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

Как работает JS: о внутреннем устройстве V8 и оптимизации кода

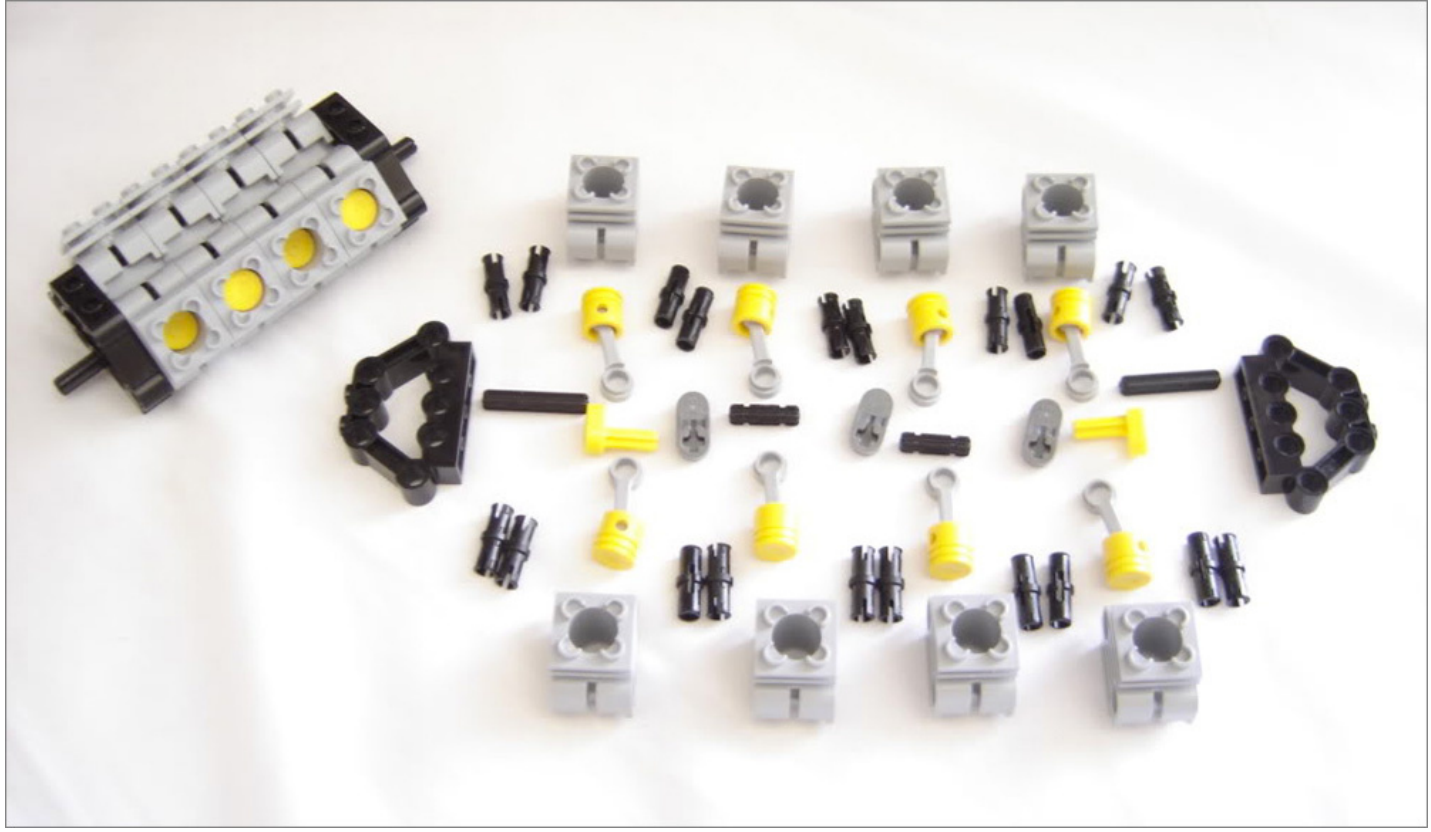
ПЕРЕВОД

ru_vds 8 сентября 2017 в 14:03 36,7k

Оригинал: Alexander Zlatkov

→ 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

Перед вами — второй материал из серии, посвящённой особенностям работы JavaScript на примере движка V8. В первом шла речь о механизмах времени выполнения V8 и о стеке вызовов. Сегодня мы углубимся в особенности V8, благодаря которым исходный код на JS превращается в исполняемую программу, и поделимся советами по оптимизации кода.



О JS-движках

JavaScript-движок — это программа, или, другими словами, интерпретатор, выполняющий код, написанный на JavaScript. Движок может быть реализован с использованием различных подходов: в виде обычного интерпретатора, в виде динамического компилятора (или JIT-компилятора), который, перед выполнением программы, преобразует исходный код на JS в байт-код некоего формата.

Вот список популярных реализаций JavaScript-движков.

- V8 — движок с открытым исходным кодом, написан на C++, его разработкой занимается Google.
- Rhino — этот движок с открытым кодом поддерживает Mozilla Foundation, он полностью написан на Java.

- SpiderMonkey — это самый первый из появившихся JS-движков, который в прошлом применялся в браузере Netscape Navigator, а сегодня — в Firefox.
- JavaScriptCore — ещё один движок с открытым кодом, известный как Nitro и разрабатываемый Apple для браузера Safari.
- KJS — JS-движок KDE, который разработал Гарри Портен для браузера Konqueror, входящего в проект KDE.
- Chakra (JScript9) — движок для Internet Explorer.
- Chakra (JavaScript) — движок для Microsoft Edge.
- Nashorn — движок с открытым кодом, являющийся частью OpenJDK, которым занимается Oracle.
- JerryScript — легковесный движок для интернета вещей.

В этом материале мы остановимся на особенностях V8.

Почему был создан движок V8?

Движок с открытым кодом V8 был создан компанией Google, он написан на C++. Движок используется в браузере Google Chrome. Кроме того, что отличает V8 от других движков, он применяется в популярной серверной среде Node.js.

Логотип V8

При проектировании V8 разработчики задались целью улучшить производительность JavaScript в браузерах. Для того, чтобы добиться высокой скорости выполнения программ, V8 транслирует JS-код в более эффективный машинный код, не используя интерпретатор. Движок компилирует JavaScript-код в машинные инструкции в ходе исполнения программы, реализуя механизм динамической компиляции, как и многие современные JavaScript-

движки, например, SpiderMonkey и Rhino (Mozilla). Основное различие заключается в том, что V8 не использует при исполнении JS-программ байт-код или любой промежуточный код.

О двух компиляторах, которые использовались в V8

Внутреннее устройство V8 изменилось с выходом версии 5.9, которая появилась совсем недавно. До этого же он использовал два компилятора:

- full-codegen — простой и очень быстрый компилятор, который выдаёт сравнительно медленный машинный код.
- Crankshaft — более сложный оптимизирующий JIT-компилятор, который генерирует хорошо оптимизированный код.

Внутри движка используются несколько потоков:

- Главный поток, который занимается тем, что от него можно ожидать: читает исходный JS-код, компилирует его и выполняет.
- Поток компиляции, который занимается оптимизацией кода в то время, когда выполняется главный поток.
- Поток профилировщика, который сообщает системе о том, в каких методах программа тратит больше всего времени, как результат, Crankshaft может эти методы оптимизировать.
- Несколько потоков, которые поддерживают механизм сборки мусора.

При первом исполнении JS-кода V8 задействует компилятор full-codegen, который напрямую, без каких-либо дополнительных

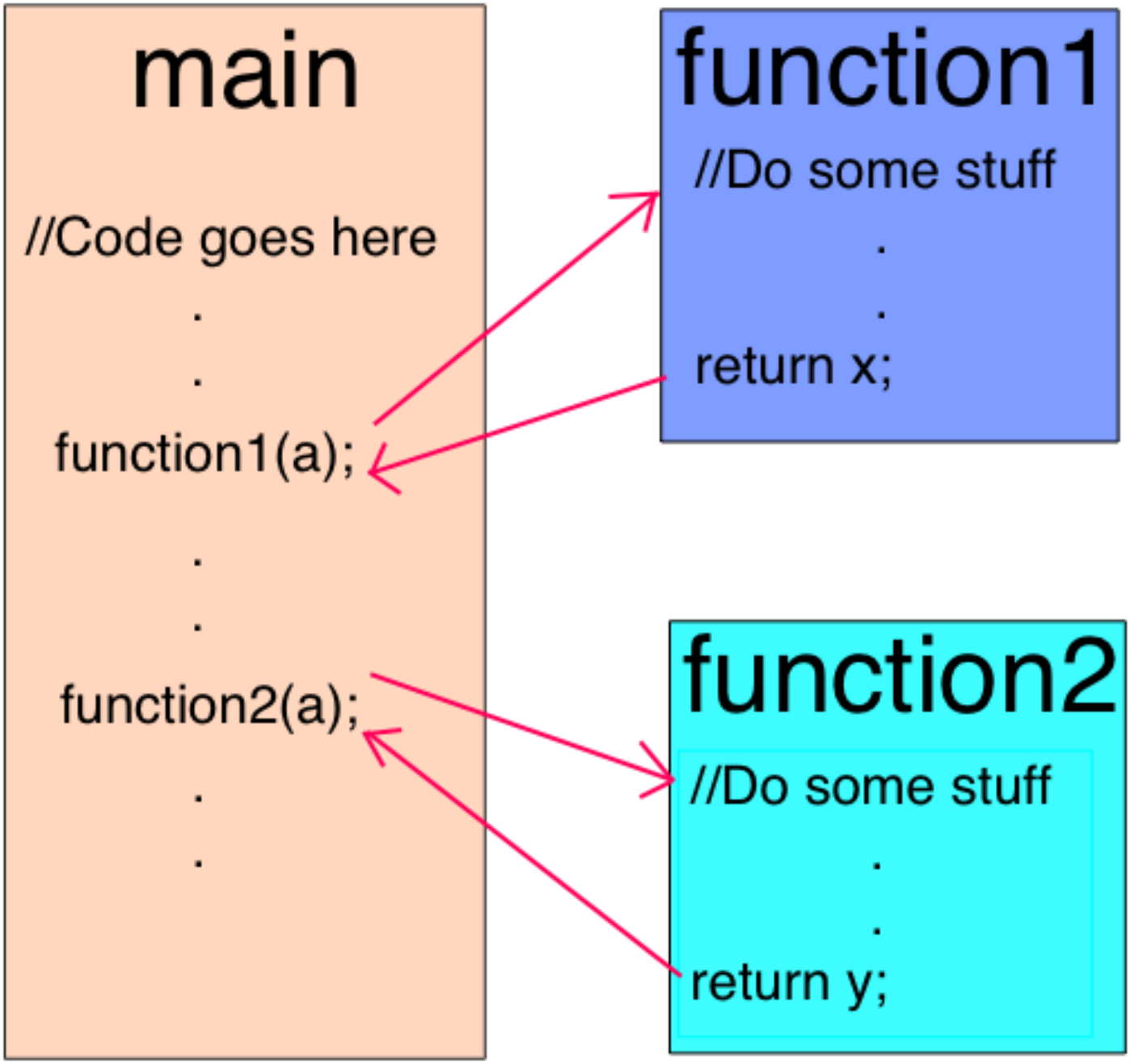
трансформаций, транслирует разобранный им JavaScript-код в машинный код. Это позволяет очень быстро приступить к выполнению машинного кода. Обратите внимание на то, что V8 не использует промежуточное представление программы в виде байт-кода, таким образом, устраняя необходимость в интерпретаторе.

После того, как код какое-то время поработает, поток профилировщика соберёт достаточно данных для того, чтобы система могла понять, какие методы нужно оптимизировать.

Далее, в другом потоке, начинается оптимизация с помощью Crankshaft. Он преобразует абстрактное синтаксическое дерево JavaScript в высокоуровневое представление, использующее модель единственного статического присваивания (static single-assignment, SSA). Это представление называется Hydrogen. Затем Crankshaft пытается оптимизировать граф потока управления Hydrogen. Большинство оптимизаций выполняется на этом уровне.

Встраивание кода

Первая оптимизация программы заключается в заблаговременном встраивании в места вызовов как можно большего объёма кода. Встраивание кода — это процесс замены команды вызова функции (строки, где вызывается функция) на её тело. Этот простой шаг позволяет сделать следующие оптимизации более результативными.



Вызов функции заменяется на её тело

Скрытые классы

JavaScript — это язык, основанный на прототипах: здесь нет классов. Объекты здесь создаются с использованием процесса клонирования. Кроме того, JS — это динамический язык программирования, это значит, что, после создания экземпляра объекта, к нему можно добавлять новые свойства и удалять из

него существующие.

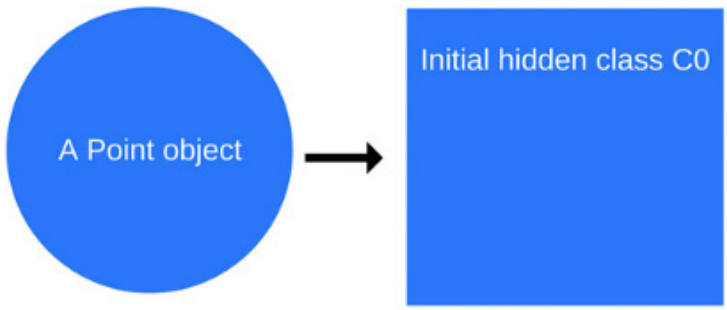
Большинство JS-интерпретаторов используют структуры, напоминающие словари (основанные на использовании хэш-функций), для хранения сведений о месте расположения значений свойств объектов в памяти. Использование подобных структур делает извлечение значений свойств в JavaScript более сложной задачей, чем в нединамических языках, таких, как Java и C#. В Java, например, все свойства объекта определяются не изменяющейся после компиляции программы схемой объекта, их нельзя динамически добавлять или удалять (надо отметить, что в C# есть динамический тип, но тут мы можем не обращать на это внимание). Как результат, значения свойств (или указатели на эти свойства) могут быть сохранены, с фиксированным смещением, в виде непрерывного буфера в памяти. Шаг смещения можно легко определить, основываясь на типе свойства, в то время как в JavaScript это невозможно, так как тип свойства может меняться в процессе выполнения программы.

Так как использование словарей для выяснения адресов свойств объекта в памяти очень неэффективно, V8 использует вместо этого другой метод: скрытые классы. Скрытые классы похожи на обычные классы в типичном объектно-ориентированном языке программирования, вроде Java, за исключением того, что создаются они во время выполнения программы. Посмотрим, как всё это работает, на следующем примере:

```
function Point(x, y) {
    this.x = x;
    this.y = y;
```

```
}
var p1 = new Point(1, 2);
```

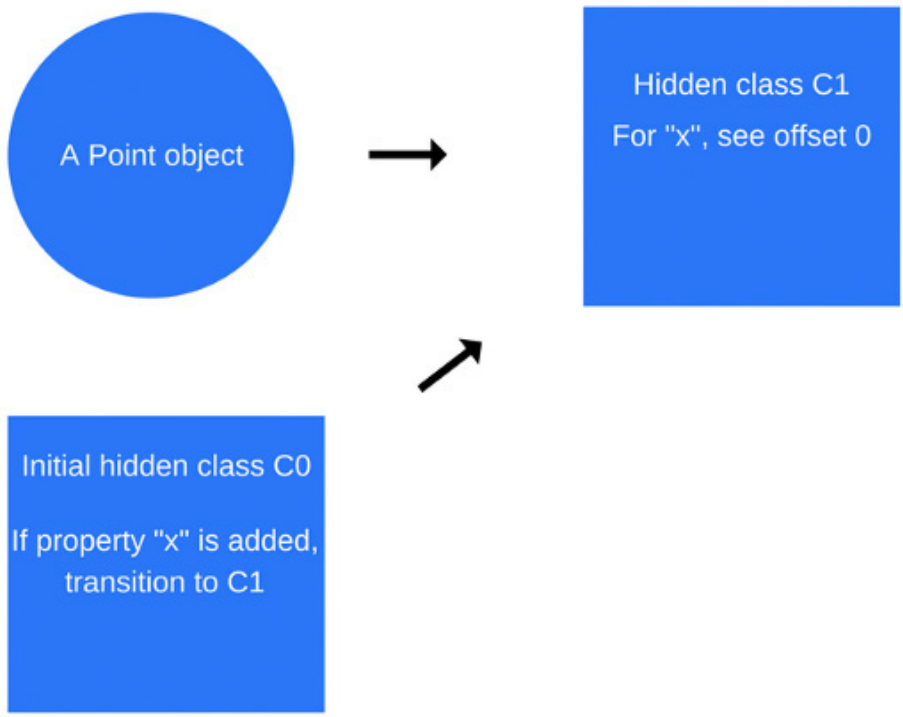
Когда происходит вызов `new Point(1, 2)`, V8 создаёт скрытый класс `C0`.



Первый скрытый класс C0

Пока, ещё до выполнения конструктора, у объекта `Point` нет свойств, поэтому класс `C0` пуст.

Как только будет выполнена первая команда в функции `Point`, V8 создаст второй скрытый класс, `C1`, который основан на `C0`. `C1` описывает место в памяти (относительно указателя объекта), где можно найти свойство `x`. В данном случае свойство x хранится по смещению 0, что означает, что если рассматривать объект `Point` в памяти как непрерывный буфер, первое смещение соответствует свойству `x`. Кроме того, V8 добавит в класс `C0` сведения о переходе к классу `C1`, где указывается, что если к объекту `Point` будет добавлено свойство `x`, скрытый класс нужно изменить с `C0` на `C1`. Скрытый класс для объекта `Point`, как показано на рисунке ниже, теперь стал классом `C1`.

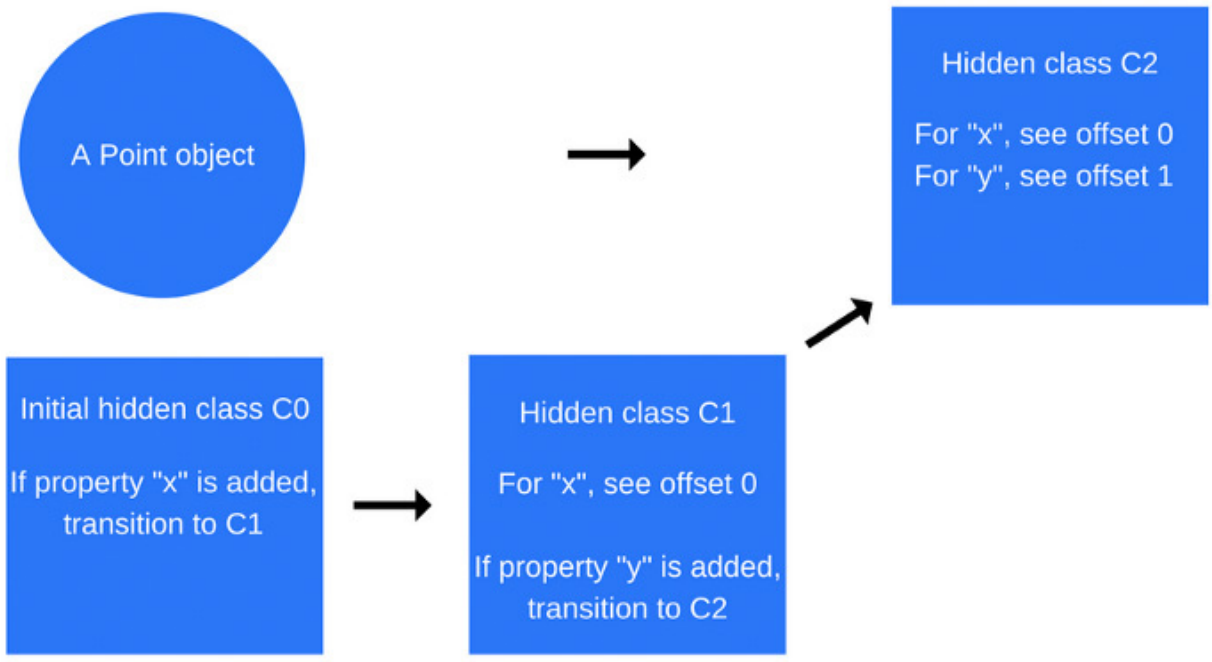


Каждый раз, когда к объекту добавляется новое свойство, в старый скрытый класс добавляются сведения о переходе к новому скрытому классу. Переходы между скрытыми классами важны, так как они позволяют объектам, которые создаются одинаково, иметь одни и те же скрытые классы. Если два объекта имеют общий скрытый класс и к ним добавляется одно и то же свойство, переходы обеспечат то, что оба объекта получат одинаковый новый скрытый класс и весь оптимизированный код, который идёт вместе с ним.

Этот процесс повторяется при выполнении команды `this.y = y` (опять же, делается это внутри функции `Point`, после вышеописанной команды по добавлению свойства `x`).

Тут создаётся новый скрытый класс, `C2`, а в класс `C1` добавляются

сведения о переходе, где указывается, что если к объекту `Point` добавляется свойство `y` (при этом речь идёт об объекте, который уже содержит свойство `x`), тогда скрытый класс объекта должен измениться на `C2`.



Переход к использованию класса C2 после добавления к объекту свойства y

Переходы между скрытыми классами зависят от порядка, в котором к объекту добавляются свойства. Взгляните на этот пример кода:

```
function Point(x, y) {
    this.x = x;
    this.y = y;
}
var p1 = new Point(1, 2);
p1.a = 5;
p1.b = 6;
var p2 = new Point(3, 4);
```

```
p2.b = 7;
p2.a = 8;
```

В подобной ситуации можно предположить, что у объектов `p1` и `p2` будет один и тот же скрытый класс и одно и то же дерево переходов скрытых классов. Однако, на самом деле это не так. В объект `p1` первым добавляется свойство `a`, а затем — свойство `b`. В объект `p2` сначала добавляют свойство `b`, а затем — `a`. В результате объекты `p1` и `p2` будут иметь различные скрытые классы — результат различных путей переходов между скрытыми классами. В подобных случаях гораздо лучше инициализировать динамические свойства в одном и том же порядке для того, чтобы скрытые классы могли быть использованы повторно.

Встроенные кэши

V8 использует и другую технику для оптимизации выполнения динамически типизированных языков, называемую встроенным кэшем вызовов. Встроенное кэширование основано на наблюдении, которое заключается в том, что повторяющиеся обращения к одному и тому же методу имеют тенденцию происходить с использованием объектов одного типа. Более подробно об этом можно почитать здесь. Если у вас нет времени слишком в это углубляться, читая вышеупомянутый материал, здесь мы изложим концепцию встроенного кэширования буквально в двух словах.

Итак, как же всё это работает? V8 поддерживает кэш типов объектов, которые мы передали в качестве параметра недавно

вызванным методам, и использует эту информацию для того, чтобы сделать предположение о типах объектов, которые будут переданы как параметры в будущем. Если V8 смог сделать правильное предположение о типе объекта, который будет передан методу, он может пропустить процесс выяснения того, как получать доступ к свойствам объекта, а, вместо этого, использовать сохранённую информацию из предыдущих обращений к скрытому классу объекта.

Как связаны концепции скрытых классов и встроенных кэшей вызовов? Когда метод вызывается для некоего объекта, движок V8 должен обратиться к скрытому классу этого объекта для того, чтобы определить смещение для доступа к конкретному свойству. После двух успешных обращений одного и того же метода к одному и тому же скрытому классу, V8 опускает операцию обращения к скрытому классу и просто добавляет сведения о смещении свойства к самому указателю объекта. Выполняя вызовов этого метода в будущем, V8 *предполагает*, что скрытый класс не менялся и идёт прямо по адресу памяти для конкретного свойства, используя смещение, сохранённое после предыдущих обращений к скрытому классу. Это очень сильно увеличивает скорость выполнения кода.

Встроенное кэширование вызовов, кроме того, является причиной того, почему так важно, чтобы объекты одного и того же типа использовали общие скрытые классы. Если вы создаёте два объекта одинакового типа, но с разными скрытыми классами (как сделано в примере выше), V8 не сможет использовать встроенное кэширование, так как, даже хотя объекты имеют один и тот же тип,

в соответствующих им скрытых классах назначено разное смещение их свойствам.

Перед нами объекты одного типа, но их свойства a и b были созданы в разном порядке и имеют разное смещение

Компиляция в машинный код

Как только граф Hydrogen оптимизирован, Crankshaft переводит его в низкоуровневое представление, которое называется Lithium. Большинство реализаций Lithium зависимо от архитектуры системы. На этом уровне, например, происходит выделение регистров.

В итоге Lithium-представление компилируется в машинный код. Затем происходит то, что называется замещением в стеке (on-stack replacement, OSR). Перед компиляцией и оптимизацией методов, в которых программа тратит много времени, нужно будет поработать с их неоптимизированными вариантами. Затем, не прерывая работу, V8 трансформирует контекст (стек, регистры) таким образом, чтобы можно было переключиться на оптимизированную версию кода. Это очень сложная задача,

учитывая то, что помимо других оптимизаций, V8 изначально выполняет встраивание кода. V8 — не единственный движок, способный это сделать.

Как быть, если оптимизация оказалась неудачной? От этого есть защита — так называемая деоптимизация. Она направлена на обратную трансформацию, возвращающую систему к использованию неоптимизированного кода в том случае, если предположения, сделанные движком и положенные в основу оптимизации, больше не соответствуют действительности.

Сборка мусора

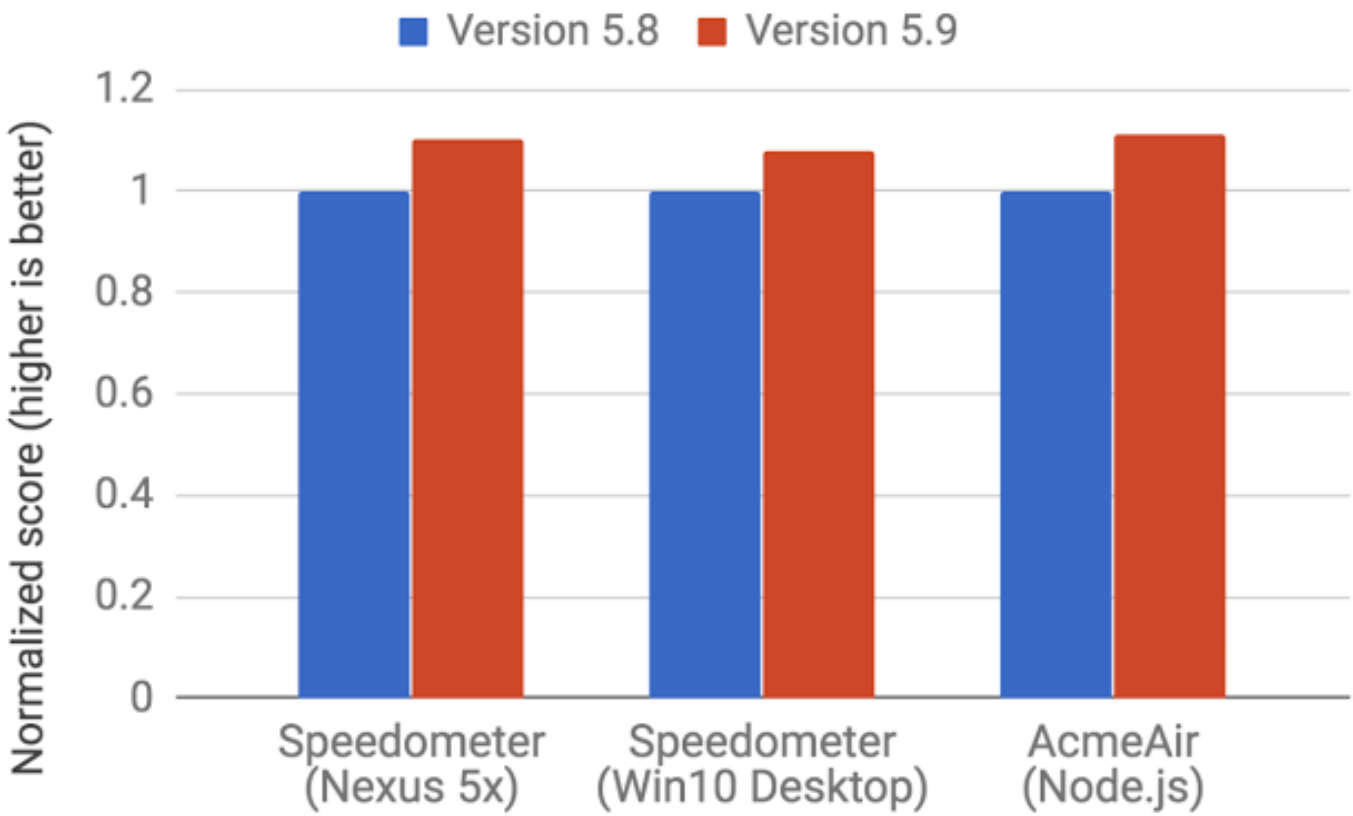
Для сборки мусора V8 использует традиционный генеалогический подход «пометь и выброси» (mark-and-sweep) для маркировки и очистки предыдущих поколений кода. Фаза маркировки предполагает остановку выполнения JavaScript. Для того, чтобы контролировать нагрузку на систему, создаваемую сборщиком мусора и сделать выполнение кода более стабильным, V8 использует инкрементный алгоритм маркирования: вместо того, чтобы обходить всю кучу, он пытается пометить всё, что сможет, обходя лишь часть кучи. Затем нормальное выполнение кода возобновляется. Следующий проход сборщика мусора по куче начинается там, где закончился предыдущий. Это позволяет добиться очень коротких пауз в ходе обычного выполнения кода. Как уже было сказано, фазой очистки памяти занимаются отдельные потоки.

Ignition и TurboFan

С выходом в этом году V8 версии 5.9. был представлен и новый конвейер выполнения кода. Этот конвейер позволяет достичь ещё большего улучшения производительности и значительной экономии памяти, причём, не в тестах, а в реальных JavaScript-приложениях.

Новая система построена на базе интерпретатора Ignition и новейшего оптимизирующего компилятора TurboFan. Подробности об этих новых механизмах V8 можно почитать в этом материале.

С выходом V8 5.9 full-codegen и Crankshaft (технологии, которые использовались в V8 с 2010-го года) больше применяться не будут. Команда V8 развивает новые средства, стараясь не отстать от новых возможностей JavaScript и внедрить оптимизации, необходимые для поддержки этих возможностей. Переход на новые технологии и отказ от поддержки старых механизмов означает развитие V8 в сторону более простой и хорошо управляемой архитектуры.



Улучшения в тестах производительности для браузерного и серверного вариантов использования JS

Эти улучшения — лишь начало. Новый конвейер выполнения кода на основе Ignition и TurboFan открывает путь к дальнейшим оптимизациям, которые улучшат производительность JavaScript и сделают V8 экономичнее.

Мы рассмотрели некоторые особенности V8, а теперь приведём несколько советов по оптимизации кода. На самом деле, кстати, всё это вполне можно вывести из того, о чём мы говорили выше.

Подходы к оптимизации JavaScript-кода для V8

1. **Порядок свойств объектов**. Всегда инициализируйте свойства объектов в одном и том же порядке. Нужно это для того, чтобы одинаковые объекты использовали одни и те же скрытые классы, и, как следствие, оптимизированный код.
2. **Динамические свойства**. Добавление свойств к объектам после создания экземпляра объекта приведёт к изменению скрытого класса и к замедлению методов, которые были оптимизированы для скрытого класса, используемого объектами ранее. Вместо добавления свойств динамически, назначайте их в конструкторе объекта.
3. **Методы**. Код, который несколько раз вызывает один и тот же метод, будет выполняться быстрее, чем код, который вызывает несколько разных методов по одному разу (из-за встроенных кэшей).
4. **Массивы**. Избегайте разреженных массивов, ключи которых не являются последовательными числами. Разреженный массив, то есть массив, некоторые из элементов которого отсутствуют, будет обрабатываться системой как хэш-таблица. Для доступа к элементам такого массива требуется больше вычислительных ресурсов. Кроме того, постарайтесь избежать заблаговременного выделения памяти под большие массивы. Лучше, если их размер будет увеличиваться по мере надобности. И, наконец, не удаляйте элементы в массивах. Из-за этого они превращаются в разреженные массивы.
5. **Числа**. V8 представляет числа и указатели на объекты, используя 32 бита. Он задействует один бит для того, чтобы определить, является ли некое 32-битное значение указателем на объект (флаг — 1), или целым числом (флаг — 0), которое называется маленьким целым числом (SMall Integer, SMI) из-за

того, что его длина составляет 31 бит. Если для хранения числового значения требуется более 31 бита, V8 упакует число, превратив его в число двойной точности и создаст новый объект для того, чтобы поместить в него это число. Постарайтесь использовать 31-битные числа со знаком везде, где это возможно, для того, чтобы избежать ресурсоёмких операций упаковки чисел в JS-объекты.

Итоги

Мы, в SessionStack, стараемся следовать вышеизложенным принципам при написании JS-кода. Надеемся, немного разобравшись в том, как работают внутренние механизмы V8, и учтя то, что мы рассказали выше, вы сможете улучшить качество и производительность ваших программ.

Уважаемые читатели! Какими советами по оптимизации JS-кода можете поделиться вы?

Похожие публикации

Может ли в JavaScript конструкция (a==1 && a==2 && a==3) оказаться равной true?

95

ПЕРЕВОД

ru_vds • 25 января в 16:08

JavaScript: путь к ясности кода

11

ПЕРЕВОД

ru_vds • 15 ноября 2017 в 13:01

Внутренние механизмы V8 и быстрая работа со свойствами объектов

8

ПЕРЕВОД

ru_vds • 7 сентября 2017 в 12:02

Популярное за сутки

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

14

alexkuku • вчера в 16:12

Unit-тестирование скриншотами: преодолеваем звуковой барьер. Расшифровка доклада

4

lahmatiy • вчера в 13:05

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

ПЕРЕВОД

Smileek • вчера в 10:32

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

ПЕРЕВОД

ru_vds • вчера в 12:04

Как адаптировать игру на Unity под iPhone X к апрелю 0

P1CACHU • вчера в 16:13

Лучшее на Geektimes

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

HostingManager • вчера в 13:49

Обзор рынка моноколес 2018 70

lozga • вчера в 06:58

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

alizar • вчера в 15:14

Стивен Хокинг и его работа — что дал ученый человечеству?

8

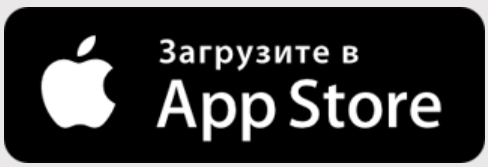
marks • вчера в 14:46

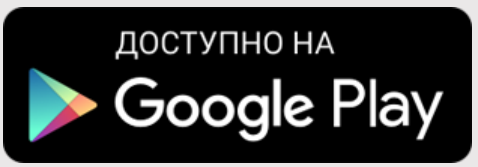
Sunlike — светодиодный свет нового поколения

17

AlexeyNadezhin • вчера в 20:32

Мобильное приложение





Полная версия