СТАТЬИ

УДК 004.021

РАБОТА ВЫДЕЛЕНИЯ ПАМЯТИ ДЛЯ JAVASCRIPT НА ПРИМЕРЕ БРАУЗЕРНОГО ДВИЖКА V8

Максимов Я.А., Мартышкин А.И.

ФГБОУ ВО «Пензенский государственный технологический университет», Пенза, e-mail: Starspeen@gmail.com, Alexey314@yandex.ru

В статье предлагается описание такого понятия, как браузерный движок для запуска JavaScript кода, и объясняется, для чего он нужен. Описаны современные движки, используемые современными браузерами. Более того, были предоставлены примеры, на которых можно увидеть, как движки оптимизируют выделение памяти на примере V8 от Google. В статье подчеркнута важность и актуальность данной темы, так как большие клиентские веб-приложения затрачивают много ресурсов. Разработчикам необходимо знать, как браузер использует и выделяет память для приложений, потому что это может помочь в разработке или отладке больших проектов. Цель статьи заключается в рассмотрении особенностей браузерных движков, с которыми могут сталкиваться разработчики при разработке клиентской части веб-приложений. Перечисленные примеры могут послужить руководством для других разработчиков при отладке или разработке подобных приложений, чтобы в результате получить углубленное понимание их работы и, как следствие, добиться разработки более оптимизированных приложений, которые работают в браузерах. Компилятор может делать все, что захочет, пока сохраняется вся языковая семантика, определенная спецификацией. В заключение приводятся основные выводы по проведенному исследованию.

Ключевые слова: память, оптимизация веб-приложений, клиентские веб-приложения, производительность, браузер, интернет, пользовательский интерфейс

THE WORK OF MEMORY ALLOCATION FOR JAVASCRIPT ON THE EXAMPLE OF THE V8 BROWSER ENGINE

Maksimov Ya.A., Martyshkin A.I.

Penza State Technological University, Penza, e-mail: Starspeen@gmail.com, Alexey314@yandex.ru

This article provides a description of such concept as a browser engine for executing JavaScript code and what it is for. Modern engines used by modern browsers are described. Moreover, examples were provided where you can see example how the engines optimize memory allocation using V8 from Google. The article emphasizes the importance and relevance of this topic, since large client-side web applications consume a lot of resources. Developers need to know how the browser uses and allocates memory for applications because it can help in developing or debugging large projects. The purpose of the article is to consider the features of browser engines that developers may encounter when developing the client side of web applications. The examples listed here can serve as a guide for other developers when debugging or developing similar applications to gain an in-depth understanding of how they work and, as a result, develop better applications that run-in browsers. The compiler can do whatever it wants, as long as all the language semantics defined by the specification are preserved. At the article concludes, the conclusions of the study are presented.

Keywords: web application optimization, client web applications, performance, browser, internet, user interface

С момента своего создания в 1995 г. *JavaScript* превратился в один из фундаментальных языков программирования для интернета. С ростом использования *JavaScript* его популярность росла большими шагами в сообществах разработчиков программного обеспечения с открытым исходным кодом с каждым годом [1]. Он лежит в основе практически в основе всех современных веб-приложений – от социальных сетей до браузерных игр. Это удобный, выразительный, позволяющий создавать полномасштабные приложения язык.

В настоящее время язык программирования *JavaScript* является одним из самых популярных и сфера его применения довольно широкая: разработка коммерческих клиентских веб-приложений, *desktop*-приложений и мобильных приложений [1]. Написано множество книг и статей об устройстве данного языка, но мало что известно о том, что представляет из себя его модель памяти. Это может быть полезно при устранении проблем, связанных с утечками памяти в результате отладки больших веб-приложений в браузере.

Материалы и методы исследования

Прежде чем начать повествование о том, что представляет из себя модель памяти, нужно сказать, что исполнение кода, написанного на *JavaScript*, зависит в первую очередь от окружения, где данный код исполняется. Это может быть один из браузеров: *Internet Explorer*, *Mozilla Firefox*, *Google Chrome*, которые в свою очередь имеют специальные программные средства для запуска *JavaScript* кода, называемые движками [2]. Движок представляет собой программный компонент, выполняющий код *JavaScript*. Первые такие движки были простыми интерпретаторами, но со време-

нем они совершенствовались и, например, сегодня все современные движки динамически управляют памятью для запуска веб-приложений, поэтому разработчикам не нужно беспокоиться об этом самим. Движок периодически просматривает память, выделенную приложению, определяет живую память и освобождает неиспользуемую память, а также они используют компиляцию *just-in-time* для повышения производительности [3]. Такие компоненты обычно разрабатываются поставщиками веб-браузеров и работают совместно с механизмом рендеринга через объектную модель документа. Ниже приведены примеры некоторых движков: *Chakra* – проприетарный движок *JScript*, разработанный *Microsoft* для использования в веб-браузере *Internet Explorer* [4]. *SpiderMonkey* – *JavaScript* и *WebAssembly Engine* от *Mozilla*, используемый в *Firefox*, *Servo* и других проектах. Он написан на *C++, Rust* и *JavaScript*. Его можно встраивать в проекты *C++* и *Rust*, а также запускать как автономную оболочку [5]. *V8* – высокопроизводительный движок *JavaScript* и *WebAssembly* от *Google* с открытым исходным кодом, написанный на *C++*. Он используется, в частности, в *Chrome* и *Node.js*. Он реализует *ECMAScript* и *WebAssembly* и работает в системах *Windows 7* или более поздних версий, *macOS 10.12+* и *Linux*, использующих процессоры *x64, IA-32, ARM* или *MIPS*. *V8* может работать автономно или может быть встроен в любое *C++* приложение [6]. В рамках данной статьи все примеры и объяснения будут производиться на примере движка *V8*, потому что он является частью популярного браузера *Google Chrome* [7].

Изначально сам по себе язык программирования *JavaScript* не требует распределения памяти. Нельзя найти термины «стек» или «куча», используемые в спецификации *ECMAScript,* на которой он основан [8]. Все значения *JavaScript* размещаются в такой структуре данных, как куча, к которой обращаются указатели, независимо от того, являются ли они объектами, массивами, строками или числами. Все значения *JavaScript* размещаются в куче, к которой обращаются указатели, независимо от того, являются ли они объектами, массивами, строками или числами. В стеке хранятся только временные, локальные и небольшие переменные (в основном указатели), и это в значительной степени все это не связано с типами *JavaScript* [9]. Вопреки распространенному мнению примитивные значения также размещаются в куче, как и объекты. Чтобы проверить это, можно ввести команду, представленную в листинге 1, на компьютере, на котором установлено программное обеспечение *Node JS*.

```
node --v8-options
```

Листинг 1. Параметры движка V8

В результате выполнения команды можно получить разную информацию, включая размер стека по умолчанию в *V8* на компьютере. В данном случае это 984 КБ (рис. 1).

```
--stack-size (default size of stack region v8 is allowed to use (in kBytes))
      type: int  default: 984
```

Рис. 1. Размер стека по умолчанию

Можно получить размер используемой кучи при помощи запуска кода, который представлен в листинге 2.

```
function memoryUsed() {
    // получить размер выделенной памяти в куче в МБ
    const mbUsed = process.memoryUsage().heapUsed / 1024 / 1024
    console.log(`Memory used: ${mbUsed} MB`);    }
memoryUsed();
```

Листинг 2. Получение размера кучи

После выполнения команды размер используемой памяти составляет 2,58504 МБ (рис. 2).

```
Memory used: 2.8416900634765625 MB
```

Рис. 2. Размер используемой памяти для кучи

Если в *JavaScript* файле создать строку с большим количеством символов с помощью команды, которая находится в листинге 3, то можно будет увидеть, что увеличилось количество потребляемой памяти для кучи.

```
function memoryUsed() {
    // получить размер выделенной памяти в куче в МБ
    const mbUsed = process.memoryUsage().heapUsed / 1024 / 1024
    console.log(`Memory used: ${mbUsed} MB`);     }
memoryUsed()
// создание большой строки
const bigString = 'x'.repeat(10*1024*1024)
console.log(bigString); // необходимая часть, для того, чтобы
движок V8 не сделал оптимазации
memoryUsed();
```

Листинг 3. Получение размера кучи

Результат выполнения кода представлен на рис. 3.

```
Memory used: 12.887123107910156 MB
```

Рис. 3. Размер используемой памяти для кучи

Разница между до и после составляет ровно 10 МБ. Если посмотреть на размер стека, который был получен ранее, он был всего 864 КБ – стек никак не может хранить такую строку. Хочется отметить, что в вышеприведенном коде происходит вывод строки *bigString* в консоль. Это необходимо для того, чтобы движок *V8* определил, что переменная будет использована в будущем в коде, и по этой причине движок не будет очищать из памяти эту строку. Еще один интересный момент заключается в том, что если строку размером 10 МБ, которая содержится в переменной *bigString*, присвоить другой переменной, например *anotherString*, то данные не будут дублироваться и для этого не будет выделена для этого значения дополнительная память.

```
function memoryUsed() {
    const mbUsed = process.memoryUsage().heapUsed / 1024 / 1024
    console.log(`Memory used: ${mbUsed} MB`); }
memoryUsed()
const bigString = 'x'.repeat(10*1024*1024)
const anotherString = bigString;
console.log(anotherString); // необходимая часть, для того, чтобы
движок V8 не сделал оптимазации
memoryUsed();
```

Листинг 4. Получения значения выделенной памяти для кучи при дублировании переменной

Присвоение одинаковых переменных *JavaScript* не влечет за собой затрат, пропорциональных размеру фактических значений – в этом суть указателей, а переменные *JavaScript* (в основном) ими и являются. Это возможно также проверить с помощью профилирования памяти с помощью инструментов разработчика в *Google Chrome*. Для этого нужно создать *html*-файл, фрагмент которого представлен в листинге 5.

```
…
      const btn = document.querySelector("#btn");
      btn.onclick = () => {
        const string1 = "foo";
        const string2 = «foo»;      };
```

Листинг 5. Пример кода для проверки выделения памяти для значений

Далее необходимо открыть этот файл через браузер *Google Chrome* и открыть инструменты разработчика, выбрать вкладку «Память» с настройками как на рис. 4.

Запустите профилирование памяти и нажмите кнопку Button, чтобы создать две переменные с одинаковым строковым значением foo. Вы увидите, что в куче выделена только одна строка (рис. 5).

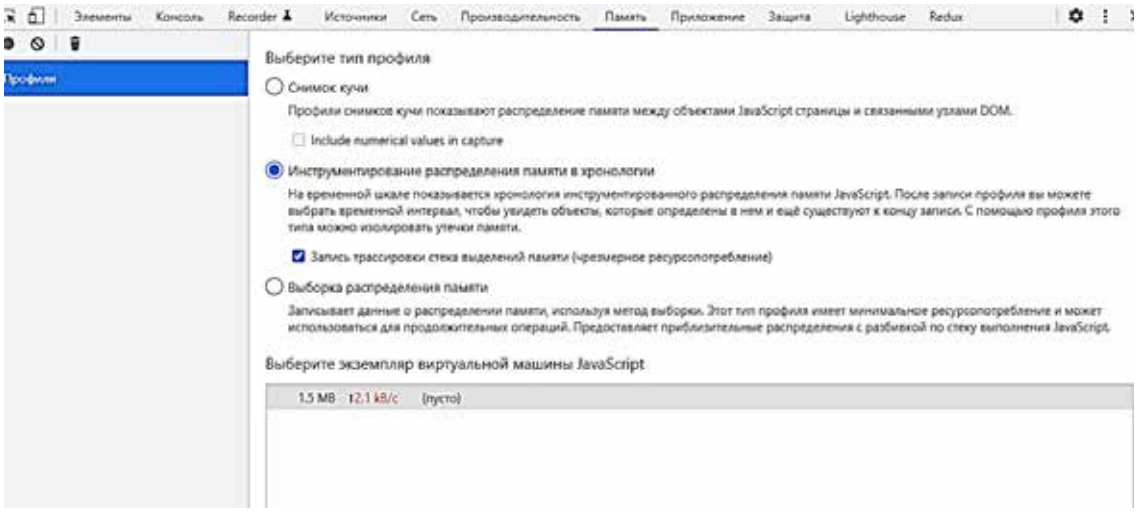

Рис. 4. Настройки для инструмента разработчика в браузере

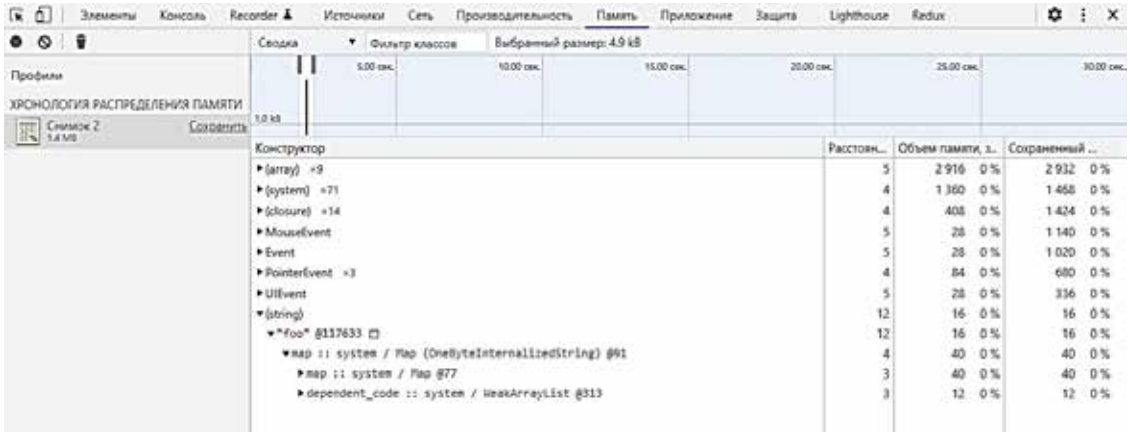

Рис. 5. Результат анализа выделенной памяти

Вкладка «Память» в инструментах разработчика в браузере Google Chrome не только показывает, где в памяти находятся указатели, но и дает представление о том, куда они указывают. Также числа, которые вы видите, например @117633, не представляют необработанные адреса памяти. Если вы хотите проверить фактическую память, вам нужно использовать собственный отладчик. Это называется интернированием строк. Внутри V8 это реализовано через *StringTable* (листинг 6).

```
  explicit StringTable(Isolate* isolate);
    ~StringTable();
    int Capacity() const;
    int NumberOfElements() const;
  // поиск строки в таблице. Если ее еще нет, то добавляем. Возвращаемое
  // значение строка
   Handle<String>LookupString(Isolate*isolate,Handle<String>key);
  // найти строку в таблице строк используя данный ключ. Если
этой строки еще нет, то создаем ее с этим ключем и добавляем. Воз-
вращаемое значение является строкой.
     template <typename StringTableKey, typename IsolateT>
    Handle<String> LookupKey(IsolateT* isolate, StringTableKey* key);
```

Листинг 6. StringTable

В V8 существует специальное подмножество примитивных значений, называемое Oddball (листинг 7).

```
type Null extends Oddball;
type Undefined extends Oddball;
type True extends Oddball;
type False extends Oddball;
type Exception extends Oddball;
type EmptyString extends String;
type Boolean = True|False;
```

Листинг 7. Виды Oddball значений

Они предварительно распределяются в куче с помощью V8 перед запуском первой строки скрипта – не имеет значения, использует ли программа JavaScript их на самом деле в будущем или нет. Они всегда используются повторно – существует только одно значение каждого типа *Oddball*. Для того, чтобы это проверить снова, необходимо создать файл *html* с кодом, аналогичным листингу 8, откройте в браузере и запустите профилировщик памяти.

```
…
      const btn = document.querySelector("#btn");
      btn.onclick = () => {
        function Oddballs() {
          this.undefined = undefined;
          this.true = true;
          this.false = false;
          this.null = null;
          this.emptyString = "";          }
        const obj1 = new Oddballs();
        const obj2 = new Oddballs();
        console.log(obj1);
        console.log(obj2);      };
…
```

Листинг 8. Пример получения Oddball значений

Делаем снимок кучи для этого скрипта выше, получаем результат, изображенный на рис. 6.

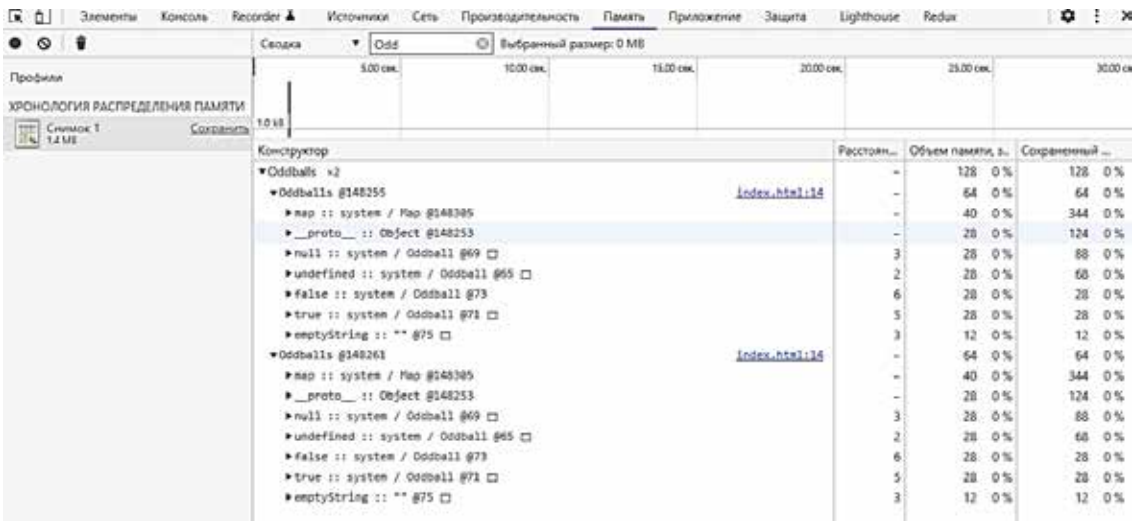

Рис. 6. Результат анализа выделенной памяти

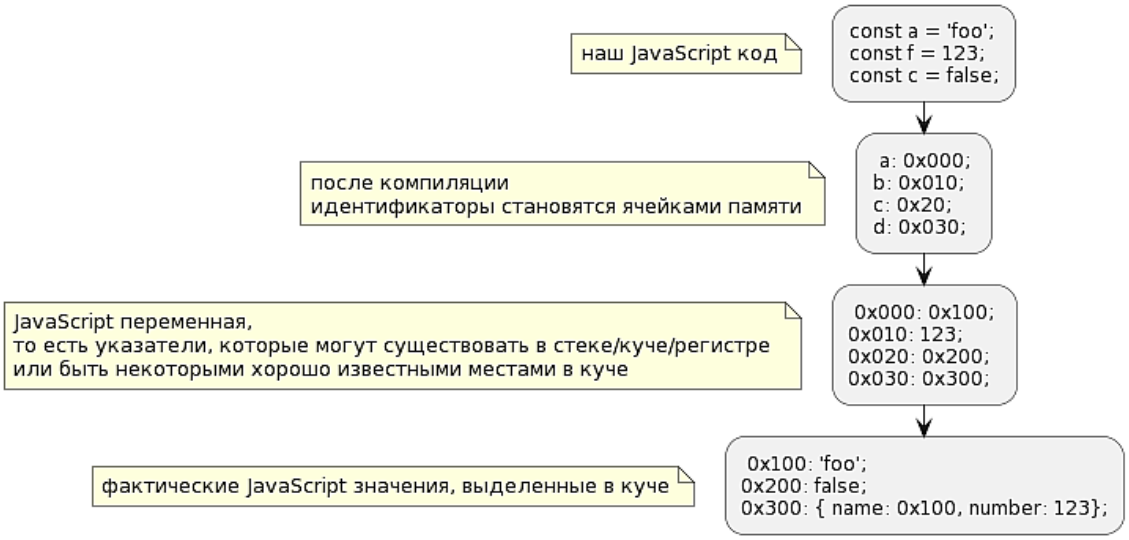

Рис. 7. Структура памяти в V8

Каждый тип *Oddball* имеет только одно и то же место в памяти в куче, даже если значения указываются разными свойствами объектов. Когда мы создаем переменные *JavaScript*, которые «имеют» значения *Oddball*, мы должны думать, что они были «вызваны» в нашей программе *JavaScript* – мы не можем создавать или уничтожать их. Углубившись в исходный код, мы можем обнаружить, что переменные, которые мы создаем в нашей программе *JavaScript*, являются просто адресами памяти, которые указывают на эти объекты *C++,* расположенные в куче. В *V8* целые числа в диапазоне от -2^{31} до 2^{31} в 64-битной архитектуре (термин *V8 – smi*) сильно оптимизированы, поэтому их можно кодировать непосредственно внутри указателя без необходимости выделения для него дополнительной памяти. И это не уникально для *V8* или *JavaScript*. Множество других языков, таких как *OCaml* и *Ruby*, тоже делают это. Таким образом, технически *smi* может существовать в стеке, поскольку им не требуется дополнительное хранилище, выделенное в куче, в зависимости от того, как объявлены переменные. Например, вариант *const a = 123* может быть в стеке, а *var a = 123* находится в куче, поскольку становится свойством глобального объекта, который является фиксированным местом в памяти. Также это зависит от того, что делает остальная часть скрипта, и от среды выполнения. Оптимизирующий компилятор хранит указатели в регистрах столько, сколько может, и выбрасывает их в стек только в таких ситуациях, как исчерпание регистров. Еще одна сложность, связанная с числами, заключается в том, что, в отличие от других типов примитивных значений, они могут не использоваться повторно. Для *smi* они закодированы как явно недействительные указатели, которые ни на что не указывают, поэтому вся концепция «повторного использования» на самом деле к ним не применима. Для *HeapNumbers* (числа, которые не считаются *smi*) в случаях, когда на них указывают свойства объекта, он становится изменяемым *HeapNumber*, что позволяет обновлять значение без выделения нового *HeapNumber* каждый раз. В других случаях их можно использовать повторно, но только в том случае, если это не приводит к дополнительной нагрузке на производительность. На рис. 7 представлена диаграмма, которая концептуально иллюстрирует структуру памяти в *V8*.

Заключение

Компьютерная память – невероятно сложная тема. Почти каждый ответ на вопрос, связанный с памятью, зависит от компилятора и архитектуры процессора. Например, переменные не всегда находятся в памяти (ОЗУ) – они могут быть загружены непосредственно в регистры назначения, стать частью инструкции в качестве непосредственного значения или даже быть полностью оптимизированными в небытие. Компилятор может делать все, что захочет, пока сохраняется вся языковая семантика, определенная спецификацией.

Список литературы

1. Top Computer Languages 2021. [Электронный ресурс]. URL: https://statisticstimes.com/tech/top-computer-languages.php (дата обращения: 20.02.2022).

2. Флэнаган Д. JavaScript. Полное руководство (7-е изд.). М.: Диалектика-Вильямс, 2021. 715 с.

3. Jochen E., Manfred E., Ross M., Hannes P. Idle time garbage collection Scheduling. 2017. 570 p.

4. Chakra-core/ChakraCore: ChakraCore is an open source Javascript engine with a C API. [Электронный ресурс]. URL: https://github.com/chakra-core/ChakraCore (дата обращения: 20.02.2022).

5. Home | SpiderMonkey JavaScript/WebAssembly Engine. [Электронный ресурс]. URL: https://spidermonkey.dev/ (дата обращения: 22.02.2022).

6. V8 JavaScript engine. [Электронный ресурс]. URL: https://v8.dev/ (дата обращения: 22.02.2022).

7. Названы самые популярные браузеры в мире – в России картина сильно отличается от общемировой. [Электронный ресурс]. URL: https://www.ixbt.com/news/2021/11/01/nazvany-samye-populjarnye-brauzery-v-mire--v-rossii-kartina-silno-otlichaetsja-ot-obshemirovoj.html свободный (дата обращения: 25.02.2022).

8. ECMAScript 2022 Language Specification. [Электронный ресурс]. URL: https://tc39.es/ecma262/ (дата обращения: 25.02.2022).

9. Хавербеке М. Выразительный JavaScript. Современное веб-программирование М.: Диалектика-Вильямс, 2021. 65 с.