередь или заполнена. Поэтому заведем отдельное свойство `length`. Сначала напишем конструктор и акцессоры; создайте файл *ch6-ringbuffer/ring-buffer.js* и скопируйте в него код из примера 6.13.

Пример 6.13 ❖ *ch6-ring-buffer/ring-buffer.js (часть 1)*

```
class RingBuffer {
  constructor(meta/*: Uint32Array[3]*/, buffer /*: Uint8Array */) {
    this.meta = meta;
    this.buffer = buffer;
  }

  get head() {
    return this.meta[0];
  }

  set head(n) {
    this.meta[0] = n;
  }

  get tail() {
    return this.meta[1];
  }

  set tail(n) {
    this.meta[1] = n;
  }

  get length() {
    return this.meta[2];
  }

  set length(n) {
    this.meta[2] = n;
  }
```

Конструктор принимает массив `Uint32Array` с тремя элементами, который мы назвали `meta`. В нем будут храниться `head`, `tail` и `length`. Для удобства мы также добавили акцессоры для чтения и изменения этих свойств, которые просто обращаются к соответствующим элементам массива. Конструктор также принимает массив типа `Uint8Array`, в котором будет храниться сам кольцевой буфер. Далее добавим метод `write()`. Скопируйте в файл код из примера 6.14.

Пример 6.14 ❖ *ch6-ring-buffer/ring-buffer.js (часть 2)*

```
  write(data /*: Uint8Array */) { ❶
    let bytesWritten = data.length;
```

```
    if (bytesWritten > this.buffer.length - this.length) { ❷
      bytesWritten = this.buffer.length - this.length;
      data = data.subarray(0, bytesWritten);
    }
    if (bytesWritten === 0) {
      return bytesWritten;
    }
    if (
      (this.head >= this.tail && this.buffer.length - this.head >= bytesWritten) ||
      (this.head < this.tail && bytesWritten <= this.tail - this.head) ❸
    ) {
      // Достаточно места после head. Просто записать данные и увеличить head.
      this.buffer.set(data, this.head);
      this.head += bytesWritten;
    } else { ❹
      // Нужно разбить блок на два.
      const endSpaceAvailable = this.buffer.length - this.head;
      const endChunk = data.subarray(0, endSpaceAvailable);
      const beginChunk = data.subarray(endSpaceAvailable);
      this.buffer.set(endChunk, this.head);
      this.buffer.set(beginChunk, 0);
      this.head = beginChunk.length;
    }
    this.length += bytesWritten;
    return bytesWritten;
  }
```

❶ Чтобы этот код работал правильно, `data` должно быть экземпляром того же типа `TypedArray`, что и `this.buffer`. Это можно проверить с помощью статической проверки типа, утверждения или того и другого сразу.

❷ Если в буфере недостаточно места для записи всех данных, то мы записываем столько, сколько помещается, и возвращаем число записанных байтов. Тем самым мы сообщаем вызывающей стороне, что она должна подождать, пока какие-то данные будут прочитаны, перед тем как записывать новые.

❸ В этом условии проверяется, достаточно ли *непрерывного* места для записи данных. Так обстоит дело, когда *либо* голова следует за хвостом в массиве и свободного места после головы хватает для записи данных, *либо* когда голова предшествует хвосту и между головой и хвостом достаточно места. Если хотя бы одно из этих условий выполнено, то мы можем просто записать данные в массив и увеличить индекс `head` на длину данных.

❹ Если же условие не выполнено, то нужно записать данные до конца массива, а затем перейти в начало и записать остаток данных туда. Для разбиения данных на части мы используем метод `subarray()`, а не `slice()`, чтобы избежать ненужных операций копирования.

Запись сводится к копированию байтов с помощью метода `set()` и соответствующего изменения индекса `head`; особый случай возникает, когда данные нужно разбить на две части из-за пересечения границы массива. Чтение очень похоже и показано в методе `read()` в примере 6.15.

Пример 6.15 ❖ *ch6-ring-buffer/ring-buffer.js (часть 3)*

```
  read(bytes) {
    if (bytes > this.length) { ❶
      bytes = this.length;
    }
    if (bytes === 0) {
```

```
      return new Uint8Array(0);
    }
    let readData;
    if (
      this.head > this.tail || this.buffer.length - this.tail >= bytes ❷
    ) {
      // Данные занимают непрерывный блок.
      readData = this.buffer.slice(this.tail, bytes)
      this.tail += bytes;
    } else { ❸
      // Читать данные отдельно из конца и из начала буфера.
      readData = new Uint8Array(bytes);
      const endBytesToRead = this.buffer.length - this.tail;
      readData.set(this.buffer.subarray(this.tail, this.buffer.length));
      readData.set(this.buffer.subarray(0, bytes - endBytesToRead), endBytesToRead);
      this.tail = bytes - endBytesToRead;
    }
      this.length -= bytes;
      return readData;
    }
  }
```

❶ **Методу `read()` передается число *запрошенных* байтов. Если в очереди нет столько байтов, то возвращаются все находящиеся в очереди байты.**

❷ **Если запрошенные данные находятся в непрерывном участке массива, начинающемся с `tail`, то мы просто передаем их вызывающей стороне, используя `slice()` для получения копии байтов. После этого индекс `tail` перемещается на позицию за последним возвращенным байтом.**

❸ **Иначе данные пересекают границу массива, поэтому нужно прочитать оба блока и склеить их в обратном порядке. Для этого мы выделяем достаточно большой массив `Uint8Array` и копируем в него данные из начала и из конца буфера. Индекс `tail` после этого перемещается на позицию за последним прочитанным байтом в начале массива.**

Читая байты из очереди, важно *копировать* их, а не возвращать ссылку на позицию в кольцевом буфере. Иначе склеивать фрагменты придется вызывающей стороне. Потому-то мы и используем `slice()` или новый `Uint8Array` для возвращенных данных.

Теперь мы имеем работающую однопоточную ограниченную очередь, реализованную в виде кольцевого буфера. Если бы мы хотели использовать ее, когда один поток (*производитель*) записывает, а другой (*потребитель*) читает, то могли бы передать конструктору очереди `SharedArrayBuffer` в качестве хранилища данных и создать по экземпляру очереди в обоих потоках. К сожалению, пока что мы не использовали атомарные операции и не определили, где находятся критические секции, защищаемые блокировками. Поэтому если буфер будут использовать несколько потоков, то все закончится состоянием гонки и повреждением данных. Это нужно исправить.

В операциях чтения и записи предполагается, что ни одно из свойств `head`, `tail` и `length` не изменяется другими потоками во время операции. Возможно, в будущем нам удастся ослабить эти ограничения, но, начав с них, мы уж точно обеспечим потокобезопасность, которая позволит избежать состояний гонки. Для определения критических секций, в которых может одновременно находиться не более одного потока, можно воспользоваться написанным ранее классом `Mutex`.

Импортируем класс `Mutex` и добавим в файл код из примера 6.16, который содержит обертку уже написанного класса `RingBuffer`.

Пример 6.16 ❖ *ch6-ring-buffer/ring-buffer.js (часть 4)*

```
const Mutex = require('../ch6-mutex/mutex.js');

class SharedRingBuffer {
  constructor(shared/*: number | SharedArrayBuffer*/) {
    this.shared = typeof shared === 'number' ?
      new SharedArrayBuffer(shared + 16) : shared;
    this.ringBuffer = new RingBuffer(
      new Uint32Array(this.shared, 4, 3),
      new Uint8Array(this.shared, 16)
    );
    this.lock = new Mutex(new Int32Array(this.shared, 0, 1));
  }

  write(data) {
    return this.lock.exec(() => this.ringBuffer.write(data));
  }

  read(bytes) {
    return this.lock.exec(() => this.ringBuffer.read(bytes));
  }
}
```

Конструктор принимает или создает экземпляр `SharedArrayBuffer`. Заметим, что мы увеличили размер буфера на 16 байт, в которых будет храниться `Mutex`, нуждающийся в одноэлементном `Int32Array`, и метаданные `RingBuffer`, состоящие из трехэлементного `Uint32Array`. Организация памяти показана в табл. 6.2.

Таблица 6.2. Организация памяти SharedRingBuffer

Данные	Type[Size]	Индекс SharedArrayBuffer
`Mutex`	`Int32Array[1]`	`0`
Метаданные `RingBuffer`	`Uint32Array[3]`	`4`
Буфер `RingBuffer`	`Uint32Array[size]`	`16`

Операции `read()` и `write()` обернуты методом `exec()` класса `Mutex`. Напомним, что это предотвращает одновременное выполнение несколькими потоками кода критической секции, защищенной мьютексом. Обернув операции таким образом, мы можем быть уверены, что, даже если несколько потоков читают и записывают в одну и ту же очередь, все равно не возникнет состояний гонки из-за модификации `head` или `tail` другим потоком во время выполнения операции.

Чтобы посмотреть на эту структуру данных в действии, создадим потоки *производителей* и *потребителей*. Заведем `SharedRingBuffer` размером 100 байт. Потоки-производители будут в цикле записывать в `SharedRingBuffer` строку `"Hello, World!\n"` настолько быстро, насколько смогут с учетом необходимо-

сти захватить блокировку. Потоки-потребители будут пытаться читать по 20 байт, а мы будем протоколировать, сколько байтов им удалось прочитать в действительности. Весь необходимый код показан в пример 6.17; добавьте его в конец файла *ch6-ring-buffer/ring-buffer.js*.

Пример 6.17 ❖ *ch6-ring-buffer/ring-buffer.js (часть 5)*

```
const { isMainThread, Worker, workerData } = require('worker_threads');
const fs = require('fs');

if (isMainThread) {
  const shared = new SharedArrayBuffer(116);
  const threads = [
    new Worker(__filename, { workerData: { shared, isProducer: true } }),
    new Worker(__filename, { workerData: { shared, isProducer: true } }),
    new Worker(__filename, { workerData: { shared, isProducer: false } }),
    new Worker(__filename, { workerData: { shared, isProducer: false } })
  ];
} else {
  const { shared, isProducer } = workerData;
  const ringBuffer = new SharedRingBuffer(shared);
  if (isProducer) {
    const buffer = Buffer.from('Hello, World!\n');
    while (true) {
      ringBuffer.write(buffer);
    }
  } else {
    while (true) {
      const readBytes = ringBuffer.read(20);
      fs.writeSync(1, `Прочитано байтов: ${readBytes.length}\n`); ❶
    }
  }
}
```

❶ **Обратите внимание, что для записи числа байтов на `stdout` мы используем не `console.log()`, а функцию синхронной записи в соответствующий дескриптор. Так сделано, потому что в бесконечном цикле нет `await`. Воспользовавшись `console.log` или любым другим асинхронным регистратором, мы приостановили бы цикл событий Node.js и не увидели бы никакого вывода.**

Для запуска этого примера выполните команду

```
$ node ring-buffer.js
```

Вывод скрипта показывает, сколько байтов прочитано на каждой итерации каждого потока-потребителя. Поскольку мы всегда запрашиваем 20 байт, счетчик, больший этого числа, мы никогда не увидим. Но иногда проскальзывает 0, это означает, что в момент чтения очередь была пуста. Встречаются и другие значения, показывающие, что очередь заполнена частично.

Этот код можно настроить разными способами. Пропускная способность зависит от размера `SharedRingBuffer`, от числа производителей и потребителей, от размера записываемого сообщения и от того, сколько байтов пытается читать потребитель. Как всегда, для нахождения оптимальных значений необходимо измерять и настраивать. Пробуйте!

Безблокировочные очереди

Наша реализация кольцевого буфера функционально корректна, но не слишком эффективна. Для выполнения *любой* операции с данными все остальные потоки блокируются. Это, конечно, самый простой подход, но есть решения, в которых блокировки вообще не используются, а синхронизацию обеспечивают тщательно подобранные атомарные операции. Платой является сложность.

Модель акторов

Модель акторов – это программный паттерн конкурентных вычислений, придуманный еще в 1970-х годах. *Актором* называется примитивный контейнер, допускающий выполнение кода. Актор может выполнять логические операции, создавать других акторов, отправлять и принимать сообщения от акторов.

Акторы взаимодействуют с внешним миром путем передачи сообщений, в остальном они работают с собственной изолированной памятью. Актор – полноправная сущность в языке программирования Erlang[1], но его можно эмулировать и в JavaScript.

Модель акторов призвана поддержать сильно распараллеленные вычисления, не задумываясь о том, где на самом деле исполняется код и даже какой протокол используется для взаимодействия. На самом деле программе должно быть безразлично, взаимодействуют акторы локально или удаленно. На рис. 6.2 показано, как акторы могут быть распределены между процессами и компьютерами.

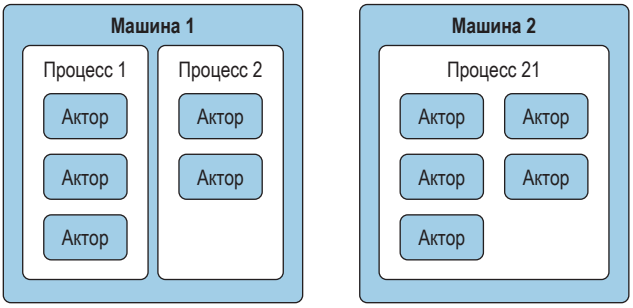


Рис. 6.2 ❖ Акторы могут быть распределены между процессами и машинами

Нюансы паттерна

Акторы могут обрабатывать получаемые сообщения или задачи по одному. Принятые сообщения сначала попадают в очередь, которая иногда назы-

[1] Еще одна достойная упоминания реализация паттерна акторов – язык Scala.

вается почтовым ящиком. Иметь очередь удобно, потому что два одновременно полученных сообщения не должны обрабатываться одновременно. Не будь очереди, один актор должен был бы проверять, готов ли другой актор, перед тем как отправить ему сообщение, что сильно усложнило бы процесс.

Хотя никакие два актора не могут писать в одно и то же место разделяемой памяти, они вправе изменять собственную память, в том числе хранить информацию о состоянии. Например, актор мог бы запоминать количество обработанных им сообщений, а затем доставлять эту информацию в отправляемых сообщениях.

Поскольку разделяемой памяти не существует, модель акторов не подвержена некоторым рассмотренным выше проблемам многопоточности, в частности в ней невозможны состояния гонки и взаимоблокировки. Во многих отношениях модель акторов ведет себя как функция в языке функционального программирования, т. е. принимает входные данные и избегает доступа к глобальному состоянию.

Так как акторы обрабатывают одну задачу в каждый момент времени, их часто можно реализовать однопоточно. И хотя один актор выполняет только одну задачу, ничто не мешает разным акторам работать параллельно.

Система на основе акторов не должна предполагать, что сообщения гарантированного обрабатываются в порядке FIFO. Напротив, она должна гибко реагировать на задержки и доставку не по порядку, особенно когда акторы распределены по сети.

Для однозначной идентификации акторов можно использовать понятие адреса. Одним из вариантов адреса является URI. Например, адрес `tcp://127.0.0.1:1234/3` мог бы принадлежать третьему актору в программе на локальном компьютере, прослушивающей порт 1234. В описываемой ниже реализации такие адреса не используются.

Акторы в JavaScript

Акторов, являющихся полноправными сущностями в таких языках, как Erlang, нельзя идеально воспроизвести в JavaScript, но попробовать-то можно. Есть десятки способов провести параллели и реализовать акторов, в этом разделе мы рассмотрим один из них.

Одна из привлекательных черт модели акторов – то, что акторы не ограничены одной машиной. То есть процессы могут работать на разных машинах и взаимодействовать по сети. Это можно реализовать с помощью процессов Node.js, которые обмениваются между собой JSON-сообщениями по протоколу TCP.

Поскольку отдельные акторы могут выполнять код параллельно с другими акторами и каждый актор одновременно обрабатывает только одну задачу, было бы разумно разместить их в разных потоках, чтобы оптимизировать использование системных ресурсов. Для этого можно было бы создавать для акторов рабочие потоки. Или иметь отдельный процесс для каждого актора, но это дороже.

Поскольку акторы не используют разделяемую память, про объекты `SharedArrayBuffer` и `Atomics` можно забыть (хотя в более развитой системе их можно было бы использовать для координации).

Акторам необходима очередь сообщений, где сообщения могут находиться, пока актор занят обработкой. Исполнители в JavaScript в какой-то мере решают это за нас, предоставляя метод `postMessage()`. Доставляемые им сообщения ждут, пока вызовы в текущем стеке JavaScript не закончатся, перед тем как выбирать следующее сообщение. Если каждый актор исполняет только синхронный код, то можно использовать эту встроенную очередь. Если же акторы могут исполнять и асинхронный код, то придется создавать очередь самостоятельно.

До сих пор модель акторов выглядела очень похоже на пул потоков, рассмотренный выше. И действительно, у них настолько много сходных черт, что можно было бы трактовать модель акторов как пул пулов потоков. Но и различий достаточно, чтобы не отождествлять эти концепции. В самом деле, модель акторов обещает уникальную парадигму вычислений, высокоуровневый паттерн программирования, который может изменить ваш подход к написанию кода. На практике модель акторов применяется к программам, которые зачастую опираются на пул потоков.

Пример реализации

Создайте новый каталог *ch6-actors/*. Скопируйте в него файл *ch6-thread-pool/rpc-worker.js* из примера 6.3 и файл *ch6-thread-pool/worker.js* из примера 6.2. Эти файлы лягут в основу пула потоков в этом примере и останутся неизменными.

Далее создайте файл *ch6-actors/server.js* и скопируйте в него код из примера 6.18.

Пример 6.18 ❖ *ch6-actors/server.js (часть 1)*

```
#!/usr/bin/env node
const http = require('http');
const net = require('net');

const [,, web_host, actor_host] = process.argv;
const [web_hostname, web_port] = web_host.split(':');
const [actor_hostname, actor_port] = actor_host.split(':');

let message_id = 0;
let actors = new Set(); // множество обработчиков акторов
let messages = new Map(); // ИД сообщения -> HTTP-ответ
```

В этом файле создается два сервера. Первый работает по базовому протоколу TCP, второй – по протоколу HTTP более высокого уровня, надстроенному над TCP. Друг от друга серверы не зависят. Первая часть файла содержит трафаретный код для разбора аргументов командой строки с параметрами серверов.

В переменной `message_id` хранится число, увеличивающееся на 1 после создания каждого HTTP-запроса. В переменной `messages` хранится отображение идентификаторов сообщений на обработчики, которые будут формировать ответы на сообщения. Это тот же паттерн, который мы использовали в разделе «Пул потоков» выше. Наконец, в переменной `actors` хранится коллекция функций-обработчиков, которые отправляют сообщения внешним процессам-акторам.

Добавьте в файл код из примера 6.19.

Пример 6.19 ❖ *ch6-actors/server.js (часть 2)*

```
net.createServer((client) => {
  const handler = data => client.write(JSON.stringify(data) + '\0'); ❶
  actors.add(handler);
  console.log('подключение к пулу акторов', actors.size);
  client.on('end', () => {
    actors.delete(handler); ❷
    console.log('отключение от пула акторов', actors.size);
  }).on('data', (raw_data) => {
    const chunks = String(raw_data).split('\0'); ❸
    chunks.pop();
    for (let chunk of chunks) {
      const data = JSON.parse(chunk);
      const res = messages.get(data.id);
      res.end(JSON.stringify(data) + '\0');
      messages.delete(data.id);
    }
  });
}).listen(actor_port, actor_hostname, () => {
  console.log(`actor: tcp://${actor_hostname}:${actor_port}`);
});
```

❶ **Между сообщениями вставляется нулевой байт `'\0'`.**
❷ **После закрытия соединение с клиентом удаляется из множества `actors`.**
❸ **Событие `data` может содержать несколько сообщений, разделенных нулевым байтом. Последний байт нулевой, т. е. последний элемент `chunks` – пустая строка.**

В этом файле создается TCP-сервер. Именно так процессы акторов подключаются к главному серверному процессу. Функция `net.createServer()` вызывается при каждом подключении процесса актора. Аргумент `client` представляет TCP-клиента, т. е. соединение с процессом актора. Каждый раз при установлении соединения в журнал выводится сообщение, а в коллекцию `actors` добавляется обработчик.

Когда клиент отключается от сервера, соответствующая ему функция-обработчик удаляется из коллекции `actors`. Акторы взаимодействуют с сервером путем отправки TCP-сообщений, активирующих событие `data`[1]. Эти

[1] Большие сообщения (например, если бы вместо нескольких чисел передавались строки) могут пересекать границы TCP-сообщений и доставляться в нескольких событиях `data`. Имейте это в виду, если захотите приспособить код к использованию в реальном приложении.

сообщения представлены в формате JSON и содержат поле `id`, идентифицирующее сообщение. Функция обратного вызова находит соответствующий обработчик в отображении `messages`. Затем сервер отправляет ответ, удаляет сообщение из `messages` и продолжает прослушивать указанный порт.

Соединение между сервером и клиентом-актором долговременное. Поэтому для событий `data` и `end` подготавливаются обработчики.

Обратите внимание, что в этом файле нет обработчика ошибок при работе с клиентским подключением. А раз так, то ошибка приведет к завершению сервера. Правильнее было бы удалять клиента из коллекции `actors`.

Между сообщениями вставляются нулевые байты, потому что нет гарантии, что каждое сообщение, отправленное одной стороной, будет доставлено другой в одном событии `data`. Напротив, если несколько сообщений посылаются быстро друг за другом, то они поступят в одном событии `data`. С такой проблемой вы не столкнетесь, если будете посылать запросы по одному с помощью `curl`, но, когда много запросов посылается командой `autocannon`, это произойдет неизбежно. В результате несколько JSON-документов конкатенируются, например: `{"id":1…}{"id":2…}`. Передача такого значения функции `JSON.parse()` приведет к ошибке. Благодаря нулевым байтам события выглядят так: `{"id":1…}\0{"id":2…}\0`. Такая строка разбивается по нулевым байтам, и каждый участок обрабатывается отдельно. Если нулевой байт встречается в JSON-объекте, то он экранируется, поэтому использовать нулевые байты для разделения JSON-документов безопасно.

Далее добавьте в файл код из примера 6.20.

Пример 6.20 ❖ *ch6-actors/server.js (часть 3)*

```
http.createServer(async (req, res) => {
  message_id++;
  if (actors.size === 0) return res.end('ERROR: EMPTY ACTOR POOL');
  const actor = randomActor();
  messages.set(message_id, res);
  actor({
    id: message_id,
    method: 'square_sum',
    args: [Number(req.url.substr(1))]
  });
}).listen(web_port, web_hostname, () => {
  console.log(`web: http://${web_hostname}:${web_port}`);
});
```

В этой части файла создает HTTP-сервер. В отличие от TCP-сервера, в этом случае запросам соответствуют кратковременные соединения. Функция `http.createServer()` вызывается при получении каждого HTTP-запроса.

Функция обратного вызова увеличивает идентификатор сообщения на 1 и смотрит на список акторов. Если список пуст, как бывает сразу после запуска сервера, когда еще ни один актор не успел подключиться, то возвращается сообщение об ошибке «ERROR: EMPTY ACTOR POOL». Если же акторы

присутствуют, то случайным образом выбирается один из них. Это не лучший подход – более правильное решение мы обсудим в конце раздела.

Затем актору отправляется JSON-сообщение. Оно содержит поле `id`, содержащее идентификатор, поле `method` с именем подлежащей вызову функции (в данном случае оно всегда равно `square_sum`) и список аргументов функции. В нашем случае путь в HTTP-запросе содержит косую черту и число, например */42*, и мы выделяем число, которое станет аргументом. Сервер продолжает прослушивать указанный порт.

Теперь добавьте в файл код из примера 6.21.

Пример 6.21 ❖ *ch6-actors/server.js (часть 4)*

```
function randomActor() {
  const pool = Array.from(actors);
  return pool[Math.floor(Math.random() * pool.length)];
}
```

Здесь мы просто выбираем из списка `actors` случайного актора. С этим файлом мы пока закончили. Далее создайте файл *ch6-actors/actor.js*. В нем не будет сервера, а будет код для подключения к другим серверным процессам. Для начала скопируйте в файл код из примера 6.22.

Пример 6.22 ❖ *ch6-actors/actor.js (часть 1)*

```
#!/usr/bin/env node
const net = require('net');

const RpcWorkerPool = require('./rpc-worker.js');
const [,, host] = process.argv;
const [hostname, port] = host.split(':');
const worker = new RpcWorkerPool('./worker.js', 4, 'leastbusy');
```

Файл начинается трафаретным кодом выделения имени сервера и номера порта из аргументов командной строки. Затем инициализируется пул потоков, представленный классом `RpcWorkerPool`. Пул содержит четыре потока, которые можно рассматривать как четырех акторов, и используется стратегия диспетчеризации `leastbusy`.

Добавьте в файл код из примера 6.23

Пример 6.22 ❖ *ch6-actors/actor.js (часть 2)*

```
const upstream = net.connect(port, hostname, () => {
  console.log('подключился к серверу');
}).on('data', async (raw_data) => {
  const chunks = String(raw_data).split('\0'); ❶
  chunks.pop();
  for (let chunk of chunks) {
    const data = JSON.parse(chunk);
    const value = await worker.exec(data.method, ...data.args);
    upstream.write(JSON.stringify({
      id: data.id,
```

```
      value,
      pid: process.pid
    }) + '\0');
  }
}).on('end', () => {
  console.log('отключение от сервера');
});
```

❶ **Актор также должен обрабатывать разделение по нулевым байтам.**

Метод `net.connect()` подключается к указанному компьютеру и порту, на котором работает серверный процесс, и печатает сообщение в случае успеха. Когда сервер отправляет этому актору сообщение, возникает событие `data`, в котором передается полезная нагрузка в виде JSON-строки (в аргументе `raw_data`). Эта строка разбирается.

Затем процесс актора вызывает один из своих рабочих потоков, передавая ему запрошенный метод и аргументы. Когда рабочий поток завершится, актор передает результат назад серверу. При этом в свойстве `id` находится тот же идентификатор сообщения, который был получен от сервера. Его необходимо указывать, потому что данный процесс актора может одновременно получать несколько запросов, а главный серверный процесс должен как-то соотносить ответы с запросами. В сообщении также передается возвращенное значение `value`. Кроме того, в ответ включены метаданные – идентификатор процесса в виде свойства `pid`, чтобы мы могли видеть, какая программа какие данные вычисляла.

И на этот раз обработка ошибок отсутствует. Если при обработке подключения произойдет ошибка, то весь процесс аварийно завершится.

На рис. 6.3 показана написанная нами реализация

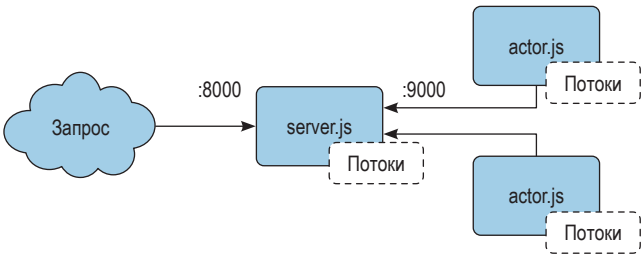


Рис. 6.3 ❖ Реализация модели акторов, созданная в этом разделе

Итак, все файлы готовы, и можно запустить программы. Сначала запустите сервер, указав имя хоста и порт для HTTP-сервера, а затем имя хоста и порт для TCP-сервера. Для этого выполните следующую команду:

```
$ node server.js 127.0.0.1:8000 127.0.0.1:9000
# web: http://127.0.0.1:8000
# actor: tcp://127.0.0.1:9000
```

Процесс выводит адреса обоих серверов.

Затем отправьте серверу запрос в новом окне терминала:

```
$ curl http://localhost:8000/9999
# ERROR: EMPTY ACTOR POOL
```

Сервер вернул ошибку. Поскольку нет ни одного работающего процесса актора, то некому выполнить работу.

Запустите процесс актора, указав имя хоста и порт сервера. Для этого выполните следующую команду:

```
$ node actor.js 127.0.0.1:9000
```

Будут напечатаны сообщения от сервера и рабочего процесса о том, что соединение установлено. Снова выполните команду `curl` в отдельном окне терминала:

```
$ curl http://localhost:8000/99999
# {"id":4,"value":21081376.519967034,"pid":160004}
```

Теперь программа напечатала содержательное сообщение. После подключения процесса актора у программы имеется не нуль, как раньше, а четыре актора, которые могут выполнить работу. Но необязательно на этом останавливаться. Откройте еще одно окно терминала и запустите в нем дополнительный рабочий процесс, а затем выполните команду `curl`:

```
$ node actor.js 127.0.0.1:9000
$ curl http://localhost:8000/8888888
# {"id":4,"value":21081376.519967034,"pid":160005}
```

Несколько раз выполнив `curl`, вы увидите, что значение `pid` в ответе изменяется. Принимайте поздравления, вам удалось динамически увеличить количество доступных приложению акторов. И сделано это во время выполнения, нам даже не пришлось останавливать приложение.

Одно из достоинств акторов заключается в том, что не имеет значения, где выполняется код. В данном случае акторы находятся во внешнем процессе. Из-за этого возникла ошибка при первом обращении к серверу: HTTP-запрос уже поступил, но процесс актора еще не подключился. Исправить это можно, включив акторов еще и в сам серверный процесс.

Измените файл *ch6-actors/server.js*, добавив код из примера 6.24.

Пример 6.24 ❖ *ch6-actors/server.js (часть 5, бонусная)*

```
const RpcWorkerPool = require('./rpc-worker.js');
const worker = new RpcWorkerPool('./worker.js', 4, 'leastbusy');
actors.add(async (data) => {
  const value = await worker.exec(data.method, ...data.args);
  messages.get(data.id).end(JSON.stringify({
    id: data.id,
    value,
    pid: 'server'
  }) + '\0');
```

```
  messages.delete(data.id);
});
```

Здесь мы создаем пул рабочих потоков в серверном процессе, включив в него четыре дополнительных актора. Остановите запущенные ранее сервер и процессы акторов, нажав Ctrl+C. Затем запустите новый сервер и отправьте ему запрос с помощью `curl`:

```
$ node server.js 127.0.0.1:8000 127.0.0.1:9000
$ curl http://localhost:8000/8888888
# {"id":8,"value":17667693458.923462,"pid":"server"}
```

В этом случае печатается зашитый в код `pid server`, означающий, что вычисление выполнял сам серверный процесс. Как и раньше, можно запустить дополнительные процессы акторов, которые подключатся к серверу, и выполнить еще несколько раз команду `curl`. Вы, скорее всего, увидите, что одни запросы обрабатываются самим сервером, а другие – внешними акторами.

Не нужно считать подключенные акторы внешним API. Лучше рассматривать их как расширение самой программы. Этот паттерн может оказаться весьма полезным, рассмотрим одно интересное применение. *Горячей загрузкой кода* называется ситуация, когда старая версия приложения заменяется новой, но приложение все это время продолжает работать. Построенный нам паттерн акторов позволяет модифицировать файлы *actor.js* и *worker.js*, изменить реализацию метода `square_sum()` и даже добавить новые методы. Затем можно запустить новых акторов и остановить старых, после чего главный сервер станет использовать новых акторов.

Стоит также отметить, что у рассмотренной в этом разделе версии модели акторов есть несколько недостатков, которые нужно устранить, прежде чем реализовывать нечто подобное в реальном проекте. Первый заключается в том, что хотя для выбора отдельных исполнителей внутри процесса актора применяется стратегия наименее занятого, сам процесс выбирается случайно. В результате возможно асимметричное распределение рабочей нагрузки. Чтобы исправить это, нужен какой-то механизм координации, запоминающий, какие акторы свободны.

Другой недостаток в том, одни акторы не допускают адресации другими, т. е. один актор не может вызвать код из другого. Архитектурно организация напоминает звездчатую топологию, в которой процессы акторов жестко связаны с серверным процессом. В идеале все акторы должны быть связаны друг с другом и иметь возможность адресовать друг друга.

Важное достоинство этого подхода – устойчивость. В описанном выше решении имеется всего один HTTP-сервер. Если он «ляжет», то остановится и все приложение. В более устойчивой системе каждый процесс должен быть одновременно HTTP- и TCP-сервером, а обратный прокси-сервер должен маршрутизировать запросы ко всем процессам. Внеся такие изменения, вы приблизитесь к реализации модели акторов, предлагаемой более надежными платформами.

Глава 7

WebAssembly

Хотя эта книга называется «Многопоточный JavaScript», современные среды выполнения JavaScript поддерживают также WebAssembly. Для тех, кто не знает, скажем, что WebAssembly (веб-сборка, часто встречается акроним WASM) – это двоичный формат команд, работающий в стековой виртуальной машине. При его проектировании много внимания уделялось безопасности, поэтому сборка работает в песочнице, где имеет доступ только к памяти и функциям, предоставляемым объемлющей средой. Главным побудительным мотивом для включения такой штуки в браузеры и другие среды выполнения JavaScript было желание выполнять особенно требовательные части приложения в среде, гораздо более быстрой, чем JavaScript. Другая цель – компилировать веб-сборки, написанные на таких языках, как C, C++ и Rust. Это открывает дверь в мир веб-разработки программистам, работающим на этих языках.

Вообще говоря, память, используемая модулями WebAssembly, представлена в виде буферов `ArrayBuffer`, но допустимы также `SharedArrayBuffer`. Кроме того, существуют команды WebAssembly для атомарных операций, аналогичных объекту `Atomics` в JavaScript. Имея `SharedArrayBuffer`, атомарные операции и веб-исполнители (или рабочие потоки в Node.js), мы располагаем всем необходимым для многопоточного программирования с использованием WebAssembly.

Но, прежде чем переходить к многопоточному коду, напишем и выполним приложение «Hello, World!», чтобы продемонстрировать сильные стороны и ограничения WebAssembly.

Ваша первая WebAssembly

Хотя WebAssembly – двоичный формат, существует также текстовый формат для представления кода в понятном человеку виде. Тут можно провести аналогию с машинным кодом и языком ассемблера. У текстового формата языка WebAssembly нет специального названия, но есть общепринятое расширение *.wat*, по которому и язык иногда называют WAT. В качестве основной синтаксической конструкции в нем используются *S-выражения*, удобные и для чтения, и для разбора. S-выражения, знакомые тем, кто пишет на языках из семейства Lisp, – это вложенные списки, заключенные в круглые скобки, элементы которых разделяются пробелами.

Чтобы лучше прочувствовать формат, напишем на WAT простую функцию сложения. Создайте файл *ch7-wasm-add/add.wat* и скопируйте в него код из примера 7.1.

Пример 7.1 ❖ *ch7-wasm-add/add.wat*

```
(module ❶
  (func $add (param $a i32) (param $b i32) (result i32) ❷
    local.get $a ❸
    local.get $b
    i32.add)
  (export "add" (func $add)) ❹
)
```

❶ **В первой строке объявлен модуль. Так начинается любой WAT-файл.**
❷ **Мы объявляем функцию $add, которая принимает два 32-разрядных целых числа и возвращает 32-разрядное целое.**
❸ **Эта строчка начинает тело функции, содержащее три предложения. Первые два помещают параметры функции в стек, один за другим. Напомним, что WebAssembly – стековая машина. Это означает, что многие операции работают с первым (если операция унарная) или с первыми двумя (если бинарная) элементами в стеке. Третье предложение – бинарная операция сложения двух значений типа i32, т. е. она снимает со стека два значения, складывает их и помещает результат на вершину стека. Возвращаемым значением функции является значение, оказавшееся на вершине стека после ее завершения.**
❹ **Чтобы функцию можно было использовать вне модуля в объемлющей среде, ее необходимо экспортировать. Здесь мы экспортируем функцию $add, сопоставляя ей внешнее имя add.**

Этот WAT-файл можно преобразовать в двоичную сборку WebAssembly с помощью инструмента `wat2wasm`, входящего в состав комплекта инструментов WebAssembly Binary Toolkit (WABT). Для этого, находясь в каталоге *ch7-wasm-add*, выполните следующую команду:

```
$ npx -p wabt wat2wasm add.wat -o add.wasm
```

Вот мы и получили свой первый файл WebAssembly! Вне объемлющей среды такие файлы бесполезны, поэтому напишем немного JavaScript-кода, чтобы загрузить и протестировать функцию `add`. Добавьте в файл *ch7-wasm-add/add.js* код из примера 7.2.

Пример 7.2 ❖ *ch7-wasm-add/add.js*

```
const fs = require('fs/promises'); // Необходима версия Node.js v14 или старше.

(async () => {
  const wasm = await fs.readFile('./add.wasm');
  const { instance: { exports: { add } } } = await WebAssembly.instantiate(wasm);
  console.log(add(2, 3));
})();
```

Если вы уже создали *wasm*-файл, как было описано выше, то теперь можете запустить его из каталога *ch7-wasm-add*.

```
$ node add.js
```

Вывод подтверждает, что мы действительно выполняем сложение с помощью модуля WebAssembly.

Простые математические операции с данными в стеке не работают с линейной памятью или концепциями, не имеющими смысла в WebAssembly, например строками. Возьмем строки в языке C. По существу, это не что иное, как указатель на начало массива байтов, завершающегося нулем. Целый массив невозможно ни передать функции WebAssembly, ни вернуть из нее, но можно передать массив по ссылке. Это означает, что для передачи строки в качестве аргумента мы должны сначала выделить необходимое число байтов в линейной памяти, записать в них строку и передать индекс первого байта функции WebAssembly. Ситуация усложняется тем, что необходимо как-то управлять доступным пространством в линейной памяти. То есть нам нужны реализации `malloc()` и `free()`, оперирующие линейной памятью[1].

Писать код WebAssembly вручную на WAT, конечно, можно, но это не самый легкий путь к продуктивности и повышению производительности. Этот язык проектировался как конечная цель компиляции с языков более высокого уровня, и вот тут-то все его плюсы проявляются в полной мере. В разделе «Компиляция программ с C на WebAssembly с помощью Emscripten» ниже эта тема обсуждается более подробно.

Атомарные операции в WebAssembly

Хотя полное рассмотрение всех команд WebAssembly (https://webassembly.github.io/spec/core/text/index.html) выходит за рамки этой книги, стоит отметить команды, относящиеся к атомарным операциям с разделяемой памятью, потому что это ключ к многопоточному коду на WebAssembly, и неважно, создан он путем компиляции с другого языка или написан вручную на WAT.

Имена команд WebAssembly часто начинаются с типа. В случае атомарных операций тип всегда равен `i32` или `i64`, что соответствует 32- и 64-разрядным целым числам. Далее в именах всех атомарных команд следует строка `.atomic.`, а после нее имя конкретной команды.

Рассмотрим некоторые атомарные команды. Мы не будем вдаваться во все детали синтаксиса, а дадим представление о том, какие виды операций доступны на уровне команд.

`[i32|i64].atomic.[load|load8_u|load16_u|load32_u]`
: Семейство команд `load` эквивалентно методу `Atomics.load()` в JavaScript. Команды с разными суффиксами позволяют загружать значения меньшей разрядности, при этом оставшиеся позиции заполняются нулями.

`[i32|i64].atomic.[store|store8|store16|store32]`
: Семейство команд `store` эквивалентно методу `Atomics.store()` в JavaScript. Команды с разными суффиксами позволяют расширить входное значение до указанного числа разрядов и сохранить в позиции с указанным индексом.

---

[1] В C и других языках, не имеющих механизмов автоматического управления памятью, память выделяется с помощью функций типа `malloc()`, а затем освобождается с помощью функций типа `free()`. Такие методы управления памятью, как сборка мусора, упрощают написание программ на языках более высокого уровня типа JavaScript, но в WebAssembly они не встроены.

`[i32|i64].atomic.[rmw|rmw8|rmw16|rmw32].[add|sub|and|or|xor|xchg|cmpxchg][|_u]`

Команды из семейства `rmw` выполняют операции чтения-модификации-записи, эквивалентные методам `add()`, `sub()`, `and()`, `or()`, `xor()`, `exchange()` и `compareExchange()` объекта `Atomics` в JavaScript. Команды с суффиксом `_u` дополняют результат нулями. Часть имени после `.atomic.` говорит, сколько битов читать.

Для следующих двух операций соглашение об именовании немного отличается.

`memory.atomic.[wait32|wait64]`

Эквивалентны методу `Atomics.wait()` в JavaScript, а суффикс определяет разрядность.

`memory.atomic.notify`

Эквивалентна методу `Atomics.notify()` в JavaScript.

Этих команд достаточно для выполнения в WebAssembly тех же атомарных операций, которые имеются в JavaScript, но есть еще одна операция, которая в JavaScript отсутствует:

`atomic.fence`

Эта команда не имеет аргументов и ничего не возвращает. Она предназначена для использования в языках высокого уровня, не имеющих средств точного задания порядка неатомарных обращений к разделяемой памяти.

Все эти операции применяются к *линейной памяти* модуля WebAssembly, песочнице, в которой можно читать и записывать значения. При инициализации модулей WebAssembly из JavaScript им можно передать линейную память, хотя это и необязательно. Чтобы память можно было разделять между потоками, в ее основе должен лежать объект `SharedArrayBuffer`.

Хотя эти команды, конечно, можно писать прямо на языке WebAssembly, им присущ тот же недостаток, что и WebAssembly в целом: очень муторно и «напряжно». К счастью, у нас есть возможность компилировать на WebAssembly код, написанный на языках более высокого уровня.

Компиляция с C на WebAssembly с помощью Emscripten

Задолго до появления WebAssembly программа Emscripten (https://emscripten.org/) применялась для компиляции программ, написанных на C и C++, на язык JavaScript. Сегодня она поддерживает многопоточный код на C и C++ с помощью веб-исполнителей в браузерах и модуля `worker_threads` в Node.js.

На самом деле Emscripten позволяет без всяких проблем откомпилировать немало существующих многопоточных программ. И в Node.js, и в браузерах Emscripten эмулирует системные вызовы из платформенного кода, так что

программы, написанные на компилируемых языках, можно выполнять почти без изменений.

Собственно говоря, код на C, написанный нами в главе 1, можно откомпилировать, не внося вообще никаких изменений! Давайте попробуем. Для работы с Emscripten воспользуемся Docker-образом. Остальные инструменты должны выбираться в соответствии с вашей платформой, но WebAssembly и JavaScript платформенно-независимы, так что Docker-образ можно использовать всюду, где поддерживается Docker .

Прежде всего установите Docker (https://www.docker.com/). Затем, находясь в каталоге *ch1-c-threads*, выполните следующую команду:

```
$ docker run --rm -v $(pwd):/src -u $(id -u):$(id -g) \
emscripten/emsdk emcc happycoin-threads.c -pthread \
-s PTHREAD_POOL_SIZE=4 -o happycoin-threads.js
```

Тут есть несколько нуждающихся в обсуждении моментов. Мы монтируем образ `emscripten/emsdk` в текущем каталоге и исполняем от имени текущего пользователя. Все, начиная с команды `emcc` включительно, выполняется внутри контейнера. Выглядит это очень похоже на компиляцию C-программы компилятором `cc`. Основное отличие заключается в том, что на выходе получается JavaScript-файл, а не исполняемый двоичный файл. Не волнуйтесь! *Wasm*-файл тоже создается. JS-файл играет роль моста к необходимым системным вызовам и подготовке потоков, потому что сделать все это в самом модуле WebAssembly невозможно.

Еще один дополнительный аргумент: `-s PTHREAD_POOL_SIZE=4`. Поскольку в файле `happycoin-threads.c` используется три потока, мы заранее выделяем для них место здесь. Emscripten предлагает несколько способов создания потоков, не нарушающих запрет на блокировку главного потока браузера. Самое простое – заранее выделить их на этапе компиляции, потому что мы знаем, сколько потоков понадобится.

Теперь можно запустить многопоточную версию Happycoin на WebAssembly. Мы выполним JavaScript в среде Node.js. На момент написания книги для этого была нужна версия Node.js v16 или старше, потому что именно ее поддерживает Emscripten.

```
$ node happycoin-threads.js
```

Вывод будет выглядеть примерно так:

```
120190845798210000 ... [ еще 106 чисел ] ... 14356375476580480000
count 108
Pthread 0x9017f8 exited.
Pthread 0x701500 exited.
Pthread 0xd01e08 exited.
Pthread 0xb01b10 exited.
```

Все точно так же, как в примерах Happycoin из предыдущих глав, но обертка, предоставленная Emscripten, также информирует нас о завершении потоков. Для выхода из программы нужно нажать Ctrl+C. Ради интереса разбе-

ритесь, что нужно сделать, чтобы процесс сам завершался после завершения всех потоков и чтобы сообщения `Pthread` не печатались.

При сравнении версий Happycoin, написанных на компилируемом языке и на JavaScript, вы, наверное, обратили внимание на время работы. Очевидно, что новая версия работает быстрее, чем JavaScript, но немного медленнее, чем написанная на C. Как всегда, важно производить измерения, чтобы убедиться, что вы движетесь в правильном направлении.

Хотя в примере Happycoin атомарные операции не используются, Emscripten в полном объеме поддерживает потоки POSIX и функции атомарных операций, встроенные в компилятор GNU (GCC). Благодаря этому Emscripten позволяет откомпилировать на WebAssembly очень многие программы, написанные на C и C++.

Другие компиляторы на WebAssembly

Emscripten не единственный способ откомпилировать код на WebAssembly. На самом деле язык WebAssembly и проектировался в первую очередь как цель компиляции, а не как полноценный универсальный язык. Существует множество инструментов для компиляции с хорошо известных языков на WebAssembly и даже нескольких языков, специально созданных в расчете на то, что они будут компилироваться на WebAssembly, а не в машинный код. Некоторые перечислены ниже, но этот список далеко не полон (https://github.com/appcypher/awesome-wasm-langs). Обратите внимание на многократное употребление фразы «на момент написания этой книги», потому что эта область сравнительно новая, и лучшие способы создания многопоточного кода на WebAssembly еще только разрабатываются! По крайней мере, на момент написания этой книги.

Clang/Clang++

Компиляторы из семейства LLVM C могут компилировать на WebAssembly, если указать параметры `-target wasm32-unknown-unknown` или `-target wasm64-unknown-unknown`. Именно на них в настоящее время основан компилятор Emscripten, благодаря которому потоки POSIX и атомарные операции работают правильно. На момент написания этой книги они обеспечивают едва ли не лучшую поддержку многопоточности в WebAssembly. Хотя `clang` и `clang++` поддерживают трансляцию на WebAssembly, рекомендуется все же использовать Emscripten, поскольку он в полной мере поддерживает и браузеры, и Node.js.

Rust

Компилятор языка Rust `rustc` поддерживает трансляцию на WebAssembly. Сайт Rust дает отличную отправную точку (https://www.rust-lang.org/what/wasm) желающим узнать, как использовать `rustc` таким способом. Для работы с потоками нужно включить крейт `wasm-bindgen-rayon` (https://docs.rs/crate/wasm-bindgen-rayon/latest), который предоставляет API параллелизма, реализованный с помощью веб-исполнителей. На момент написания этой книги поддержка потоков в стандартной библиотеке Rust еще не работала.

AssemblyScript
: Компилятор AssemblyScript принимает подмножество языка TypeScript и порождает код на WebAssembly. Он не поддерживает запуск потоков, но поддерживает атомарные операции и использование `SharedArrayBuffer`, так что если работу с потоками вы реализуете сами на стороне JavaScript с помощью веб-исполнителей или модуля `worker_threads`, то можете наслаждаться всеми прелестями многопоточного программирования в программе на AssemblyScript. В следующем разделе мы поговорим об этом подробнее.

Конечно, есть много других вариантов, и их число с каждым днем растет. Поищите в сети, может ли ваш любимый компилируемый язык транслироваться на WebAssembly и поддерживаются ли при этом атомарные операции.

AssemblyScript

AssemblyScript (https://www.assemblyscript.org/) – подмножество TypeScript, транслируемое на WebAssembly. Вместо того чтобы взять существующий язык и предоставить реализации имеющегося системного API, создатели AssemblyScript решили спроектировать его, так чтобы порождать код на WebAssembly с помощью синтаксиса, гораздо более привычного, чем WAT. Основным аргументом в пользу AssemblyScript является то, что TypeScript уже используется во многих проектах, поэтому для добавления небольшого объема кода на AssemblyScript с целью получить все преимущества WebAssembly не потребуется не то что изучения совершенно нового языка программирования, но даже мысленного переключения контекста.

Модуль AssemblyScript очень похож на модуль TypeScript. Если вы незнакомы с TypeScript, то можете считать, что это обычный JavaScript с некоторыми добавлениями, позволяющими включить информацию о типах. Ниже приведен простой модуль TypeScript для сложения двух чисел.

```
export function add(a: number, b: number): number {
  return a + b
}
```

Выглядит это почти так же, как простой модуль ECMAScript, с тем отличием, что имеется информация о типе вида `: number` после каждого аргумента функции и после самой функции (определяет тип возвращаемого значения). С помощью этой информации компилятор TypeScript проверяет, что при вызове данной функции всегда передаются правильные типы и что использование возвращенного значения не противоречит его типу.

AssemblyScript отличается только тем, что вместо типа JavaScript `number` используются встроенные в WebAssembly типы. Если бы мы захотели написать эквивалентный модуль сложения на TypeScript, то в предположении, что используются 32-разрядные целые, он мог бы выглядеть, как в примере 7.3. Добавьте этот код в файл *ch7-wasm-add/add.ts*.

Пример 7.3 ❖ *ch7-wasm-add/add.ts*

```
export function add(a: i32, b: i32): i32 {
  return a + b
}
```

Поскольку файлы на AssemblyScript и на TypeScript по сути дела ничем не отличаются, в обоих случаях используется расширение *.ts*. Для компиляции файла AssemblyScript на WebAssembly служит команда `asc` из модуля `assemblyscript`. Выполните следующую команду, находясь в каталоге *ch7-wasm-add*:

```
$ npx -p assemblyscript asc add.ts --binaryFile add.wasm
```

Этот код на WebAssembly можно выполнить с использованием того же файла *add.js*, что в примере 7.2. Результат должен быть таким же, потому что код одинаковый.

Если опустить параметр `--binaryFile add.wasm`, то результатом будет модуль на языке WAT, показанный в примере 7.4. Как видите, он похож на код в примере 7.1.

Пример 7.4 ❖ Результат трансляции AssemblyScript-функции `add` на язык WAT

```
(module
  (type $i32_i32_=>_i32 (func (param i32 i32) (result i32)))
  (memory $0 0)
  (export "add" (func $add/add))
  (export "memory" (memory $0))
  (func $add/add (param $0 i32) (param $1 i32) (result i32)
    local.get $0
    local.get $1
    i32.add
  )
)
```

AssemblyScript не позволяет запускать потоки, но их можно запустить в среде JavaScript и использовать объекты `SharedArrayBuffer` в качестве памяти WebAssembly. Важно, что AssemblyScript поддерживает атомарные операции с помощью глобального объекта `atomics`, не особенно отличающегося от стандартного `Atomics` в JavaScript. Главное отличие заключается в том, что методы работают не с `TypedArray`, а с линейной памятью модуля WebAssembly, а вместо индекса используется указатель и необязательное смещение. Детали см. в документации по AssemblyScript (https://www.assemblyscript.org/stdlib/builtins.html).

Чтобы увидеть, как это работает на практике, создадим еще одну реализацию примера Happycoin, который развиваем еще с главы 1.

Happycoin на AssemblyScript

Как и в предыдущих версиях Happycoin, мы распределим вычисления по нескольким потокам, которые будут отправлять результаты главному потоку. Это лишь намек на то, как может работать многопоточная программа на

AssemblyScript. В реальном приложении вы, наверное, воспользовались бы разделяемой памятью и атомарными операциями, но, чтобы не усложнять, мы просто раздали работу потокам.

Создайте каталог *ch7-happycoin-as* и перейдите в него. Создайте новый проект и добавьте необходимые зависимости:

```
$ npm init -y
$ npm install assemblyscript
$ npm install @assemblyscript/loader
```

Пакет `assemblyscript` включает компилятор AssemblyScript, а пакет `assemblyscript/loader` предоставляет удобные средства для взаимодействия с созданными нами модулем.

В объект `scripts` вновь созданного файла *package.json* добавьте свойства `"build"` и `"start"`, чтобы упростить компиляцию и выполнение программы:

```
"build": "asc happycoin.ts --binaryFile happycoin.wasm --exportRuntime",
"start": "node --no-warnings --experimental-wasi-unstable-preview1 happycoin.mjs"
```

Параметр `--exportRuntime` дает нам некоторые высокоуровневые средства для взаимодействия со значениями из AssemblyScript. Мы вернемся к этому вопросу позже.

При вызове Node.js в скрипте `"start"` мы передаем экспериментальный флаг WASI. Он активирует системный интерфейс (WebAssembly System Interface – WASI) (https://wasi.dev/), предоставляющий WebAssembly доступ к системной функциональности, которая в противном случае осталась бы недоступной. Она понадобится нам для генерирования случайных чисел. Поскольку на момент написания этой книги флаг был экспериментальным, мы добавляем еще флаг `--no-warnings`[1], чтобы подавить предупреждения. Экспериментальный статус означает также, что флаг может измениться в будущем, поэтому обязательно обращайтесь к документации по той версии Node.js, с которой работаете.

Пора уже написать код на AssemblyScript! В примере 7.5 приведена версия Happycoin на этом языке. Скопируйте код в файл *happycoin.ts*.

Пример 7.5 ❖ *ch7-happycoin-as/happycoin.ts*

```
import 'wasi'; ❶

const randArr64 = new Uint64Array(1);
const randArr8 = Uint8Array.wrap(randArr64.buffer, 0, 8); ❷
function random64(): u64 {
  crypto.getRandomValues(randArr8); ❸
  return randArr64[0];
}

function sumDigitsSquared(num: u64): u64 {
```

[1] Вообще говоря, компилировать производственное приложение с этим флагом не рекомендуется. Надеемся, что, когда вы будете читать этот текст, поддержка WASI будет уже штатной. И тогда лишние флаги можно будет убрать.

```
  let total: u64 = 0;
  while (num > 0) {
    const numModBase = num % 10;
    total += numModBase ** 2;
    num = num / 10;
  }
  return total;
}

function isHappy(num: u64): boolean {
  while (num != 1 && num != 4) {
    num = sumDigitsSquared(num);
  }
  return num === 1;
}

function isHappycoin(num: u64): boolean {
  return isHappy(num) && num % 10000 === 0;
}

export function getHappycoins(num: u32): Array<u64> {
  const result = new Array<u64>();
  for (let i: u32 = 1; i < num; i++) {
    const randomNum = random64();
    if (isHappycoin(randomNum)) {
  result.push(randomNum);
    }
  }
  return result;
}
```

❶ Здесь импортируется модуль `wasi`, чтобы загрузить необходимые глобальные объекты.
❷ Мы инициализировали массив `Uint64Array` для случайных чисел, но метод `crypto.getRandomValues()` работает только с типом `Uint8Array`, поэтому мы создали соответствующее представление этого буфера. Кроме того, конструкторы `TypedArray` в AssemblyScript не перегружены, а вместо этого существует статический метод `wrap()` для построения новых экземпляров `TypedArray` из экземпляров `ArrayBuffer`.
❸ Ради этого метода мы и активировали WASI.

Если вы знакомы с TypeScript, то, возможно, подумали, что этот файл очень напоминает результат переноса на TypeScript кода из раздела «И снова Happycoin» главы 3. И вы правы! Это одно из главных достоинств AssemblyScript. Мы не пишем на совершенно новом языке, и тем не менее написанный код очень хорошо отображается на WebAssembly. Заметим, что экспортированная функция возвращает значение типа `Array<u64>`. В WebAssembly экспортируемые функции не могут возвращать массивы, но могут вернуть индекс позиции в памяти модуля (в действительности, указатель); именно это здесь и происходит. Мы могли бы сделать это вручную, но, как мы увидим, загрузчик AssemblyScript сильно упрощает дело.

Поскольку AssemblyScript не имеет средств для запуска потоков, мы должны сделать это в JavaScript. В данном случае воспользуемся модулями ECMAScript, позволяющими применить `await` на верхнем уровне. Скопируйте код из примере 7.6 в файл *happycoin.mjs*.

Пример 7.6 ❖ *ch7-happycoin-as/happycoin.mjs*

```
import { WASI } from 'wasi'; ❶
import fs from 'fs/promises';
import loader from '@assemblyscript/loader';
import { Worker, isMainThread, parentPort } from 'worker_threads';

const THREAD_COUNT = 4;

if (isMainThread) {
  let inFlight = THREAD_COUNT;
  let count = 0;
  for (let i = 0; i < THREAD_COUNT; i++) {
    const worker = new Worker(new URL(import.meta.url)); ❷
    worker.on('message', msg => {
    count += msg.length;
    process.stdout.write(msg.join(' ') + ' ');
    if (--inFlight === 0) {
      process.stdout.write('\ncount ' + count + '\n');
    }
  });
}
} else {
  const wasi = new WASI();
  const importObject = { wasi_snapshot_preview1: wasi.wasiImport };
  const wasmFile = await fs.readFile('./happycoin.wasm');
  const happycoinModule = await loader.instantiate(wasmFile, importObject);
  wasi.start(happycoinModule);

  const happycoinsWasmArray =
    happycoinModule.exports.getHappycoins(10_000_000/THREAD_COUNT);
  const happycoins = happycoinModule.exports.__getArray(happycoinsWasmArray);
  parentPort.postMessage(happycoins);
}
```

❶ **Это нельзя сделать без флага `--experimental-wasi-unstable-preview1`.**

❷ **Для читателей, незнакомых с ESM, это может показаться странным. Нам недоступна переменная `__filename`, как в модулях CommonJS. Вместо этого свойство `import.meta.url` дает полный путь в виде URL-адреса. Этот путь нужно передать конструктору `URL`, чтобы его можно было подать на вход конструктору `Worker`.**

Как и в разделе «И снова Happycoin» главы 3, мы проверяем, находимся ли в главном потоке, и если да, то запускаем четыре рабочих потока. В главном потоке мы ожидаем только сообщения в порт `MessagePort` по умолчанию, которое будет содержать массив найденных Happycoin'ов. Когда все рабочие потоки пришлют такие сообщения, мы печатаем на консоль их и общее количество.

В ветви `else` мы находимся в одном из рабочих потоков и инициализируем экземпляр WASI для передачи модулю WebAssembly. А затем с помощью `@assemblyscript/loader` создаем модуль, который дает нам все необходимое для работы с массивом, возвращенным из функции `getHappycoins`. Мы вызываем метод `getHappycoins()`, экспортированный модулем, а тот дает нам указатель на массив в линейной памяти WebAssembly. Функция `__getArray`, предостав-

ляемая загрузчиком, преобразует этот указатель в массив JavaScript, который затем можно использовать как обычно. Этот массив передается главному потоку для вывода.

Чтобы запустить этот пример, выполните две приведенные ниже команды. Первая компилирует AssemblyScript в WebAssembly, а вторая запускает результат с помощью только что написанной JavaScript-программы.

```
$ npm run build
$ npm start
```

Результат выглядит примерно так же, как в предыдущих вариантах Harpycoin:

```
7641056713284760000 ... [ еще 134 числа ] ... 10495060512882410000
count 136
```

Как всегда, важно сравнивать результат с подходящим эталоном. В качестве упражнения замерьте время работы этой реализации и сравните с написанными ранее. Она быстрее или медленнее? Можете ли вы объяснить, в чем причина? Что можно улучшить?

Глава 8

Анализ

Сейчас вы уже довольно хорошо знакомы с разработкой многопоточных приложений на JavaScript – браузерных, серверных и даже комбинированных. И, хотя в этой книге приведено немало сценариев и справочных материалов, ни разу мы не сказали «вот сюда следует добавить многопоточность». И тому есть важная причина.

Вообще говоря, основной причиной для добавления рабочих потоков в приложение является желание повысить производительность. Но за это приходится расплачиваться дополнительной сложностью. *Принцип KISS* – «Keep It Simple, Stupid» (Будь проще, дурашка)[1] – гласит, что приложение должно быть до такой степени простым, что разобраться в нем можно с одного взгляда на код. Возможность прочитать написанный код чрезвычайно важна, а добавление в программу потоков без определенной цели, вне всякого сомнения, нарушает принцип KISS.

Существуют убедительные обоснования для добавления в приложение потоков, и если, измерив производительность, вы убедились, что выигрыш в скорости перевешивает затраты на сопровождение, то, наверное, ситуация складывается в пользу потоков. Но как понять, помогут потоки или нет, не реализовав их? И как измерить их влияние на производительность?

Когда не стоит использовать потоки

Потоки не панацея, способная решить все проблемы производительности приложения. И обычно это не самый легкий способ добиться цели, так что выбирать его нужно, лишь когда все другие возможности исчерпаны. Особенно это относится к JavaScript, где сообщество понимает многопоточность не так хорошо, как в других языках. Добавление многопоточности может потребовать внесения масштабных изменений в приложение, поэтому отношение затраченных усилий к достигнутому результату будет выше, если сначала устранить другие причины неэффективности кода.

Сделав это и добившись хорошей производительности в других частях приложения, вы оказываетесь лицом к лицу с вопросом: «Не пора ли доба-

---

[1] О происхождении названия и вариантах его перевода см. статью *https://ru.wikipedia.org/wiki/KISS_(принцип)*. – *Прим. перев.*

вить многопоточность?» Далее в этом разделе мы опишем некоторые ситуации, когда добавление потоков, скорее всего, не принесет никакой выгоды. Это поможет вам избежать кропотливой исследовательской работы.

Ограничения на объем памяти

С запуском нескольких потоков в JavaScript связаны дополнительные издержки в части памяти, поскольку браузер должен выделить дополнительную память для новой среды JavaScript; сюда входят такие вещи, как глобальные объекты и API, доступные вашему коду, а также память, используемая самим движком. Эти издержки могут оказаться несущественными в серверном окружении при использовании Node.js или на «прокачанном» ноутбуке в случае браузеров. Но если речь идет о встраиваемом устройстве с процессором ARM и 512 МБ памяти или о купленном на спонсорские деньги нетбуке для школьного кабинета, то такие издержки могут стать серьезным «тормозом».

И как же дополнительные потоки влияют на потребление памяти? Дать количественную оценку довольно трудно, да и зависит она от движка JavaScript и платформы. Безопасный ответ такой: как и для любых вопросов, связанных с производительностью, нужно измерять в реальных условиях. Но некоторые цифры мы попробуем получить.

Для начала рассмотрим совсем примитивную программу для Node.js, которая просто запускает таймер и не импортирует никаких сторонних модулей, например:

```
#!/usr/bin/env node

const { Worker } = require('worker_threads');
const count = Number(process.argv[2]) || 0;

for (let i = 0; i < count; i++) {
  new Worker(__dirname + '/worker.js');
}

console.log(`PID: ${process.pid}, ADD THREADS: ${count}`);
setTimeout(() => {}, 1 * 60 * 60 * 1000);
```

Запустив эту программу и померив потребление памяти, получим:

```
# Терминал 1
$ node leader.js 0
# PID 10000

# Терминал 2
$ pstree 10000 -pa # только в Linux
$ ps -p 10000 -o pid,vsz,rss,pmem,comm,args
```

Команда `pstree` отображает потоки в программе. Она показывает главный поток V8 JavaScript, а также некоторые фоновые потоки, описанные в разделе «Скрытые потоки» главы 1:

```
node,10000 ./leader.js
  ├─{node},10001
  ├─{node},10002
  ├─{node},10003
  ├─{node},10004
  ├─{node},10005
  └─{node},10006
```

Команда ps отображает сведения о процессе и, прежде всего, о потреблении им памяти, например:

```
  PID     VSZ   RSS   %MEM  COMMAND COMMAND
66766 1409260 48212    0.1  node    node ./leader.js
```

Здесь есть две важных переменных, существенных для оценки потребления памяти программой, обе выражены в килобайтах. Первая, VSZ, или *размер виртуальной памяти*, показывает, сколько памяти процесс может использовать; сюда включается выгруженная на диск память, выделенная память и даже память, используемая разделяемыми библиотеками (например, TLS), – всего приблизительно 1.4 ГБ. Вторая, RSS, или *размер резидентного набора*, показывает, сколько физической памяти процесс использует в настоящее время – приблизительно 48 МБ.

Измерение памяти – вещь приблизительная; оценить, сколько процессов реально может поместиться в памяти, трудно. В данном случае нас больше интересует значение RSS.

Теперь рассмотрим более сложный вариант этой программы, с использованием потоков. Мы снова запустим простейший таймер, но теперь в программе будет несколько потоков. Потребуется импортировать файл *worker.js*:

```
console.log(`WPID: ${process.pid}`);
setTimeout(() => {}, 1 * 60 * 60 * 1000);
```

Если запустить программу *leader.js* с числовым аргументом, большим 0, то будут созданы дополнительные рабочие потоки. В табл. 8.1 показано, как изменяется потребление памяти (измеренное ps) в зависимости от числа потоков.

Таблица 8.1. Потребление памяти при увеличении числа потоков в Node.js v16.5

Дополнительных потоков	VSZ	RSS	SIZE
0	318 124 КБ	31 836 КБ	47 876 КБ
1	787 880 КБ	38 372 КБ	57 772 КБ
2	990 884 КБ	45 124 КБ	68 228 КБ
4	1 401 500 КБ	56 160 КБ	87 708 КБ
8	2 222 732 КБ	78 396 КБ	126 672 КБ
16	3 866 220 КБ	122 992 КБ	205 420 КБ

На рис. 8.1 показана связь между RSS и числом потоков.

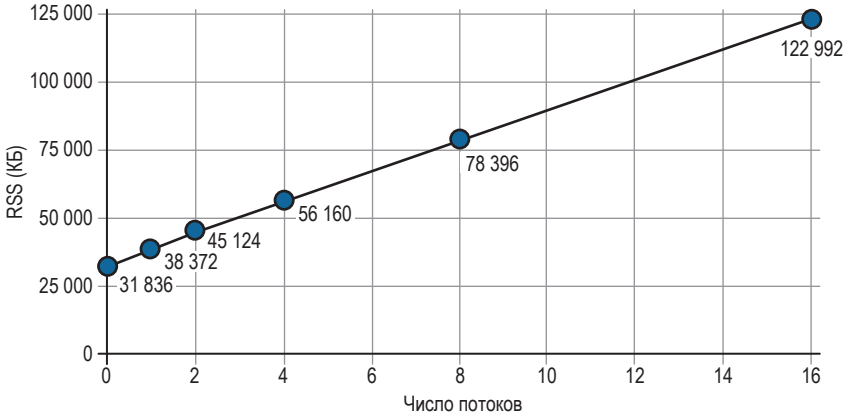


Рис. 8.1 ❖ Потребление памяти увеличивается при добавлении каждого нового потока

Из этого графика следует, что приращение памяти при добавлении каждого потока в случае Node.js 16.5 на процессоре x86 составляет примерно 6 МБ. Повторим – это очень приблизительная оценка, вы должны провести измерения в своей ситуации. Конечно, накладные расходы возрастут, если потоки импортируют больше модулей. Если вы создаете в каждом потоке экземпляры тяжелых каркасов, а то и веб-серверы, то вполне может статься, что дополнительный поток будет привносить в процесс сотни мегабайтов.

Хотя в настоящее время все реже встречаются программы, работающие на 32-разрядном компьютере или смартфоне, стоит отметить, что для них максимальный объем адресуемой памяти составляет 4 ГБ. И эта память разделяется между всеми потоками программы.

Недостаточное число ядер

Программа может работать медленно, если число ядер мало. В особенности это относится к одноядерным машинам, но может быть верно и для двухъядерных. Даже если в приложении используется пул потоков, масштабируемый в зависимости от числа ядер, приложение все равно замедлится, если создаст единственный рабочий поток. Действительно, после этого общее число потоков в приложении увеличится до двух (главный и рабочий), и оба будут конкурировать за процессор.

Еще одна причина замедления – накладные расходы на взаимодействие между потоками. При наличии одного ядра и двух потоков, даже если они не конкурируют за ресурсы, т. е. главному потоку нечего делать, когда исполняется рабочий поток, и наоборот, все равно имеются накладные расходы на передачу сообщений между двумя потоками.

Быть может, это и не страшно. Например, если допускающее распределенное выполнение приложение может работать во многих средах, чаще всего в многоядерных системах, но изредка в одноядерных, то с такими накладны-

ми расходами можно смириться. Но если вы создаете приложение, которое почти всегда работает в одноядерной среде, то лучше вообще не использовать потоки. То есть не стоит разрабатывать приложение, которое пользуется всеми ресурсами вашего «навороченного» ноутбука, а затем развертывать его в производственной среде, где оркестратор контейнеров выделяет приложению всего одно ядро.

О каких потерях производительности идет речь? В ОС Linux очень просто сказать операционной системе, что некоторая программа и все ее потоки должны использовать только подмножество процессорных ядер. Соответствующая команда позволяет разработчику проверить, как поведет себя многопоточное приложение в среде с небольшим числом ядер. Если вы работаете с компьютером под управлением Linux, то выполните приведенные ниже примеры, для остальных будет приведена итоговая информация.

Перейдите в каталог *ch6-thread-pool/*, созданный в разделе «Пулы потоков» главы 6. Запустите приложение, так чтобы оно создало пул с двумя рабочими потоками:

```
$ THREADS=2 STRATEGY=leastbusy node main.js
```

Заметим, что если размер пула потоков равен 2, то в среде JavaScript приложение работает с тремя потоками, а `libuv` по умолчанию имеет пул размером 5, так что всего в Node.js v16 будет восемь потоков. Разрешив приложению использовать все имеющиеся ядра, вы сможете получить эталонный тест. Выполните следующую команду, которая посылает пачку запросов серверу:

```
$ npx autocannon http://localhost:1337/
```

В данном случае нас интересует средний темп запросов, который в результирующей таблице показан в столбце Avg строки Req/Sec. При одном прогоне было получено значение 17.5.

Остановите сервер нажатием **Ctrl**+**C** и запустите снова. Но на этот раз воспользуйтесь командой `taskset`, чтобы заставить процесс (и все его дочерние потоки) использовать одно и то же ядро:

```
# Только в Linux
$ THREADS=2 STRATEGY=leastbusy taskset -c 0 node main.js
```

В данном случае заданы две переменные среды `THREADS` и `STRATEGY`, а потом запущена команда `taskset`. Флаг `-c 0` говорит, что программе разрешено использовать только нулевой CPU. Последующие аргументы интерпретируются как запускаемая команда. Отметим, что команду `taskset` можно использовать и для модификации уже работающего процесса. В таком случае команда выведет полезную информацию о происходящем. Ниже показано, что печатает команда на компьютере с 16 ядрами.

```
pid 211154's current affinity list: 0-15
pid 211154's new affinity list: 0
```

В данном случае программа, которая раньше имела доступ ко всем 16 ядрам (0–15), теперь может получить в свое распоряжение только одно (0).

Теперь, когда программа ограничена одним ядром с целью эмулировать среду с небольшим числом процессором, выполните тот же тест еще раз:

```
$ npx autocannon http://localhost:1337/
```

При одном прогоне среднее число запросов в секунду упало до 8.32. Это означает, что попытка использовать три потока JavaScript в одноядерной среде снизила пропускную способность данной конкретной программы до 48 % от уровня, достигнутого при использовании всех ядер!

Возникает естественный вопрос: каким должен быть размер пула потоков и сколько ядер следует предоставить приложению *ch6-threadpool*, чтобы максимизировать его пропускную способность? Чтобы ответить на него, мы проверили 16 комбинаций параметров теста и измерили производительность в каждом случае. Продолжительность теста была увеличена до двух минут, чтобы избавиться от выбросов. Табличная версия полученных данных приведена в табл. 8.2.

Таблица 8.2. Зависимость пропускной способности от числа доступных ядер и размера пула потоков

	1 ядро	2 ядра	3 ядра	4 ядра
1 поток	8.46	9.08	9.21	9.19
2 потока	8.69	9.60	17.61	17.28
3 потока	8.23	9.38	16.92	16.91
4 потока	8.47	9.57	17.44	17.75

Графическое изображение тех же данных показано на рис. 8.2.

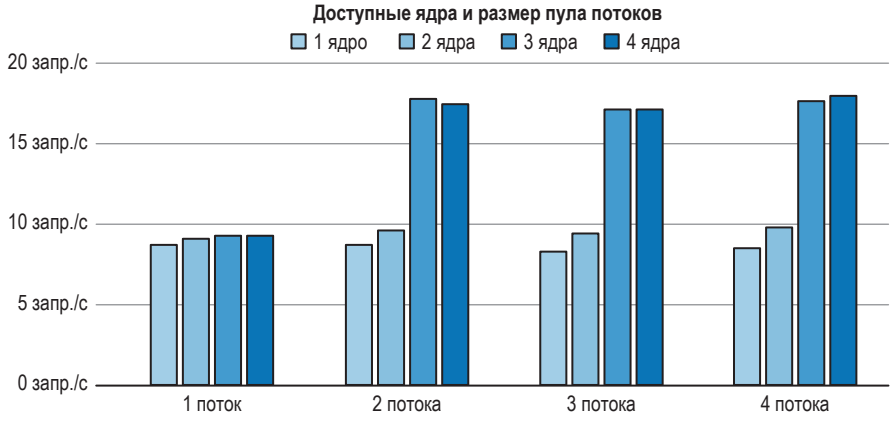


Рис. 8.2 ❖ Зависимость пропускной способности от числа доступных ядер и размера пула потоков

В данном случае налицо очевидный выигрыш в производительности, когда число потоков в пуле не меньше двух и приложению доступно не менее трех ядер. В остальных отношениях данные не представляют особого интереса.

При измерении зависимости производительности от числа ядер и потоков в реальном приложении вы, вероятно, увидите более интересные эффекты.

При изучении этих данных возникает вопрос: почему добавление дополнительных потоков сверх двух или трех не дает ускорения? Для ответа на подобные вопросы нужно выдвинуть гипотезу, поставить эксперимент и попытаться расшить узкие места. В данном случае проблема, возможно, заключается в том, что главный поток настолько занят координацией, обработкой запросов и взаимодействием между потоками, что рабочим потокам просто нечего делать.

Контейнеры и потоки

При написании серверных программ, например для Node.js, есть эвристическое правило: процессы должны горизонтально масштабироваться. Этот ученый термин означает, что нужно выполнять несколько автономных версий программы, изолированных, например, в Docker-контейнере. Горизонтальное масштабирование полезно тем, что позволяет разработчикам точно настраивать производительность целой когорты приложений. Это не так легко сделать, когда примитивы масштабируемости скрыты внутри программы, например в форме пула потоков.

Оркестраторы, например Kubernetes, – это инструменты, которые распределяют контейнеры по нескольким серверам. Они упрощают масштабирование по запросу; на период отпусков инженер может вручную увеличить количество работающих экземпляров. Оркестраторы могут также динамически изменять масштаб в зависимости от таких эвристик, как потребление CPU, пропускная способность сети и даже размер очереди работ.

Как могло бы выглядеть такое динамическое масштабирование, если бы выполнялось в самом приложении во время выполнения? Понятно, что нужно было бы изменять размер пула потоков. Также нужно было реализовать какой-то механизм, который позволил бы инженеру отправлять процессам сообщения с требованием изменить размер пула; быть может, стоило бы завести дополнительный сервер, который прослушивал бы порт, в который направляются такие административные команды. Подобная функциональность увеличивает сложность приложения.

Хотя добавление дополнительных процессов вместо увеличения числа потоков увеличивает общее потребление ресурсов, не говоря уже о накладных расходах, связанных с помещением процессов в контейнер, крупные компании обычно отдают предпочтение такому подходу, ценя присущую ему гибкость масштабирования.

Когда стоит использовать потоки

Иногда нам везет, поскольку решаемая задача сильно выигрывает от внедрения многопоточности. Приведем несколько простых характеристик таких задач, на которые стоит обращать внимание.

Естественная параллельность

В этом случае большую задачу можно разбить на меньшие части, у которых вообще или почти нет разделяемой информации. Одна из таких задач – игра «Жизнь» Конвея – была рассмотрена в главе 5. В ней сетку можно разбить на части и поручить обработку каждой части отдельному потоку.

Трудоемкие математические вычисления

Хорошо приспособлены для многопоточности также задачи, в которых производятся сложные математические вычисления, т. е. счетные задачи. Можно, конечно, сказать, что процессор ничего, кроме математических вычислений, и не делает, но противоположностью счетных задач считаются, например, те, что занимаются в основном вводом–выводом или сетевыми операциями. Взять, к примеру, программу взлома паролей со слабым SHA1-хешем. Она выполняет алгоритм Secure Hash Algorithm 1 (SHA1) для каждой возможной комбинации 10 символов пароля, и уж тут вычислений точно хватает.

Задачи, ориентированные на MapReduce

MapReduce – это модель программирования, вдохновленная функциональным стилем программирования. Она часто применяется для крупномасштабной обработки данных, распределенных между несколькими машинами. MapReduce состоит из двух этапов. На первом, Map (распределение), принимается один список значений и порождается другой. На втором, Reduce (редукция), новый список обрабатывается еще раз, и порождается одно значение. Однопоточную версию этой модели можно было бы написать на JavaScript с помощью функций `Array#map()` и `Array#reduce()`, но для многопоточной нужно, чтобы разные потоки обрабатывали подмножества списков данных. Поисковая система использует Map для поиска ключевых слов в миллионах документов, а затем Reduce, чтобы оценить и ранжировать их, предложив пользователю страницу наиболее релевантных результатов. Технология MapReduce используется также в таких базах данных, как Hadoop и MongoDB.

Обработка графики

Многие задачи обработки графической информации также выигрывают от наличия нескольких потоков. Как и игра «Жизнь», которая разворачивается на сетке клеток, изображения представлены сеткой пикселей. В обоих случаях значение в каждой позиции представляется числом, хотя в игре «Жизнь» это одно однобитовое число, а для изображения обычно нужно 3 или 4 байта (красный, зеленый, синий каналы и необязательный канал альфа, описывающий прозрачность). Тогда фильтрация изображений сводится к разбиению изображения на меньшие части, которые обрабатываются потоками из пула, и обновлению интерфейса по завершении обработки.

Это не полный список всех ситуаций, в которых следует использовать многопоточность, а лишь самые очевидные случаи.

Часто можно встретить утверждение, что задачи, не требующие разделения данных или, по крайней мере, не нуждающиеся в координации чтения

и записи разделяемых данных, проще моделировать с помощью нескольких потоков. Хотя код, не имеющий побочных эффектов, вообще лучше, особенно это верно в отношении многопоточного кода.

Еще одна ситуация, в которой JavaScript-приложения могут выиграть особенно много, – отрисовка шаблонов. Библиотеки бывают разные, но часто для отрисовки используется шаблон в виде строки и объект, содержащий подставляемые в шаблон переменные. В таких случаях глобальное состояние невелико, всего два элемента входных данных, и возвращается одна выходная строка. Так отрисовывают шаблоны популярные пакеты `mustache` и `handlebars`. Передача отрисовки шаблона в приложении Node.js отдельному рабочему потоку – то место, где, похоже, можно достичь выигрыша в производительности.

Проверим это предположение. Создайте каталог *ch8-template-render/*. Скопируйте в него файл *ch6-thread-pool/rpc-worker.js* из примера 6.3. Этот файл будет работать и без модификации, но все же закомментируйте предложение `console.log()`, чтобы не замедлять тестирование производительности.

Необходимо также инициализировать npm-проект и установить парочку пакетов. Для этого выполните следующие команды:

```
$ npm init -y
$ npm install fastify@3 mustache@4
```

Затем создайте файл *server.js*. Он представляет HTTP-приложение, которое выполняет простую отрисовку HTML-кода при получении запроса. В этой программе мы воспользуемся сторонними пакетами вместо встроенных модулей. Скопируйте в файл код из примера 8.1.

Пример 8.1 ❖ *ch8-template-render/server.js (часть 1)*

```
#!/usr/bin/env node

// npm install fastify@3 mustache@4
const Fastify = require('fastify');
const RpcWorkerPool = require('./rpc-worker.js');
const worker = new RpcWorkerPool('./worker.js', 4, 'leastbusy');
const template = require('./template.js');
const server = Fastify();
```

Сначала мы создаем экземпляр веб-каркаса Fastify, а также пул с четырьмя рабочими потоками. Кроме того, приложение загружает модуль *template.js*, который будет отрисовывать шаблоны.

Теперь все готово, чтобы объявить несколько маршрутов и начать прослушивание запросов. Добавьте в файл код из примера 8.2.

Пример 8.2 ❖ *ch8-template-render/server.js (часть 2)*

```
server.get('/main', async (request, reply) =>
  template.renderLove({ me: 'Thomas', you: 'Katelyn' }));

server.get('/offload', async (request, reply) =>
  worker.exec('renderLove', { me: 'Thomas', you: 'Katelyn' }));
```

```
server.listen(3000, (err, address) => {
  if (err) throw err;
  console.log(`listening on: ${address}`);
});
```

В приложении два маршрута. Первый, `GET /main`, выполняет отрисовку ответа на запрос в главном потоке. Он представляет однопоточное приложение. Второй, `GET /offload`, соответствует отрисовке в отдельном рабочем потоке. Наконец, в последнем предложении сервер начинает прослушивать порт 3000.

Сейчас приложение функционально готово. Но в качестве дополнительного бонуса хорошо бы количественно оценить работу, выполняемую сервером. Хотя основным средством тестирования эффективности этого приложения является тест производительности, отправляющий HTTP-запросы, иногда полезно взглянуть и на другие цифры, Добавьте в файл последний фрагмент – код из примера 8.3.

Пример 8.3 ❖ *ch8-template-render/server.js (часть 3)*

```
const timer = process.hrtime.bigint;
setInterval(() => {
  const start = timer();
  setImmediate(() => {
    console.log(`delay: ${(timer() - start).toLocaleString()}ns`);
  });
}, 1000);
```

В этом коде имеется вызов `setInterval`, срабатывающий каждую секунду. Он обертывает вызов `setImmediate()`, который измеряет время в наносекундах между моментами вызова функции и возврата из нее. Способ не идеальный, но позволяет приближенно оценить загрузку процесса. По мере того как цикл событий становится более занятым, печатаемое число должно увеличиваться. Кроме того, занятость цикла событий влияет на задержку асинхронных операций во всем процессе. Поэтому чем меньше это число, тем выше производительность приложения.

Далее создайте файл *worker.js* и добавьте в него код из примера 8.4.

Пример 8.4 ❖ *ch8-template-render/worker.js*

```
const { parentPort } = require('worker_threads');
const template = require('./template.js');

function asyncOnMessageWrap(fn) {
  return async function(msg) {
    parentPort.postMessage(await fn(msg));
  }
}

const commands = {
  renderLove: (data) => template.renderLove(data)
};
```

```
parentPort.on('message', asyncOnMessageWrap(async ({ method, params, id }) => ({
  result: await commands[method](...params), id
})));
```

Это модифицированная версия исполнителя, созданного ранее. В данном случае имеется только одна команда, `renderLove()`, – она принимает объект, содержащий пары ключ–значение, который будет использован функцией отрисовки шаблона.

Наконец, создайте файл *template.js* и скопируйте в него код из примера 8.5.

Пример 8.5 ❖ *ch8-template-render/template.js*

```
const Mustache = require('mustache');
const love_template = "<em>{{me}} loves {{you}}</em> ".repeat(80);

module.exports.renderLove = (data) => {
  const result = Mustache.render(love_template, data);
  // Mustache.clearCache();
  return result;
};
```

В реальном приложении этот файл можно было бы использовать для чтения файлов шаблонов с диска и подстановки в них значений. Но в этом простом примере экспортируется всего один отрисовщик шаблона и имеется один зашитый в код шаблон. В шаблоне есть две переменных, `me` и `you`. Строка повторяется много раз, чтобы аппроксимировать длину шаблона в реальном приложении. Чем длиннее шаблон, тем дольше он отрисовывается.

Итак, все файлы созданы, и можно запускать приложение. Выполните следующие команды, которые запускают сервер и прогоняют для него тест производительности:

```
# Терминал 1
$ node server.js

# Терминал 2
$ npx autocannon -d 60 http://localhost:3000/main
$ npx autocannon -d 60 http://localhost:3000/offload
```

В тестовом прогоне на мощном 16-ядерном ноутбуке, когда шаблоны отрисовывались целиком в главном потоке, средняя пропускная способность приложения составляла 13 285 запросов в секунду. В том же тесте, но при отрисовке шаблонов в рабочем потоке средняя пропускная способность составила 18 981 запросов в секунду. В данном примере налицо рост пропускной способности на 43 %.

Задержка цикла событий тоже резко уменьшилась. Среднее время, измеряемое в вызове `setImmediate()`, когда процесс ничем не занят, равно 87 мкс. Если шаблоны отрисовываются в главном потоке, то средняя задержка составляет 769 мкс. А при переносе отрисовки в рабочий поток она составляет 232 мкс. Вычтя задержку в состоянии простоя из обоих значений, получим ускорение в 4.7 раз за счет использования потоков. На рис. 8.3 приведены результаты выборочного сравнения на протяжении теста продолжительностью 60 с.

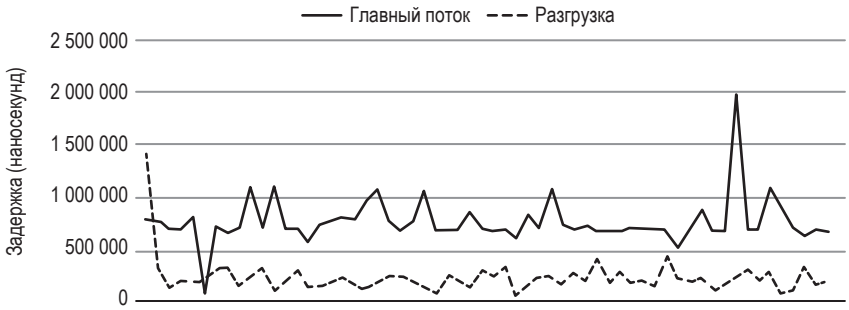


Рис. 8.3 ❖ Задержка цикла событий в однопоточной и многопоточной версиях

Означает ли это, что все приложения нужно перерабатывать с целью перенести отрисовку в другой поток? Необязательно. В нашем искусственном примере приложение удалось ускорить с помощью дополнительных потоков, но это было сделано на 16-ядерной машине. Весьма вероятно, что в производственной среде у ваших приложений не будет доступа к такому большому числу ядер.

Но надо признать, что важнейшим фактором, определяющим производительность, был размера шаблона. Если бы шаблон был меньше, например если бы мы убрали повтор строки, то и отрисовывать его в одном потоке было бы быстрее. Причина замедления в том, что накладные расходы на передачу данных шаблона между потоками гораздо выше, чем время, требуемое для отрисовки шаблона.

Как обычно, воспринимайте результаты тестов производительности с долей скептицизма. Такие изменения в приложении нужно проверять в производственной среде, только тогда вы будете точно знать, принесли дополнительные потоки какую-то выгоду или нет.

Подводные камни

Ниже приводится сводка всех упомянутых в книге подводных камней при работе с потоками в JavaScript.

Сложность

При использовании разделяемой памяти приложения становятся сложнее. Особенно это заметно, когда вы вручную вызываете методы `Atomics` и работаете с экземплярами `SharedBufferArray`. Впрочем, признаем, что значительную часть сложности можно скрыть, пользуясь сторонними модулями. В таком случае рабочие потоки можно представить вполне элегантно, а координацию и взаимодействие между ними и главным потоком абстрагировать.

Избыточное потребление памяти

Каждый добавленный в программу поток требует памяти. Эти накладные расходы тем выше, чем больше модулей загружается в поток. И хотя в современных компьютерах памяти обычно достаточно, для пущей уверенности имеет смысл тестировать программу на том оборудовании, на

котором ее предполагается эксплуатировать. В какой-то мере сгладить эту проблему можно, отслеживая, какой код импортируется в отдельных потоках. Не нужно загружать вагон и маленькую тележку!

Отсутствуют разделяемые объекты
Невозможность разделять объекты между потоками может затруднить преобразование однопоточной программы в многопоточную. Когда нужно изменить состояние объекта, приходится передавать сообщения, так чтобы получатель мог изменить принадлежащий ему и только ему объект.

Отсутствует доступ к DOM
Только главный поток браузерного приложения имеет доступ к DOM. Поэтому передать задачи отрисовки UI другому потоку проблематично. Но ничто не мешает главному потоку отвечать за изменение DOM, а дополнительным – заниматься трудной работой и возвращать главному ее результаты для обновления UI.

Модифицированные API
Потокам доступны не все API, эта проблема сродни отсутствию доступа к DOM. В браузере это означает, например, запрет на вызов `alert()`, кроме того, действуют дополнительные ограничения в исполнителях конкретных типов: запрет блокирующих запросов `XMLHttpRequest#open()`, ограничения на локальное хранилище (`localStorage`), на `await` на верхнем уровне и т. д. Хотя некоторые отличия не слишком существенны, все равно не всякий код можно выполнить, не внося изменений, в любом контексте JavaScript. В этом отношение документация – ваш лучший друг.

Ограничения со стороны алгоритма структурированного клонирования
Алгоритм структурированного клонирования налагает некоторые ограничения, которые затрудняют передачу некоторых экземпляров классов между потоками. В настоящее время, даже если два потока имеют доступ к одному и тому же определению класса, экземпляры этого класса, передаваемые из одного потока в другой, становятся экземплярами `Object`. Хотя привести данные к типу исходного класса возможно, это требует ручных действий.

Браузеры требуют специальных заголовков
При работе с разделяемой памятью в браузере посредством `SharedArrayBuffer` сервер должен добавить два специальных заголовка в запрос HTML-документа, нужного странице. Если вы полностью контролируете сервер, то сделать это легко. Но если сервер размещен на хостинге, то иногда добавить такие заголовки трудно, а то и невозможно. Даже пакет, использованный в этой книге для размещения локального сервера, пришлось модифицировать, чтобы добавить эти заголовки.

Определение готовности потока
Не существует встроенной возможности узнать, когда запущенный поток будет готов к работе с разделяемой памятью. Вместо этого нужно построить решение так, чтобы оно сначала прозвонило поток, а потом дождалось ответа.

Приложение

Алгоритм структурированного клонирования

Алгоритм структурированного клонирования – это механизм, которым движки JavaScript пользуются при копировании объектов с помощью некоторых API. Прежде всего речь идет о передаче данных между исполнителями, хотя другие API тоже используют этот алгоритм. Суть механизма сводится к тому, что данные сериализуются, а затем десериализуются в новый объект в другой области JavaScript.

Когда объекты клонируются таким способом, например для передачи из главного потока в рабочий или из одного рабочего потока в другой, модификация объекта на одной стороне никак не отражается на объекте по другую сторону. По сути дела, мы теперь имеем две копии данных. Цель алгоритма структурированного клонирования – предоставить обработчикам более удобный механизм, чем `JSON.stringify`, наложив при этом разумные ограничения.

Браузеры применяют алгоритм структурированного клонирования при копировании данных между веб-исполнителями. В Node.js он также используется для копирования данных между рабочими потоками. Вообще, видя вызов `.postMessage()`, можно сразу сказать, что данные клонируются этим алгоритмом. Браузеры и Node.js следуют одним и тем же правилам, но поддерживают разные множества объектов, допускающих копирование.

Простое, но не точное правило звучит так: любые данные, которые можно представить в формате JSON, допускают безопасное клонирование. Если не отклоняться от такого представления данных, то сюрпризов будет очень мало. Однако алгоритм структурированного клонирования поддерживает и некоторые другие типы.

Во-первых, можно клонировать все примитивные типы данных, имеющиеся в JavaScript, за исключением типа `Symbol`. К ним относятся `Boolean`, `null`, `undefined`, `Number`, `BigInt` и `String`.

Экземпляры классов `Array`, `Map` и `Set`, служащие для хранения коллекций данных, тоже можно клонировать. Можно даже передавать экземпляры `ArrayBuffer`, `ArrayBufferView` и `Blob`, применяемые для хранения двоичных данных.

Более сложные объекты, если они универсальны и не допускают двояких толкований, тоже можно передавать. Сюда относятся объекты, созданные конструкторами `Boolean` и `String`, `Date` и даже экземпляры `RegExp`[1].

В браузере можно клонировать более сложные и не так хорошо известные объекты, в том числе `File`, `FileList`, `ImageBitmap` и `ImageData`.

В Node.js к числу специальных объектов, допускающих клонирование, относятся `WebAssembly.Module`, `CryptoKey`, `FileHandle`, `Histogram`, `KeyObject`, `MessagePort`, `net.BlockList`, `net.SocketAddress` и `X509Certificate`. Копировать можно даже экземпляры классов `ReadableStream`, `WritableStream` и `TransformStream`.

Алгоритм структурированного клонирования, в отличие от копирования JSON-объектов, позволяет клонировать рекурсивные объекты (некоторые свойства которых ссылаются на другие свойства). Алгоритм понимает, что, встретив вложенный объект-дубликат, следует прекратить сериализацию.

Имеется несколько недостатков, которые могут повлиять на ваш код. Во-первых, таким образом нельзя клонировать функции. Функция может иметь весьма сложную природу. Например, у нее есть область видимости, но она может обращаться к переменным, объявленным вне нее. Передача чего-то подобного между областями не имеет смысла.

Запрещается также передавать элементы DOM в браузере. Значит ли это, что ничего из того, что делает веб-исполнитель, нельзя показать пользователю в DOM? Вовсе нет. Нужно просто, чтобы веб-исполнитель вернул значение, которая можно будет трансформировать и показать пользователю в главной области JavaScript. Например, если вы собираетесь выполнить 1000 итераций алгоритма `fibonacci` в веб-исполнителе, то можете затем вернуть числовое значение, и вызывающий JavaScript-код поместит его в DOM.

Объекты в JavaScript довольно сложно устроены. Иногда их можно создать с помощью синтаксиса объектного литерала. А иногда – с помощью создания экземпляра базового класса. А еще их можно модифицировать с помощью дескрипторов свойств и акцессоров чтения и записи. Алгоритм структурированного клонирования сохраняет только базовые свойства объектов.

Это прежде всего означает, что если вы определили свой класс и попросили клонировать экземпляр, то клонированы будут только собственные свойства этого экземпляра, и результирующий объект будет экземпляром класса `Object`. Свойства, определенные в прототипе, тоже не клонируются. Даже если вы определите `class Foo {}` на вызывающей стороне и внутри веб-исполнителя, результат клонирования все равно будет иметь тип `Object`. Связано это с тем, что нет никакого способа убедиться, что на обеих сторонах определение класса `Foo` одинаково[2].

---

[1] С объектами `RegExp` есть небольшой подвох. Они содержат свойство `.lastIndex`, которое используется, когда регулярное выражение применяется несколько раз к одной и той же строке, и позволяет узнать, где закончилось последнее соответствие выражению. Это свойство не передается.

[2] Существуют предложения, как разрешить сериализацию и десериализацию экземпляров класса, например «User-defined structured clone for JavaScript objects» (https://github.com/littledan/serializable-objects), поэтому в будущем это ограничение, возможно, будет снято.

Некоторые объекты категорически отказываются клонироваться. Например, попытавшись передать `window` из главного потока в рабочий или вернуть `self` в обратном направлении, вы получите одно из следующих сообщений об ошибке в зависимости от браузера:

```
Uncaught DOMException: The object could not be cloned.
DataCloneError: The object could not be cloned.
```

Различные движки JavaScript не во всем согласованы, поэтому лучше тестировать код в нескольких браузерах. Например, Chrome и Node.js поддерживают клонирование объектов `Error`, а Firefox пока нет[1].

Общее правило таково: JSON-совместимые объекты никогда не должны вызывать проблем, но более сложные типы могут. Поэтому лучше всегда передавать как можно более простые данные.

[1] Разработчики Firefox планируют поддержать эту возможность (см. «Allow structured cloning of native error types» (https://bugzilla.mozilla.org/show_bug.cgi?id=1556604)).

Предметный указатель

Б

В

Г

Д

И

К

Р

С

Т

У

Ф

Ц

Книги издательства «ДМК ПРЕСС»
можно купить оптом и в розницу
в книготорговой компании «Галактика»
(представляет интересы издательств
«ДМК ПРЕСС», «СОЛОН ПРЕСС», «КТК Галактика»).
Адрес: г. Москва, пр. Андропова, 38;
тел.: **(499) 782-38-89**, электронная почта: **books@alians-kniga.ru**.
При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.
Эти книги вы можете заказать и в интернет-магазине: **www.a-planeta.ru**.

Томас Хантер II, Брайан Инглиш

Многопоточный JavaScript

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Зам. главного редактора *Сенченкова Е. А.*
Перевод *Снастин А. В.*
Корректор *Абросимова Л. А.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Гарнитура PT Serif. Печать цифровая.
Усл. печ. л. 15,28. Тираж 200 экз.

Веб-сайт издательства: **www.dmkpress.com**

O'REILLY®

Готовы ли вы к разработке многопоточных приложений на JavaScript? Прочитав данную книгу, JavaScript-разработчик среднего уровня узнает о сильных и слабых сторонах API веб-исполнителей в браузерах и рабочих потоков в Node.js.

Авторы рассказывают о двух подходах к построению многопоточных приложений: на основе передачи сообщений и на основе разделяемой памяти. Описываются API для обоих случаев, объясняется, когда использовать тот или другой, а когда — их сочетание. Вы также увидите, как реализуются высокоуровневые паттерны на базе этих подходов.

- Как получить отдачу от многопоточного программирования.
- Различие между выделенными, разделяемыми и сервисными исполнителями.
- Когда стоит, а когда не стоит использовать потоки в приложении.
- Организация взаимодействия потоков с помощью объекта Atomics.
- Применение полученных знаний к построению высокопроизводительных приложений.
- Тестирование производительности для оценки полезности потоков.

«Брайан и Томас квалифицированно описывают основы многопоточной разработки и искусно иллюстрируют, как различные среды выполнения JavaScript допускают параллельные вычисления».

Джеймс Снелл, член технического комитета Node.js

«Такую книгу я хотел бы прочитать, когда начинал углубленное изучение рабочих потоков. Авторы вдаются в детали и предлагают хороший справочный материал. Отдельное спасибо за главу, посвященную анализу».

Маттео Коллина, Главный архитектор программного обеспечения, компания NearForm

Томас Хантер II участвовал в разработке сервисов Node.js и работал в компании, занимающейся обеспечением безопасности Node.js. Выступал на нескольких конференциях по Node.js и JavaScript, имеет сертификат JSNSD/JSNAD. Организатор NodeSchool SF.

Брайан Инглиш разрабатывает проекты с открытым исходным кодом на JavaScript и Rust. Занимался крупными корпоративными системами, безопасностью на уровне приложений. В настоящее время старший инженер на проектах с открытым исходным кодом в компании Datadog. Является соразработчиком ядра Node.js, внес немалый вклад в различные аспекты Node.js.

ISBN 978-5-93700-129-0

Интернет-магазин: www.dmkpress.com

Оптовая продажа: КТК «Галактика»
books@alians-kniga.ru

ДМК ПРЕСС ИЗДАТЕЛЬСТВО
www.дмк.рф

9 785937 001290 >