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

Как работает JS: особенности и сфера применения WebAssembly

ПЕРЕВОД

ru_vds 30 ноября 2017 в 14:21 11,7k

Оригинал: Alexander Zlatkov

Сегодня мы представляем вам шестую часть серии материалов, которые посвящены особенностям работы всего того, что связано с JavaScript. Здесь мы поговорим о WebAssembly. А именно, детально проанализируем эту технологию, рассмотрим особенности её работы, а так же то, как она соотносится с обычным JavaScript в плане производительности. Речь пойдёт о времени загрузки кода, о скорости выполнения программ, о сборке мусора, об использовании памяти, о доступе к API платформы, об отладке, о многопоточности и о переносимости WebAssembly-кода. Эта технология, хотя и находится сейчас в самом начале своего развития, уже начала менять взгляды на разработку веб-приложений. Если разработчику нужна высочайшая производительность браузерного кода, ему просто необходимо познакомиться с WebAssembly.

Что такое WebAssembly

WebAssembly (сокращённо — wasm) — это эффективный низкоуровневый байт-код для веб-приложений. Wasm даёт возможность разработки функционала веб-страниц на языках, отличных от JavaScript (например, это C, C++, Rust и другие). Код на этих языках компилируется (статически) в WebAssembly. В результате получается веб-приложение, которое быстро загружается и отличается очень высокой производительностью.

Время загрузки

Для того, чтобы запустить JavaScript-программу, браузеру сначала нужно загрузить все .js-файлы, которые хранятся и передаются по сети в виде обычного текста.

Wasm — это низкоуровневый язык, похожий на ассемблер.

WebAssembly-программы загружаются браузером быстрее, так как через интернет нужно передать уже скомпилированные файлы в весьма компактном бинарном формате.

Выполнение

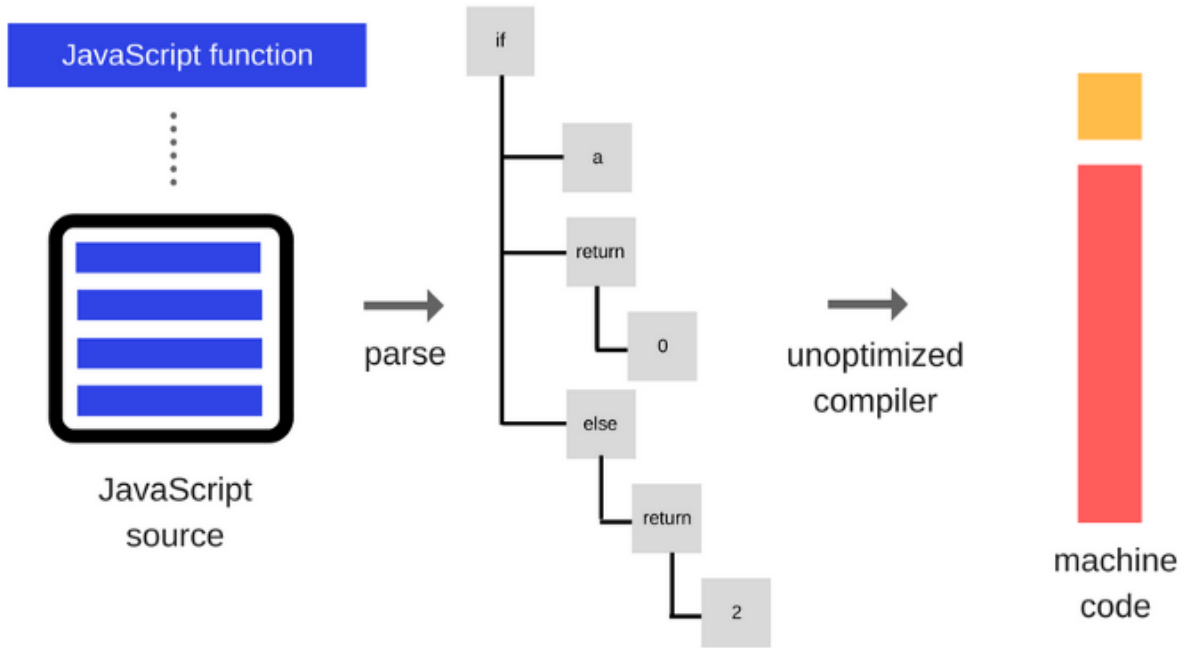
Сегодня wasm-программы выполняются лишь на 20% медленнее чем машинный код. Это, без сомнения, достойный результат. Ведь речь идёт о формате, который компилируется в особом окружении и запускается с применением множества ограничений, которые обеспечивают высокий уровень безопасности. Подобное замедление в сравнении с машинным кодом в этом свете выглядит не таким уж и большим. Кроме того, в будущем ожидается повышение производительности wasm-кода. Ещё интереснее то, что wasm платформенно-независим. Его поддержка имеется во всех ведущих браузерных движках, которые демонстрируют примерно одинаковую производительность при выполнении wasm-кода.

Для того, чтобы сопоставить особенности wasm и JavaScript, вы можете обратиться к этому материалу, в котором идёт речь об особенностях JS-движков на примере V8. А сейчас мы поговорим о том, как wasm-код соотносится с другими механизмами V8.

Wasm и JS-движок V8

Вот схема устройства движка V8, а именно, тот путь, который проходит программа на JavaScript от простого текстового файла до

исполняемого кода.

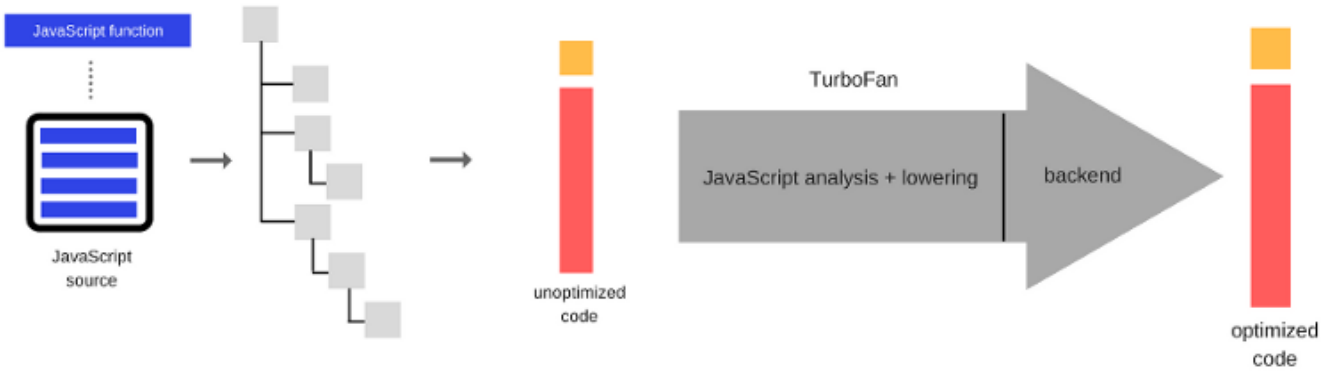


Динамическая компиляция в V8

Слева представлен исходный код на JavaScript, который содержит некую функцию. Сначала этот код подвергается парсингу, строки превращаются в токены и генерируется абстрактное синтаксическое дерево (Abstract Syntax Tree, AST). AST — это представление логики JS-программы. После создания AST V8 преобразует то, что получилось, в машинный код. Производится обход абстрактного синтаксического дерева и то, что раньше было функцией, существующей в виде текста, преобразуется в её скомпилированный вариант. При этом V8 не прилагает особых усилий для того, чтобы оптимизировать код.

Посмотрим теперь на то, что происходит на следующих стадиях

работы движка.



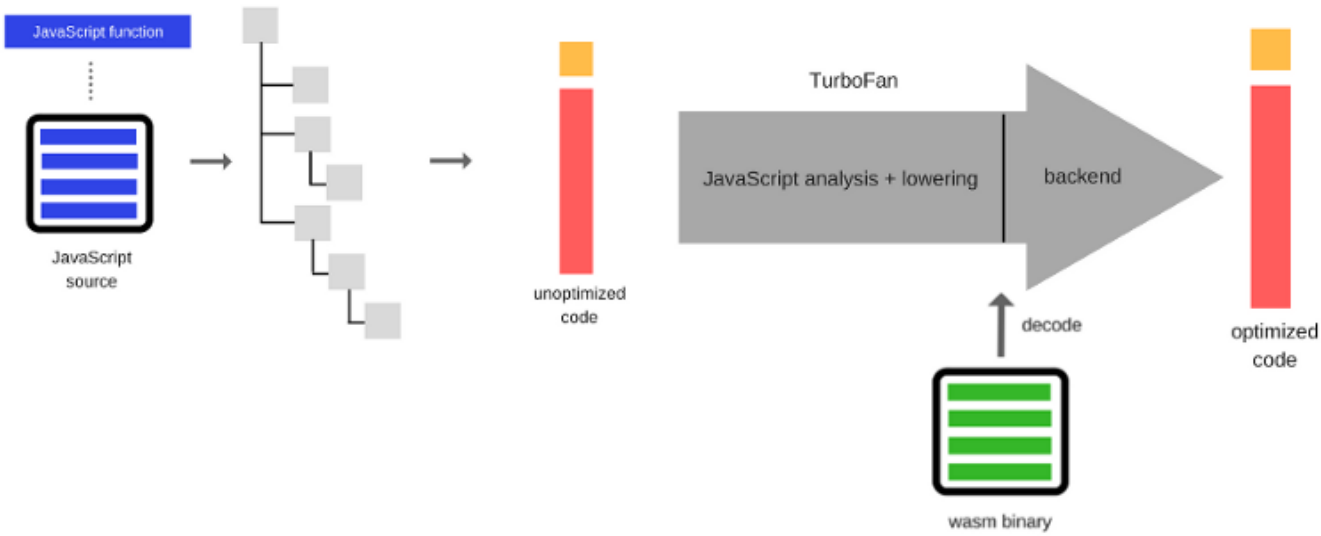
Конвейер движка V8

Здесь показано, что теперь приходит время TurboFan — одного из оптимизирующих компиляторов V8. Во время работы JavaScript-приложения в V8 производится множество вспомогательных действий. А именно, TurboFan наблюдает за происходящим в поисках узких мест и наиболее часто используемых фрагментов программы для того, чтобы оптимизировать соответствующие участки кода. После этого TurboFan обработает наиболее ответственные части программы, пропустив их через оптимизирующий JIT-компилятор, что приведёт к тому, что те функции, которые «съедают» большую часть времени процессора, станут выполняться гораздо быстрее.

Этот подход позволяет повысить производительность выполнения JavaScript, решает проблему скорости работы программ, но и тут не всё гладко. Дело в том, что операции по анализу кода и по

принятию решений о том, что нужно оптимизировать, а что не нужно, так же потребляют ресурсы. Это, в свою очередь, означает более высокое энергопотребление систем, что, в случае с мобильными устройствами, ведёт к сокращению срока их жизни от одной зарядки.

Если же включить в вышеприведённую схему wasm, то окажется, что этот код не нуждается в анализе и в нескольких проходах компиляции. Он уже оптимизирован и готов к использованию.

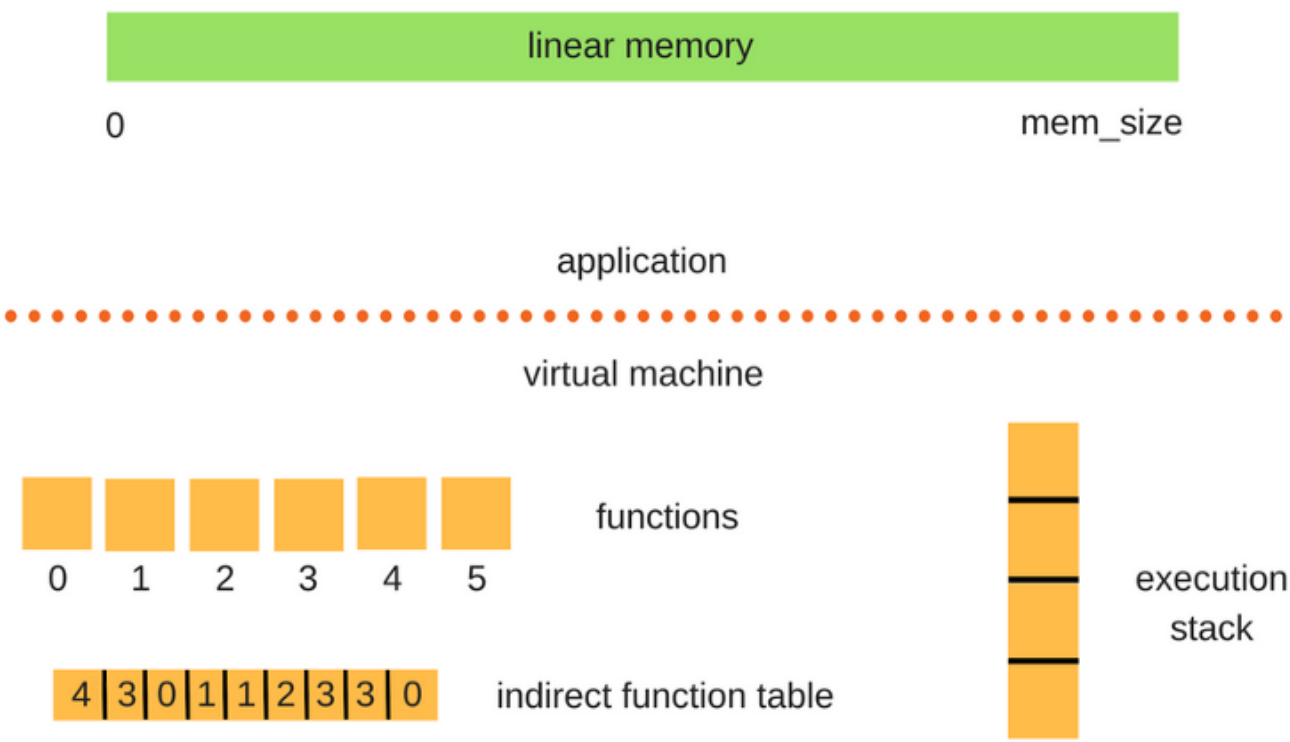


Wasm и конвейер V8

Wasm-код оптимизируется в ходе статической компиляции. При работе с ним не нужно разбирать текстовые файлы. Благодаря wasm в нашем распоряжении оказываются бинарные файлы, которые достаточно лишь преобразовать в машинный код. Все улучшения в этот код были внесены при компиляции, которая производится до того, как он попадает в браузер.
Всё это делает выполнение wasm гораздо более эффективным,

так как немало шагов по превращению текста программы в оптимизированный машинный код можно пропустить.

Модель памяти



Доверенная и недоверенная области памяти WebAssembly

Память программы, написанной, например, на C++ и скомпилированной в WebAssembly, представляет собой непрерывный блок, в котором нет «дыр». Одна из особенностей wasm, которая способствует повышению безопасности, заключается в отделении стека выполнения от линейного адресного пространства. В C++-программах есть куча, память под стек выделяется в её верхней части. При подобной организации работы с памятью можно, воспользовавшись указателем, получить доступ к памяти стека для того, чтобы воздействовать на

состояние переменных, взаимодействие с которыми на текущем этапе выполнения программы не предусмотрено. Именно эту возможность используют многие вредоносные программы.

WebAssembly пользуется совершенно другой моделью работы с памятью. Стек выполнения отделён от памяти, где хранится сама wasm-программа, в результате нет возможности получить несанкционированный доступ к этой памяти, и, например, изменить состояние каких-то переменных. Кроме того, функции используют не указатели, а целочисленные смещения. Здесь применяется механизм косвенной адресации. Необходимые прямые адреса вычисляются в процессе работы программы. Этот механизм построен так, что можно одновременно загрузить несколько wasm-модулей, адреса будут находиться с использованием смещений, в итоге всё будет работать как надо.

Подробнее о механизмах управления памятью в JavaScript можно почитать здесь.

Сборка мусора

В предыдущих материалах этой серии мы уже говорили о том, что в управлении памятью JS-программ участвует сборщик мусора (Garbage Collector, GC).

В случае с WebAssembly всё выглядит немного иначе. Эта технология поддерживают языки с ручным управлением памятью. В результате вместе с wasm-модулями можно использовать и собственный сборщик мусора, но это — непростая задача.

Сейчас WebAssembly ориентирован на способы работы с памятью, применяемые в C++ и Rust. Так как wasm — низкоуровневая технология, вполне логично то, что языки программирования, находящиеся лишь на одну ступень выше ассемблера, будут легко компилироваться в wasm. Так, при программировании на C можно использовать обычную команду `malloc`, в C++ можно применять интеллектуальные указатели. Rust задействует совершенно другой подход (не будем в это углубляться, так как там всё совершенно иначе). Эти языки не применяют сборщики мусора, в результате им не нужны сложные механизмы времени выполнения, которые отвечают за управление памятью. WebAssembly отлично вписывается в подобные модели работы с памятью.

Кроме того, эти языки не созданы для выполнения сложных операций, которые обычно реализуются с помощью JavaScript, таких, как манипуляции с DOM. Нет смысла писать HTML-приложение целиком на C++, так как C++ просто не рассчитан на такое применение. В большинстве случаев код для веб-приложений на C++ или Rust пишут для работы с WebGL или создают на нём высокооптимизированные библиотеки, например, такие, которые отвечают за математические вычисления.

В будущем, однако, ожидается поддержка языков, которые используют другие модели работы с памятью.

Доступ к внешним API

В зависимости от среды выполнения, JavaScript программа может

напрямую взаимодействовать со специализированными API. Например, если программа написана для браузера, в её распоряжении оказывается набор Web API, которые приложение использует для управления браузером или устройством, и для работы с DOM, CSSOM, WebGL, IndexedDB, Web Audio API и так далее.

У модулей WebAssembly нет прямого доступа к API, который предоставляет платформа. Модули могут взаимодействовать с API только при посредничестве JavaScript. Если, из wasm-модуля, нужно обратиться к подобному API, этот вызов нужно выполнять через JavaScript. Например, если нужно выполнить команду `console.log`, вызывать её придётся через JS. Подобные обращения к средствам JavaScript сказываются на производительности.

Нельзя говорить, что так будет всегда. В будущем ожидается появление соответствующих API для непосредственного использования их в wasm-коде. Как результат, wasm-приложения можно будет создавать, не используя обращения к JavaScript.

Карты кода (source maps)

Если после минификации JS-кода его нужно отлаживать, в дело вступают карты кода (source maps). Это — способ установления соответствий между JS-кодом, который минифицирован или скомбинирован из различных файлов, и его исходным состоянием. Когда проект собирают для продакшна, производят минификацию и комбинирование файлов, создают и карту кода, которая хранит

информацию об исходных файлах. При обращении к конкретному месту в сгенерированном коде можно, сверившись с картой кода, найти фрагмент исходной программы, который выглядит гораздо понятнее, нежели упакованный вариант программы.

WebAssembly не поддерживает карты кода (source maps), так как пока нет соответствующей спецификации, но, вполне возможно, что уже весьма скоро такая возможность появится.

В результате, при отладке wasm-кода, полученного, например, из C++, можно будет видеть исходный код, устанавливать в нём точки останова. По крайней мере, такова цель внедрения карт кода в wasm.

Многопоточность

JavaScript выполняется в одном потоке, хотя он поддерживает асинхронную модель программирования. Подробнее об этом можно почитать здесь. Кроме того, JS поддерживает технологию Web Workers, но у неё довольно специфическая область применения.

Преимущественно — вычисления, интенсивно использующие ресурсы процессора, которые могут заблокировать главный поток пользовательского интерфейса. Однако, код Web Workers не может работать с DOM.

В настоящий момент WebAssembly не поддерживает многопоточность. Однако, вероятнее всего, эта возможность

появится совсем скоро. Wasm будет близок к низкоуровневым потокам (то есть — тем, которые используются в C++). Возможность работать с «настоящими» потоками создаст множество новых возможностей в разработке браузерных приложений. Но, конечно, многопоточность означает и появление новых сложностей.

Переносимость кода

JavaScript-приложения могут работать практически везде: в браузерах, на серверах, даже во встраиваемых системах.

WebAssembly спроектирован с учётом безопасности и возможности переносимости кода. Этим он очень похож на JavaScript. Он будет работать в любом окружении, поддерживающем wasm, то есть, например, в любом браузере.

В плане переносимости перед WebAssembly стоят те же цели, которых в своё время пытался достичь Java посредством апплетов.

Итоги

В ранних версиях WebAssembly особое внимание уделялось тяжёлым вычислениям, например, математическим. Очевидная область применения таких вычислений — игры, в которых приходится работать с мириадами пикселей. Подобные приложения можно писать на C++/Rust с использованием привычных методов работы с OpenGL и компилировать то, что

получится, в wasm. После чего всё это будет работать в браузере. В качестве примера откройте эту ссылку в Firefox. Тут применяется Unreal Engine.

Ещё один вариант использования WebAssembly, ориентированный на повышение производительности веб-приложений, заключается в реализации с использованием этой технологии библиотек, выполняющих ресурсоёмкие вычисления, например — это библиотеки для обработки изображений.

Использование wasm может довольно серьёзно снизить потребление энергии аккумуляторов на мобильных устройствах (конечно, зависит это и от движка), так как большинство вспомогательных операций по подготовке программы к выполнению будет выполнено в ходе статической компиляции кода.

Ожидается, что в будущем у нас появится возможность использовать бинарные wasm-файлы, даже не занимаясь написанием кода, который в них компилируется. В NPM можно найти проекты, которые реализуют такой подход.

Можно ли говорить о том, что wasm заменит JS? На данном этапе развития технология — определённо нельзя. Например, если речь идёт о работе с DOM или об использовании API платформы, на которой выполняется код, у JavaScript пока нет альтернативы, так как эти API доступны из JS-программ напрямую, без добавления дополнительных уровней абстракции.

Автор материала отмечает, что в SessionStack с интересом наблюдают за WebAssembly, рассчитывая ускорить с помощью этой технологии наиболее ресурсоёмкие части своих разработок.

Предыдущие части цикла статей:

Часть 1: Как работает JS: обзор движка, механизмов времени выполнения, стека вызовов
Часть 2: Как работает JS: о внутреннем устройстве V8 и оптимизации кода
Часть 3: Как работает JS: управление памятью, четыре вида утечек памяти и борьба с ними
Часть 4: Как работает JS: цикл событий, асинхронность и пять способов улучшения кода с помощью async / await
Часть 5: Как работает JS: WebSocket и HTTP/2+SSE. Что выбрать?

Уважаемые читатели! Планируете ли вы пользоваться WebAssembly в своих проектах?

Похожие публикации

Разработка статического блога на Gatsby и Strapi

7

ПЕРЕВОД

ru_vds • 1 февраля в 12:38

JavaScript ES8 и переход на async / await

107

ПЕРЕВОД

ru_vds • 10 октября 2017 в 14:58

WebAssembly – путь к новым горизонтам производительности

33

ПЕРЕВОД

ru_vds • 18 января 2017 в 12:59

Популярное за сутки

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

14

alexkuku • вчера в 16:12

Unit-тестирование скриншотами: преодолеваем звуковой барьер. Расшифровка доклада

4

lahmatiy • вчера в 13:05

Люди не хотят чего-то действительно нового — они хотят привычное, но сделанное иначе 25

ПЕРЕВОД

Smileek • вчера в 10:32

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

ПЕРЕВОД

ru_vds • вчера в 12:04

Как адаптировать игру на Unity под iPhone X к апрелю 0

P1CACHU • вчера в 16:13

Лучшее на Geektimes

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

HostingManager • вчера в 13:49

Обзор рынка моноколес 2018 70

lozga • вчера в 06:58

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

alizar • вчера в 15:14

Стивен Хокинг и его работа — что дал ученый человечеству?

8

marks • вчера в 14:46

Sunlike — светодиодный свет нового поколения

17

AlexeyNadezhin • вчера в 20:32