УДК 004.9
ББК 32.072
Б28

Баттальини Р.
Б28 Искусство WebAssembly / пер. с англ. П. М. Бомбаковой. – М.: ДМК Пресс, 2021. – 310 с.: ил.

ISBN 978-5-97060-976-7

В книге подробно рассматриваются принципы работы WebAssembly – компактной межплатформенной технологии, которая оптимизирует производительность ресурсоемких веб-приложений и программ.

Вы узнаете, как оптимизировать, компилировать и отлаживать низкоуровневый код, сравнивать его производительность с JavaScript, а также представлять код в удобном для прочтения текстовом формате WebAssembly Text (WAT). Затем сможете создать программу обнаружения столкновений на базе браузера, поработать с технологиями рендеринга в браузере для создания графики и анимации и выяснить, как WebAssembly взаимодействует с другими языками программирования.

Книга адресована веб-разработчикам, желающим понять, как создавать и развертывать приложения на основе WebAssembly, а также пользователям, которые хотят изучить и применять эту технологию.

УДК 004.9
ББК 32.072

ISBN 978-1-7185-0144-7 (англ.)
ISBN 978-5-97060-976-7 (рус.)

СОДЕРЖАНИЕ

Об авторе

Рик Баттальини (Rick Battagline) – разработчик игр и автор книги «Hands-On Game Development with WebAssembly» (Packt Publishing, 2019). Занимается браузерными технологиями с 1994 г. В 2006 г. он основал независимую студию разработки веб-игр BattleLine Games LLC. Разработанная студией игра Epoch Star была номинирована на награду конкурса Slamdance Guerilla Games Competition. На сегодняшний день под его авторством созданы сотни игр, созданных с использованием различных веб-технологий, таких как WebAssembly, HTML5, WebGL, JavaScript, TypeScript, ActionScript и PHP.

О техническом рецензенте

Конрад Уатт (Conrad Watt) – научный сотрудник Питерхауса, Кембриджский университет (Peterhouse, University of Cambridge). Прежде чем получить степень доктора компьютерных наук в Кембриджском университете, он успешно прошел обучение по программе бакалавриата в Имперском колледже Лондона (Imperial College London). Исследования Конрада Уатта сосредоточены преимущественно на формальной семантике и характеристиках безопасности WebAssembly. Он является разработчиком WasmCert-Isabelle – первой в мире механизации семантики языка WebAssembly с помощью инструмента Isabelle/HOL.

Конрад является активным членом WebAssembly Community Group и продолжает участвовать в создании спецификаций новых языковых возможностей, делая упор на многопоточность и параллелизм. Его исследование характеристик слабых моделей памяти JavaScript и WebAssembly является ключевым компонентом спецификации потоков WebAssembly. Вне профессиональной жизни он увлекается хоровым пением, а также много путешествует (напоминаем, что ввиду ситуации, сложившейся в мире на момент написания этой книги, заниматься тем и другим не рекомендуется).

ПРЕДИСЛОВИЕ

Сегодня, когда большинство языков успешно компилируются в JavaScript, WebAssembly представляет собой новый виток развития технологий, который позволит выйти за пределы устоявшихся рамок. WebAssembly является универсальным эффективным инструментом для выполнения кода на вашем любимом языке в браузере, который к тому же позволяет переосмыслить способы взаимодействия многократно используемых программных компонентов не только в сети, но и на других платформах, начиная от блокчейна и заканчивая граничными вычислениями интернета вещей (IoT – Internet of things).

Безусловно, WebAssembly – это молодая технология, на развитие которой уйдет немало времени. Однако уже сегодня ее явный потенциал вдохновил огромное количество самых разных людей. В качестве примера хотелось бы привести разрабатываемый совместно с Риком (Rick) проект AssemblyScript. Здесь мы рассматриваем WebAssembly не как язык системного программирования, а как инструмент для объединения лучших концепций данной технологии с JavaScript. AssemblyScript позволяет компилировать варианты кода JavaScript, схожие с TypeScript, в WebAssembly, формируя при этом сверхмалые и эффективные модули. Все это дает возможность ощутить преимущества WebAssembly тем, кто работает с JavaScript.

WebAssembly обладает множеством интересных функций и аспектов, которые будут интересны тем, кто стремится исследовать новые технологии и внести свою лепту в их развитие. Книга «Искусство WebAssembly» поможет овладеть базовыми знаниями для дальнейшего более глубокого погружения в тонкости работы технологий, способных совершить революцию в области вычислений и во всей сети.

– ДЭНИЕЛ ВИРТЦ (DANIEL WIRTZ),
создатель AssemblyScript

БЛАГОДАРНОСТИ

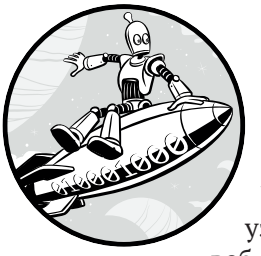
В первую очередь хотелось бы выразить благодарность Лиз Чедвик (Liz Chadwick), работавшей над первыми набросками этой книги. Благодаря неутомимой работе и приложенным усилиям она воплотила мой смутный поток мыслей в связную письменную форму и превратила его в полноценный проект. Если вам понравится читать эту книгу, будьте уверены, что это заслуга Лиз (Liz) и результат ее усилий, приложенных на каждом этапе создания.

Я хочу поблагодарить Конрада Уатта (Conrad Watt), который является приглашенным экспертом рабочей группы W3C WebAssembly, за предоставление экспертных технических рецензий. Я искренне восхищаюсь его талантом. Его технический опыт в данной области невозможно переоценить. Конрад Уатт (Conrad Watt) провел глубокий и доскональный технический анализ данной книги.

Также я хочу сказать спасибо Катрине Тейлор (Katrina Taylor) и Энн Мари Уокер (Anne Marie Walker). Я искренне признателен за вашу работу по подготовке книги к печати. И конечно, я благодарю своих друзей Винита Капура (Vineet Kapur), Стива Тэка (Steve Tack) и Терри Коэна (Terri Cohen), которые нашли время, чтобы ознакомиться с моим первым черновиком и дать по этому поводу свои комментарии. Все вы помогли мне сделать эту книгу лучше.

Особую благодарность хотелось бы выразить Биллу Поллоку (Bill Pollock), чья помощь в некоторых критических моментах помогла мне продвинуться дальше и завершить эту книгу.

ВВЕДЕНИЕ



Добро пожаловать в «Искусство WebAssembly». Цель этой книги – показать, как читать, понимать и писать код WebAssembly на уровне виртуальной машины. Здесь вы узнаете, как WebAssembly взаимодействует с JavaScript, веб-браузером и средой встраивания. После прочтения книги вы поймете, что такое WebAssembly, узнаете об оптимальных вариантах его применения, а также научитесь писать коды WebAssembly с высокой скоростью их исполнения в браузере.

Для кого предназначена эта книга

Данная книга предназначена для веб-разработчиков, желающих понять, когда и зачем стоит использовать WebAssembly. Если вы действительно хотите понять WebAssembly, вам необходимо изучить его подробно. На сегодняшний день существует множество книг о различных наборах инструментальных средств WebAssembly. Однако данная книга создана не для того, чтобы научить вас писать коды на C/C++, Rust или другом языке. Вместо этого она расскажет вам о механизмах и возможностях WebAssembly.

Также данная книга предназначена для пользователей, которые хотят понять, что такое WebAssembly, на что он способен и как применять его наиболее оптимальным образом. WebAssembly превосходит JavaScript по части производительности, объема загружаемых данных и потребления памяти. Однако разработка высокопроизводительных приложений WebAssembly представляет собой более сложный процесс, нежели просто написание приложения на C++/Rust или AssemblyScript и его дальнейшая компиляция в WebAssembly. Для создания приложения, в три раза превосходящего по скорости свой эквивалент в JavaScript, необходимо погрузиться в работу WebAssembly чуть глубже.

Читателю необходимы базовые знания о веб-технологиях, таких как JavaScript, HTML и CSS. При этом быть экспертом в какой-либо из них вовсе необязательно. Изучать WebAssembly в нынешнем воплощении, не имея фундаментальных знаний о работе сетей, будет крайне непросто. Безусловно, в рамках этой книги у меня не получится раскрыть все аспекты функционирования веб-страниц. Однако я предполагаю, что большинство читателей уже знакомы с данной темой.

Чем интересен WebAssembly для пользователей

На первом саммите WebAssembly Эшли Уильямс (Ashley Williams) (@ag_dubs) представила результаты проведенного ею опроса в Твиттере (Twitter). Первый вопрос для пользователей WebAssembly звучал как «Чем вас привлекает WebAssembly?». Результаты были следующие:

- возможность поддержки нескольких языков, 40,1 %;
- сокращение объема кода и более быстрое его исполнение, 36,8 %;
- изолированность (безопасность), 17,3 %.

Затем она попросила пользователей, выбравших первый вариант (возможность поддержки нескольких языков), рассказать, почему для них важен этот аспект. В результате ответы были следующими:

- недостаточность JavaScript для удовлетворения всех потребностей, 43,5 %;
- возможность повторного использования существующих библиотек, 40,8 %;
- наличие дистрибутива, 8,1 %.

Далее пользователей, выбравших первый вариант ответа (недостаточность JavaScript для удовлетворения всех потребностей), спросили, почему они так считают. Среди причин были:

- низкая или нестабильная производительность, 42 %;
- неудовлетворенность экосистемой, 17,4 %;
- нежелание или невозможность разобраться, 31,3 %.

Вы можете ознакомиться с докладом Эшли (Ashley) «Why the #wasmsummit Website Isn't Written in Wasm» на YouTube по адресу *https://www.youtube.com/watch?v=J5Rs9oG3FdI*.

Несмотря на то что описанный выше опрос не был частью научного исследования, он все же представляется достаточно информативным. Как минимум, по его результатам выяснилось, что более 55 % пользователей заинтересованы в повышении производительности своих приложений посредством WebAssembly. И несомненно, повышение производительности кода с помощью WebAssembly более чем возможно. В свою очередь, реализация WebAssembly – это не волшебство; для этого просто необходимо знать, что и зачем вы дела-

ете. После прочтения данной книги вы будете знать о WebAssembly достаточно, чтобы значительно повысить производительность ваших веб-приложений.

Почему миру необходим WebAssembly

Я занимаюсь разработкой веб-приложений с середины 1990-х гг. Изначально веб-страницы были не более чем документами с изображениями. Ситуация изменилась с появлением Java и JavaScript. В то время JavaScript был «языком-игрушкой», который позволял добавлять эффекты при наведении курсора на кнопки на веб-страницах. Большинство считало Java весьма стоящим продуктом, а виртуальную машину Java (JVM – Java virtual machine) новой захватывающей технологией. Однако Java так и не смогла реализовать весь свой потенциал на веб-платформе. Дело в том, что Java требовала использования встраиваемых расширений, которые в конечном итоге вышли из моды, поскольку очень часто они были источником вредоносных ПО и не соответствовали требованиям безопасности.

К сожалению, Java является проприетарной технологией, что не позволяет интегрировать поддержку этого языка напрямую в веб-браузер. WebAssembly же отличается своим открытым форматом. Данная технология является продуктом сотрудничества между многими поставщиками аппаратного и программного обеспечения, такими как Google, Mozilla, Microsoft и Apple. Она не требует встраиваемых расширений и может быть успешно интегрирована во все современные веб-браузеры. Объединив использование данной технологии с Node.js, вы можете создавать аппаратно-независимое ПО. Ввиду открытого формата технология может быть использована без каких-либо лицензионных отчислений или разрешения на любой аппаратной либо программной платформе. Так что WebAssembly является воплощением мечты 1990-х – один двоичный код на все случаи жизни (one binary to rule them all).

Структура книги

В этой книге мы расскажем о работе WebAssembly на низком уровне, познакомив вас с текстовым форматом WebAssembly Text. Мы рассмотрим множество вопросов низкоуровневой реализации, а также покажем, как WebAssembly взаимодействует с JavaScript в Node.js и веб-приложениях. Данную книгу следует читать последовательно, поскольку описываемые концепции опираются друг на друга. Также здесь содержатся ссылки на примеры кода, которые можно найти на *https://wasmbook.com*.

Глава 1 «Введение в WebAssembly»

В этой главе мы подробно рассмотрим, что представляет собой технология WebAssembly, а также поговорим о наиболее оптимальных

вариантах ее применения. Вы познакомитесь с WebAssembly Text (WAT), который позволяет понять, как работает WebAssembly на самом низком уровне. Также стоит отметить, что мы создали специальную среду, включающую все примеры из данной книги.

Глава 2 «Основы работы с WebAssembly Text»

Здесь мы рассмотрим основы WAT и его взаимосвязь с высокоуровневыми языками, которые могут быть развернуты в WebAssembly. Также вы напишете свою первую программу на WAT, а затем мы обсудим такую фундаментальную концепцию, как контроль потока с помощью переменных.

Глава 3 «Функции и таблицы»

В этой главе мы рассмотрим создание функций в модулях WebAssembly и их вызов в JavaScript. Затем вы напишете программу для проверки простых чисел, что послужит отличной иллюстрацией для описанных концепций. После мы исследуем вызов функций из таблиц и влияние данного процесса на производительность.

Глава 4 «Низкоуровневые битовые операции»

Здесь вы узнаете о низкоуровневых концепциях, которые можно использовать для повышения производительности модулей WebAssembly. Среди них: системы счисления, побитовое маскирование и дополнительный код.

Глава 5 «Строки в WebAssembly»

В WebAssembly нет встроенной поддержки строкового типа данных. В этой главе вы узнаете, как представлены строки в WebAssembly и как ими управлять.

Глава 6 «Линейная память»

В этой главе вы познакомитесь с линейной памятью и тем, как модули WebAssembly используют ее для обмена большими наборами данных в JavaScript или альтернативной среде. Здесь вы начнете создание программы обнаружения столкновений, которая заставляет объекты перемещаться случайным образом и проверяет наличие столкновений между ними. Мы будем обращаться к данной программе на протяжении всей книги.

Глава 7 «Веб-приложения»

Здесь вы узнаете, как создаются простые веб-приложения с помощью HTML, CSS, JavaScript и WebAssembly.

Глава 8 «Работа с canvas»

Здесь мы обсудим, как использовать элемент HTML canvas в WebAssembly для создания быстрой веб-анимации. Также мы научимся применять canvas для улучшения созданной ранее программы обнаружения столкновений.

Глава 9 «Оптимизация производительности»

В этой главе вы узнаете, почему WebAssembly хорошо подходит для выполнения ресурсоемких задач, таких как обнаружение столкновений. Также вы научитесь использовать профилировщики Chrome и Firefox и другие инструменты оптимизации для повышения производительности приложений.

Глава 10 «Отладка WebAssembly»

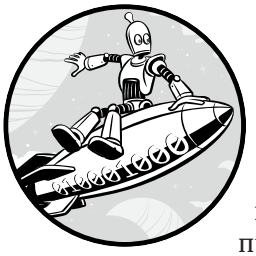
Здесь мы рассмотрим основы отладки, такие как вывод данных в веб-консоль посредством уведомлений и трассировки стека. Также вы научитесь использовать отладчики в Chrome и Firefox для пошагового исполнения кода WebAssembly.

Глава 11 «AssemblyScript»

В этой главе мы обсудим использование WAT с точки зрения понимания высокоуровневых языков и оценки AssemblyScript – высокоуровнего языка, разработанного для эффективного развертывания в WebAssembly.

1

ВВЕДЕНИЕ В WEBASSEMBLY



В этой главе вы получите базовые знания о технологии WebAssembly и изучите инструменты, которые понадобятся вам для начала работы с ней и ее текстовым воплощением – WebAssembly Text (WAT). Мы обсудим преимущества WebAssembly, такие как повышение производительности, возможность интеграции уже существующих библиотек, портируемость на другие платформы, безопасность и использование WebAssembly в качестве альтернативы JavaScript. Мы рассмотрим взаимосвязь JavaScript и WebAssembly, а также то, чем же на самом деле является WebAssembly. Вы научитесь синтаксису встроенных WAT- и S-выражений (символьные выражения). Мы познакомимся с концепциями среды встраивания и обсудим встраивание WebAssembly в веб-браузеры, Node.js и системный интерфейс WebAssembly (WASI – WebAssembly System Interface).

Затем мы рассмотрим преимущества использования Visual Studio Code в качестве среды разработки для WAT. Вы познакомитесь с основами Node.js и научитесь встраивать в данную платформу Web-

Assembly. Мы покажем, как использовать пакетный менеджер npm (Node Package Manager) для установки инструмента wat-wasm, с помощью которого вы сможете создавать приложения WebAssembly на основе WAT. Кроме того, вы создадите свое первое приложение WebAssembly, встроенное в среду Node.js.

Что такое WebAssembly?

WebAssembly – это технология, которая в перспективе следующих нескольких лет позволит значительно повысить производительность веб-приложений. Поскольку WebAssembly является молодой технологией и требует некоторых пояснений, многие люди используют ее не совсем правильно. Данная книга расскажет вам, что представляет собой технология WebAssembly и как использовать ее для создания высокопроизводительных веб-приложений.

WebAssembly – это *виртуальная архитектура набора команд* (*ISA – Instruction Set Architecture*) для стековой машины. Как правило, ISA представляется в двоичном формате и предназначается для конкретной машины. В свою очередь, WebAssembly разработан для работы на *виртуальной* машине, то есть не предназначен для физического аппаратного обеспечения. Виртуальная машина позволяет WebAssembly работать на разнообразном аппаратном обеспечении компьютеров и цифровых устройствах. ISA для WebAssembly предполагает компактность, межплатформенность и безопасность, а также наличие небольших двоичных файлов, что позволяет сократить время загрузки при развертывании в качестве части веб-приложения. Байт-код легко перенести на самые разные аппаратные платформы, а также он может быть безопасно развернут в интернете.

Большинство современных браузеров уже поддерживают WebAssembly. По данным Mozilla Foundation, код WebAssembly работает на 10–800 % быстрее, чем эквивалентный код JavaScript. К примеру, исполнение одного проекта eBay с помощью WebAssembly было в 50 раз быстрее, чем с JavaScript. В последующих главах мы расскажем, как создать программу обнаружения столкновений, которая может использоваться для измерения производительности. При запуске проведенные тесты производительности показали, что код обнаружения столкновений WebAssembly работает более чем в четыре раза быстрее того же JavaScript в Chrome и более чем в два раза быстрее, чем JavaScript в Firefox.

WebAssembly предлагает наиболее значительное повышение производительности в сети с момента появления динамических JIT-компиляторов (JIT – just-in-time) JavaScript. Современные браузерные движки JavaScript могут загружать и анализировать двоичный формат WebAssembly на порядок быстрее, чем JavaScript. Дело в том, что WebAssembly является генератором двоичного целевого кода для среды выполнения, а не языком программирования, как JavaScript, что

позволяет разработчикам выбирать языки программирования, которые лучше всего соответствуют потребностям его приложения. Сегодня многие говорят, что «JavaScript – это ассемблерный язык для сети», но на самом деле формат JavaScript является не лучшей целевой платформой для компиляции. JavaScript значительно менее эффективен, чем двоичный формат WebAssembly. Также любой код целевой платформы JavaScript должен уметь справляться со всеми особенностями языка JavaScript.

WebAssembly позволяет значительно повысить производительность веб-приложений в двух аспектах. Один из них – скорость запуска. В настоящее время наиболее компактным форматом JavaScript является минифицированный JavaScript, который позволяет уменьшить размеры загружаемых приложений, но при этом должен анализировать, интерпретировать, осуществлять JIT-компиляцию и оптимизировать код JavaScript. Для более компактного двоичного файла WebAssembly перечисленные выше действия не требуются. WebAssembly по-прежнему нуждается в парсинге, однако этот процесс происходит гораздо быстрее, ввиду того что байт-код анализировать гораздо проще, чем текстовый формат. Также WebAssembly требуется оптимизация веб-движками, однако и этот процесс происходит намного быстрее ввиду большей четкости языка.

Более того, WebAssembly предлагает значительное улучшение пропускной способности. WebAssembly упрощает оптимизацию движка браузера. JavaScript – это очень динамичный и гибкий язык программирования, который полезен для разработчика JavaScript, но абсолютно не подходит для оптимизации кода. WebAssembly не имеет никаких веб-предпочтений (несмотря на свое название) и может использоваться вне браузера.

В скором времени WebAssembly сможет делать все, что умеет JavaScript. К сожалению, текущая версия – приложение с минимальным функционалом (MVP – Minimum Viable Product) версии 1.0 – на это неспособна. MVP-версия WebAssembly подходит для выполнения конкретных задач, но она не предназначена для замены JavaScript или фреймворков, таких как Angular, React или Vue. Если вы хотите работать с WebAssembly прямо сейчас, у вас должен быть конкретный проект с интенсивными вычислениями, который требует очень высокой производительности. Онлайн-игры, WebVR, 3D-математика и криптовалюта являются теми областями, где может эффективно применяться WebAssembly.

Причины использовать WebAssembly

Прежде чем перейти к детальному изучению WebAssembly, давайте рассмотрим несколько причин, по которым вам может быть интересно его использовать. Вы поймете, что такое WebAssembly, а также для чего и как его использовать.

Повышение производительности

JavaScript требует от разработчиков программного обеспечения делать выбор, который повлияет на то, как они проектируют движок JavaScript. Например, вы можете оптимизировать движок JavaScript для достижения максимальной производительности с помощью оптимизирующего компилятора JIT, который может выполнять код быстрее, но требует больше времени на запуск. В качестве альтернативы вы можете использовать интерпретатор, который сразу же запускает выполнение кода, но не достигает максимальной производительности оптимизирующего компилятора JIT. Решение, которое большинство разработчиков движков JavaScript используют в своих веб-браузерах, заключается в реализации обоих подходов, но для этого требуется гораздо больший объем памяти. Каждое ваше решение – это компромисс.

WebAssembly позволяет ускорить запуск и повысить пиковую производительность без чрезмерного увеличения объема памяти. Но, к сожалению, вы все же не сможете просто переписать свой JavaScript на AssemblyScript, Rust или C++ и ожидать, что это произойдет без каких-либо дополнительных действий. WebAssembly – это не волшебство, поэтому простой перенос программы JavaScript на другой язык и его компиляция без понимания того, что делает WebAssembly на более низком уровне, могут привести к разочаровывающим результатам. Написание кода C++ и его компиляция в WebAssembly с помощью флагов оптимизации обычно занимают меньше времени, чем JavaScript. Иногда программисты жалуются на то, что они потратили весь день на переписывание своего приложения на C++, а оно работает лишь на 10 % быстрее. Если это так, то, вероятно, этим приложениям не очень поможет перенос в WebAssembly, и их код на C++ пусть и дальше компилируется в JavaScript. Выделите некоторое количество времени, чтобы изучить WebAssembly, а не C++, и заставьте свои веб-приложения работать с молниеносной скоростью.

Интеграция существующих библиотек

Две популярные библиотеки для переноса существующих библиотек на WebAssembly – это wasm-pack для Rust и Emscripten для C/C++. Использование WebAssembly идеально подходит тогда, когда у вас есть существующий код, написанный на C/C++ или Rust, который вы хотите сделать доступным для веб-приложений, или же хотите перенести любые существующие настольные приложения, чтобы сделать их доступными в сети. Набор инструментальных средств Emscripten особенно эффективен при переносе существующих настольных приложений C++ в сеть с помощью WebAssembly. Если вы следуете этому пути, то, вероятно, захотите, чтобы ваше приложение после переноса работало как можно ближе к исходной скорости вашего существующего приложения, что может быть осуществимо, только если прило-

жение не потребляет много ресурсов. Однако у вас также может быть приложение, для которого придется повозиться с оптимизацией быстродействия, чтобы оно работало так, как на компьютере. К концу этой книги вы сможете оценить модуль WebAssembly, созданный вашим набором инструментальных средств на основе существующего кода.

Портируемость на другие платформы и безопасность

Мы объединили их в один раздел, потому что они часто зависят друг от друга. WebAssembly изначально был лишь технологией для запуска в браузере, а затем стал быстро расширяться, превращаясь в изолированную среду, которую можно запускать где угодно. Рабочая группа WebAssembly создает высокозащищенную среду выполнения, которая не позволяет злоумышленникам скомпрометировать ваш код – от серверного кода WASI до WebAssembly для встраиваемых систем и интернета вещей (IoT). Я рекомендую послушать превосходный доклад Лин Кларк (Lin Clark) о безопасности WebAssembly и повторном использовании пакетов на первом саммите WebAssembly (*https://www.youtube.com/watch?v=lBZFJzGnBoU/*).

Несмотря на то что рабочая группа WebAssembly внимательно относится к безопасности, ни одна система не является полностью безопасной. Изучение WebAssembly на низком уровне подготовит вас к любым будущим угрозам безопасности.

Противники JavaScript

Некоторым людям просто не нравится JavaScript, и они не хотят, чтобы он был доминирующим языком веб-программирования. К сожалению, WebAssembly все же не в состоянии свергнуть JavaScript. Сегодня JavaScript и WebAssembly должны мирно сосуществовать и хорошо работать вместе, как показано на рис. 1.1.

Но в мире есть и хорошие новости для противников JavaScript: наборы инструментов WebAssembly предлагают множество вариантов написания веб-приложений без использования JavaScript. Например, Emscripten позволяет писать веб-приложения на C/C++ с очень малым количеством кода JavaScript, если таковой потребуется, конечно. Вы также можете писать целые веб-приложения, используя Rust и wasm-pack. Эти наборы инструментов не только генерируют WebAssembly, но также создают обширный связующий код JavaScript для вашего приложения. Причина в том, что в настоящее время возможности WebAssembly ограничены, а наборы инструментов заполняют эти пробелы с помощью кода JavaScript. Прелесть полноценных наборов инструментов, таких как Emscripten, в том, что они делают все за вас. Если вы разрабатываете с помощью одного из этих наборов инструментов, полезно понимать, когда ваш код превратится в WebAssembly, а когда – в JavaScript. Эта книга поможет вам узнать, когда это произойдет.

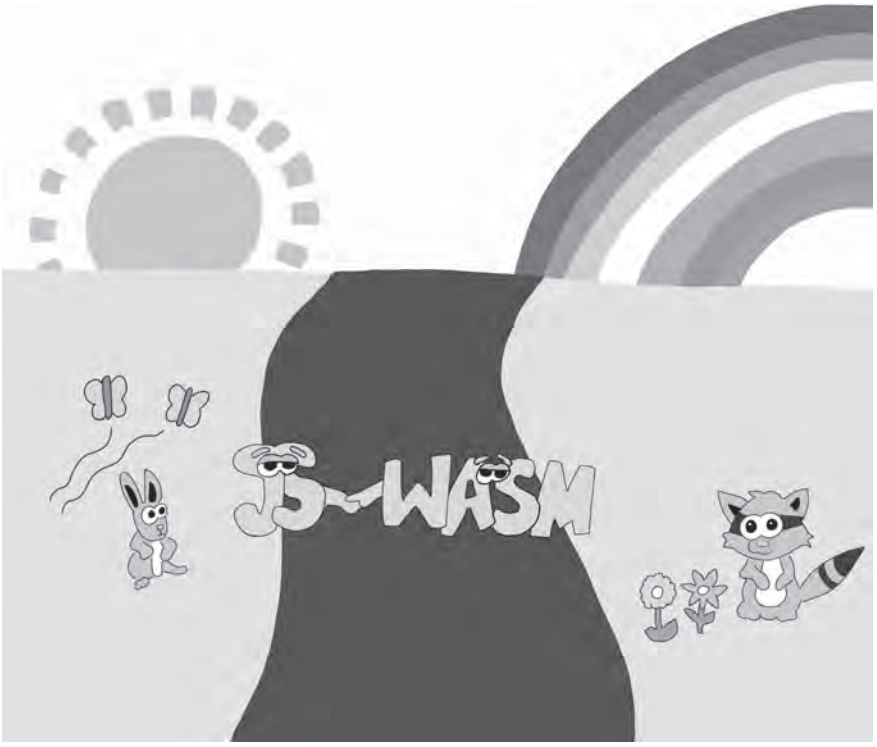


Рис. 1.1. JavaScript и WebAssembly могут сосуществовать в гармонии

Связь WebAssembly с JavaScript

Важно прояснить, как WebAssembly используется и соотносится с JavaScript. WebAssembly не является прямой заменой JavaScript; скорее WebAssembly:

- быстрее загружается, компилируется и выполняется;
- позволяет писать приложения для сети на языках, отличных от JavaScript;
- при правильном использовании может обеспечивать скорость, близкую к исходной для вашего приложения;
- работает с JavaScript для повышения производительности ваших веб-приложений, когда используется надлежащим образом;
- не является ассемблерным языком, хотя с ним связан псевдоассемблерный язык (WAT);
- предназначен не только для сети и также может выполняться небраузерными движками Javascript, такими как Node.js, или может выполнять байт-код в средах выполнения, реализующих WASI;
- пока что не универсальное решение для создания веб-приложений.

WebAssembly – это результат сотрудничества всех основных производителей браузеров над созданием новой платформы для распространения приложений через интернет. Язык JavaScript эволюционировал от потребностей веб-браузеров в конце 1990-х гг. до полноценного языка сценариев, которым он является сегодня. Хотя JavaScript стал достаточно быстро работающим языком, веб-разработчики заметили, что иногда он показывает плохие результаты. WebAssembly – это решение многих проблем с производительностью JavaScript.

Несмотря на то что WebAssembly не может делать абсолютно все, что умеет JavaScript, он может выполнять определенные операции намного быстрее, чем JavaScript, при этом потребляя меньше памяти. В этой книге мы сравниваем код JavaScript с WebAssembly. Мы будем многократно тестировать и профилировать код для сравнения. К концу книги вы будете уметь определять, когда вам следует использовать WebAssembly, а когда имеет смысл продолжать пользоваться JavaScript.

Зачем учить WAT?

Многие книги и руководства по WebAssembly сосредоточены на конкретных наборах инструментов, таких как вышеупомянутый wasm-pack для Rust или Emscripten для C/C++. Наборы инструментов для других языков, таких как AssemblyScript (разновидность TypeScript) и Go, в настоящее время находятся в разработке. Эти наборы инструментов – основная причина, по которой программисты обращаются к WebAssembly, и постоянно становятся доступными все новые языковые наборы инструментов WebAssembly. В будущем веб-разработчики смогут выбирать язык, на котором они будут разрабатывать, исходя из потребностей проекта, а не доступности языков.

Одним из важных факторов для любого из этих языков является понимание того, что делает WebAssembly на самом низком уровне. Глубокое понимание WAT даст вам ответ на вопрос, почему код может работать не так быстро, как бы вам хотелось. Также поможет понять, как WebAssembly взаимодействует со своей средой встраивания. Написание модуля в WAT – лучший способ работать в веб-браузере как можно ближе к «железу» (на низком уровне). Знание WAT поможет вам создавать максимально производительные веб-приложения, разбирать и оценивать любое веб-приложение, написанное для платформы WebAssembly. Вы сможете оценить любые потенциальные риски безопасности в будущем. Кроме того, он позволяет писать код, который максимально приближен к скорости обычного приложения, но без написания специфичного кода для программно-аппаратной платформы.

Так что же такое WAT? WAT похож на ассемблерный язык для виртуальной машины WebAssembly. В практическом смысле это означает, что при написании программ WebAssembly на таком языке, как Rust

или C++, используется набор инструментов, который, как упоминалось ранее, компилирует двоичный файл WebAssembly, а также связующий код JavaScript и HTML, в который встроен модуль WebAssembly. Файл WebAssembly очень похож на машинный код, поскольку он включает разделы, коды операций и данные, которые хранятся в виде последовательности двоичных чисел. Когда у вас есть исполняемый файл в машинном коде, вы можете деассемблировать этот файл в машинный ассемблерный язык, который находится на самом низком уровне языков программирования. Ассемблирование заменяет числовые коды операций в двоичном формате мнемоническими кодами, которые читабельны для человека. WAT действует как ассемблерный язык для WebAssembly.

Стили кодирования WAT

Есть два основных стиля кодирования WAT. Первый – это *стиль линейного набора команд*. Такой стиль кодирования требует, чтобы разработчик параллельно отслеживал элементы в стеке. Большинство команд WAT проталкивают элементы в стек, выталкивают их из стека или и то, и другое. Если вы решите писать в стиле линейного набора команд, то должны знать, что существует неявный стек, в котором параметры ваших команд должны быть помещены перед вызовом самих команд. Другой стиль кодирования называется *S-выражениями*. S-выражения представляют собой древовидную структуру кодирования, в которой параметры передаются в дерево способом, который похож на вызовы функций в JavaScript. В случае если у вас есть проблемы с визуализацией стека и его элементов, синтаксис S-выражения, скорее всего, подойдет больше. Вы также можете комбинировать два данных стиля в зависимости от неявного стека для менее сложных команд, а когда становится сложно отслеживать количество параметров – использовать S-выражение.

Применение стиля линейного набора команд

Рассмотрим простую функцию сложения в листинге 1.1, которая написана на языке JavaScript.

Листинг 1.1. Код JavaScript, выполняющий сложение переменных `a_val` и `b_val`

```
function main() {
    let a_val = 1;
    let b_val = 2;
    let c_val = a_val + b_val;
}
```

После выполнения этих строк значение переменной `c_val` теперь равно 3, что является результатом сложения `a_val` и `b_val`. Чтобы

выполнить ту же задачу в WAT, вам понадобится больше строк кода. В листинге 1.2 показана та же программа, использующая WAT.

Листинг 1.2. WebAssembly прибавляет `$a_val` к `$b_val`

```
(module
 ❶ (global $a_val (mut i32) (i32.const 1))
 ❷ (global $b_val (mut i32) (i32.const 2))
   (global $c_val (mut i32) (i32.const 0))
   (func $main (export "main")
       global.get $a_val
       global.get $b_val

       i32.add
       global.set $c_val
   )
)
```

Листинг 1.2 содержит больше строк кода, потому что WAT должен быть более развернутым, чем JavaScript. Пока код не будет запущен, JavaScript понятия не имеет, являются ли типы в предыдущих двух примерах данными в форме с плавающей запятой, целыми числами, строками или комбинацией. WebAssembly заранее компилируется в байт-код, и при компиляции необходимо знать, какие типы он использует. JavaScript должен провести синтаксический анализ и выделить лексемы, прежде чем JIT-компилятор сможет преобразовать его в байт-код. Как только оптимизирующий компилятор начинает работать с этим байт-кодом, компилятор должен следить за тем, являются ли переменные последовательными целыми числами. Если это так, JIT может создать байт-код, который делает это предположение.

Однако у JavaScript нет уверенности в том, что в конечном итоге он не получит строковые данные или данные с плавающей запятой, хотя ожидал целые числа; поэтому в любой момент он должен быть готов сбросить свой оптимизированный код и начать все сначала. Код WAT может быть и сложнее написать и понять, но веб-браузеру его намного проще запустить. WebAssembly передает большую часть работы из браузера в компилятор средства разработки или разработчику. Отсутствие необходимости выполнять столько работы делает браузеры счастливыми, и приложения работают быстрее.

Стековые машины

Как упоминалось ранее, WebAssembly – это виртуальная стековая машина. Давайте разберемся, что это значит. Представьте стопку тарелок. Каждая тарелка в этой метафоре – фрагмент данных. Когда вы добавляете тарелку в стопку, вы кладете ее поверх уже имеющихся тарелок. Вы убираете тарелку из стопки сверху, а не снизу. По этой причине последняя тарелка, которую вы кладете в стопку, становится первой, которую вы уберете. В информатике это называется «послед-

ним пришел – первым вышел» (LIFO – last-in, first-out). Добавление данных в стек называется *проталкиванием*, а извлечение данных из стека – *выталкиванием*. Когда вы используете стековую машину, почти все команды взаимодействуют со стеком, либо добавляя дополнительные данные в верхнюю часть стека с помощью проталкивания, либо удаляя данные сверху с помощью выталкивания. На рис. 1.2 показано взаимодействие со стеком.

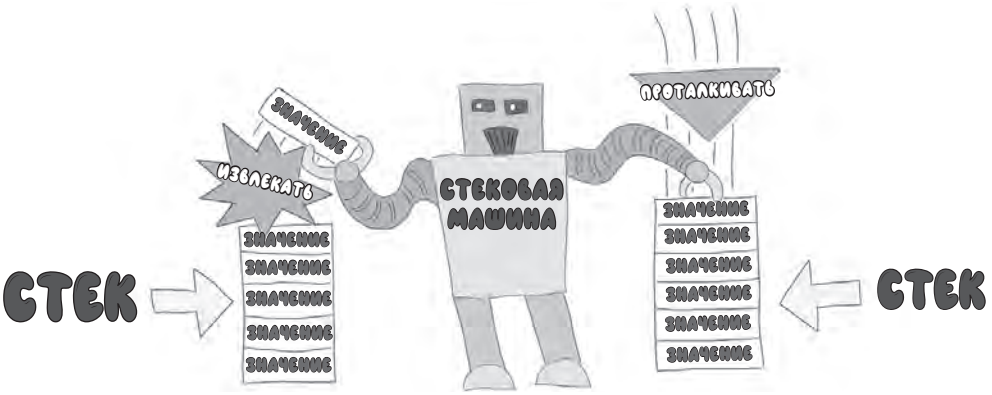


Рис. 1.2. Стековая машина выталкивает и проталкивает значения в стек

Как вы могли увидеть, первые две строки функции `$main` в листинге 1.2 проталкивают `$a_val` в верхнюю часть стека ❶, а затем проталкивают `$b_val` в верхнюю часть стека ❷. В результате получается стек с двумя значениями на нем. Нижняя часть стека имеет значение `$a_val`, потому что мы добавили ее первой, а верхняя часть имеет значение `$b_val`, потому что она была добавлена последней.

Важно проводить различие между ISA для стековой машины (WebAssembly) и ISA для регистровой машины (x86, ARM, MIPS, PowerPC или любой другой популярной аппаратной архитектурой за последние 30 лет). Регистровые машины должны перемещать данные из памяти в регистры ЦП, чтобы выполнять над ними математические операции. WebAssembly – это виртуальная стековая машина, которая должна работать на регистровых машинах. Когда мы будем писать код в формате WAT, вы увидите это взаимодействие подробнее.

Для выполнения вычислений стековые машины проталкивают в стек и выталкивают данные из него. Аппаратные стековые машины – редкая разновидность компьютеров. Виртуальные стековые машины, такие как WebAssembly, более распространены (например, JVM Java, AVM2 Adobe Flash player, EVM Ethereum и интерпретатор байт-кода CPython). Преимущество машин виртуального стека заключается в том, что они создают байт-код меньшего размера, что удобно для любого байт-кода, предназначенного для загрузки или потоковой передачи через интернет.

Стековые машины не делают никаких предположений о количестве регистров общего назначения, доступных среде встраивания.

Это позволяет оборудованию выбирать, какие регистры использовать и когда. Код WAT может немного сбить с толку, если вы не знаете, как работает стековая машина, поэтому давайте еще раз взглянем на первые две строки функции `$main` с учетом стека (Листинг 1.3).

Листинг 1.3. Получение `$a_val` и `$b_val`, а затем их проталкивание в стек

```
global.get $a_val ;; проталкиваем $a_val в стек
global.get $b_val ;; проталкиваем $b_val в стек
```

Первая строка получает значение `$a_val`, которое мы определяем как глобальное значение, а вторая строка получает глобальную переменную `$b_val`. Оба элемента попадают в стек в ожидании обработки.

Функция `i32.add` берет из стека две 32-битные целочисленные переменные, складывает их вместе, а затем проталкивает результат обратно в верхнюю часть стека. Как только два значения окажутся в стеке, мы можем вызвать `i32.add`. Если вы запустите функцию, которая выталкивает из стека больше значений, чем доступно, инструменты, которые вы используете для преобразования вашего WAT в двоичный файл WebAssembly, не позволят этого и вызовут ошибку компилятора. Мы используем последнюю строку в функции `$main`, чтобы установить для переменной `$c_val` значение в стеке. Это значение является результатом вызова функции `i32.add`.

Применение стиля S-выражений

S-выражения – это стиль кодирования с вложенной древовидной структурой, используемый в языках программирования, таких как Lisp. В листинге 1.3 мы применлия стиль линейного набора команд для написания WAT. В этом стиле используется неявный стек для каждого вызываемого оператора вызова и выражения. Тем, у кого есть некоторый опыт работы с ассемблерным языком, этот метод может показаться удобным. Но если вы пришли к WebAssembly с языка высокого уровня, такого как JavaScript, синтаксис S-выражений для WAT, вероятно, покажется вам более по душе. S-выражения организуют ваши обращения к операторам и выражениям WAT во вложенной структуре. Линейный же стиль требует, чтобы вы мысленно проталкивали элементы в стек и выталкивали их при написании кода. S-выражения больше похожи на вызовы функций JavaScript, чем на линейный стиль.

В листинге 1.2 мы присваиваем `c_val` значение `a_val` + `b_val`, используя стек. Код в листинге 1.4 – это фрагмент кода в листинге 1.2, где мы складываем два значения:

Листинг 1.4. Добавление и установка `$c_val` в WebAssembly

```
❶ global.get $a_val ;; проталкиваем $a_val в стек
  global.get $b_val ;; проталкиваем $b_val в стек
```

```
❷ i32.add           ;; выталкиваем два значения и помещаем результат в стек
  global.set $c_val ;; выталкиваем значение из стека и получаем $c_val
```

Мы помещаем в стек две 32-битные целочисленные переменные, которые получили из глобальных переменных с помощью `global.get` ❶. Затем мы извлекли эти два значения из стека с помощью вызова `i32.add`. После сложения этих двух значений функция `i32.add` ❷ поместила полученное значение обратно в стек. Так работает стековая машина. Каждый инструмент либо проталкивает значение в стек, либо выталкивает значение, либо и то, и другое.

В листинге 1.5 показана та же функция с использованием альтернативного синтаксиса S-выражения.

Листинг 1.5. Модуль WebAssembly для сложения двух значений

```
(module
  (global $a_val (mut i32) (i32.const 1))
  (global $b_val (mut i32) (i32.const 2))
  (global $c_val (mut i32) (i32.const 0))
  (func $main (export "main")
 ❶ (global.set $c_val
     (i32.add (global.get $a_val) (global.get $b_val))
    )
  )
)
```

Пусть вас не сбивают с толку круглые скобки: они работают так же, как символы {} во многих языках для создания блоков кода. При написании WAT-функции мы заключаем ее в круглые скобки. Когда вы помещаете соответствующую круглую скобку под открывающую скобку с таким же отступом, это то же самое, если сделать отступ для символов { и } в JavaScript. Например, обратите внимание, что отступ перед вызовом (`global.set` ❶ на одном уровне со скобкой закрытия под ним.

Этот код больше похож на обычный язык программирования, чем тот, что в листинге 1.2, потому что он в явном виде передает параметры в функцию, а не проталкивает и выталкивает значения из стека. Этот код компилируется в тот же двоичный файл. При написании своего кода в стиле S-выражений вы по-прежнему проталкиваете элементы в стек и выталкиваете из него. Такой стиль написания WAT – всего лишь синтаксический «сахар» (синтаксический прием, упрощающий чтение кода). Когда вы научитесь деассемблировать файлы WebAssembly в WAT, вы обнаружите, что синтаксис S-выражений не поддерживается дизассемблерами, такими как wasm2wat.

Среда встраивания

Как упоминалось ранее, WebAssembly не взаимодействует напрямую с аппаратным оборудованием. Вы должны встроить двоич-

ный файл WebAssembly в среду хоста, которая управляет загрузкой и инициализацией модуля WebAssembly. В этой книге мы работаем с движками JavaScript, такими как Node.js, и веб-браузерами как средами для встраивания. Другие среды, например среда выполнения wasmtime, включают в себя WASI. Но даже несмотря на то, что мы все же обсудим WASI, мы не будем использовать его в этой книге, потому что он все еще очень новый и находится в стадии разработки. Реализовать стек-машину должна среда встраивания. Поскольку современное оборудование обычно представляет собой регистровую машину, среда встраивания управляет стеком с помощью аппаратных регистров.

Браузер

Скорее всего, ваш интерес к WebAssembly вызван стремлением к повышению производительности ваших веб-приложений. Все современные браузерные движки JavaScript реализуют WebAssembly. В настоящее время Chrome и Firefox имеют лучшие инструменты для отладки WebAssembly, поэтому мы предлагаем выбрать для разработки один из этих браузеров. Ваши приложения WAT также должны нормально работать в Microsoft Edge, а Internet Explorer больше не вводит новые функции, и поэтому, к сожалению, Internet Explorer никогда не будет поддерживать WebAssembly.

Когда вы пишете WAT для веб-браузера, очень важно понимать, какие части приложения вы можете писать в WAT, а какие лучше в JavaScript. Также может случиться, что повышение производительности, которое вы получаете с помощью WebAssembly, может не стоить траты дополнительного времени на разработку. Вы сможете принимать эти решения, если разбираетесь в WAT и WebAssembly. Когда вы работаете с WebAssembly, вам часто приходится во время разработки жертвовать производительностью или циклами процессора ради памяти, или наоборот. Оптимизация производительности должна быть превыше всего.

WASI

WASI – это спецификация среды выполнения для приложений WebAssembly и стандарт взаимодействия WebAssembly с операционной системой. Он позволяет WebAssembly использовать файловую систему, выполнять системные вызовы и обрабатывать входные и выходные данные. Mozilla Foundation создали среду выполнения WebAssembly под названием *wasmtime*, которая реализует стандарт WASI. С WASI WebAssembly может делать все, что и исходное приложение, но более безопасным и независимым от платформы способом. Он делает все это с тем же уровнем производительности, что и исходное приложение.

Node.js также может запускать экспериментальный предварительный просмотр WASI с помощью командной строки `--experi`

`mental-wasi-unstable-preview1`. Вы можете использовать его для запуска приложений WebAssembly, которые взаимодействуют с операционной системой вне веб-браузера. Windows, macOS, Linux или любая другая операционная система может реализовать среду выполнения WASI, поскольку она предназначена для обеспечения портируемости, безопасности и в конечном итоге универсальности WebAssembly.

Visual Studio Code

Visual Studio Code (VS Code) – это интегрированная среда разработки (IDE) с открытым исходным кодом, которую я использовал для написания примеров в этой книге. VS Code доступен для Windows, macOS и Linux по адресу *https://code.visualstudio.com/download*. Мы используем VS Code с расширением WebAssembly, написанным Дмитрием Цветцих (Dmitriy Tsvettsikh), которое доступно по адресу *https://marketplace.visualstudio.com/items?ItemName=dtsvet.vscode-wasm*. Расширение обеспечивает подсветку синтаксиса кода для формата WAT, а также дает несколько других полезных пунктов в меню. Например, если у вас есть файл WebAssembly, вы можете дизассемблировать его в WAT, кликнув на файл правой кнопкой мыши и выбрав пункт меню **Show WebAssembly**. Это очень полезно, если вы хотите посмотреть код WebAssembly, который был написан кем-то другим, или код, который был скомпилирован с использованием набора инструментов. Расширение также может компилировать ваши файлы WAT в двоичный файл WebAssembly. Вы можете щелкнуть правой кнопкой мыши файл *.wat* и выбрать **Save as WebAssembly binary file**. Появится окно запроса на сохранение файла, где можно указать имя файла, в который вы хотите сохранить новый файл WebAssembly.

На рис. 1.3 показан снимок экрана с данным расширением.

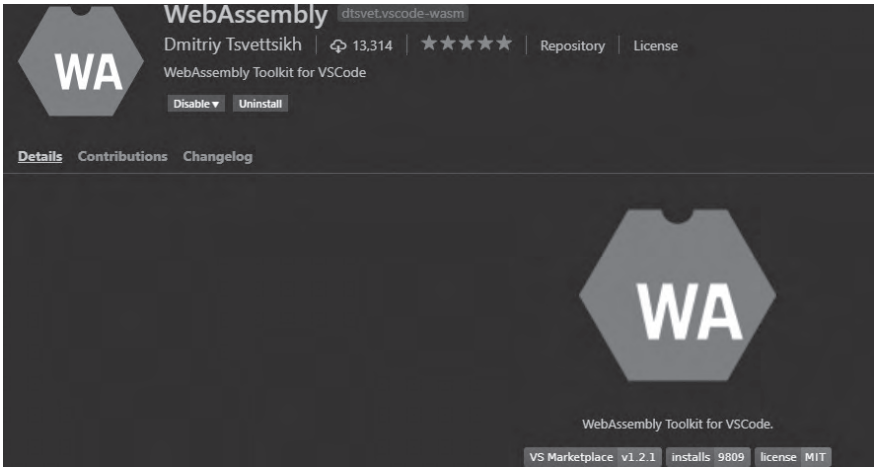


Рис. 1.3. Установка расширения WebAssembly для VS Code

Node.js

Node.js – отличный инструмент для тестирования производительности модулей WebAssembly по сравнению с существующими модулями JavaScript, а также это среда выполнения JavaScript, которую мы будем часто использовать в данной книге. Node.js поставляется с *npm* (*Node Package Manager*), который можно использовать для простой установки пакетов кода. WebAssembly – отличная альтернатива написанию исходного модуля в Node.js, который блокирует использование определенного аппаратного оборудования. Если вы хотите создать модуль npm для общего использования, написав его под WebAssembly, вы можете получить производительность исходного модуля в сочетании с портируемостью и безопасностью модуля JavaScript. Мы будем выполнять многие приложения, описанные в этой книге, с помощью Node.js.

Node.js – наш предпочтительный инструмент разработки для выполнения WebAssembly, будь то из JavaScript или через веб-сервер. Мы начнем с использования Node.js для выполнения модулей WebAssembly из JavaScript, а в главе 7 напишем простой веб-сервер для обслуживания веб-приложений WebAssembly.

Node.js поставляется с npm, что упрощает установку некоторых инструментов, которые используются для разработки WebAssembly. Далее мы покажем вам, как использовать npm для установки модуля *wat-wasm*, инструмента для компиляции, оптимизации и дизассемблирования WebAssembly. Также рассмотрим, как использовать Node.js для написания простого приложения WebAssembly. Многие читатели, возможно, уже знакомы с Node.js, но если нет, существует масса документации по Node.js, которую стоит прочитать.

Установка Node.js

У вас должен быть заранее установлен Node.js, чтобы выполнять примеры по мере прочтения. К счастью, сделать это несложно. Если вы используете Windows или macOS, установщики для обеих операционных систем доступны по адресу *https://nodejs.org/en/download/*.

Для Ubuntu Linux вы можете установить Node, используя следующую команду apt:

```
sudo apt install nodejs
```

После установки Node выполните следующую команду из командной строки (на любой платформе), чтобы убедиться, что все установлено должным образом:

```
node -v
```

Если все правильно установлено, вы увидите установленную версию Node.js. Когда мы запускаем команду `node -v` на нашей машине с Windows, она выдает следующий вывод:

```
v12.14.0
```

Это означает, что мы работаем с версией 12.14.0.

Примечание *Если у вас возникли какие-либо проблемы с выполнением кода из этой книги, вы можете установить конкретную версию Node.js со страницы* ***Предыдущие выпуски*** *по адресу https://nodejs.org/ru/download/Release/.*

Установка wat-wasm

На сегодняшний день доступно множество инструментов для преобразования кода WAT в двоичный файл WebAssembly. При написании этой книги я использовал многие из этих инструментов. В конце концов, я написал *wat-wasm* поверх *WABT.js* и *Binaryen.js*, чтобы уменьшить количество пакетов, необходимых для функций, которые я хотел продемонстрировать. Чтобы установить `wat-wasm`, выполните следующую команду `npm`:

```
npm install -g wat-wasm
```

Флаг `-g` устанавливает `wat-wasm` глобально. На протяжении всей книги мы будем использовать в окне терминала инструменты командной строки, такие как `wat2wasm`. Чтобы использовать инструменты и дальше, вам необходимо установить его глобально. После установки `wat-wasm` убедитесь, что вы можете запустить его через команду `wat2wasm` из командной строки:

```
wat2wasm
```

После этого вы должны увидеть вывод `wat-wasm` в вашу консоль. Также вы увидите множество флагов, о которых узнаете позже в этой книге.

Вы можете протестировать `wat2wasm`, создав простейший модуль WAT, как показано в листинге 1.6. Создайте новый файл с именем *file.wat* и введите в него следующий код:

Листинг 1.6. Простейший возможный модуль WebAssembly

```
(module)
```

После установки `wat-wasm` вы можете использовать команду из листинга 1.7 для компиляции файла *file.wat* в *file.wasm*, который является двоичным файлом WebAssembly:

Листинг 1.7. Трансляция файла file.wat с помощью wat2wasm

```
wat2wasm file.wat
```

В этой книге мы будем использовать Node.js для запуска приложений командной строки WebAssembly и обслуживания веб-приложений WebAssembly для открытия в браузере. В следующем разделе мы напишем первое приложение WebAssembly с помощью Node.js.

Наше первое приложение WebAssembly с помощью Node.js

Мы начнем книгу с использования Node.js в качестве среды встраивания вместо веб-браузера, чтобы исключить необходимость в HTML и CSS в примерах кода и сделать их проще. Позже, когда вы поймете основу, рассмотрим использование браузера в качестве среды встраивания.

Код WAT в наших приложениях Node.js будет работать так же, как и в браузере. Механизм WebAssembly внутри Node.js такой же, как и в Chrome, а часть приложения WebAssembly вообще не знает, в какой среде оно работает.

Давайте начнем с создания простого файла WAT и его компиляции с помощью wat2wasm. Создайте файл с именем *AddInt.wat* и добавьте к нему код WAT из листинга 1.8.

Листинг 1.8. Модуль WebAssembly с функцией, которая складывает два целых числа

AddInt.wat

```
(module
    (func (export "AddInt")
    (param $value_1 i32) (param $value_2 i32)
    (result i32)
        local.get $value_1
        local.get $value_2
        i32.add
    )
)
```

Сейчас вы должны понять этот код. Выделите время, чтобы внимательно просмотреть код и понять его логику. Это простой модуль WebAssembly с единственной функцией `AddInt`, которую мы экспортируем в среду встраивания. Теперь скомпилируйте *AddInt.wat* в *AddInt.wasm* с помощью `wat2wasm`, как показано в листинге 1.9.

Листинг 1.9. Компиляция AddInt.wat в AddInt.wasm

```
wat2wasm AddInt.wat
```

Теперь мы готовы написать фрагмент JavaScript нашего первого приложения Node.js.

Вызов модуля WebAssembly из Node.js

Мы можем вызвать модуль WebAssembly из Node.js с помощью JavaScript. Создайте файл с именем *AddInt.js* и добавьте код JavaScript, как в листинге 1.10.

Листинг 1.10. Вызов функции AddInt WebAssembly из асинхронного IIFE

AddInt.js
```
❶ const fs = require ('fs');
  const bytes = fs.readFileSync (__dirname+'/AddInt.wasm');
❷ const value_1 = parseInt (process.argv[2]);
  const value_2 = parseInt (process.argv[3]);
❸ (async () => {
❹ const obj = await WebAssembly.instantiate (
                                  new Uint8Array (bytes));
❺ let add_value = obj.instance.exports.AddInt( value_1,value_2 );
❻ console.log(`${value_1}+${value_2}=${add_value}`);
  })();
```

Node.js может читать файл WebAssembly прямо с жесткого диска, на котором запущено приложение, благодаря встроенному модулю `fs` ❶, который считывает файлы из локального хранилища. Мы загружаем этот модуль с помощью функции `require` в Node.js. Также используем его для чтения файла *AddInt.wasm* с помощью функции `readFileSync`. Мы также извлекаем два аргумента из командной строки, используя массив `process.argv` ❷. В массиве `argv` есть все аргументы, переданные из командной строки в Node.js. Запустим функцию из командной строки; `process.argv[0]` будет содержать командный узел, а `process.argv[1]` будет содержать имя файла JavaScript *AddInt.js*. Когда мы запускаем программу, то передаем в командную строку два числа, которые устанавливают `process.argv[2]` и `process.argv[3]`.

Для создания экземпляра модуля WebAssembly, вызова функции WebAssembly и вывода результатов на консоль мы используем асинхронное *немедленно вызываемое функциональное выражение* (*IIFE – immediately invoked function expression*). Для тех, кто незнаком с синтаксисом IIFE: это средство, с помощью которого JavaScript может дождаться обещания (promise), перед тем как выполнить остальную часть кода. Создание экземпляра модуля WebAssembly требует определенного количества времени, поэтому вы не захотите задерживать браузер или узел в ожидании завершения этого процесса. Синтаксис `(async () =>{})();` ❸ сообщает движку JavaScript: «объект-обещание в пути, так что можно сделать что-нибудь еще, пока ждешь». Внутри IIFE мы вызываем `WebAssembly.instantiate` ❹, передавая байты, полученные из файла WebAssembly ранее, с вызо-

вом `readFileSync`. После создания экземпляра модуля мы вызываем функцию `AddInt` ❺, экспортированную из кода WAT. Затем вызываем оператор `console.log` ❻ для вывода добавляемых значений и результата.

Теперь, когда у нас есть модуль WebAssembly и файл JavaScript, мы можем запустить приложение из командной строки, используя вызов Node.js, как показано в листинге 1.11.

Листинг 1.11. Запуск AddInt.js с использованием Node.js

```
node AddInt.js 7 9
```

Выполнение этой команды приводит к следующему результату:

```
7 + 9 = 16
```

Сложение двух целых чисел происходит в WebAssembly. Прежде чем мы продолжим, давайте быстро посмотрим, как использовать синтаксис `.then` в качестве альтернативы асинхронному IIFE.

Синтаксис .then

Другой широко используемый синтаксис ожидания возврата обещаний – это синтаксис `.then`. Предпочтительнее использовать синтаксис IIFE из листинга 1.10, но любой из этих синтаксисов вполне приемлем.

Создайте файл с именем *AddIntThen.js* и добавьте код из листинга 1.12, чтобы заменить асинхронный синтаксис IIFE в листинге 1.10 кодом `.then`.

Листинг 1.12. Использование синтаксиса `.then` для вызова функции WebAssembly

AddIntThen.js

```
const fs = require ('fs');
const bytes = fs.readFileSync (__dirname +'/AddInt.wasm');
const value_1 = parseInt(process.argv[2]);
const value_2 = parseInt (process.argv[3]);

❶WebAssembly.instantiate (new Uint8Array (bytes))
❷.then (obj => {
    let add_value = obj.instance.exports.AddInt(value_1, value_2);
    console.log(`${value_1} + ${value_2}=${add_value}`);
  });
```

Основное различие здесь заключается в функции `WebAssembly.instantiate` ❶, за которой следует `.then` ❷, и содержащей обратный вызов функции стрелки, который передает объект `obj`.

Удачное время

Сейчас самое время, чтобы изучать WAT. На момент написания этой книги текущая версия WebAssembly 1.0 имела относительно небольшой набор инструментов, в общей сложности 172 различных кода операций в двоичном файле WebAssembly, и вам нет необходимости запоминать их все. WebAssembly поддерживает четыре разных типа данных: `i32`, `i64`, `f32` и `f64`, и многие коды операций являются повторяющимися командами для каждого типа (например, `i32.add` и `i64.add`). Так, для устранения повторяющихся кодов операций вам нужно знать всего около 50 различных мнемоник, чтобы знать весь язык. Количество кодов операций, поддерживаемых WebAssembly, со временем будет увеличиваться. Изучение WebAssembly сейчас, когда он только зарождается, дает вам преимущество, ведь в будущем запоминание каждого кода операции станет крайне трудно или невозможно.

Как упоминалось ранее, написание ваших модулей в WAT – лучший способ работать в веб-браузере как можно ближе к «железу». То, как сегодня реализован JavaScript в браузере, может привести к провалам в производительности в зависимости от множества факторов. WebAssembly может устранить эти провалы, а WAT может помочь вам оптимизировать код, чтобы сделать его максимально быстрым.

Чтобы использовать набор инструментальных средств Emscripten, вам понадобится только базовое понимание платформы WebAssembly. Однако лишь базовых знаний вряд ли будет достаточно, чтобы повысить производительность вашего приложения до максимума, и как следствие вы подумаете, что WebAssembly не стоит потраченных усилий. В таком случае эта мысль будет ошибкой. Если вы хотите получить максимально возможную производительность от своего веб-приложения, то должны узнать как можно больше о WebAssembly. Так, необходимо знать, что он может и что лучше не стоит делать. Вы должны понимать, в каких случаях лучше выбрать WebAssembly, а когда JavaScript. Лучший способ получить эти знания – написать код WAT. В конце концов, вы можете и не писать свое приложение в WAT, но знание языка поможет вам понять WebAssembly и веб-браузер.