# Fastersite

On web performance

Sunday, August 7, 2011

## Finding memory leaks

Over lunch last week Mikhail Naganov (creator of the DevTools Heap Profiler) and I were discussing how invaluable it has been to have the same insight into JavaScript memory usage that we have into applications written in languages like C++ or Java. But the heap profiler doesn't seem to get as much attention from developers as I think it deserves. There could be two explanations: either leaking memory isn't a big problem for web sites or there is a problem but developers aren't aware of it.

### Are memory leaks a problem?

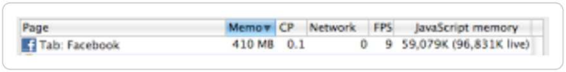
For traditional pages where the user is encouraged to navigate from page to page, memory leaks should almost never be a problem. However, for any page that encourages interaction, memory management must be considered. Most realize that ultimately if too much memory is consumed the page will be killed, forcing the user to reload it. However, even before all memory is exhausted performance problems arise:

- A large JavaScript heap means garbage collections may take longer.
- Greater system memory pressure means fewer things can be cached (both in the browser and the OS).
- The OS may start paging or thrashing which can make the whole system feel sluggish.

These problems are of course exacerbated on mobile devices which have less RAM.

### A real world walkthrough

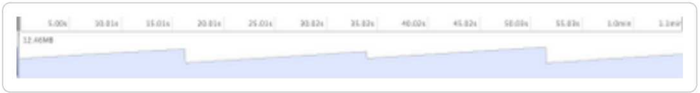
So, in order to demonstrate this is a real world problem and how easily the heap profiler can diagnose it, I set out to find a memory leak in the wild. A peak at the task manager (`Wrench > Tools > Task Manager`) for my open tabs showed a good candidate for investigation: Facebook is consuming 410MB!!



| Page | Memo▾ | CP | Network | FPS | JavaScript memory |
|---|---|---|---|---|---|
| Tab: Facebook | 410 MB | 0.1 | 0 | 9 | 59,079K (96,831K live) |

### Pinpoint the leaky action

The first step in finding a memory leak is to isolate the action that leaks. So I loaded facebook.com in a new tab. The fresh instance used only 49MB -- another indicator the 410MB might have been due to a leak. To observe memory use over time, I opened the Inspector's `Timeline` panel, selected the `Memory` tab and pressed the record button. At rest, the page displays a typical pattern of allocation and garbage collection. This is not a leak.



While keeping an eye on the graph, I began navigating around the site. I eventually noticed that each time I clicked the Events link on the left side, memory usage would rise but never

About Me

Tony Gentilcore

I'm a software engineer at Google who enjoys hacking on Chrome/WebKit to make the web faster.

View my complete profile

Popular Posts

Finding memory leaks

How a web page loads

Chrome's 10 Caches

How (not) to trigger a layout in WebKit

Optimizing with the timeline panel

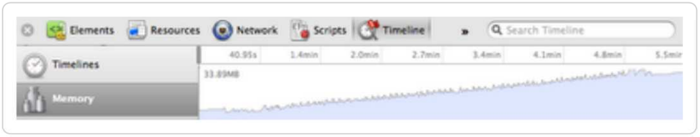
Blog Archive

- ► 2012 (4)
- ▼ 2011 (8)
  - ► December (1)
  - ▼ August (1)
    - Finding memory leaks
  - ► May (2)
  - ► March (1)
  - ► February (1)
  - ► January (2)
- ► 2008 (4)
- ► 2006 (2)

There was an error in this

gadget

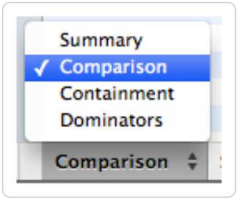
be collected. This is how the usage grows as I repeatedly click the link. A quintessential leak.



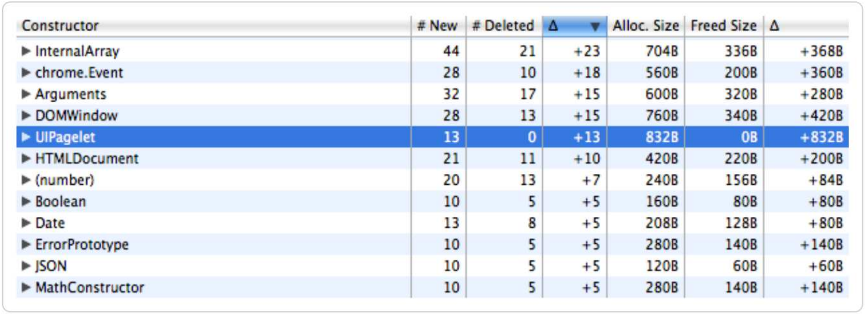
As an aside, this leak isn't a browser bug. The OS task manager shows similar memory growth when performing the same action in Firefox.

## Find the leaked memory

Now that we know we have a leak, the obvious next question is what is leaking. The heap profiler's ability to compare heap snapshots is the perfect tool to answer it. To use it, I reloaded a new instance and took a snapshot by clicking the heap snapshot button at the bottom of the `Profiles` panel. Next, I performed the leaky action a prime number of times in hopes that it might be easy to spot. So I clicked Events 13 times and immediately took a second snapshot. To compare before and after, I highlighted the second snapshot and selected `Comparison` view.

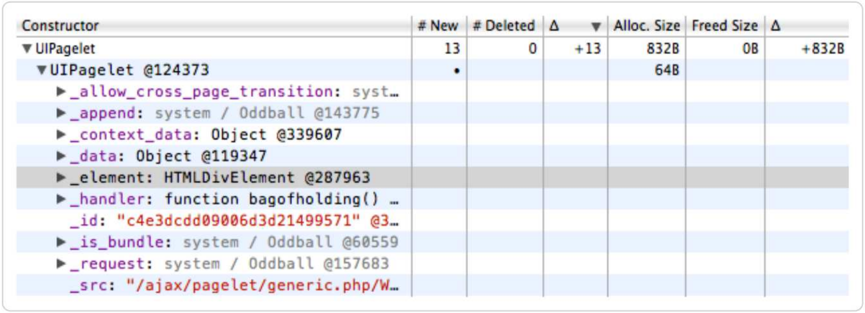


The comparison view displays the difference between any two snapshots. I sorted by delta to look for any objects that grew by the same number of times I clicked: 13. Sure enough, there were 13 more `UIPagelets` on the heap after my clicks than before.



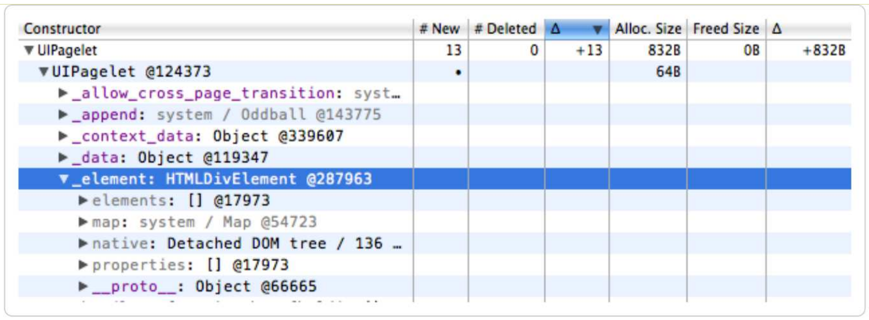
| Constructor | # New | # Deleted | Δ ▼ | Alloc. Size | Freed Size | Δ |
|---|---|---|---|---|---|---|
| ▶ InternalArray | 44 | 21 | +23 | 704B | 336B | +368B |
| ▶ chrome.Event | 28 | 10 | +18 | 560B | 200B | +360B |
| ▶ Arguments | 32 | 17 | +15 | 600B | 320B | +280B |
| ▶ DOMWindow | 28 | 13 | +15 | 760B | 340B | +420B |
| ▶ UIPagelet | 13 | 0 | +13 | 832B | 0B | +832B |
| ▶ HTMLDocument | 21 | 11 | +10 | 420B | 220B | +200B |
| ▶ (number) | 20 | 13 | +7 | 240B | 156B | +84B |
| ▶ Boolean | 10 | 5 | +5 | 160B | 80B | +80B |
| ▶ Date | 13 | 8 | +5 | 208B | 128B | +80B |
| ▶ ErrorPrototype | 10 | 5 | +5 | 280B | 140B | +140B |
| ▶ JSON | 10 | 5 | +5 | 120B | 60B | +60B |
| ▶ MathConstructor | 10 | 5 | +5 | 280B | 140B | +140B |

Expanding the `UIPagelet` shows us each instance. Let's look at the first.



| Constructor | # New | # Deleted | Δ ▼ | Alloc. Size | Freed Size | Δ |
|---|---|---|---|---|---|---|
| ▼ UIPagelet | 13 | 0 | +13 | 832B | 0B | +832B |
| ▼ UIPagelet @124373 | • |  |  | 64B |  |  |
| ▶ _allow_cross_page_transition: syst… |  |  |  |  |  |  |
| ▶ _append: system / Oddball @143775 |  |  |  |  |  |  |
| ▶ _context_data: Object @339607 |  |  |  |  |  |  |
| ▶ _data: Object @119347 |  |  |  |  |  |  |
| ▶ _element: HTMLDivElement @287963 |  |  |  |  |  |  |
| ▶ _handler: function bagofholding() … |  |  |  |  |  |  |
| _id: "c4e3dcdd09006d3d21499571" @3… |  |  |  |  |  |  |
| ▶ _is_bundle: system / Oddball @60559 |  |  |  |  |  |  |
| ▶ _request: system / Oddball @157683 |  |  |  |  |  |  |
| _src: "/ajax/pagelet/generic.php/W… |  |  |  |  |  |  |

Each instance has an `_element` property that points to a DOM node. Expanding that node, we can see that it is part of a detached DOM tree of 136 nodes. This means that 136 nodes are no longer visible in the page, but are being held alive by a JavaScript reference. There are legitimate reasons to do this, but it is also easy and common to do it by accident.

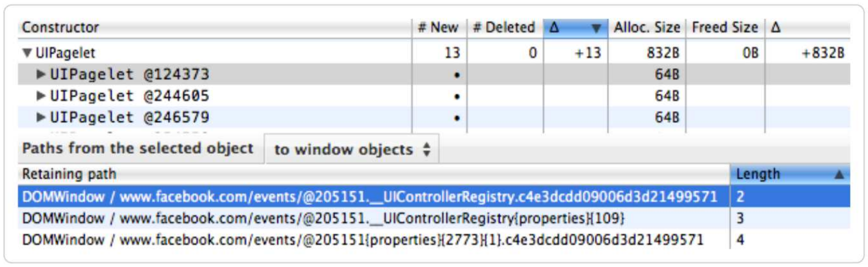


| Constructor | # New | # Deleted | Δ | Alloc. Size | Freed Size | Δ |
|---|---|---|---|---|---|---|
| ▼UIPagelet | 13 | 0 | +13 | 832B | 0B | +832B |
| ▼UIPagelet @124373 | • | | | 64B | | |
| ▶_allow_cross_page_transition: syst… | | | | | | |
| ▶_append: system / Oddball @143775 | | | | | | |
| ▶_context_data: Object @339607 | | | | | | |
| ▶_data: Object @119347 | | | | | | |
| ▼_element: HTMLDivElement @287963 | | | | | | |
| ▶elements: [] @17973 | | | | | | |
| ▶map: system / Map @54723 | | | | | | |
| ▶native: Detached DOM tree / 136 … | | | | | | |
| ▶properties: [] @17973 | | | | | | |
| ▶__proto__: Object @66665 | | | | | | |

Note that all memory statistics reported by the tool reflect only the memory allocated in the JavaScript heap. This does not include native memory used by the DOM objects. So we cannot readily determine how much memory those 136 nodes are using. It all depends on their content -- for example leaking images can burn through memory very quickly.

## Determine what prevents collection

After finding the leaked memory the last question is what is preventing it from being collected. To answer this we simply highlight any node and the retaining path will be shown (I typically change it to show paths to window objects instead of paths to GC roots). Here we see a very simple path. The `UIPagelet`s are stored in a `__UIControllerRegistry` object.



| Constructor | # New | # Deleted | Δ | Alloc. Size | Freed Size | Δ |
|---|---|---|---|---|---|---|
| ▼UIPagelet | 13 | 0 | +13 | 832B | 0B | +832B |
| ▶UIPagelet @124373 | • | | | 64B | | |
| ▶UIPagelet @244605 | • | | | 64B | | |
| ▶UIPagelet @246579 | • | | | 64B | | |

Paths from the selected object: to window objects

| Retaining path | Length |
|---|---|
| DOMWindow / www.facebook.com/events/@205151.__UIControllerRegistry.c4e3dcdd09006d3d21499571 | 2 |
| DOMWindow / www.facebook.com/events/@205151.__UIControllerRegistry[properties][109] | 3 |
| DOMWindow / www.facebook.com/events/@205151[properties][2773][1].c4e3dcdd09006d3d21499571 | 4 |

At first I wondered if this might intentionally keep DOM nodes alive as a cache. However, that doesn't seem to be the case. A search of the source JS shows several places where items are added to the `__UIControllerRegistry`, but I couldn't find anywhere where they are cleaned up. So this appears to be a case where retaining the DOM nodes is purely accidental. The fix is to remove references to these nodes so they may be collected.

## Takeaway

The point of the post is not that facebook has a leak. Facebook is an extremely well engineered site and large apps all deal with memory leaks from time to time. The point is to demonstrate how readily leaks can be diagnosed even with no knowledge of the source.

For anyone with an interactive web site, I highly recommend using your site for a few minutes with the memory timeline enabled to watch for any suspicious growth. If you have to solve any issues, the manual has excellent tutorials.

Posted by Tony Gentilcore at 5:44 AM

## 30 comments:

**daniel15 said...**

Great article! Thanks. :)
This is one of the advantages of Chrome (and IE9+) running every site as a separate process. If a site starts using a large amount of RAM, you can just close the tab and open it again. With other browsers, often you need to restart the whole browser.

August 7, 2011 6:49 PM