Санкт-Петербургский государственный университет
Математико-механический факультет
Кафедра системного программирования

Allocation Profiler для движка V8 в Chromium

Курсовая работа студентки 444-ой группы
Михайловой Александры Алексеевны

Научный консультант . . . . . Семихатский Ю.С.

Санкт-Петербург
2014

Оглавление

Введение

В настоящее время, когда веб-приложения приобретают особую популярность, растёт важность их эффективной и удобной отладки. Особенно ценными в таком случае являются инструменты для анализа работы и отладки приложений в веб-браузере – так называемые Web Developer Tools[1] (далее Developer Tools). Это обусловлено тем, что сейчас для веб-разработчиков сам браузер является полноценным инструментом разработки, тестирования и сопровождения продукта. Создание прототипов страниц, проектирование интерфейсов, вёрстка макетов, тестирование программных модулей – ни на одном из этапов работы над веб-приложением специалист не может обойтись без браузера.

Инструменты для веб-разработчиков

Цель Developer Tools заключается в предоставлении разработчику как можно большего количества подробной информации о внутренних компонентах и состоянии как его веб-приложения, так и браузера. Например, следующие сценарии использования Developer Tools являются особенно популярными:

- проверка корректности DOM-структуры и исправление вёрстки веб-страниц;
- инспектирование CSS-стилей и работа с ними, в том числе в реальном времени;
- отладка кода приложения на JavaScript, расположение точек останова в коде;
- просмотр сообщений об ошибках, возникших в процессе работы приложения;
- отслеживание сетевых запросов и загруженных ресурсов;
- отслеживание памяти, потребляемой веб-приложением.

Инструменты для разработчиков, предоставляющие подобные функции, есть во всех наиболее популярных современных веб-браузерах. Ниже представлена таблица наиболее популярных браузеров [1] и соответствующих им инструментов.

Веб-браузер	Инструменты для веб-разработчика
Google Chrome / Chromium	Chrome DevTools [2]
Safari	Safari Web Inspector [3]

[1] Инструменты (для) веб-разработчиков.

Mozilla Firefox	Расширение для браузера Firebug [4]
Internet Explorer	IE Developer Tools
Opera	Opera Dragonfly [5]

Профилирование памяти

Одной из самых актуальных и перспективных функций Developer Tools является профилирование памяти, которую потребляет веб-приложение, поскольку данная функциональность делает возможным определение и предсказание утечек памяти. Концепция профилирования памяти подразумевает:

- подсчёт общего количества занятой приложением памяти в зависимости от времени;
- предоставление информации о размере и типе объектов, занимающих ресурсы памяти в данный момент;
- предоставление информации о времени создания и цепочки вызовов, которая привела к созданию объектов, занимающих ресурсы памяти в данный момент.

В настоящее время инструменты разработчика веб-браузеров предоставляют не все описанные выше функции профилирования памяти. Данная работа посвящена исследованию различных подходов к реализации профилирования памяти для инструментов разработчика в браузере Google Chrome / Chromium[2] в первую очередь для архитектуры x64, а также интеграции наиболее эффективного из них в продукт.

Chromium Developer Tools

Инструменты разработчика в браузере Chromium предоставляют большое количество способов инспектирования веб-приложений. В данном случае DevTools встроены в браузер (в отличие, например, от Firebug, который является расширением, заточенным под Mozilla Firefox) и представляют собой окно с организованными по задачам вкладками-панелями, которые содержат группы различных инструментов. Ниже на рис. 1 представлен скриншот, иллюстрирующий пример работы DevTools со структурой

[2] Google Chrome является коммерческим продуктом с закрытым кодом, созданным на основе открытого проекта Chromium [6].

и со стилями страницы. В левой части окна отображается DOM-дерево[3] страницы, в то время как справа отслеживаются CSS-стили и их переопределения.

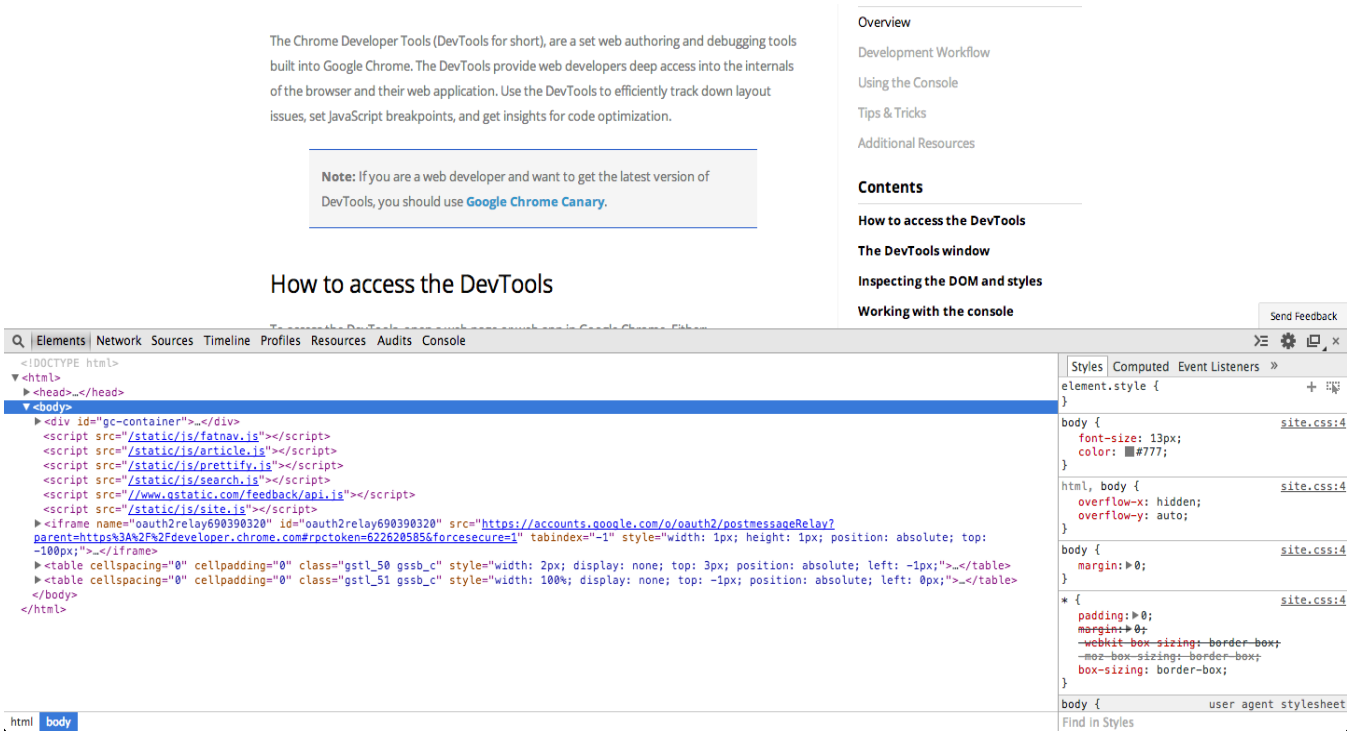

рис. 1

[3] Document Object Model.

Текущее решение задачи

V8: JavaScript-движок в Chromium

За компиляцию и запуск клиентского кода на JavaScript, а также за распределение памяти и сборку мусора в браузере Chromium отвечает движок V8 [7]. V8 написан на языке C++ и является проектом с открытым кодом, как и Chromium. Данный движок реализует стандарт ECMA-262 [8] для языка ECMAScript, который является встраиваемым расширяемым языком программирования, используемым в качестве основы для построения других языков, в том числе JavaScript. На основе клиентского кода V8 генерирует машинный код, который затем исполняется.

Внутри движка V8 все объекты, которые создаются в процессе работы клиентского кода, хранятся в структуре данных, представляющей собой кучу (heap). Соответствующим классом, реализующим эту структуру данных в V8, является v8::internal::Heap. Для профилирования работы и отслеживания состояния кучи в движке используется класс v8::internal::HeapProfiler, предоставляющий возможность получать информацию об объектах, занимающих место в куче в данный момент времени. Следует отметить, что по адресу объекта в куче можно определить только тип и размер объекта.

“Простое” профилирование памяти в Chromium

До завершения данной работы в V8 использовалась простая реализация профилирования памяти, основанная на регулярном периодическом взятии и сравнении друг с другом так называемых снимков кучи (heap snapshots) [9]. Ниже на рис. 2 и 3 представлены скриншоты пользовательского интерфейса инструментов профилирования памяти.

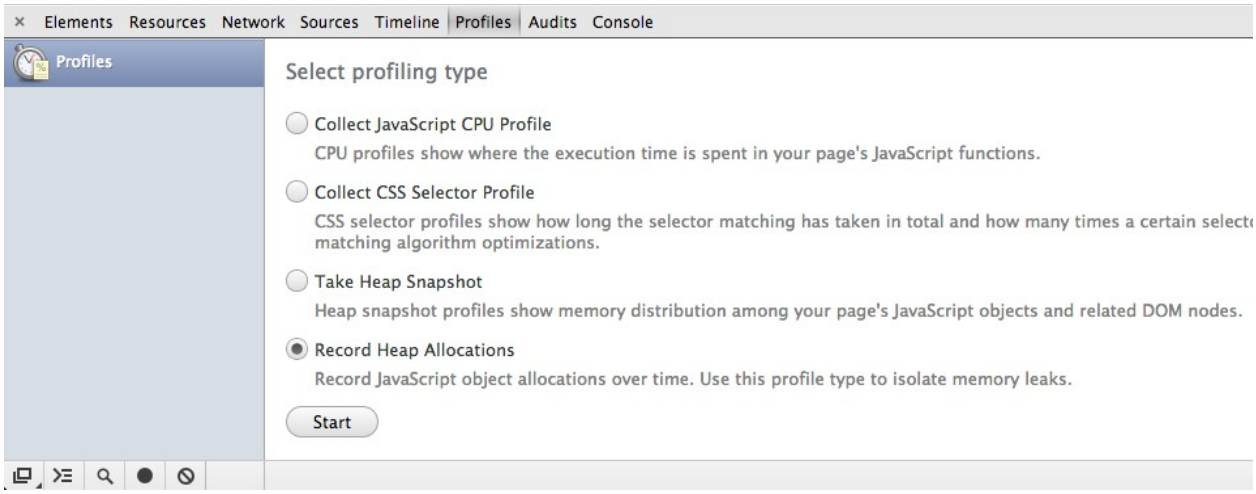

рис. 2

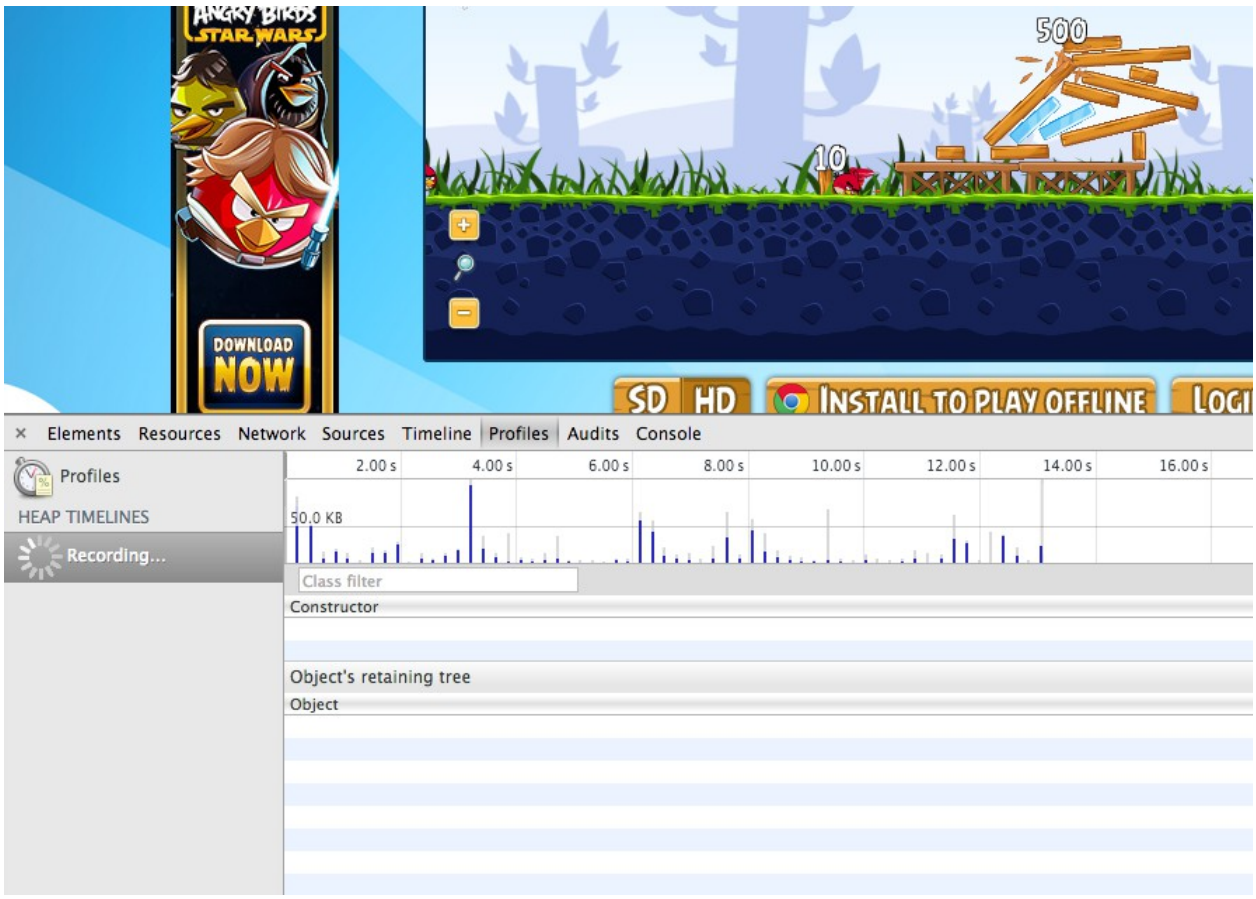

рис. 3

Снимок кучи представляет собой список записей с информацией о достижимых объектах, содержащихся в куче, где на один достижимый объект приходится одна запись,

хранящая тип и размер соответствующего ей объекта. Объект называется достижимым в одном из следующих случаев:

- данный объект является одним из корней структуры данных “куча”;
- на данный объект ссылается другой достижимый объект.

В любой выбранный момент времени в куче могут храниться также и недостижимые объекты, но в снимке информации о них содержаться не будет. Недостижимые объекты периодически собираются сборщиком мусора движка V8 [10]. Для построения снимка совершается обход кучи по ссылкам, связывающим объекты, начиная с корней. Такой обход кучи имеет линейную вычислительную сложность и, таким образом, сравним с простым итеративным проходом по всем записям в куче.

Алгоритм профилирования памяти, основанный на взятии снимков кучи, состоит из следующих шагов.

1. Взятие i-го снимка кучи.
2. Сравнение снимков кучи i и i - 1.
    a. Объекты, присутствующие в снимке i - 1 и отсутствующие в снимке i, считаются “мёртвыми”, то есть недостижимыми либо уже собранными сборщиком мусора.
    b. Объекты, отсутствующие в снимке i - 1 и присутствующие в снимке i, считаются аллоцированными в интервале между взятием данных снимков.
    c. Объекты, присутствующие в обоих снимках, считаются “живыми” (достижимыми) на протяжении всего интервала между взятием данных снимков.

Описанный алгоритм имеет три значительных недостатка в глазах пользователей профилирующих инструментов.

1. Проход по куче занимает значительное количество времени, а также блокирует кучу на изменение.
2. Получаемая информация об объектах неточна: вместо точного времени выделения памяти под объект известен только временной интервал. Кроме того, в интервале между взятием двух снимков объект по адресу X мог “умереть”, уступив место (X) новому объекту, но сравнение снимков в таком случае постановит, что старый объект остаётся “живым” до сих пор.

3. Получаемая информация довольно скудна: неизвестно точное время создания объекта, а также цепочка вызовов, которая к нему привела.

Обозначенные недостатки послужили отправной точкой для данной работы. Была предложена новая модель профилирования памяти, которая заключается в так называемой “мгновенной” регистрации объектов, то есть записи информации об объекте непосредственно после выделения памяти под этот объект. Была высказана гипотеза: “мгновенная” регистрация объектов решит описанные выше проблемы. В то же время, существенным ограничением является отсутствие падения производительности движка в условиях выключенного профилирования памяти.

Новые подходы к профилированию памяти

Генерация машинного кода в V8

Движок V8 транслирует клиентский код на языке JavaScript в момент его первого исполнения в машинный код на языке ассемблера без какого-либо промежуточного языка и без использования интерпретатора. Таким образом, появляется возможность генерации кода, содержащего инструкции, которые совершают вызов специальной инструментирующей функции и следуют сразу же за инструкциями выделения памяти под объект. Инструментирующая функция в данном случае представляет собой метод класса v8::internal::HeapProfiler, который записывает информацию об объекте в соответствующую структуру данных. В качестве такой структуры данных был выбран тип HashMap, который отображает адрес объекта в куче в запись с информацией о нём. Класс v8::internal::HeapObjectsMap служит обёрткой для данного типа. Ниже представлены выдержки из кода, иллюстрирующие инструментирующий интерфейс класса v8::internal::HeapProfiler и схему вызова инструментирующего метода v8::internal::HeapProfiler::RecordObjectAllocationFromMasm из машинного кода.

src/heap-profiler.h:

```
namespace v8 {
namespace internal {
class HeapProfiler {
 public:
  <...>
  static void RecordObjectAllocationFromMasm(Isolate* isolate,
                                             Address obj,
                                             int size);
  <...>
};
} }
```

src/x64/macro-assembler-x64.cc:

```
void MacroAssembler::RecordObjectAllocation(Isolate* isolate,
                                            Register object,
                                            int object_size) {
  <...>
```

```
  PushSafepointRegisters();
  PrepareCallCFunction(3);
  movq(arg_reg_2, object);
  movq(arg_reg_3, Immediate(object_size));
  movq(arg_reg_1, isolate, RelocInfo::EXTERNAL_REFERENCE);
  CallCFunction(
     ExternalReference::record_object_allocation_function(isolate), 3);
  PopSafepointRegisters();
  <...>
}
```

src/assembler.cc:

```
ExternalReference ExternalReference::record_object_allocation_function(
  Isolate* isolate) {
  return ExternalReference(
     Redirect(isolate,
           FUNCTION_ADDR(HeapProfiler::RecordObjectAllocationFromMasm)));
}
```

Следует заметить, что здесь MacroAssembler::RecordObjectAllocation является макровставкой, используемой в процессе генерации машинного кода и следующей сразу за выделением кусков памяти.

В обозначенных условиях возможны два подхода к использованию макровставки MacroAssembler::RecordObjectAllocation, содержащей вызов инструментирующего метода. Первый подход заключается в том, чтобы внутри данной вставки проверять флаг, обозначающий режим работы профайлера (профилирование памяти включено/выключено), с помощью инструкции cmp и в зависимости от значения флага делать вызов инструментирующего метода (профайлер включен) либо посредством инструкции jmp (j) пропускать вызов инструментации (профайлер выключен). Второй же подход предлагает генерировать машинный код в зависимости от значения данного флага и, как следствие, перегенерировать весь машинный код, когда пользователь переключает режим работы профайлера. Ниже представлены подробные детали реализации каждого из подходов и результаты сравнения их производительности.

Профилирование без перегенерации кода

Профилирование без перегенерации кода является наиболее прямым и простым подходом к инструментированию памяти. В данном случае логика инструментирования сосредоточена внутри макровставки MacroAssembler::RecordObjectAllocation, приведённой выше. Схема работы профайлера выглядит следующим образом.

1. В машинном коде на ассемблере производится проверка внешнего флага, символизирующего режим работы профайлера (включен/выключен).
2. Если профайлер выключен, посредством инструкции jmp (j) совершается “прыжок” через следующие далее инструкции вызова инструментирующего C++-метода из ассемблера.
3. Если профайлер включен, исполняются следующие далее инструкции вызова инструментирующего C++-метода.
4. При этом генерация машинного кода по клиентскому коду на JavaScript производится только один раз – в момент загрузки веб-страницы. Таким образом, машинный код, отвечающий за выделение и распределение памяти, остаётся неизменным до обновления веб-страницы.

Ниже приведены выдержки из кода, иллюстрирующие детали реализации данного подхода.

src/x64/macro-assembler-x64.cc:

```
void MacroAssembler::RecordObjectAllocation(Isolate* isolate,
                                            Register object,
                                            int object_size) {
  Label done;
  cmpb(ExternalOperand(
     ExternalReference::is_tracking_allocations_address(isolate)),
     Immediate(0));
  j(zero, &done);
  FrameScope frame(this, StackFrame::MANUAL);
  PushSafepointRegisters();
  PrepareCallCFunction(3);
  movq(arg_reg_2, object);
  movq(arg_reg_3, Immediate(object_size));
  movq(arg_reg_1, isolate, RelocInfo::EXTERNAL_REFERENCE);
```

```
  CallCFunction(
      ExternalReference::record_object_allocation_function(isolate), 3);
  PopSafepointRegisters();
  bind(&done);
}
```

src/assembler.cc:

```
ExternalReference ExternalReference::is_tracking_allocations_address(
  Isolate* isolate) {
  return ExternalReference(
    isolate->heap_profiler()->is_tracking_allocations_address());
}
```

Здесь is_tracking_allocations_address возвращает адрес внешнего флага, значение которого символизирует режим работы профайлера.

Следует заметить, что в случае выключенного профилирования памяти всегда будет выполняться “прыжок” после сравнения флага: j(zero, &done);, что обусловлено неизменяемостью сгенерированного при загрузке веб-страницы кода.

Профилирование с перегенерацией кода

Основное отличие данного подхода от предыдущего заключается в том, что генерация нового машинного кода происходит не только при загрузке страницы, но также и всякий раз, когда переключается режим работы профайлера. Ниже приведена схема работы профайлера в этом случае.

1. При загрузке веб-страницы и при включении/выключении профилирования памяти запускаются процессы генерации всего машинного кода и перегенерации кода, отвечающего за работу с памятью, соответственно.
2. В процессе генерации кода, отвечающего за работу с памятью, совершается проверка флага режима работы профайлера.
3. Если профайлер включен, в машинный код вставляются инструкции, производящие вызов инструментирующего метода.
4. Если профайлер выключен, инструкции, соответствующии инструментации, не вставляются.

Выдержки из кода ниже объясняют подробности реализации этого подхода.

src/x64/macro-assembler-x64.cc:

```
void MacroAssembler::Allocate(int object_size,
                              Register result,
                              Register result_end,
                              Register scratch,
                              Label* gc_required,
                              AllocationFlags flags) {
  <...>
  if (isolate()->heap_profiler()->is_tracking_allocations()) {
    RecordObjectAllocation(isolate(), result, object_size);
  }
  <...>
}

void MacroAssembler::RecordObjectAllocation(Isolate* isolate,
                                            Register object,
                                            int object_size) {
  FrameScope frame(this, StackFrame::EXIT);
  PushSafepointRegisters();
  PrepareCallCFunction(3);
  movq(arg_reg_2, object);
  movq(arg_reg_3, Immediate(object_size));
  movq(arg_reg_1, isolate, RelocInfo::EXTERNAL_REFERENCE);
  CallCFunction(
      ExternalReference::record_object_allocation_function(isolate), 3);
  PopSafepointRegisters();
}
```

src/heap-profiler.cc:

```
void HeapProfiler::StartHeapAllocationsRecording() {
  StartHeapObjectsTracking();
  is_tracking_allocations_ = true;
  DropCompiledCode();
  snapshots_->UpdateHeapObjectsMap();
}
```

```
void HeapProfiler::StopHeapAllocationsRecording() {
  StopHeapObjectsTracking();
  is_tracking_allocations_ = false;
  DropCompiledCode();
}
```

Комментарий: здесь метод v8::internal::HeapProfiler::DropCompiledCode отвечает за перегенерацию частей машинного кода, отвечающих за выделение и распределение памяти в куче.

Инструменты сравнения: Octane benchmark

В качестве инструмента сравнения двух описанных выше реализаций была выбрана система Octane [11] – инструмент измерения производительности движка JavaScript с помощью набора тестов, содержащих различные конструкции языка. Данные тесты разделены на смысловые группы по тестируемой функциональности. В качестве примеров таких групп можно привести:

- основные конструкции языка (Core language features): Richards, Deltablue, Raytrace;
- работа со строками и с массивами (Strings & arrays): Regexp, NavierStokes, pdf.js;
- выделение, распределение и освобождение памяти (Memory & GC): EarleyBoyer, Splay;
- сборка мусора (GC latency, Virtual machine & GC): SplayLatency, Typescript.

Наиболее интересной в рамках данной работы группой является набор Memory & GC.

В качестве итогового результата Octane считает среднее геометрическое всех промежуточных значений. В данной работе при измерении производительности способов профилирования были проведены многократные измерения на Octane при включенном и при выключенном профайлере и подсчитано среднее арифметическое полученных по группам тестов результатов. При этом 5% самых низких и 5% самых высоких результатов были отброшены.

Результаты

Сравнение подходов

Описанный выше метод сравнения способов профилирования выявил, что профилирование без перегенерации машинного кода при выключенной инструментации вызывает недопустимое в проекте V8 падение производительности (порядка 1%), в то время как профилирование с перегенерацией кода укладывается в поставленные рамки. Таблица ниже иллюстрирует результаты измерения производительности обоих подходов в сравнении с работой базовой версии движка V8 (инструментация кучи с помощью снимков) при выключенном профайлере.

	Базовая версия	**Без перегенерации**	**С перегенерацией**
Профайлер выключен	100%	- 1%	- 0.1%
Профайлер включен	- 50-70%	- 30-50%	- 25-30%

Такая разница между подходами обусловлена тем, что перегенерация кода удаляет недостижимый код (вызовы инструментирующего метода) и позволяет избежать jmp-инструкций.

Выводы

В процессе проведения данного исследования были выполнены все поставленные в начале работы задачи.

1. Были реализованы оба предложенных подхода в рамках модели “мгновенного” профилирования памяти.
2. Был выбран инструмент сравнения производительности различных реализаций.
3. Было проведено сравнение трёх реализаций: базовой, без перегенерации кода, с перегенерацией кода. Была выбрана наиболее производительная их них (с перегенерацией кода).
4. Итоговая реализация, основанная на перегенерации машинного кода, была интегрирована в проект V8 [12].

Дальнейшие исследования

Данная работа открыла возможности для дальнейшего усовершенствования профайлера памяти в проекте V8 для инструментов разработчика браузера Chromium (например, добавление информации о стеке вызова функции в месте создания каждого объекта), а также для исследований в области предсказания утечек памяти. Ниже перечислены направления, кажущиеся наиболее интересными и перспективными.

1. Предсказание утечек памяти на основе анализа работы приложения на данном временном промежутке с использованием методов машинного обучения.
2. Текущая инструментация пропускает объекты, выделенные на стадии генерации кода. По итогам данной работы был предложен новый подход в к профилированию, основная идея которого состоит в регистрации объектов на уровне построения списка виртуальных инструкций для кодогенератора, на основе которого впоследствии генерируется сам машинный код.
3. Текущая инструментация не учитывает объекты, выделенные с использованием одной из оптимизаций внутри V8 – так называемых folded allocations. Folded allocation возникает в тех случаях, когда под набор объектов, которые появляются друг за другом в контексте условного перехода, места выделяется больше, чем требуется, а указатели по выделенной памяти расставляются позднее. Folded allocation позволяет вместо нескольких обращений к аллокатору совершить только одно, но затрудняет инструментацию создания отдельных объектов.

Источники

[1] http://en.wikipedia.org/wiki/Usage_share_of_web_browsers Usage share of web browsers

[2] https://developer.chrome.com/devtools/index Chrome Developer Tools

[3] https://developer.apple.com/safari/tools/ Safari Web Inspector

[4] http://getfirebug.com/ Официальный сайт расширения Firebug

[5] http://www.opera.com/dragonfly/ Opera Dragonfly

[6] http://www.chromium.org/Home Chromium: an open-source browser project

[7] https://code.google.com/p/v8/ V8 JavaScript Engine

[8] http://www.ecma-international.org/publications/standards/Ecma-262.htm Стандарт ECMA-262

[9] http://www.html5rocks.com/en/tutorials/developertools/revolutions2013/ Chrome DevTools Revolutions 2013

[10] Michael Starzinger, V8 Garbage Collection and Write Barriers // Google Inc., 2012.

[11] https://developers.google.com/octane/ Octane JavaScript benchmark

[12] https://code.google.com/p/v8/source/detail?r=17191 V8 revision 17191

[13] https://code.google.com/p/chromium/issues/detail?id=277984 Chromium issue 277984: Allocation profiler: show allocation statistic for each function, including stack traces