Javascript profiling

#### При профайлинге:

- не должны быть запущены другие программы;

- браузер должен быть запущен с дефолтными настройками;

- открытой нужно оставить только одну вкладку с тестируемым сайтом (это ограничение связано с тем, что Хром может рендерить несколько вкладок в одном процессе, и, соответственно, в результатах профилирования будут лишние объекты);

- не должны быть установлены плагины (или они должны быть выключены);

- лучше всего делать профайлинг в режиме инкогнито.

Recording

Timeline имеет 3 режима исследования производительности приложения: Events, Frames,  Memoryusage.

Timeline изначально не содержит никакой статистики.

☻ - начать запись;

* - очистить текущую статистику;
* - принудительный запуск сборщика мусора;
* - фильтрация событий.

Examine

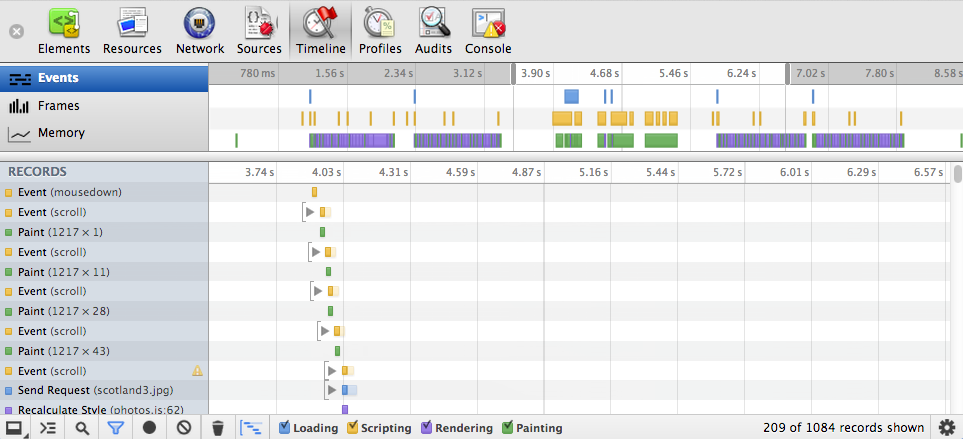
Events mode позволяет проанализировать, на что уходит наибольшее количество ресурсов:

**-** network and HTML parsing - синий,

**-** JavaScript - желтый,

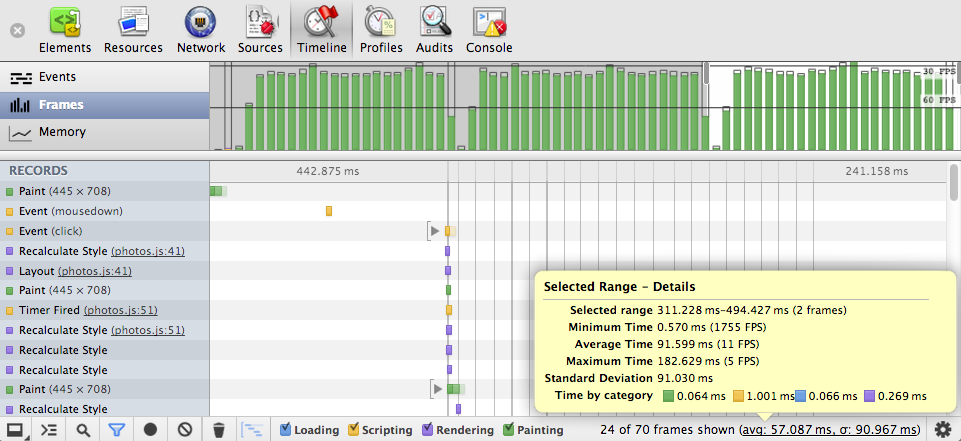
**-** style recalculation and layout**–** фиолетовый,

**-** painting and compositing  - зеленый.



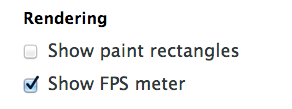
Improving frame rate, animations and responsiveness

Frames mode – позволяет оценивать сколько фреймов за секунду успевает отрисоваться. Норма 60 – 30.



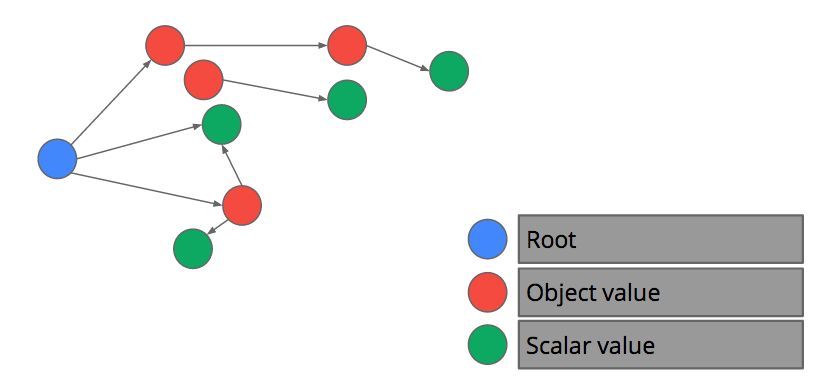
***Related tip:***

Можно включить real-time FPS counter в настройках DevTools



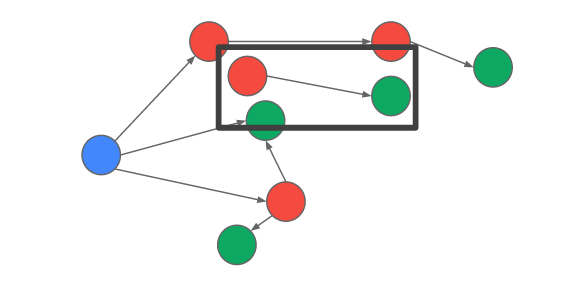
#### Terminology

### Все значения в JavaScript – части объектного графа. Граф начинается с корня - [window object](https://developer.mozilla.org/en-US/docs/DOM/window).



*When Does a Value Become Garbage?*

Значение становится мусором, когда к нему нет путь от корня графа

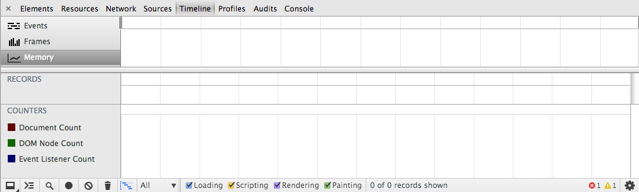


### *What is a Memory Leak in JavaScript?*

### Утечка памяти – это когда к элементу нет пути от корня графа, но тем не менее, в JavaScript есть ссылки на этот элемент.

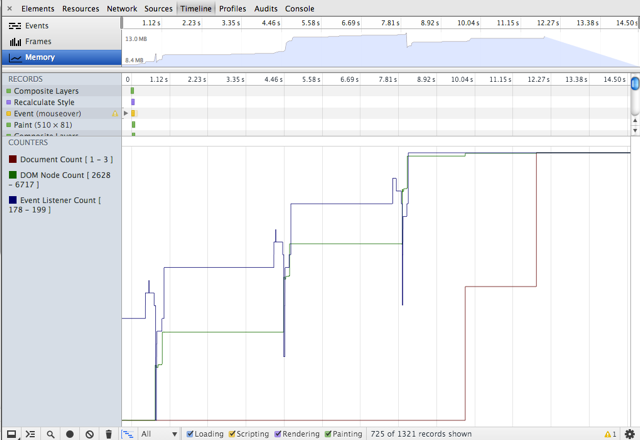
#### **Определяем кейсы, в которых могут быть утечки**

Чтобы определить случаи, когда есть возможность утечки памяти, мы воспользуемся панелью timeline в Chrome Developer Tools, а конкретнее режимом Memory:



Открываем тестируемый сайт и начинаем запись (Cmd+E). Дальше несколько раз совершаем действие, которое, по нашему мнению, может привести к утечке памяти.

Во время записи получаем нечто такое:



Тут важно обратить внимание на график в самой верхней панели и график со статистикой в панели «counters». На верхнем графике мы видим, как аллоцируется память и, собственно, размер JSHeap в разный момент времени.

Вообще рост горочкой — это нормально, потому что еще не было сборки мусора. Она будет со временем, и тогда значение занимаемой памяти должно вернуться в норму (в нашем случае — 8,4 МБ). Можно не ждать GC и принудительно вызвать сборку мусора, если нажать на ведерко в левой нижней части.

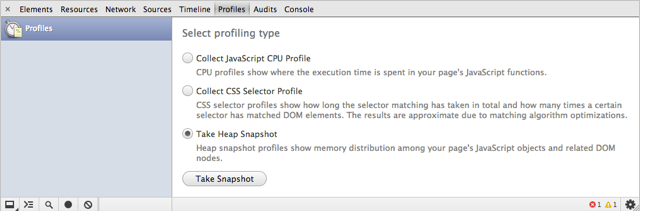
В панели «counters» есть три показателя:

* Document Count — количество html документов (сюда входят и фреймы);
* DOM Node Count — количество ДОМ нод;
* Event Listener Count — количество обработчиков событий,

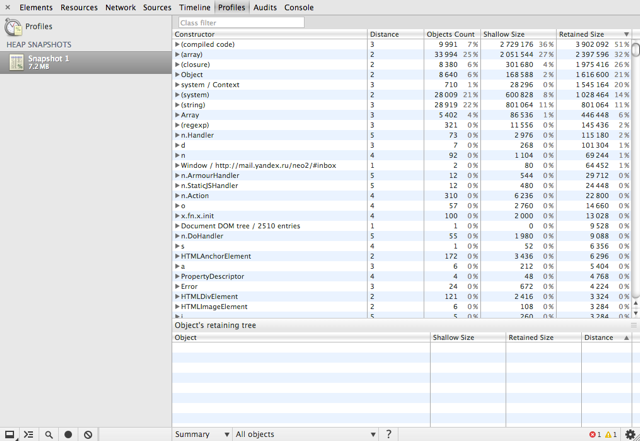
И есть график изменения этих показателей за указанный выше промежуток времени. Эти показатели важны, потому что они не учитываются на графике занимаемой памяти наверху. Так вот, если после сборки мусора горочка не падает до базовой линии, или показатели в панели Counters не возвращаются к прежним, значит, мы нашли кейс, который приводит к утечке.

#### **Находим причину утечки**

Чтобы найти причину утечки воспользуемся панелью Profiles и пунктом Take Heap Snapshot в ней.



Нажав на кнопку Take Snapshot, можно получить снапшот JSHeap на текущий момент:



В левой колонке под названием снапшота (у нас Snapshot 1) указан общий размер памяти занимаемой живыми объектами — такими, у которых есть retaining path. Только живыми, потому что после каждого нажатия на эту кнопку первым делом вызывается сборщик мусора. В панели, занимающей основную часть правой части скриншота, можно просматривать сами объекты, размеры занимаемой ими памяти (абсолютные значения в байтах или же процент от общего размера снапшота) и другую полезную информацию. По умолчанию в этой панели объекты показываются в режиме Summary, где они сгруппированы по имени своего конструктора. Колонка Distance — количество ссылок до этого объекта от корня (GC Roots). Objects Count — количество объектов с этим конструктором. 

Если название конструктора в скобках, то это внутренний тип объектов или примитив (за исключением array). В принципе, на большинство объектов с конструктором в скобках можно не обращать внимание ("(compiled code)", "(closure)", "(system)"), за исключением, пожалуй, "(string)" и "(array)" — и то только если у них большой shallow size.

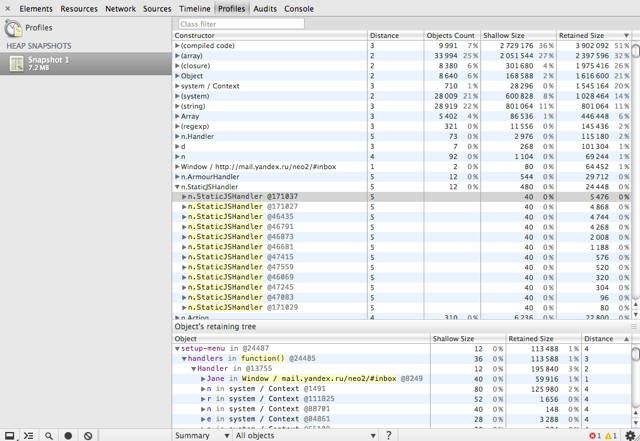
Представить содержимое снапшота можно и в других режимах:

* Comparison — сравнивает два снапшота и показывает только те объекты,

которые изменились.

* Containment — показывает весь граф в виде дерева с глобальными объектами. Удобен для обнаружения неиспользуемых ДОМ нод (Detached DOM tree).
* Dominators — Показывает доминаторы (объекты, которые есть в как можно большем числе retaining path'ей).

В режиме Summary для удобства можно фильтровать объекты по имени конструктора через строку поиска наверху. Если выбрать какой-то объект из группы, внизу можно увидеть retaining path в виде дерева:

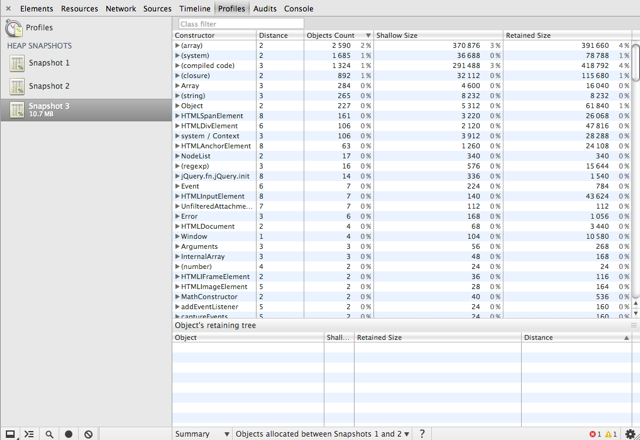


Серым указаны айдишники объектов. Если объект подсвечен желтым, значит, где-то есть ссылка на него, которая удерживает его от сборки мусора. Если объект подсвечен красным — это ДОМ нода, которая была задетачена, но на нее осталась ссылка из JS.

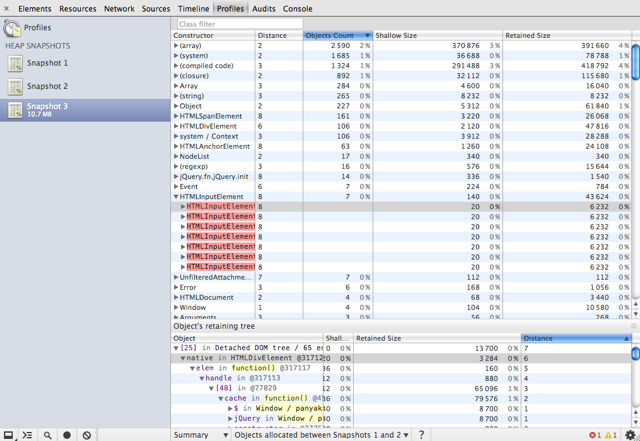
#### **Техника трех снапшотов**

* Делаем первый снапшот перед действиями из кейса (он нужен для базовой линии).
* Повторяем несколько раз действия из кейса (лучше повторять нечетное количество раз, чтобы можно было легче определить текущие объекты при анализе снапшота) и делаем второй снапшот.
* Повторяем снова эти же действия столько же раз и делаем третий снапшот.
* Дальше выбираем третий снапшот и в режиме Summary в селекте внизу выделяем «Objects allocated between Snapshots 1 and 2» (или 2 и 3 — как угодно).

Вот мы и получили все утекшие объекты. Отсортируем результаты по колонке Objects Count:



В топе после объектов с системными конструкторами и массивов кэшированных писем, видим какие-то подозрительные дивы, спаны, ссылки и инпуты. И их ровно 7 — именно столько раз я нажимал на кнопку «Проверить». Если раскрыть HTMLInputElement, то мы увидим, что все 7 объектов — это задетаченные инпуты с классом b-mail-dropdown\_\_search\_\_input, на которые просто остались ссылки в JS:

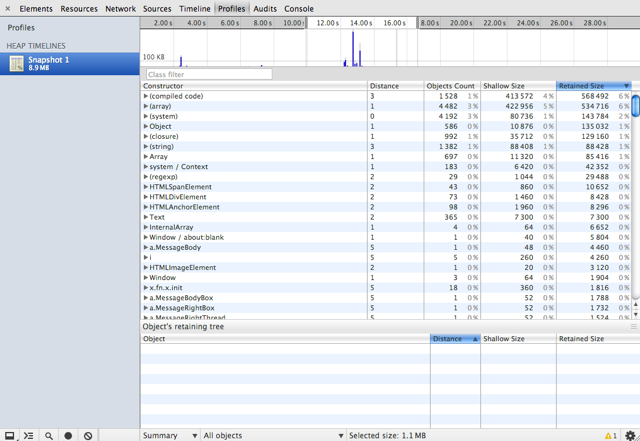


На скриншоте не видно, но, чтобы узнать класс, я просто навел курсор на один из инпутов. По retaining path можно быстро понять, что это инпут из дропдауна (первый HTMLDivElement имеет класс b-mail-dropdown), на который в свою очередь есть ссылка в обработчике какого-то события (можно понять по стандартной jQuery структуре $ — cache — [..] — handle — elem — наш элемент). Есть еще проблема с лоадером в тулбаре — див с лоадером не очищается после сборки мусора.

**Фиксим утечку**

Для избавления от утечки, просто анбиндим эти обработчики в onhtmldestroy и зануляем ссылки на this.$searchInput и this.$dropdown:

#### **Record Heap Allocations**

Чтобы упростить нахождение утечек, в Хром, начиная с 29 версии, введен дополнительный пункт в панели «Profiles» — «Record Heap Allocations». При нажатии на Start он начинает непрерывно раз в N секунд делать снапшот и сравнивать его с предыдущим, а в верхней панели столбиками показывать отношение очищенных объектов к живым. Живые — синие, очищенные — серые. После остановки записи можно выбрать любой интервал времени и в нижней панели посмотреть на объекты в привычном виде (как после Take Snapshot).  
  


***Related tips***

#### **Improving jQuery Selection and Processing**

Самый правый секлектор должен быть наиболее конкретизированным.

Например, вместо div.wrapper-class h1 лучше использовать div h1.wrapper-title

Быстрее всего Jquery ищет по айди и классам.

Не писать псевдоклассы на ряду с классами: $("input:hidden"), заменить на $("input").filter(":hidden")

Сохранять в локальнх переменнтых элементы, к которым будет часто е обращение, а не искать их каждый раз в DOM.

#### **Reducing and Optimizing CSS Performance**

Можно добавлять атрибут async на тег script

Использовать плагин PageSpeed

Минифицировать и собирать js в один файл, чтоб уменьшить количество запросов

Оптимизировать изображения и хранить соответсвующий размер, использовать png по возможности

Утечкой памяти могут послужить detached DOM tree items - то что удалено из DOM, но не удалено сборщиком мусора. Такая ситуация может возникнуть тогда, когда хоть DOM больше и не ссылается на объект, какая-то функция все же ссылается

***Оценка производительности из кода:***

console.time("Execution time took");

// Some code to execute

console.timeEnd("Execution time took");

Помимо замера времени можно профилировать код и вывести стек профилирования, который подробно показывает, где и сколько времени потратил браузер.

console.profile();  
// Some code to execute

console.profileEnd();

Links

<http://addyosmani.com/blog/performance-optimisation-with-timeline-profiles/>

<http://habrahabr.ru/company/yandex/blog/195198/>

<https://crowdfavorite.com/blog/2014/07/15/javascript-profiling-and-optimization/>

[developers.google.com/chrome-developer-tools/docs/memory-analysis-101](https://developers.google.com/chrome-developer-tools/docs/memory-analysis-101)

[developers.google.com/chrome-developer-tools/docs/heap-profiling](https://developers.google.com/chrome-developer-tools/docs/heap-profiling)

[addyosmani.com/blog/taming-the-unicorn-easing-javascript-memory-profiling-in-devtools/](http://addyosmani.com/blog/taming-the-unicorn-easing-javascript-memory-profiling-in-devtools/)

[coding.smashingmagazine.com/2012/11/05/writing-fast-memory-efficient-javascript/](http://coding.smashingmagazine.com/2012/11/05/writing-fast-memory-efficient-javascript/)

[www.html5rocks.com/en/tutorials/memory/effectivemanagement/](http://www.html5rocks.com/en/tutorials/memory/effectivemanagement/)

[www.html5rocks.com/en/tutorials/speed/static-mem-pools/](http://www.html5rocks.com/en/tutorials/speed/static-mem-pools/)