JavaScript

Memory Management Masterclass

@addyosmani
+AddyOsmani

DevTools Demos

http://github.com/addyosmani/memory-mysteries

Chrome Task Manager
Memory Timeline
Heap Profiler
Object Allocation Tracker

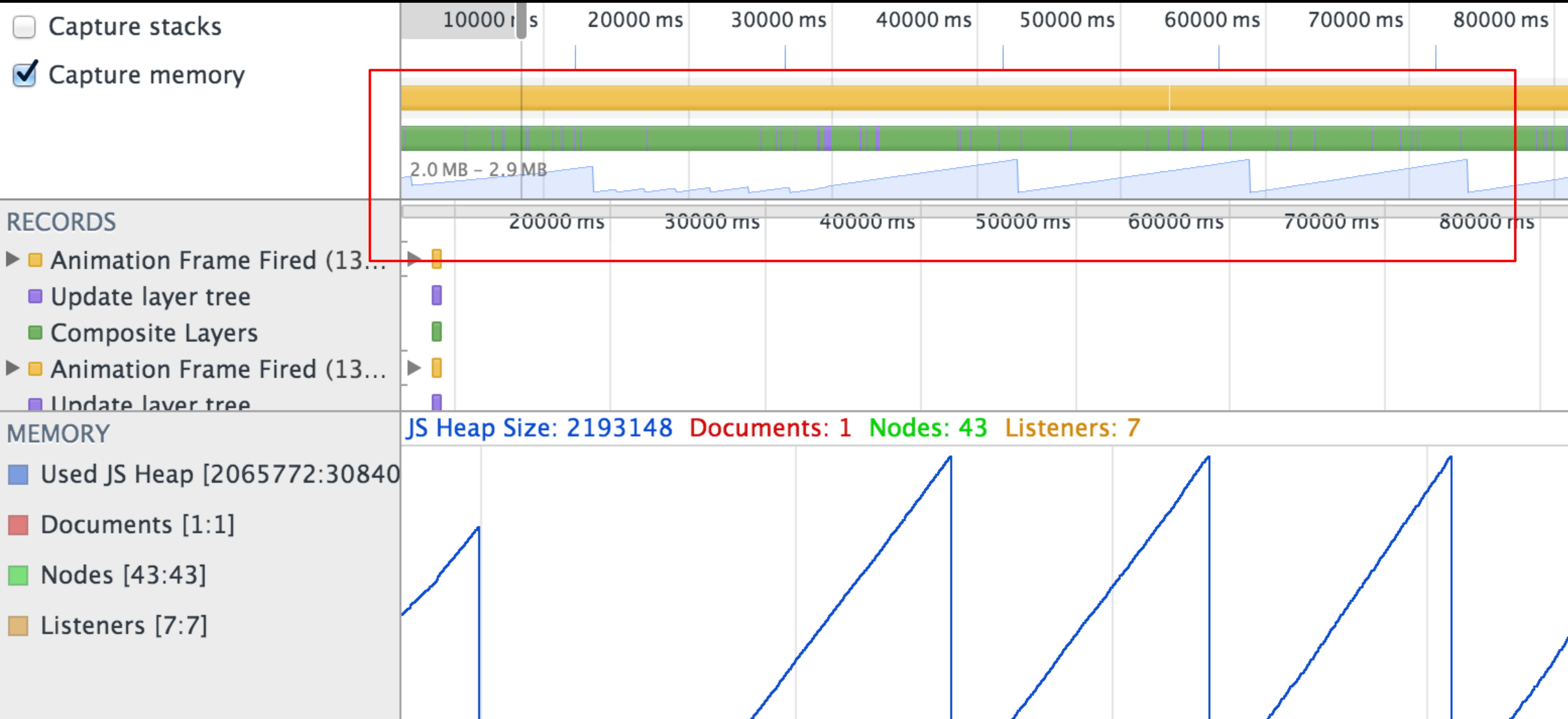
The Sawtooth Curve

If after a few Timeline iterations you see a **sawtooth** shaped graph (in the pane at the top), you are allocating lots of shortly lived objects.

When the chart dips suddenly, it’s an instance when the garbage collector has run, and cleaned up your referenced memory objects.

But if the sequence of actions is **not** expected to result in any retained memory, and the DOM node count does not drop down back to the baseline where you began, you have good reason to suspect there is a leak.

Memory Leak Pattern (sawtooth)



“Do I have a leak?”

1. Check Chrome Task Manager to see if the tab’s memory usage is growing
2. ID the sequence of actions you suspect is leaking
3. Do a Timeline recording and perform those actions
4. Use the Trash icon to force GC. If you don’t objects will be alive in memory until the next GC cycle.
5. If you iterate and see a Sawtooth curve, you’re allocating lots of short life objects. If the sequence of actions is not expected to retain memory and your DOM node count doesn’t drop - you may have a leak.
6. Use the Object Allocation Tracker to narrow down the cause of the leak. It takes heap snapshots periodically through the recording.

V8’s Hidden Classes

V8’s optimizing compiler makes many assumptions about your code. Behind the scenes, it creates hidden classes representing objects.

Using these hidden classes, V8 works much faster. If you **delete** properties, these assumptions may no longer be valid and code can be de-optimized slowing it down.

That said, **delete** has a purpose in JS and is used in plenty of libraries. The takeaway is to avoid modifying the structure of hot objects at runtime. Engines like V8 can detect such “hot” objects and attempt to optimize them.

Accidental de-optimization

Take care with the *delete* keyword

"o" becomes a SLOW object.

It's better to set "o" to "null".

Only when the **last** reference to an object is removed does that object get eligible for collection.

```
var o = {x: "y"};
delete o.x;
o.x; // undefined

var o = {x: "y"};
o = null;
o.x; // TypeError
```

Fast object

```
function FastPurchase(units, price) {
    this.units = units;
    this.price = price;
    this.total = 0;
    this.x = 1;
}
var fast = new FastPurchase(3, 25);
```

“fast” objects are faster

Slow object

```
function SlowPurchase(units, price) {
    this.units = units;
    this.price = price;
    this.total = 0;
    this.x = 1;
}
var slow = new SlowPurchase(3, 25);
//x property is useless
//so I delete it
delete slow.x;
```

“slow” should be using a smaller memory footprint than “fast” (1 less property), shouldn”t it?

Reality: “Slow” uses 15 times more memory

Constructor	Distance	Objects Count		Shallow Size		Retained Size ▼	
▶ SlowPurchase	3	300 001	31 %	3 600 012	3 %	127 200 104	89 %
▶ FastPurchase	3	300 001	31 %	8 400 012	6 %	8 400 104	6 %

Closures

Closures are powerful. They enable inner functions to retain access to an outer function’s variables even after the outer function returns.

Unfortunately, they’re also excellent at hiding circular references between JavaScript objects and DOM objects. Make sure to understand what references are retained in your closures.

The inner function may need to still access all variables from the outer one, so as long as a reference to it exists, variables from the outer function can’t be GC’d and continue to consume memory after it’s done invoking.

Closures

Closures can be a source of memory leaks too. Understand what references are retained in the closure.

```
var a = function () {
  var largeStr = new Array(1000000).join('x');
  return function () {
     return largeStr;
  };
}();

var a = function () {
    var smallStr = 'x',
           largeStr = new Array(1000000).join('x');
   return function (n) {
      return smallStr;
   };
}();

var a = (function() { // `a` will be set to the return of this function
    var smallStr = 'x', largeStr = new Array(1000000).join('x');
    return function(n) {
         // which is another function; creating a closure
        eval('');
        return smallStr;
    };
}());
```

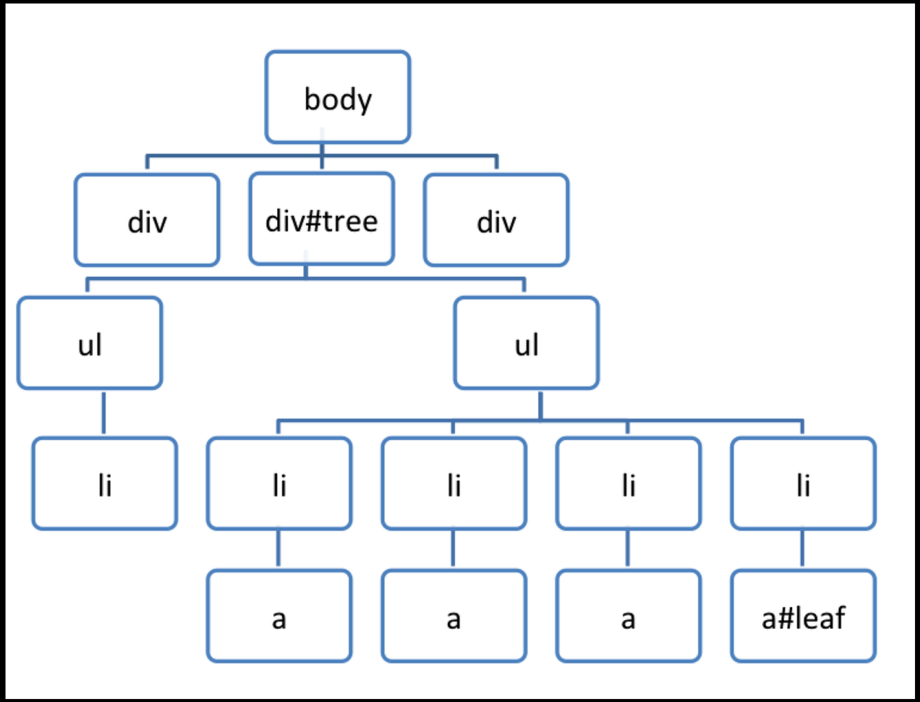
DOM Leaks

DOM leaks usually occur when an element gets appended to the DOM, additional elements are appended to the first element and then the original element is removed from the DOM without removing the secondary elements.

In the next example, #leaf maintains a reference to its parentNode and recursively maintains references up to #tree. It's only when leafRef is nullified is the entire tree under #tree a candidate to be garbage collected.

DOM Leaks.

When is #tree GC'd?




```
var select = document.querySelector;
var treeRef = select("#tree");
var leafRef = select("#leaf");
var body = select("body");
body.removeChild(treeRef);

//#tree can't be GC yet due to treeRef
//let’s fix that:
treeRef = null;

//#tree can't be GC yet, due to
//indirect reference from leafRef

leafRef = null;
//NOW can be #tree GC
```

Timers

Timers are a common source of memory leaks.

Anything you're repetitively doing in a timer should ensure it isn't maintaining refs to DOM objects that could accumulate leaks if they can be GC'd.

If we run this loop..
This introduces a memory leak:

```
for (var i = 0; i < 90000; i++) {
    var buggyObject = {
        callAgain: function() {
            var ref = this;
            var val = setTimeout(function() {
                ref.callAgain();
            }, 90000);
        }
    }

    buggyObject.callAgain();
    buggyObject = null;
}
```

ES6 WeakMaps

WeakMaps help us avoid memory leaks by holding references to properties weakly. If a WeakMap is the only objects with a reference to another object, the GC may collect the referenced object.

In the next example, *Person* is a closure storing private data as a **strong** reference. The garbage collector can collect an object if there are only weak or no references to it.

WeakMaps hold keys weakly so the *Person* instance and its private data are eligible for garbage collection when a *Person* object is no longer referenced by the rest of the app.

ES6 WeakMaps

```
var Person = (function() {

   var privateData = {}, // strong reference
        privateId = 0;

   function Person(name) {
      Object.defineProperty(this, "_id", { value:
privateId++ });

      privateData[this._id] = {
         name: name
      };
   }

   Person.prototype.getName = function() {
      return privateData[this._id].name;
   };

   return Person;
}());
```

Avoid memory leaks by holding refs to properties weakly.

```
var Person = (function() {

   var privateData = new WeakMap();

   function Person(name) {
      privateData.set(this, { name: name });
   }

   Person.prototype.getName = function() {
      return privateData.get(this).name;
   };

   return Person;
}());
```

Cheat sheet


cheats?!

Design first.
Code from the design.
Then profile the result.

Optimize at the right time.

> *“Premature optimization is the root of all evil.”*

Donald Knuth

Memory Checklist

Memory Checklist

- Is my app using too much memory?

Timeline memory view and Chrome task manager can help you identify if you're using too much memory. Memory view can track the number of live DOM nodes, documents and JS event listeners in the inspected render process.

Memory Checklist

- Is my app using too much memory?
- Is my app free of memory leaks?

The Object Allocation Tracker can help you narrow down leaks by looking at JS object allocation in real-time. You can also use the heap profiler to take JS heap snapshots, analyze memory graphs and compare snapshots to discover what objects are not being cleaned up by garbage collection.

Memory Checklist

- Is my app using too much memory?
- Is my app free of memory leaks?
- How frequently is my app forcing garbage collection?

If you are GCing frequently, you may be allocating too frequently. The Timeline memory view can help you identify pauses of interest.

Good rules to follow

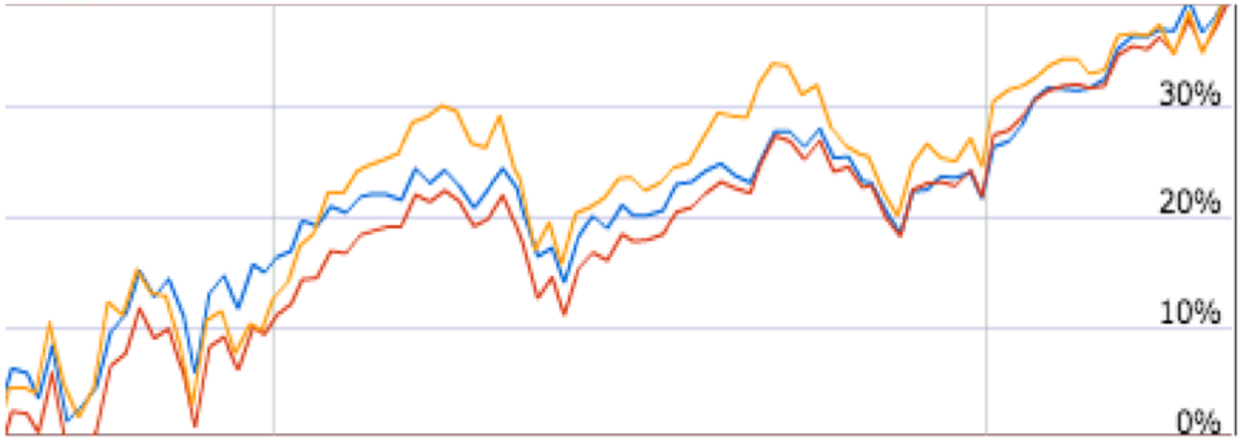
- Avoid long-lasting refs to DOM elements you no longer need
- Avoid circular object references
- Use appropriate scope
- Unbind event listeners that aren’t needed anymore
- Manage local cache of data. Use an aging mechanism to get rid of old objects.

V8 Deep Dive.

Why does #perfmatter?

Silky smooth apps.

Longer battery life
Smoother interactions
Apps can live longer



Nothing is free.

Tools > Task Manager

Task Manager – Google Chrome

Task	Memory	CPU	Network	Process ID
GPU Process	419 MB	0.0	N/A	46802
Browser	175 MB	1.7	0	46799
Tab: WebGL Water	160 MB	0.0	0	46844
Tab: Eye texture raytracing demo	160 MB	0.0	0	46840
Tab: Chrome	148 MB	0.2	0	46812
Tab: Circles	147 MB	0.6	0	46933

You will always pay a price for the resources you use.

JavaScript Execution Time

Google apps

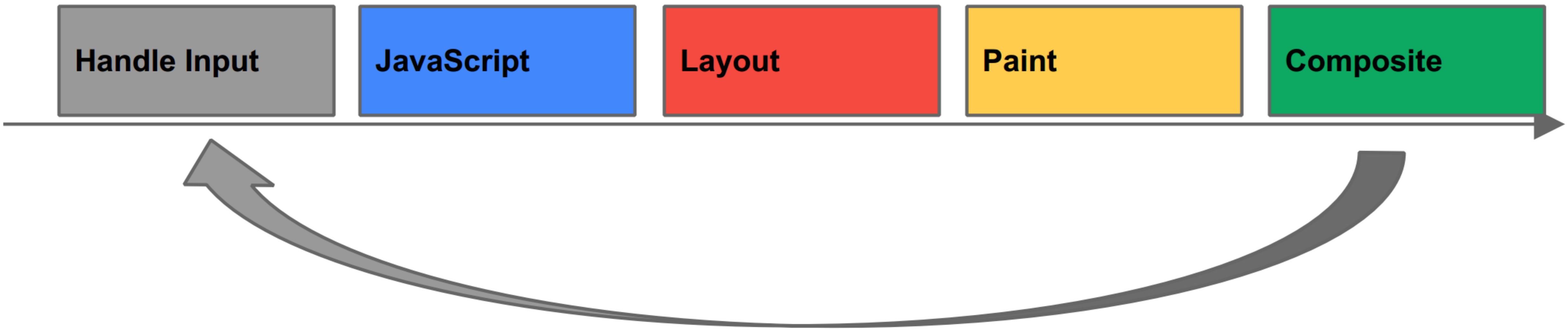
50-70% of time in V8

Popular sites

20-40% of time in V8

16ms to do everything.

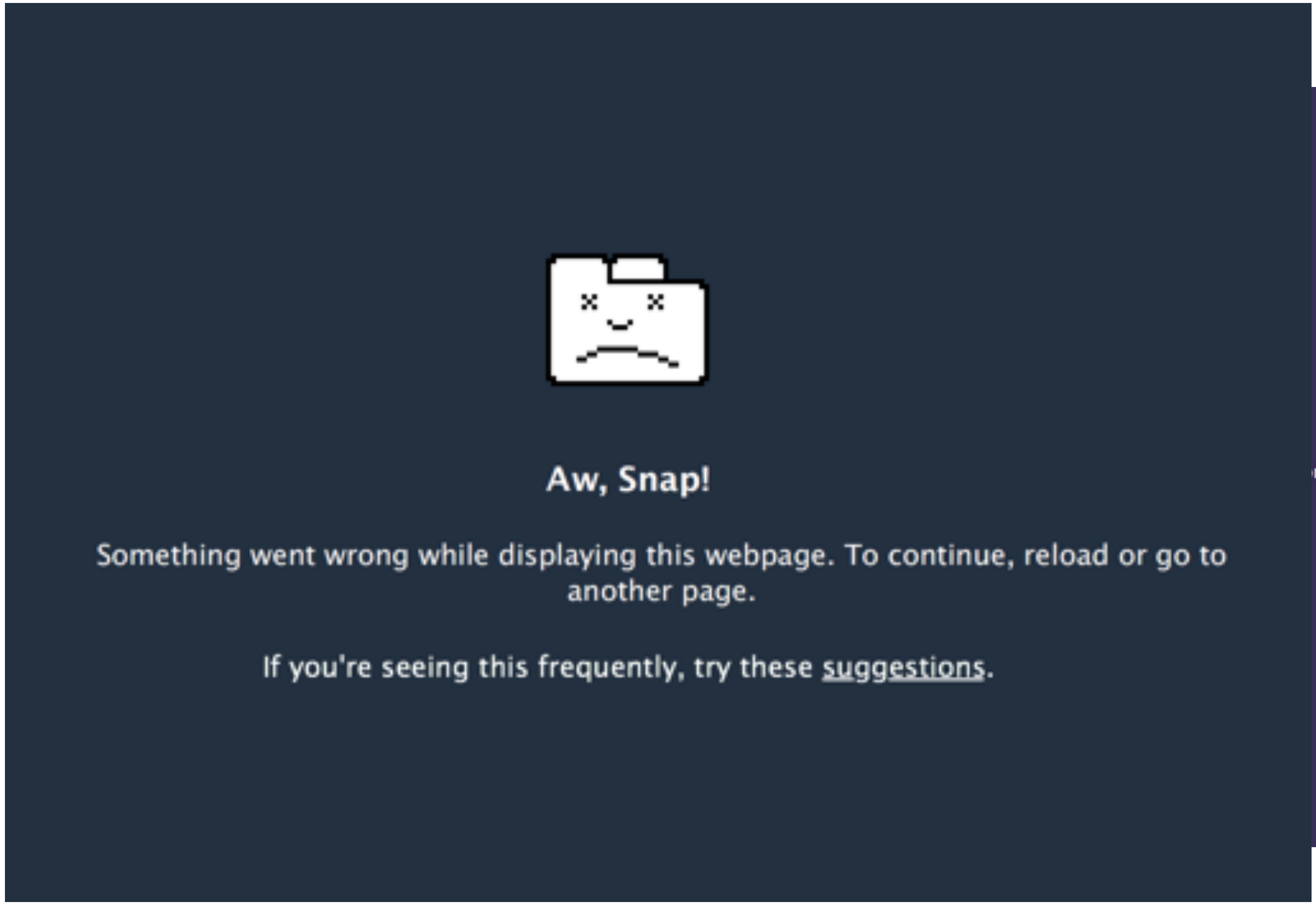
Workload for a frame:

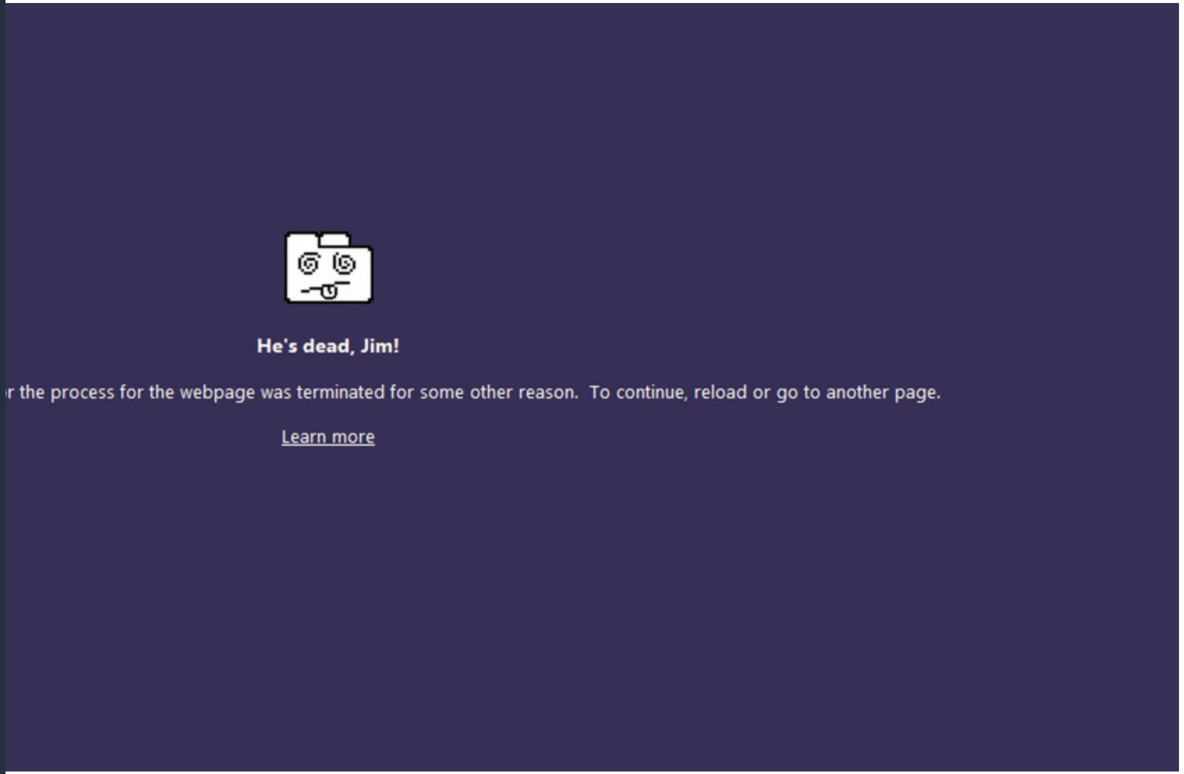


Miss it and you'll see...

JANK

Blow memory & users will be sad.





Performance vs. Memory

My app's tab is using a gig of RAM. #worstDayEver

So what? You've got 32GB on your machine!

Yeah, but my grandma's Chromebook only has 4GB. #stillSad

When it comes down to the age-old *performance vs. memory* tradeoff, we usually opt for **performance**.

Memory management basics

Core Concepts

- What types of values are there?
- How are values organized in memory?
- What is garbage?
- What is a leak?

With thanks to John Mccutchan & Loreena Lee

Four primitive types

- **boolean**
  - true or false
- **number**
  - double precision IEEE 754 number
  - 3.14159
- **string**
  - UTF-16 string
  - “Bruce Wayne”
- **objects**
  - key value maps

Always leafs or terminating nodes.

An object.

```
object[key] = value;
```

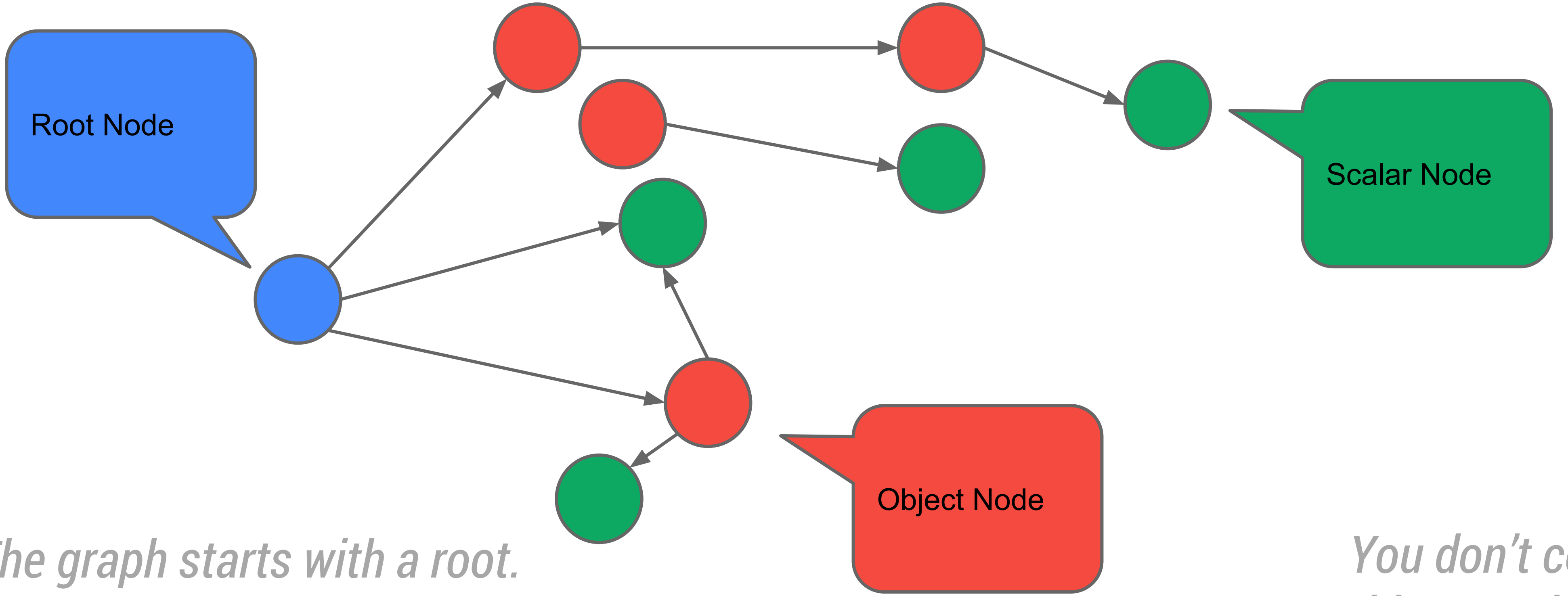
String only

Any variable type

Think of memory as a graph

The value graph

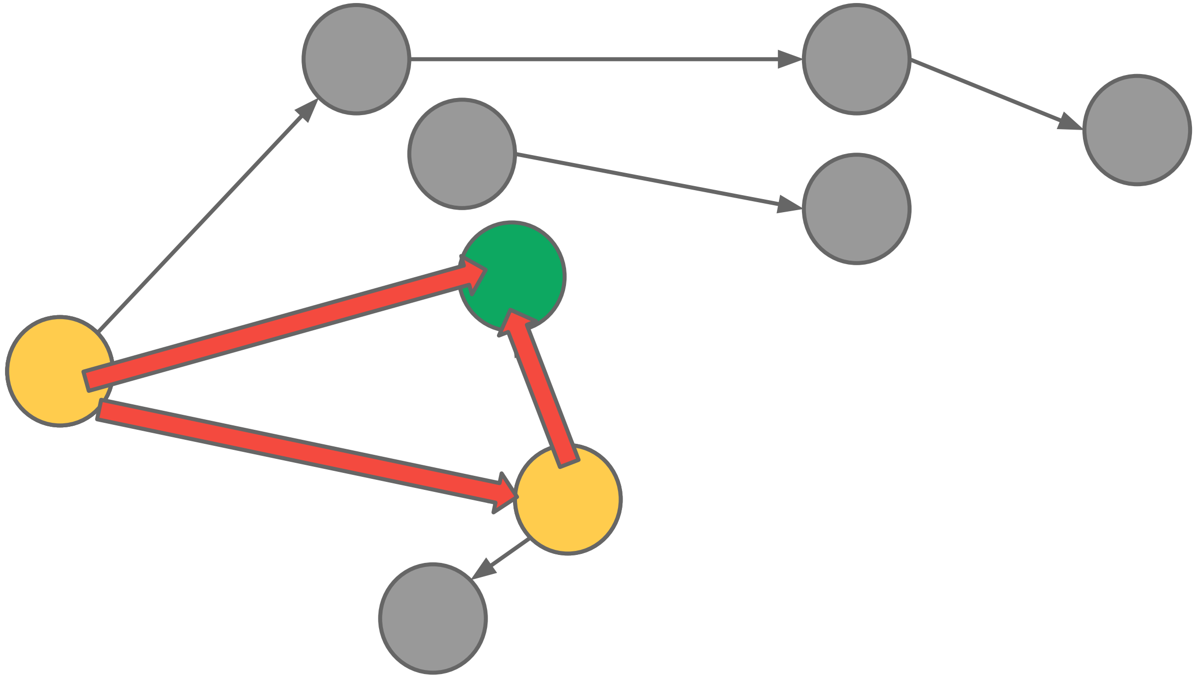
Root could be browser "window" or Global object of a Node module.



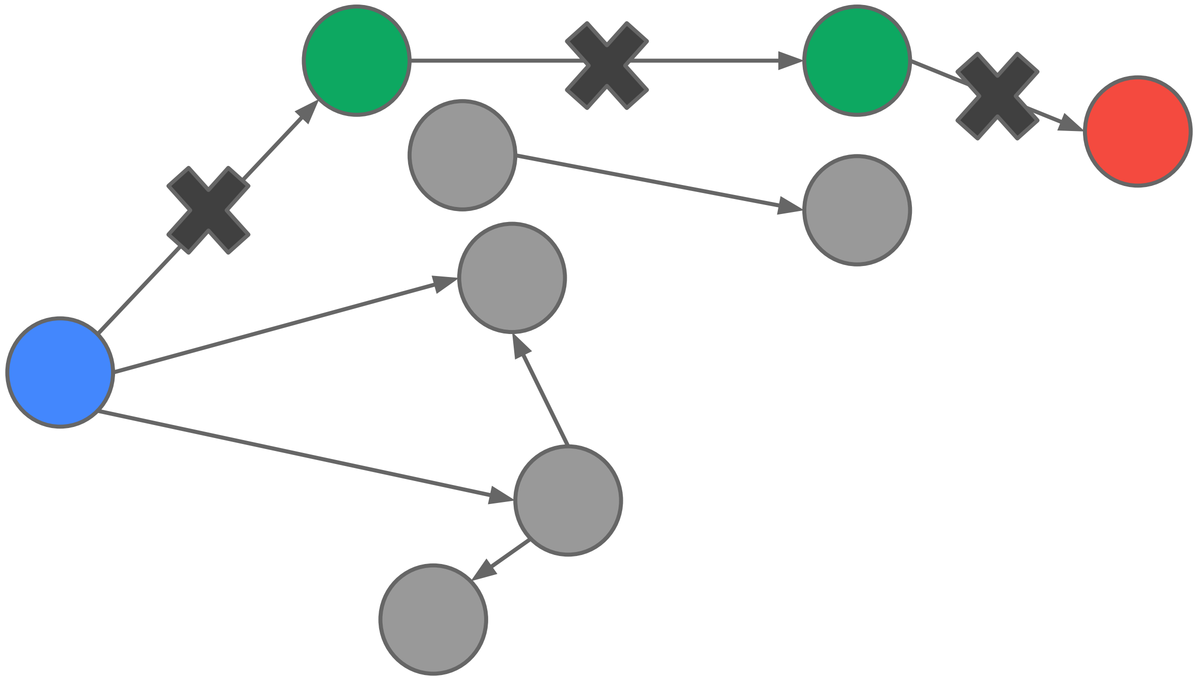
The graph starts with a root.

You don't control how this root object is GC.

A value's retaining path(s)

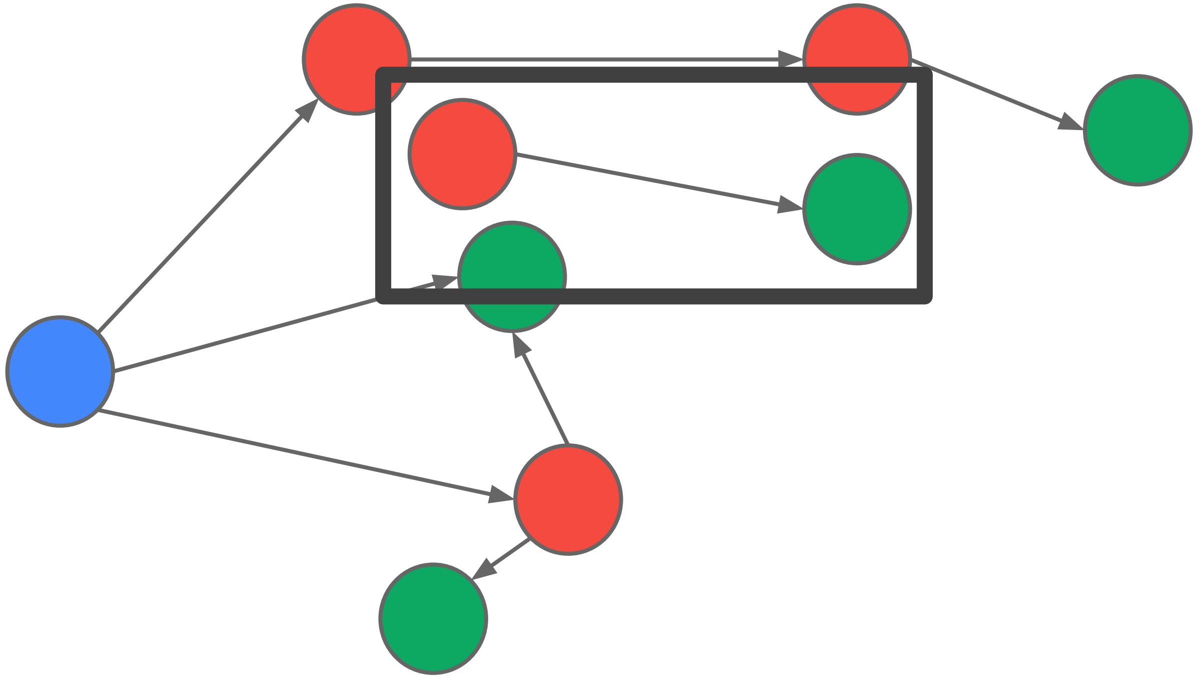


Removing a value from the graph



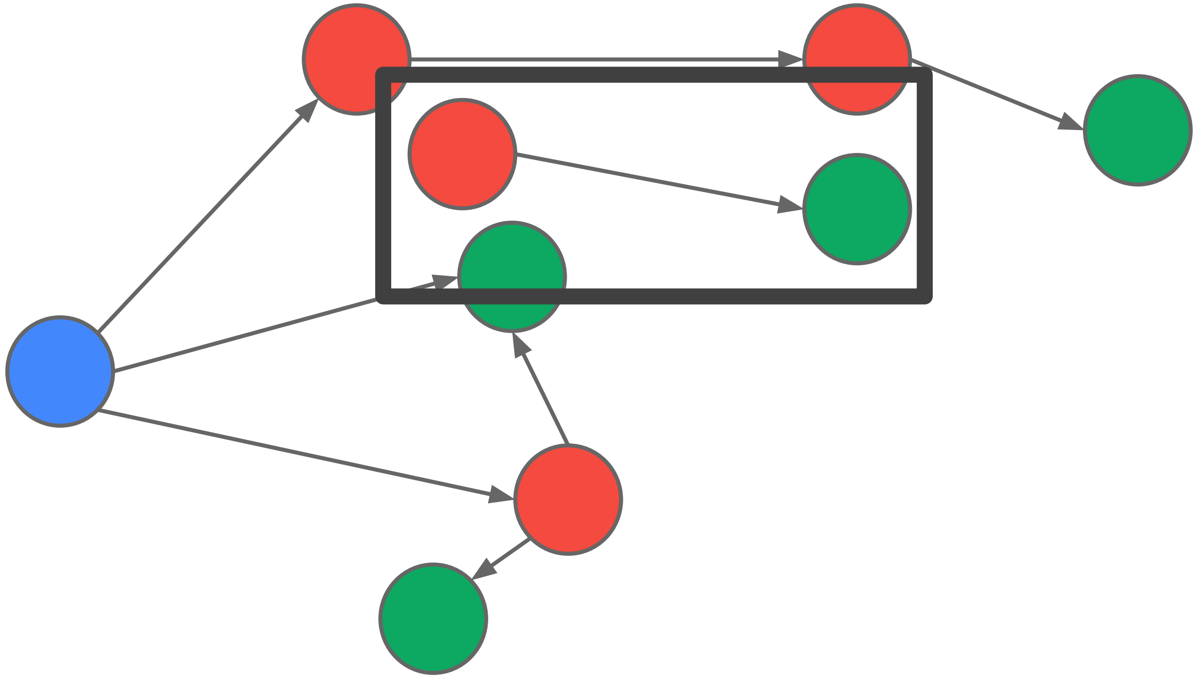
What is garbage?

- Garbage: All values which cannot be reached from the root node.

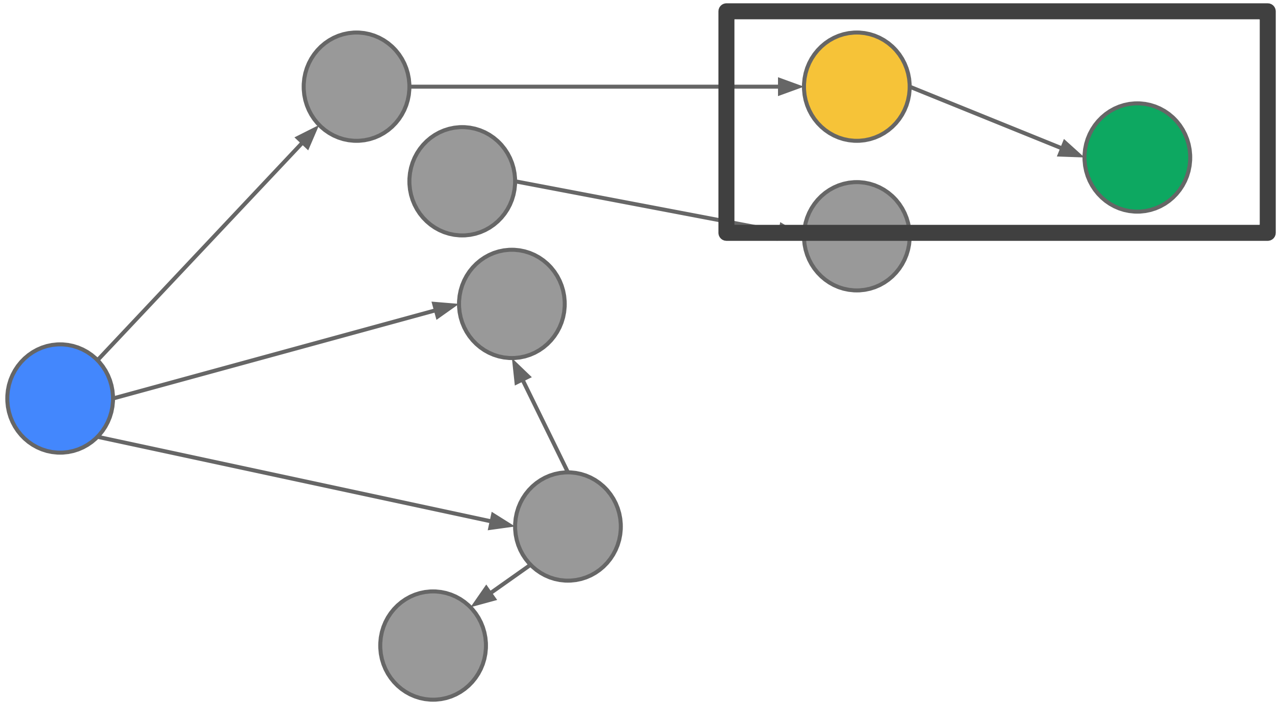


What is garbage collection?

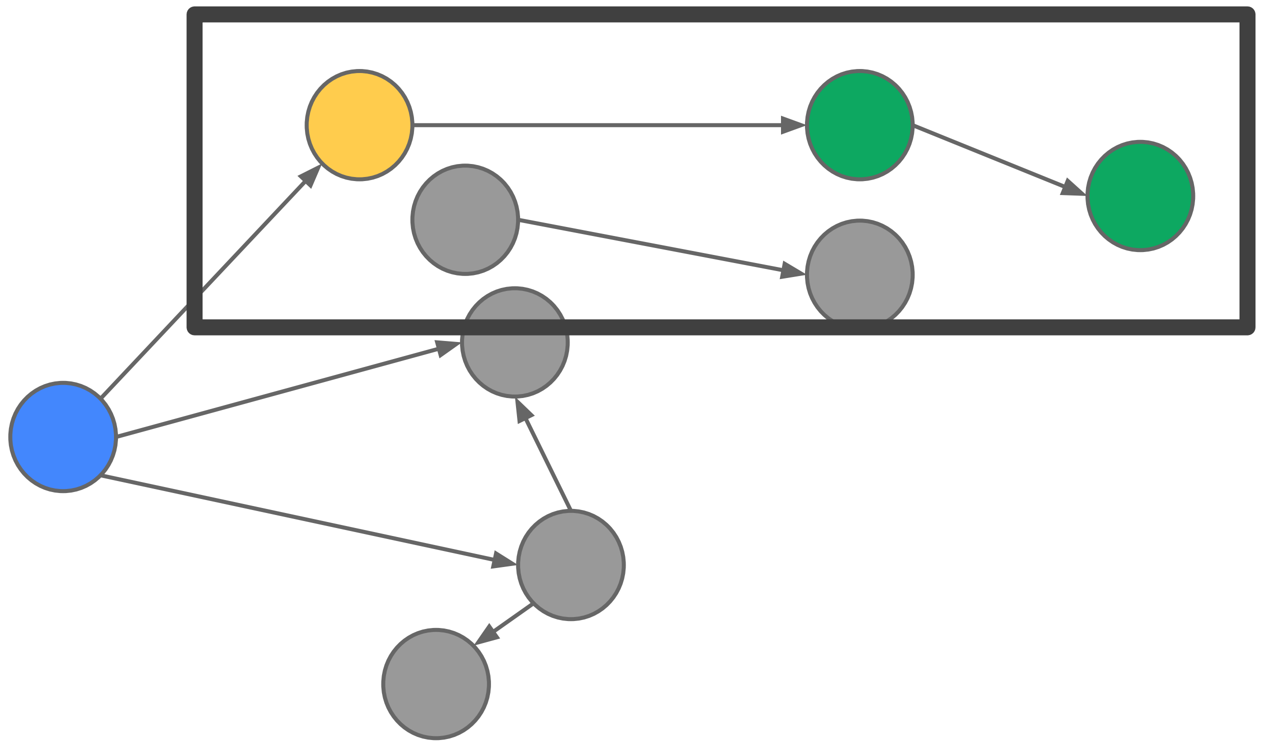
1. Find all live values
2. Return memory used by dead values to system



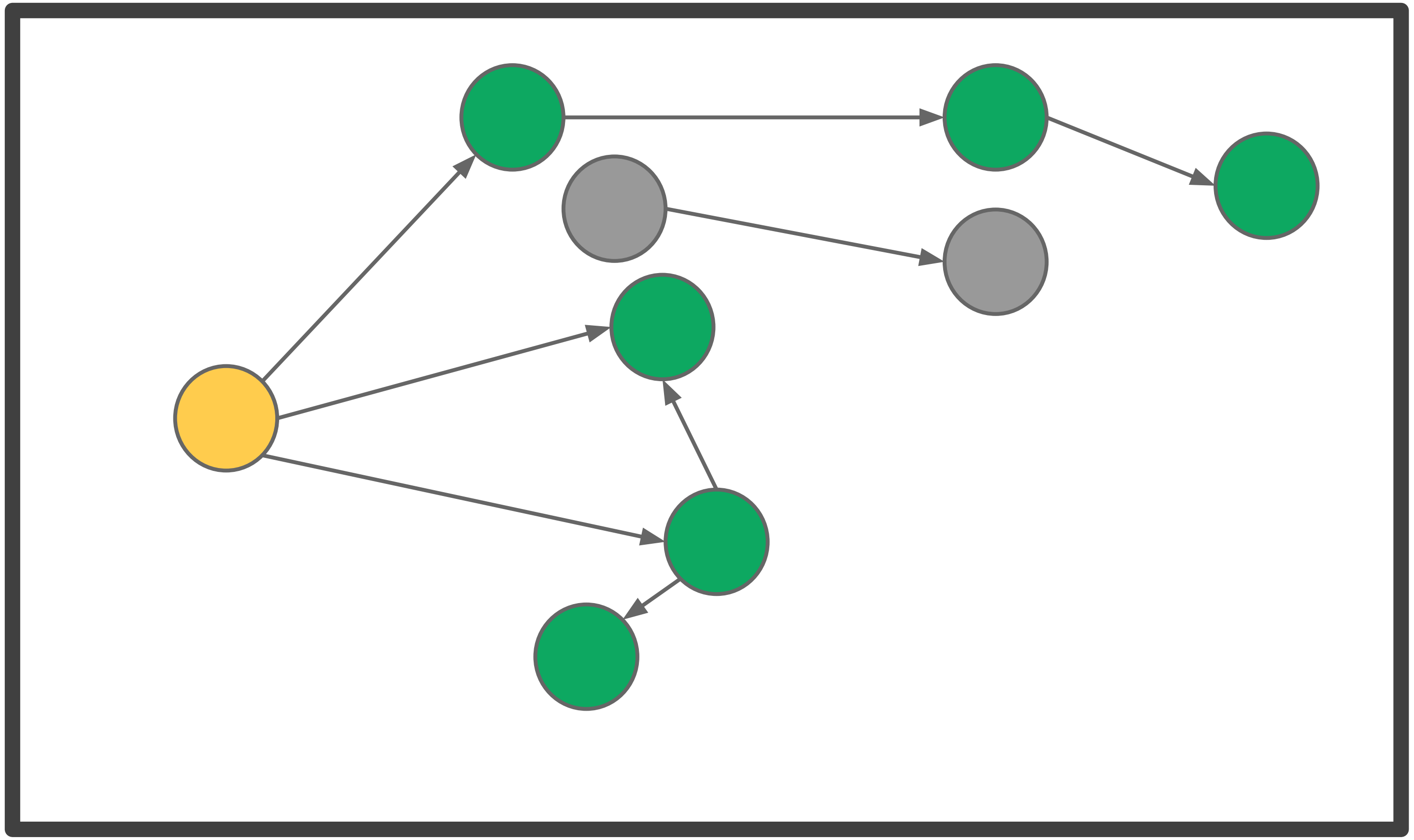
A value's retained size



A value's retained size



A value's retained size



What is a memory leak?

“Gradual loss of available computer memory”

When a program repeatedly fails to return memory obtained for temporary use.

Leaks in JavaScript

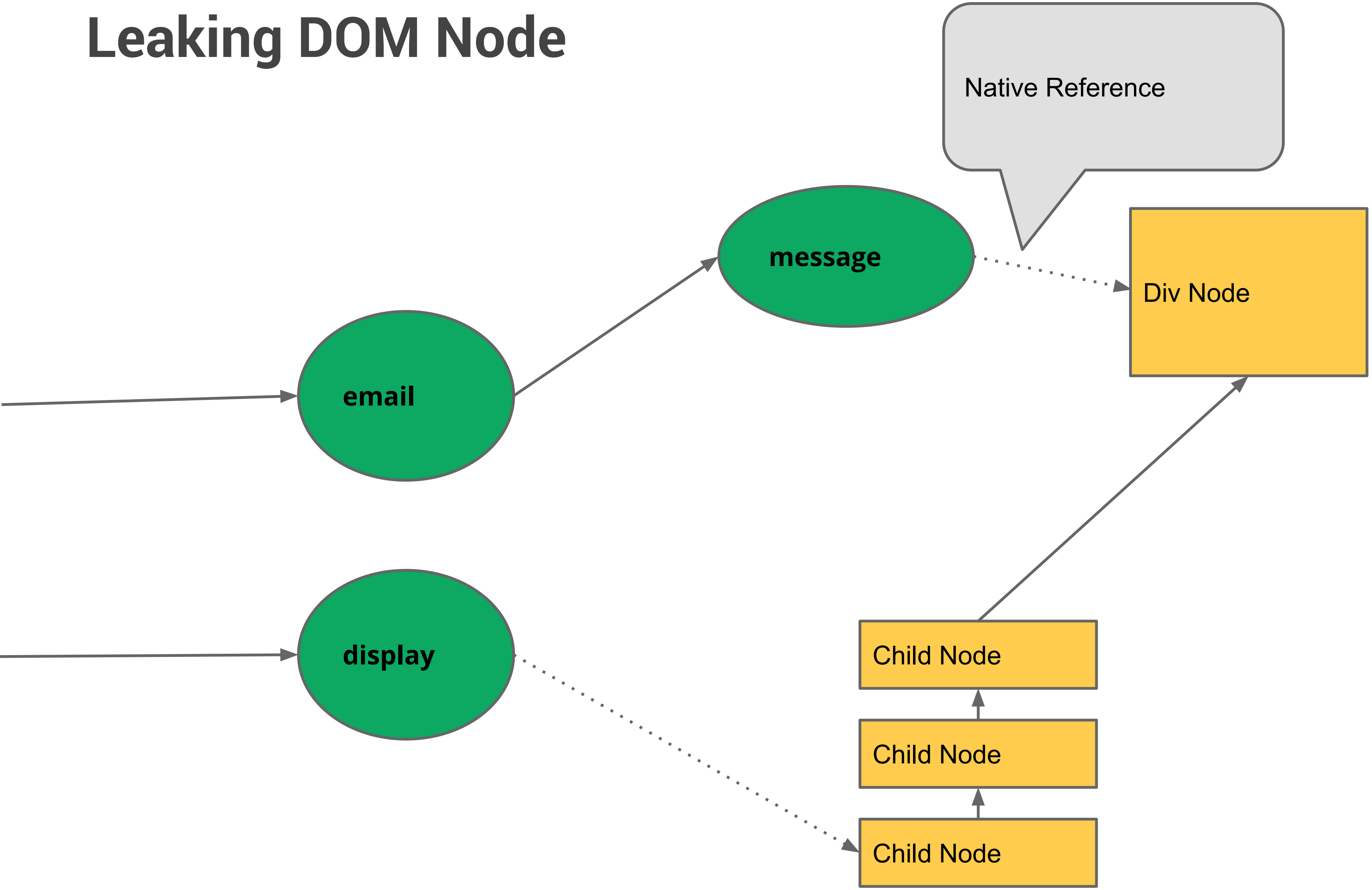
- A value that erroneously still has a retaining path
  - Programmer error

JavaScript

```
email.message = document.createElement("div");

display.appendChild(email.message);
```

Leaking DOM Node



Leaks in JavaScript

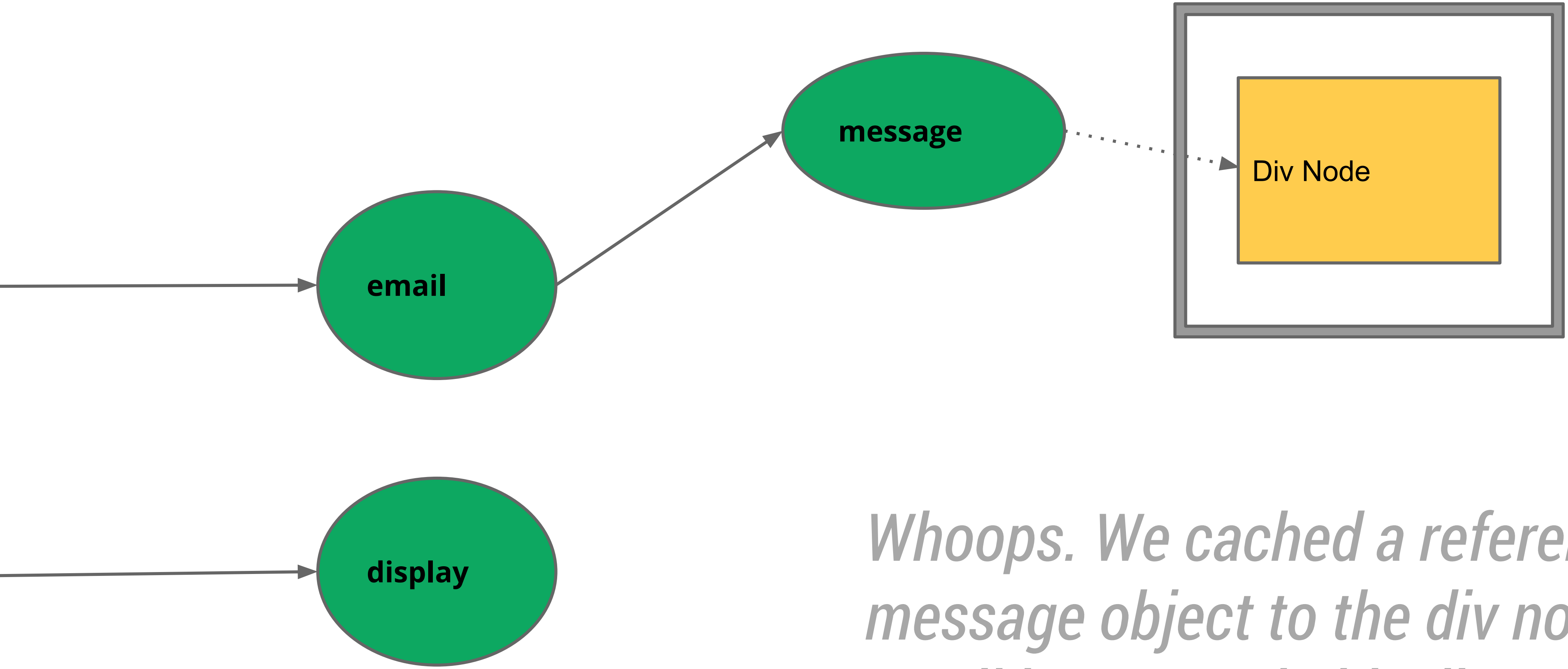
Are all the div nodes actually gone?

JavaScript

```
// ...

display.removeAllChildren();
```

Leaking DOM Node



Whoops. We cached a reference from the message object to the div node. Until the email is removed, this div node will be pinned in memory and we've leaked it.

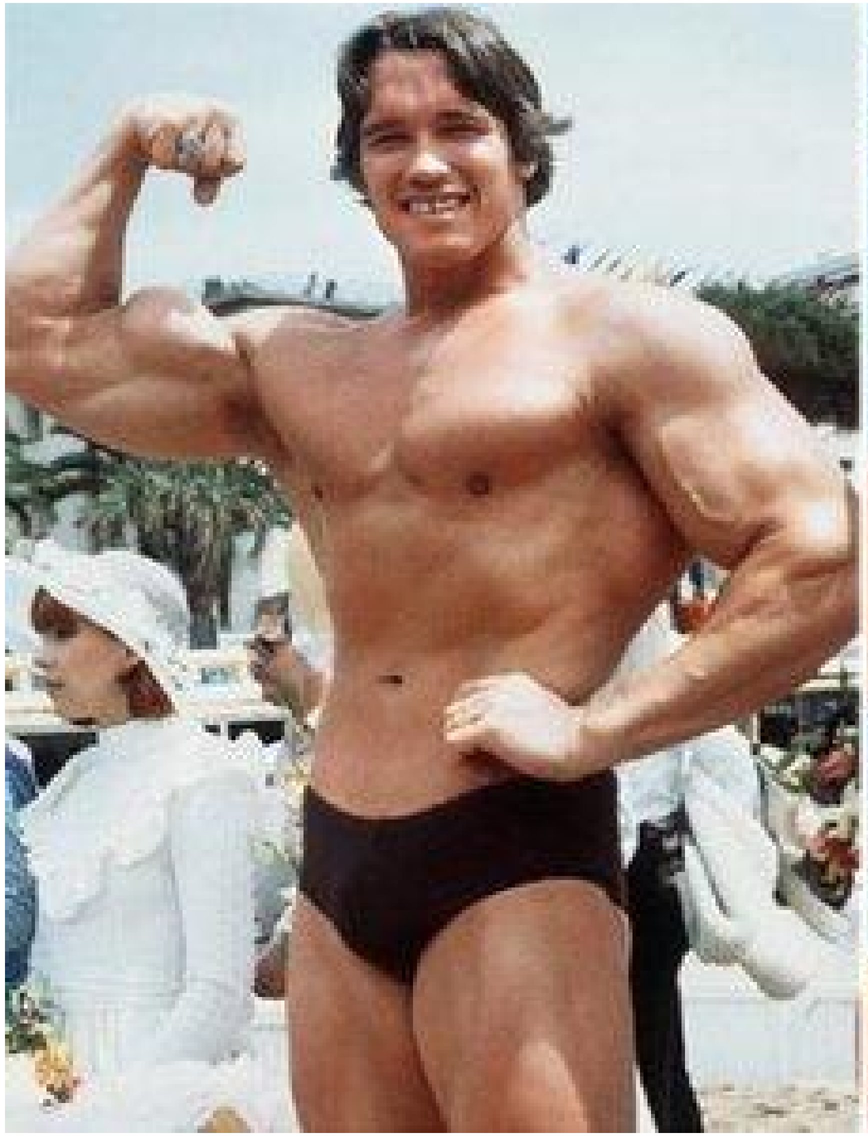
Memory Management Basics

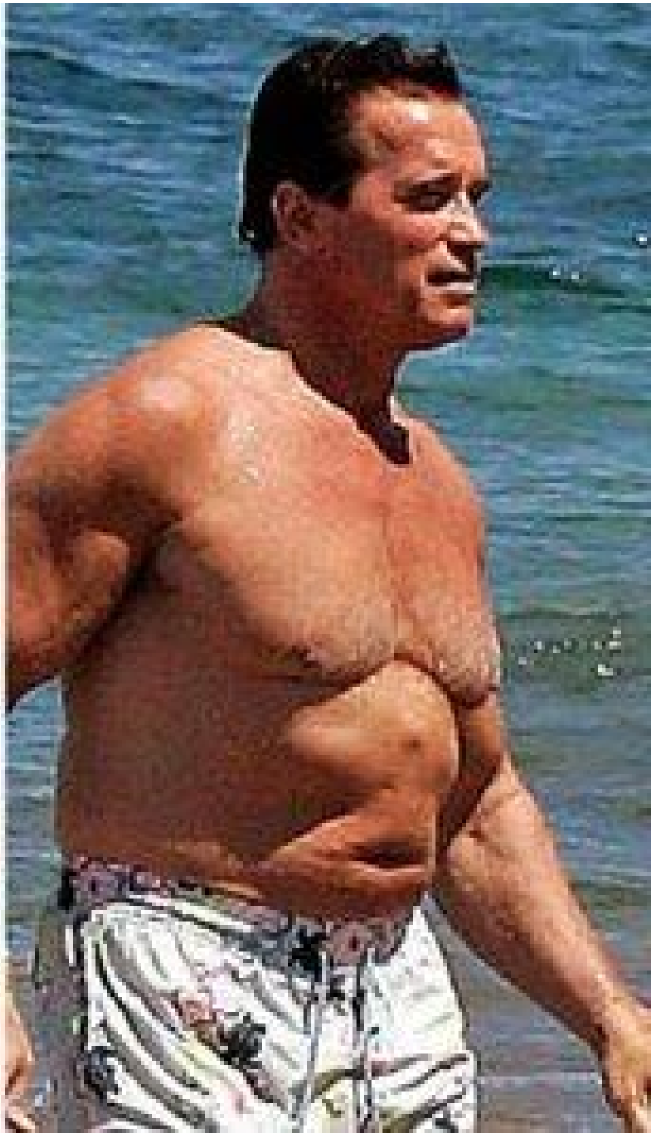
- Values are organized in a graph
- Values have retaining path(s)
- Values have retained size(s)

V8 memory management

Where is the cost in allocating memory?

- Every call to new or implicit memory allocation
  - Reserves memory for object
  - Cheap until...
- Memory pool **exhausted**
  - Runtime forced to perform a garbage collection
  - Can take milliseconds (!)
- Applications must be careful with object allocation patterns
  - Every allocation brings you closer to a GC pause





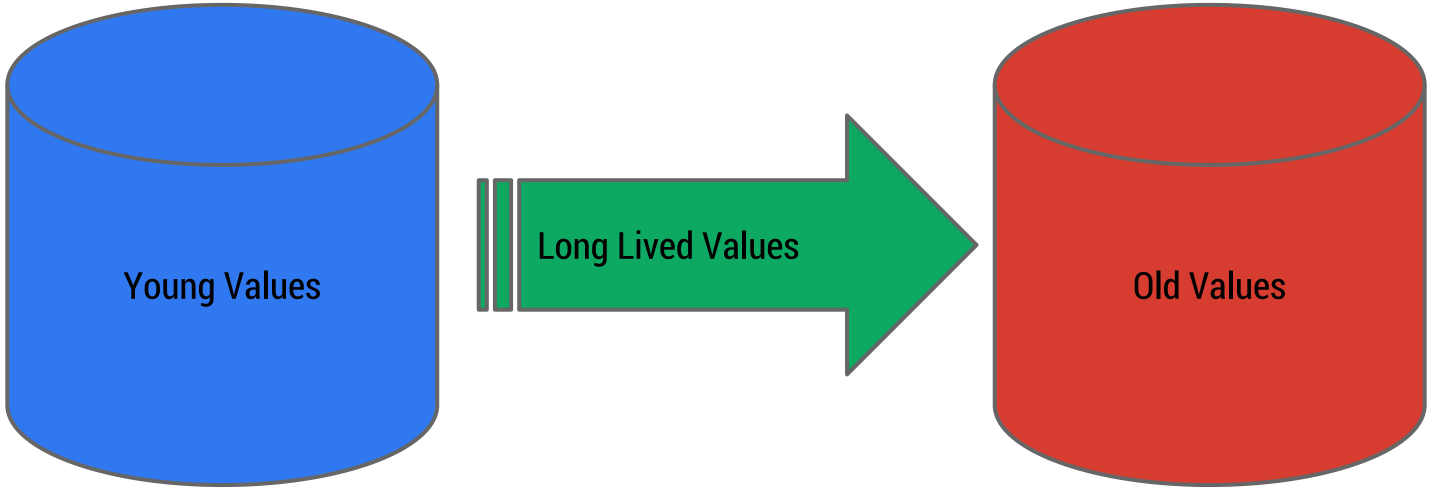
Young generation *Old generation*

How does V8 manage memory?

- Generational
  - Split values between young and old
  - Overtime young values promoted to old

By young and old we mean how long has the JS value existed for.

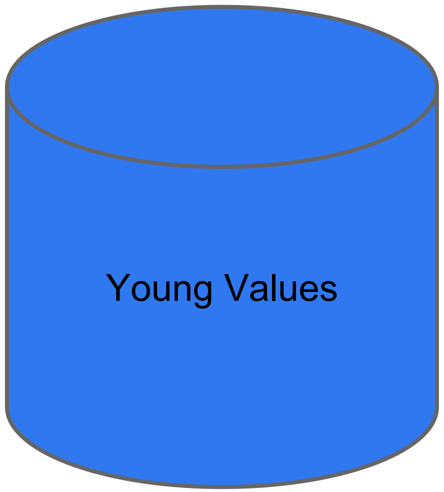
After a few garbage collections, if the value survives (i.e there's a retaining path) eventually it gets promoted to the old generation.

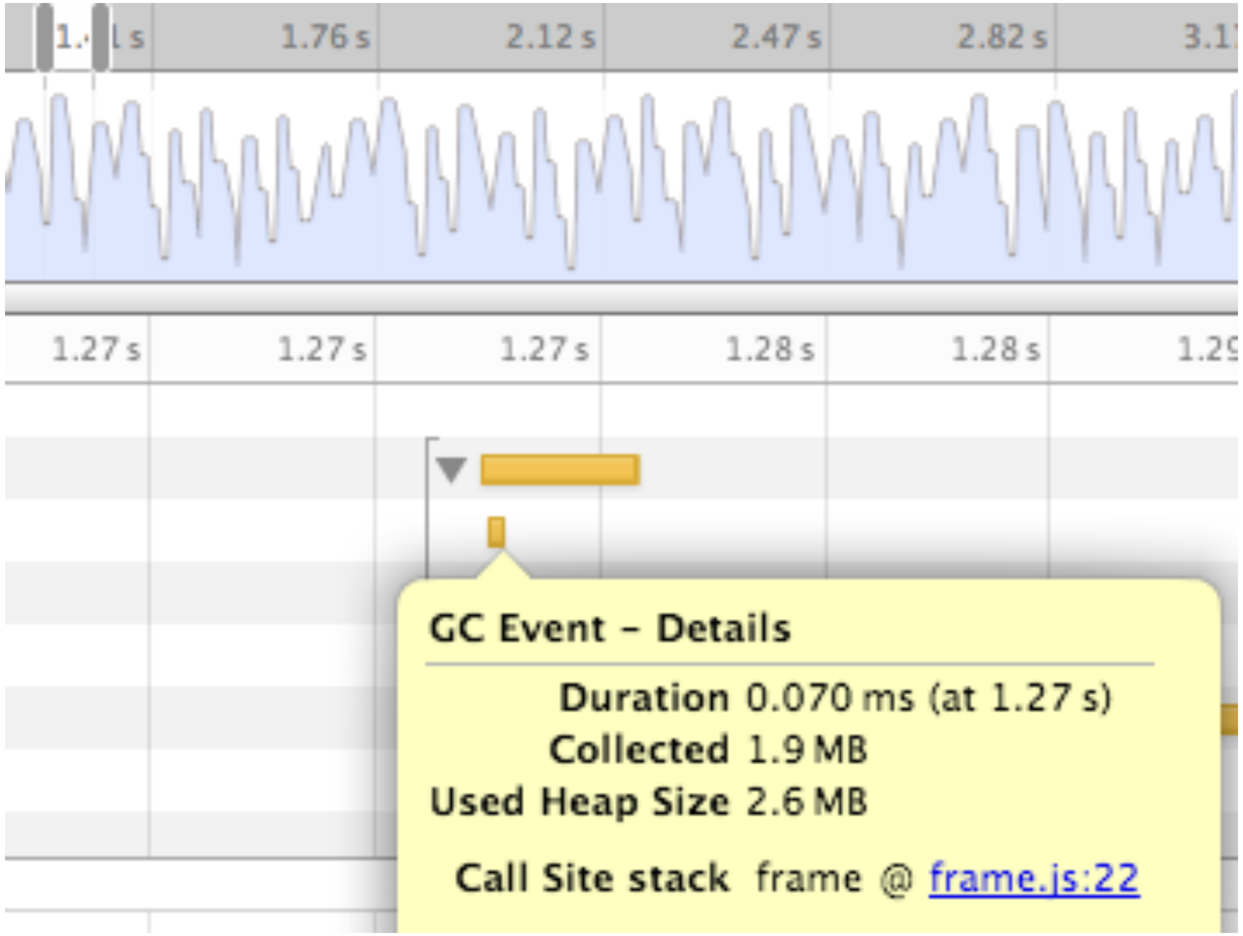


How does V8 manage memory?

DevTools Timeline shows the GC event on it. Below is a young generation collection.

- **Young Generation**
  - Fast allocation
  - Fast collection
  - Frequent collection





How does V8 manage memory?

Some of the old generation's collection occurs in parallel with your page's execution.

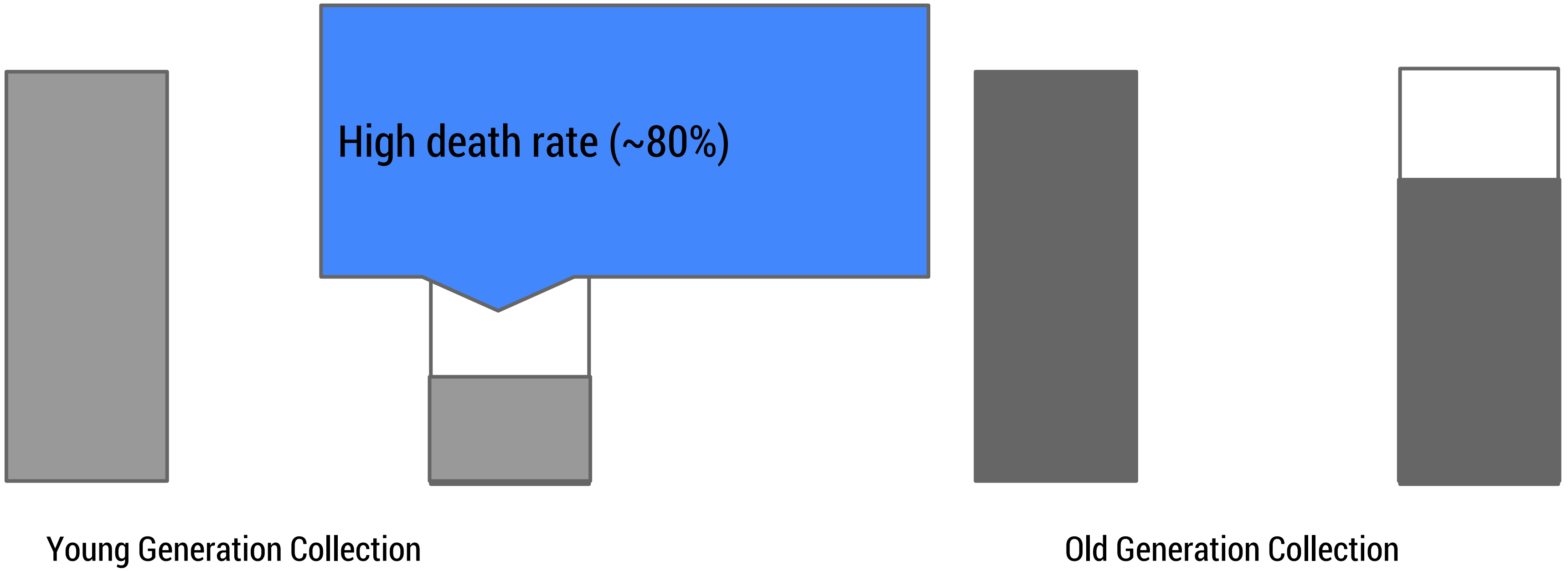
- Old Generation
  - Fast allocation
  - Slower collection
  - **Infrequently** collected

- Parts of collection run concurrently with mutator
  - Incremental Marking
- Mark-sweep
  - Return memory to system
- Mark-compact
  - Move values

How does V8 manage memory?

After GC, most values in the young generation don't make it. They have no retaining path because they were used briefly and they're gone.

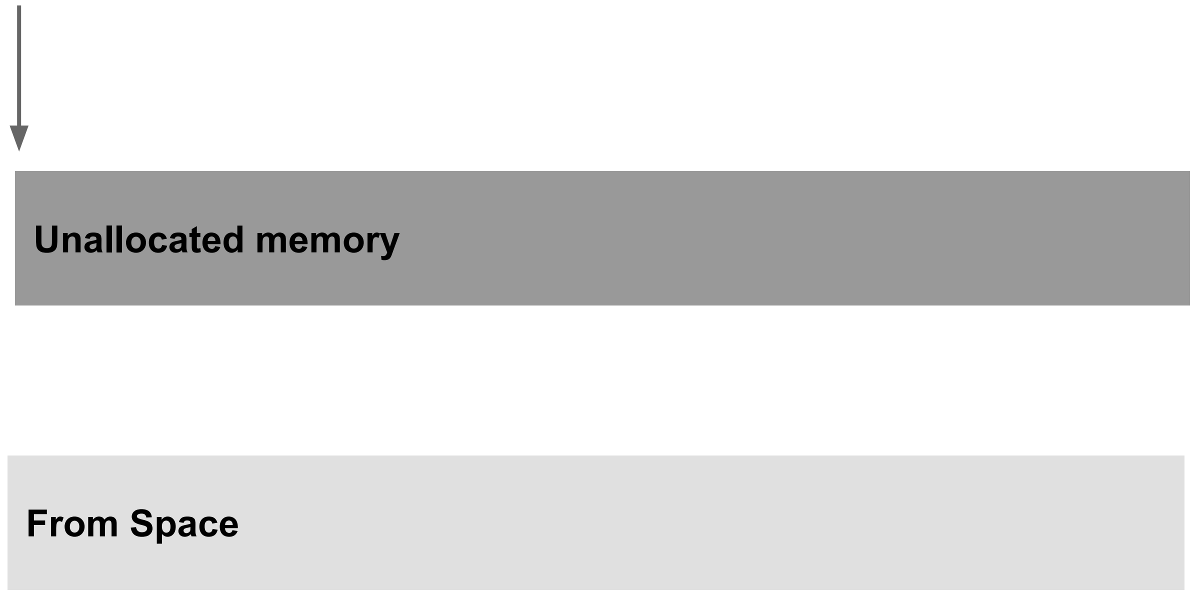
- Why is collecting the young generation faster
  - Cost of GC is proportional to the number of live objects



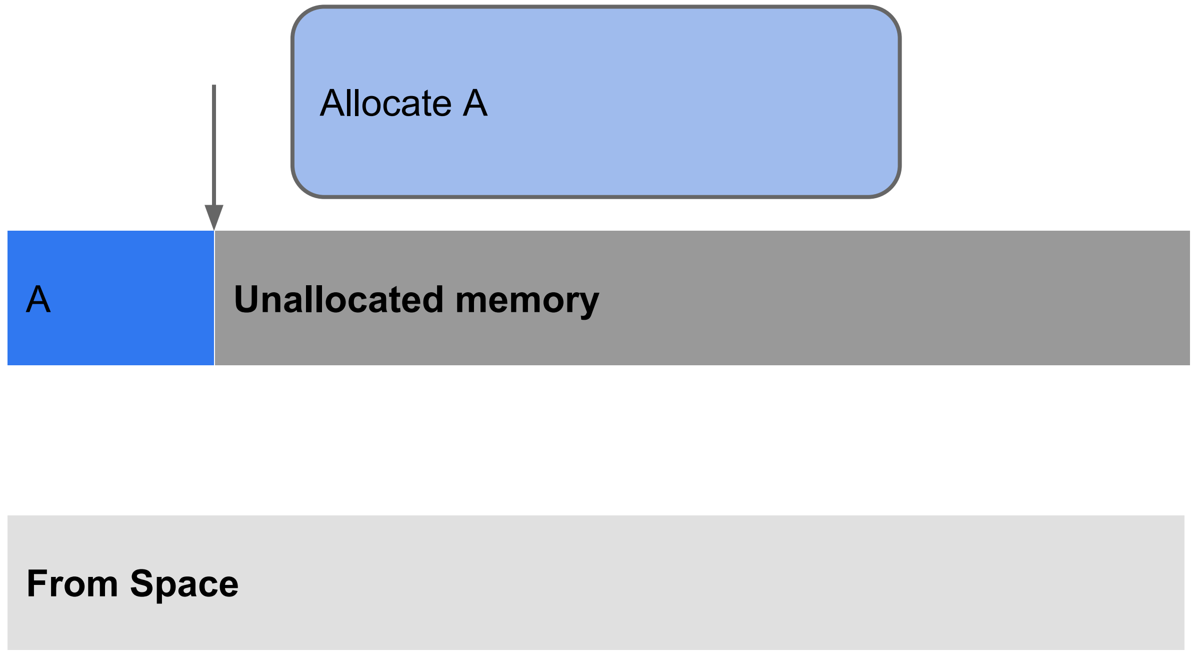
Young Generation In Action

Young Generation In Action

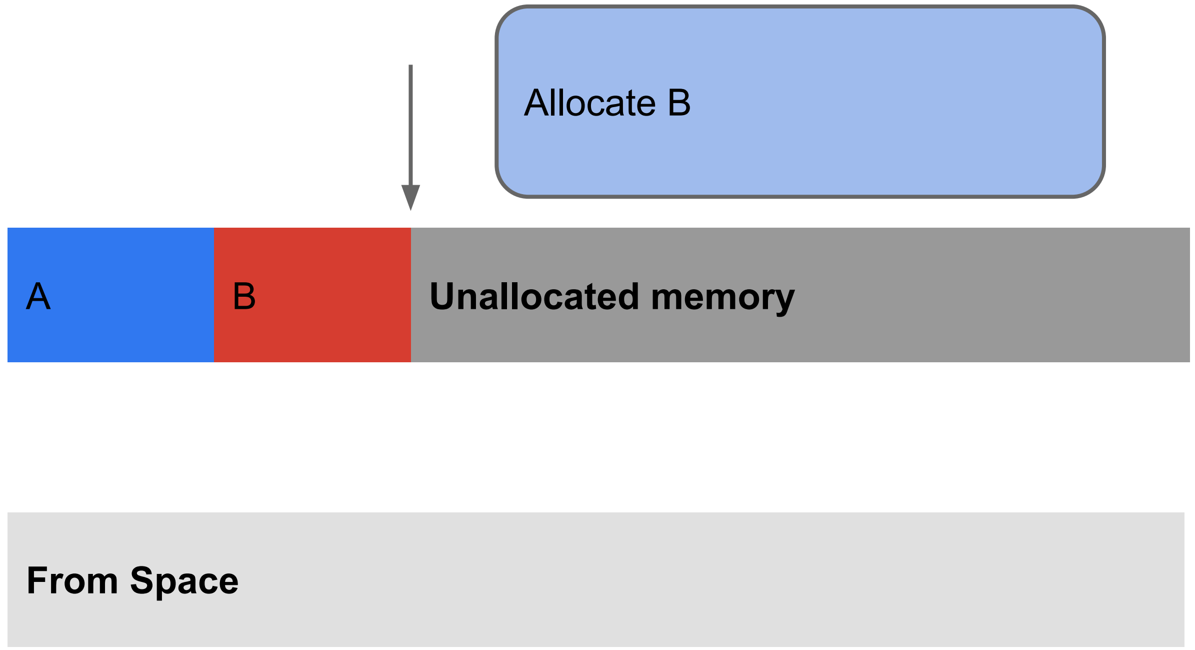
Assume the To Space started off empty and your page starts allocating objects..



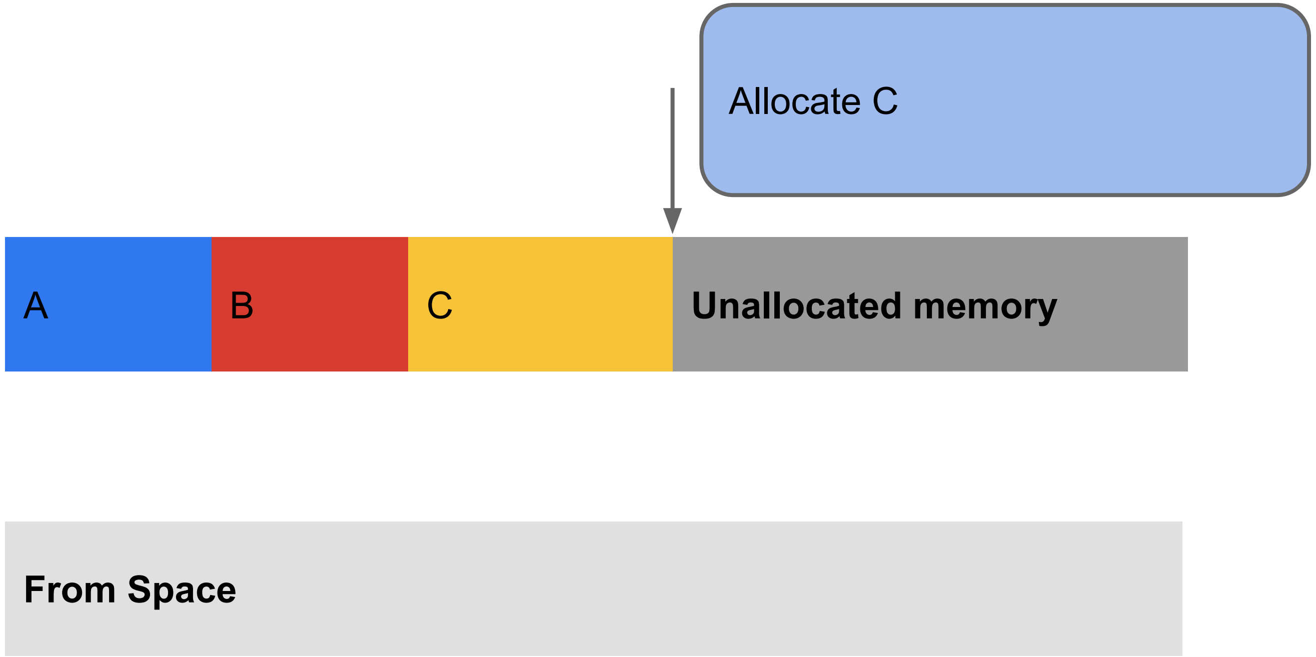
Young Generation In Action



Young Generation In Action

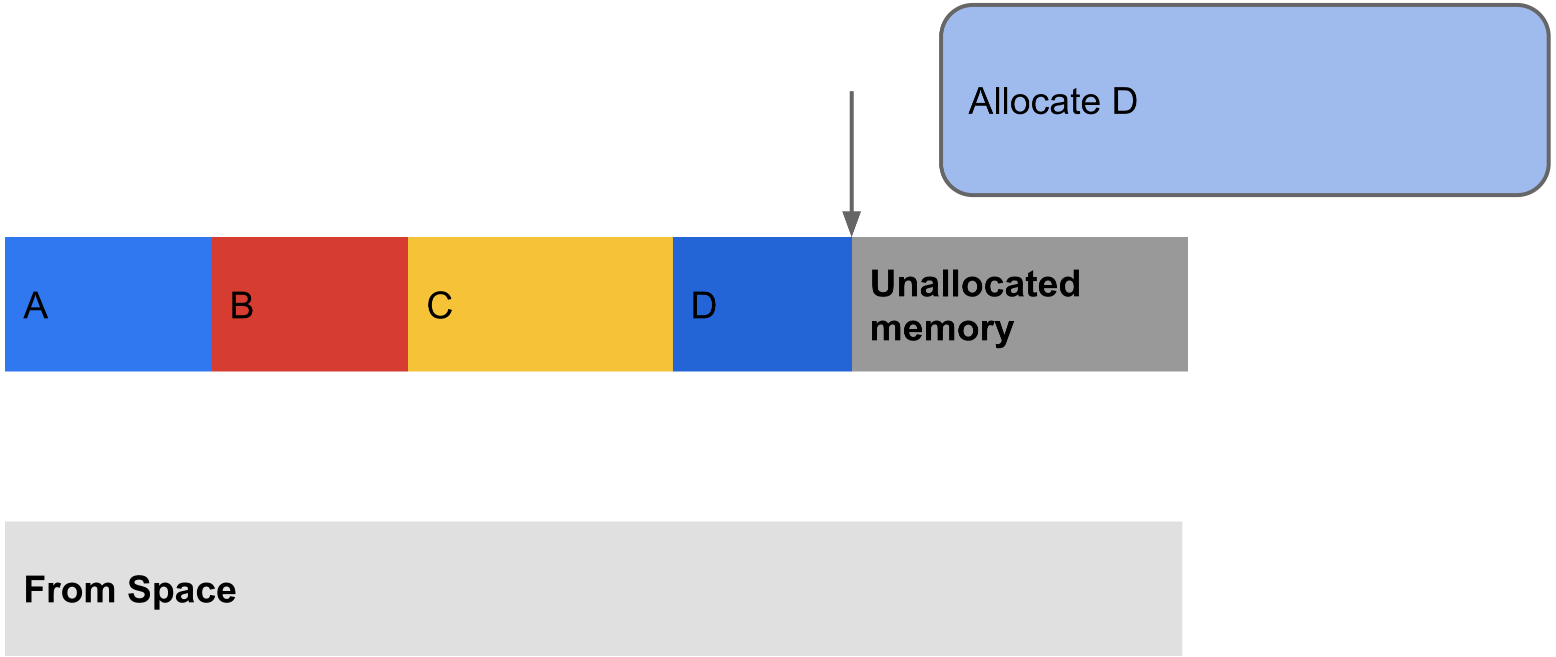


Young Generation In Action



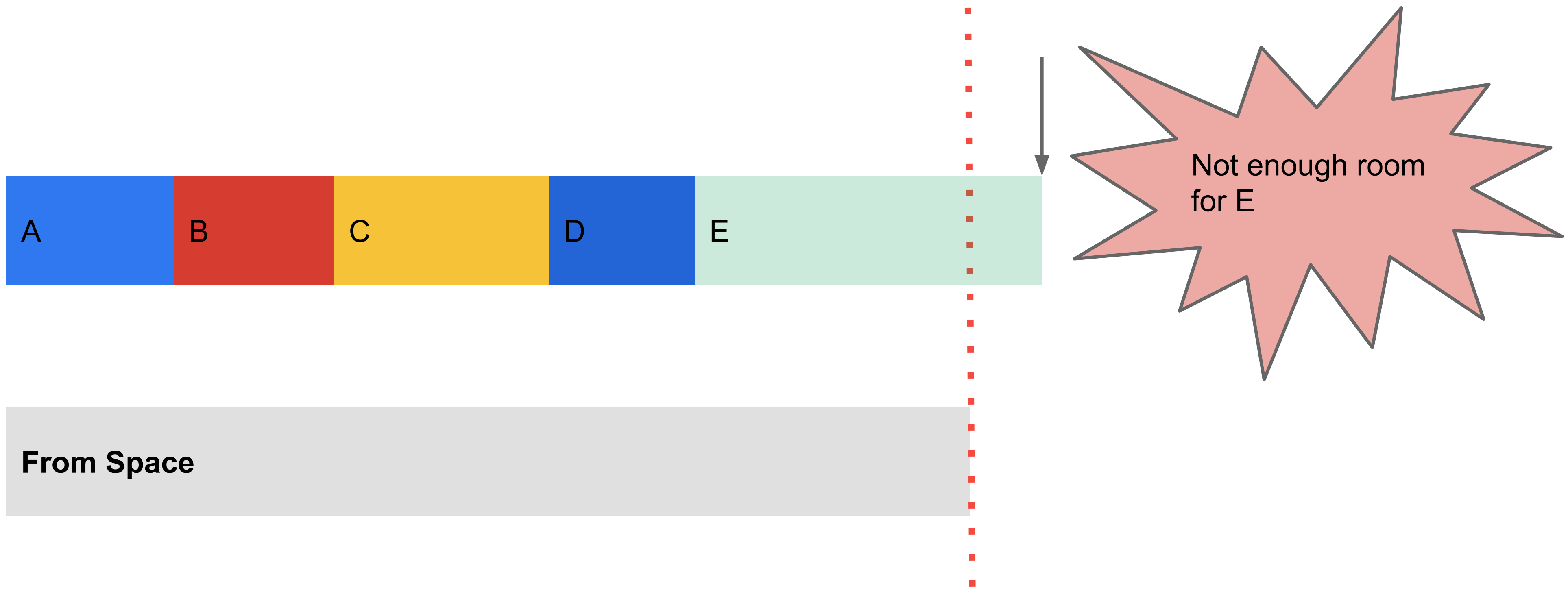
Young Generation In Action

Until this point, everything has been fast. There's been no interruption in your page's execution.



Young Generation In Action

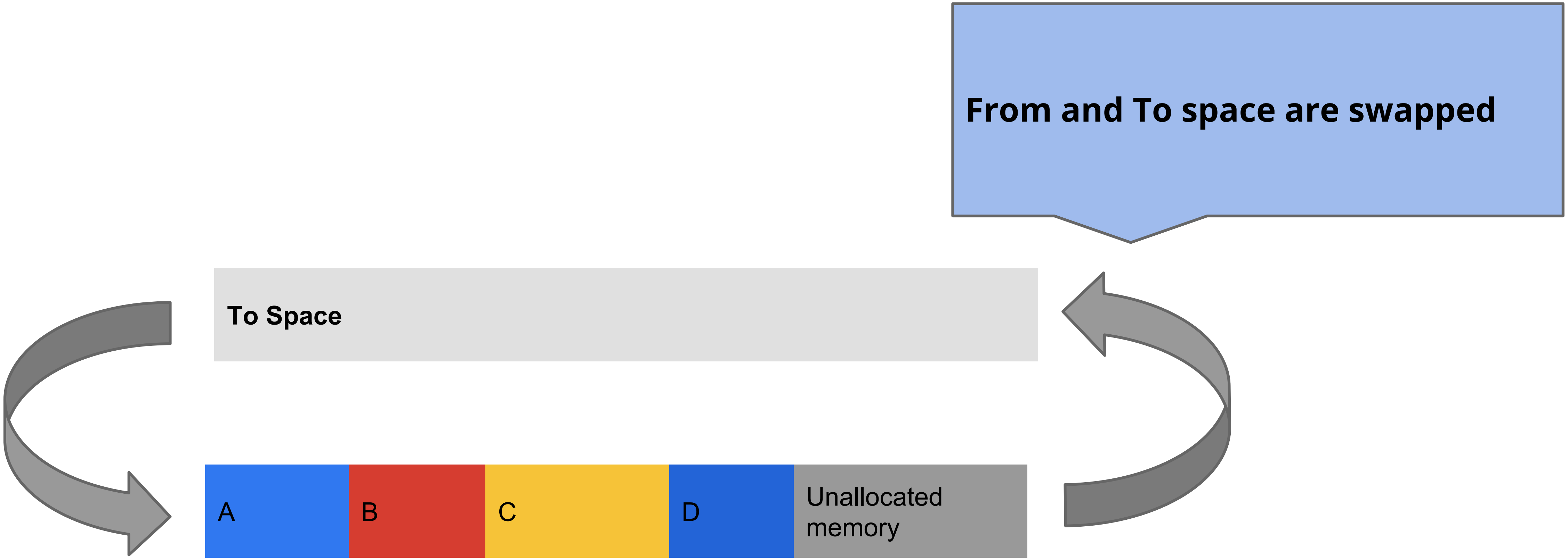
*Then we do **new** E() and..it's too big. We moved closer to this GC pause and we've actually just triggered it.*



Young Generation In Action

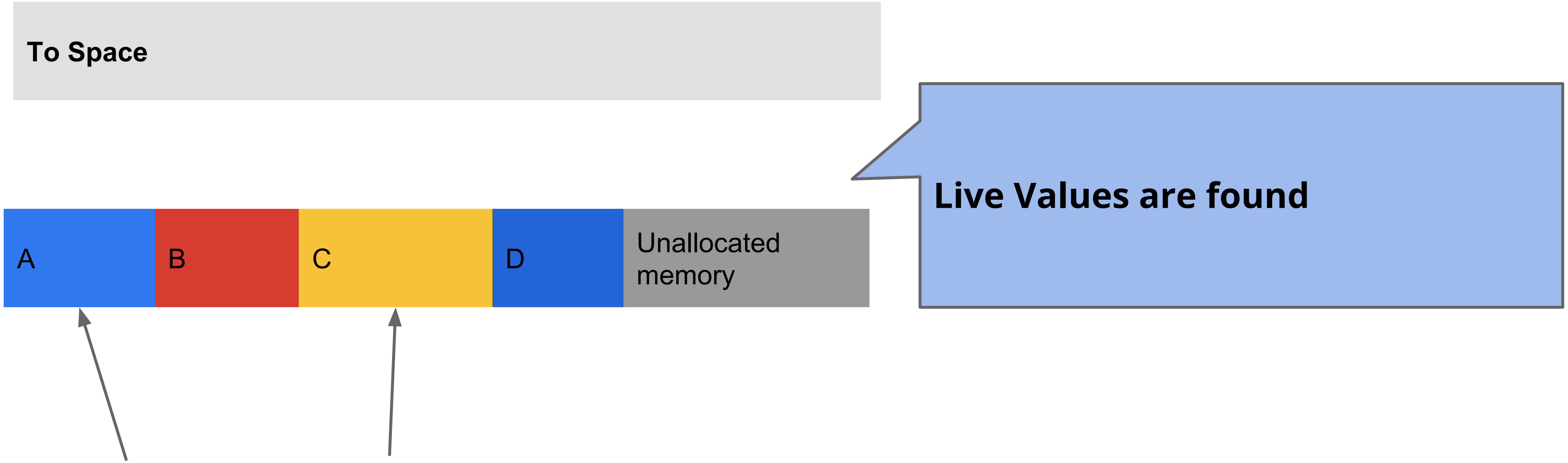
So, E doesn't happen. It's kind of paused. The page is paused, everything halts and the collection is triggered.

Young Generation In Action



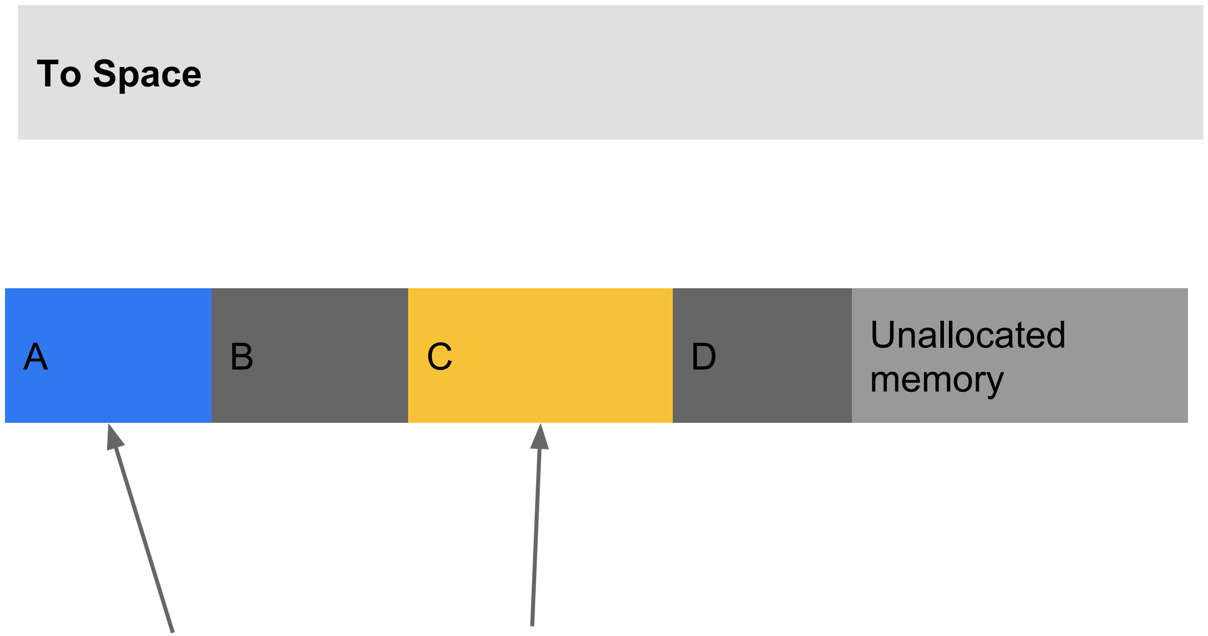
Young Generation In Action

Labels are flipped internally and then the live values are found.



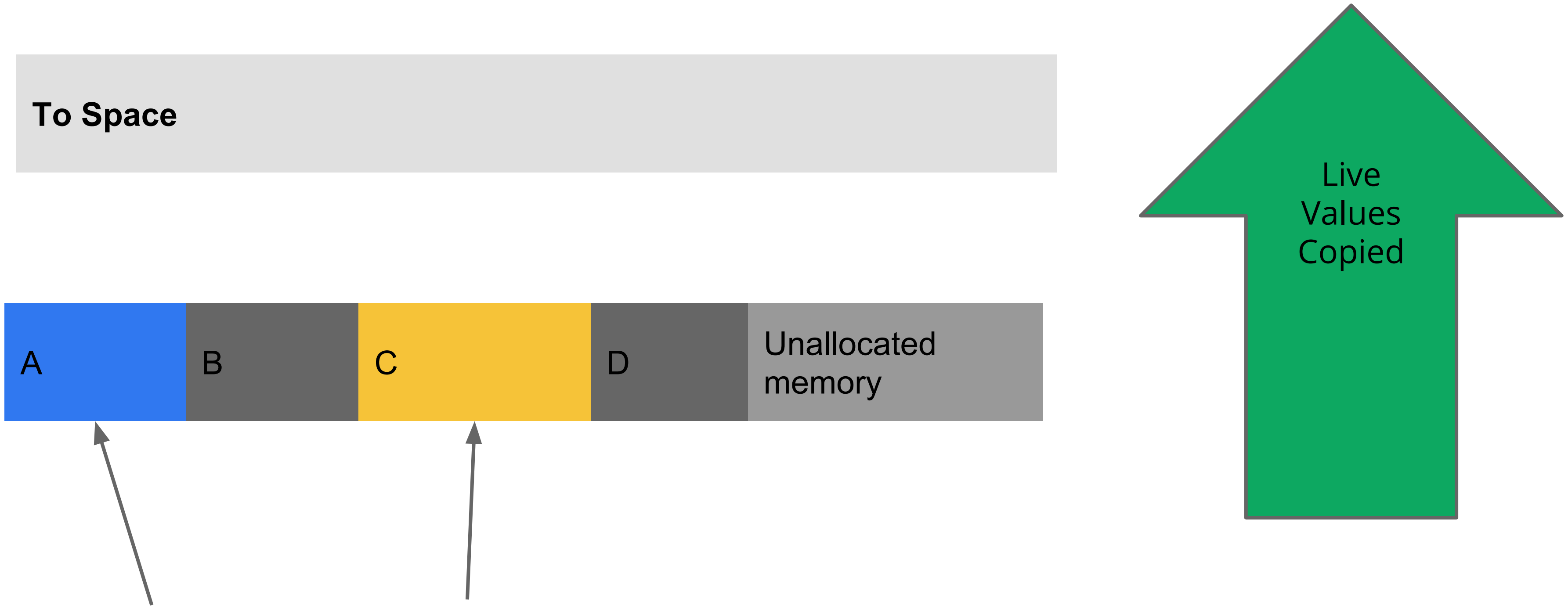
Young Generation In Action

A and C are marked. B and D are not marked so they're garbage. They're not going anywhere.



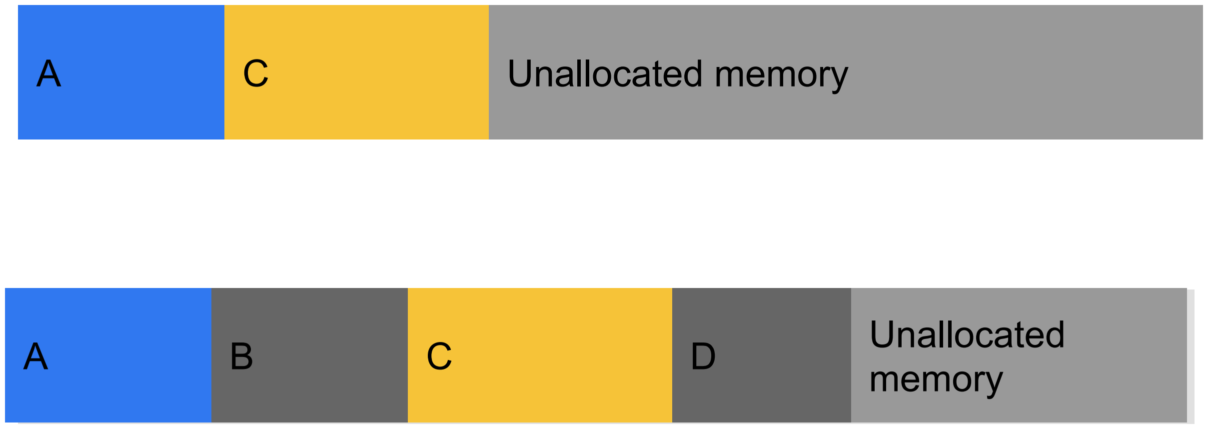
Young Generation In Action

This is when the live values are copied from the From Space to the To Space.



Young Generation In Action

So here we've done the copy. We've done the collection. Copied the live objects from one semispace to the next.

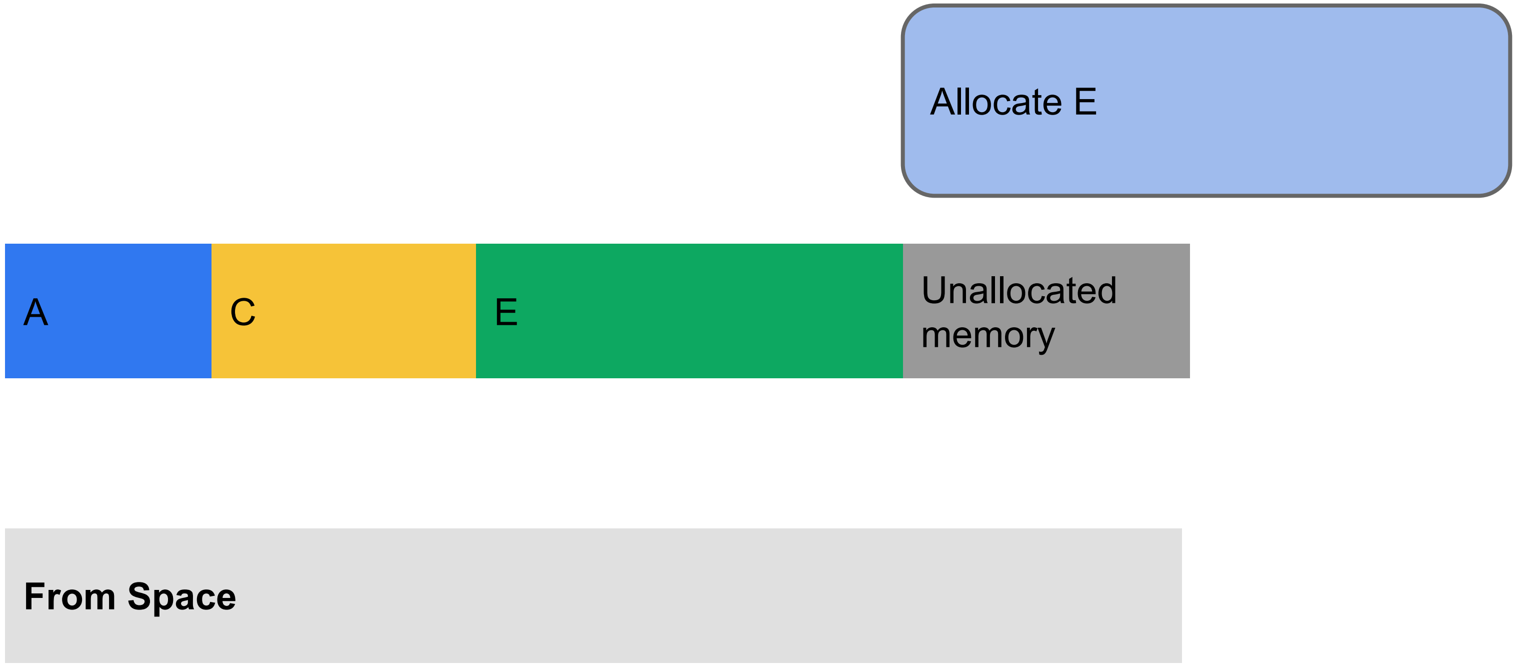


Young Generation In Action

There's no other work done to it. It's just ready for use the next time there's a collection that needs to happen.

Young Generation In Action

At this point, your page is resumed and the object E is allocated.



How does V8 manage memory?

- **Each allocation moves you closer to a collection**
  - Not always obvious when you are allocating
- **Collection pauses your application**
  - Higher latency
  - Dropped frames
  - Unhappy users

Remember: Triggering a collection pauses your app.

Performance Tools

performance.memory

Great for field measurements.

performance.memory

jsHeapSizeLimit

the amount of memory (in bytes) that the JavaScript heap is limited to

performance.memory

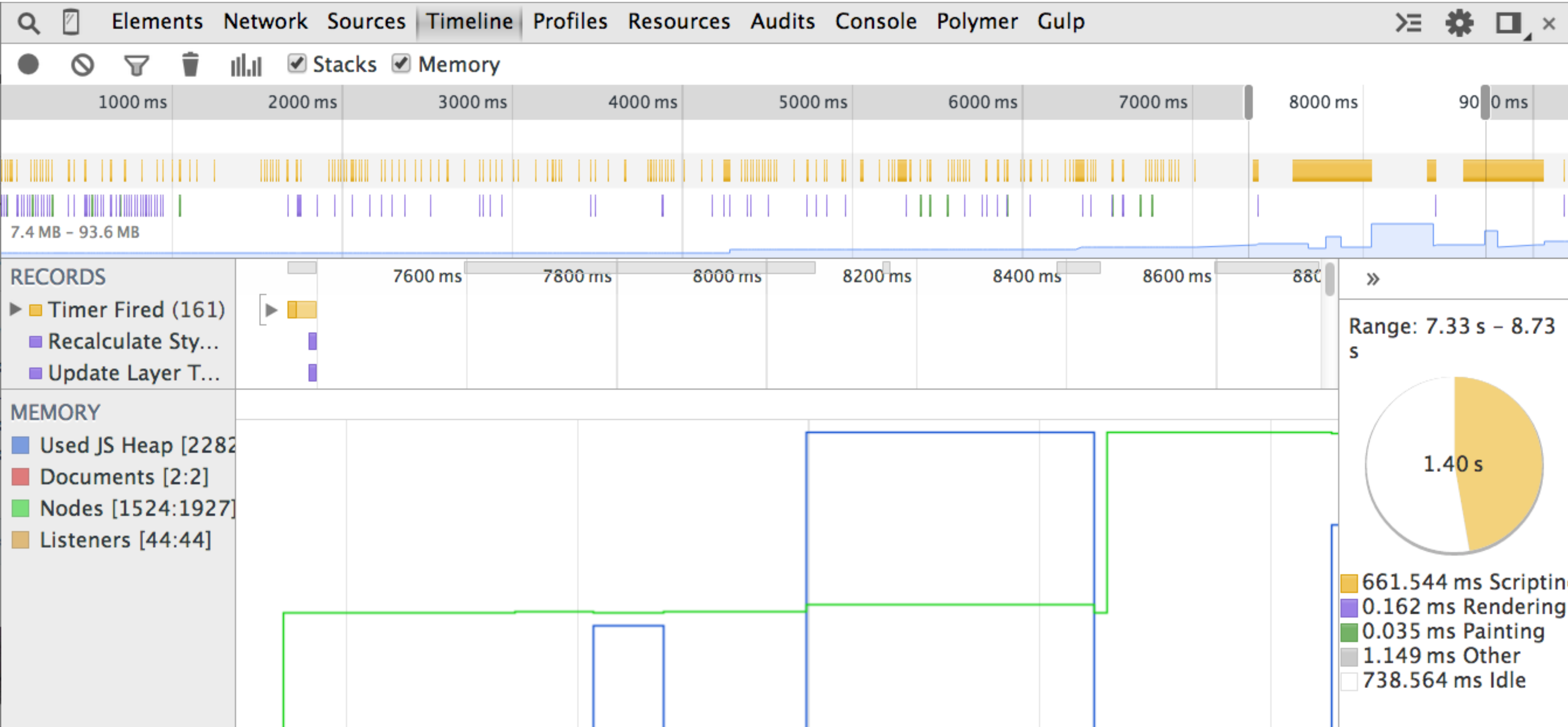
jsHeapSizeLimit	the amount of memory (in bytes) that the JavaScript heap is limited to
totalJSHeapSize	the amount of memory (in bytes) currently being used

performance.memory

jsHeapSizeLimit	the amount of memory (in bytes) that the JavaScript heap is limited to
totalJSHeapSize	the amount of memory (in bytes) currently being used
usedJSHeapSize	the amount of memory (in bytes) that the JavaScript heap has allocated, including free space

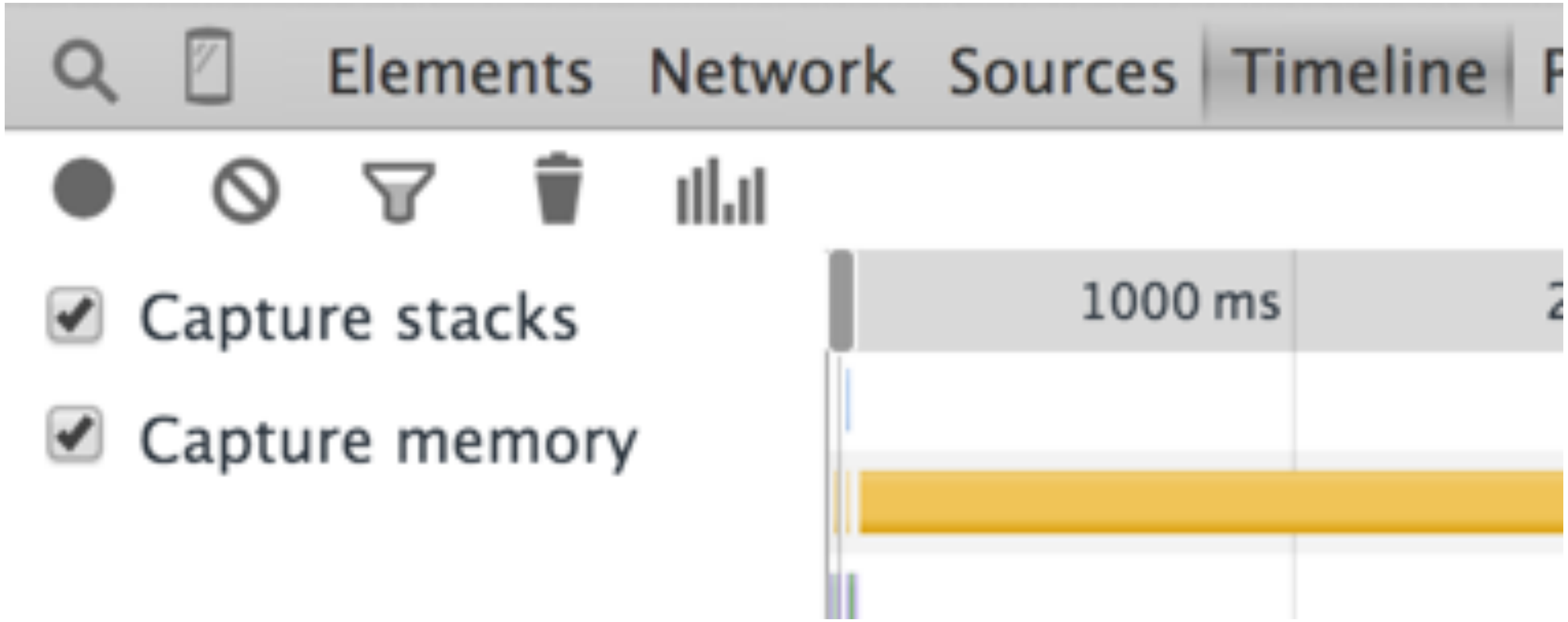
Chrome DevTools

DevTools Memory Timeline



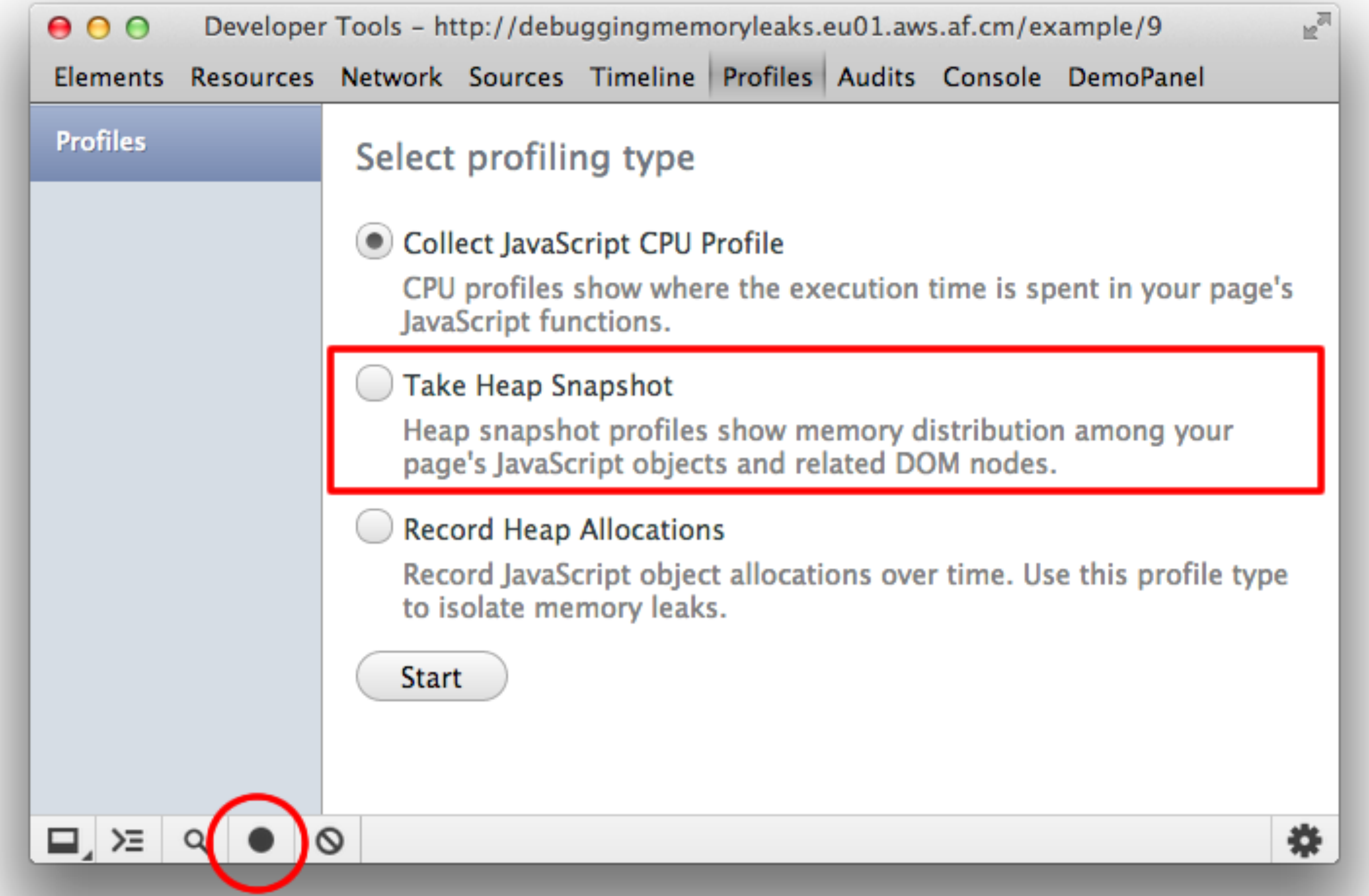
Force GC from DevTools

Snapshots automatically force GC. In Timeline, it can be useful to force a GC too using the Trash can.



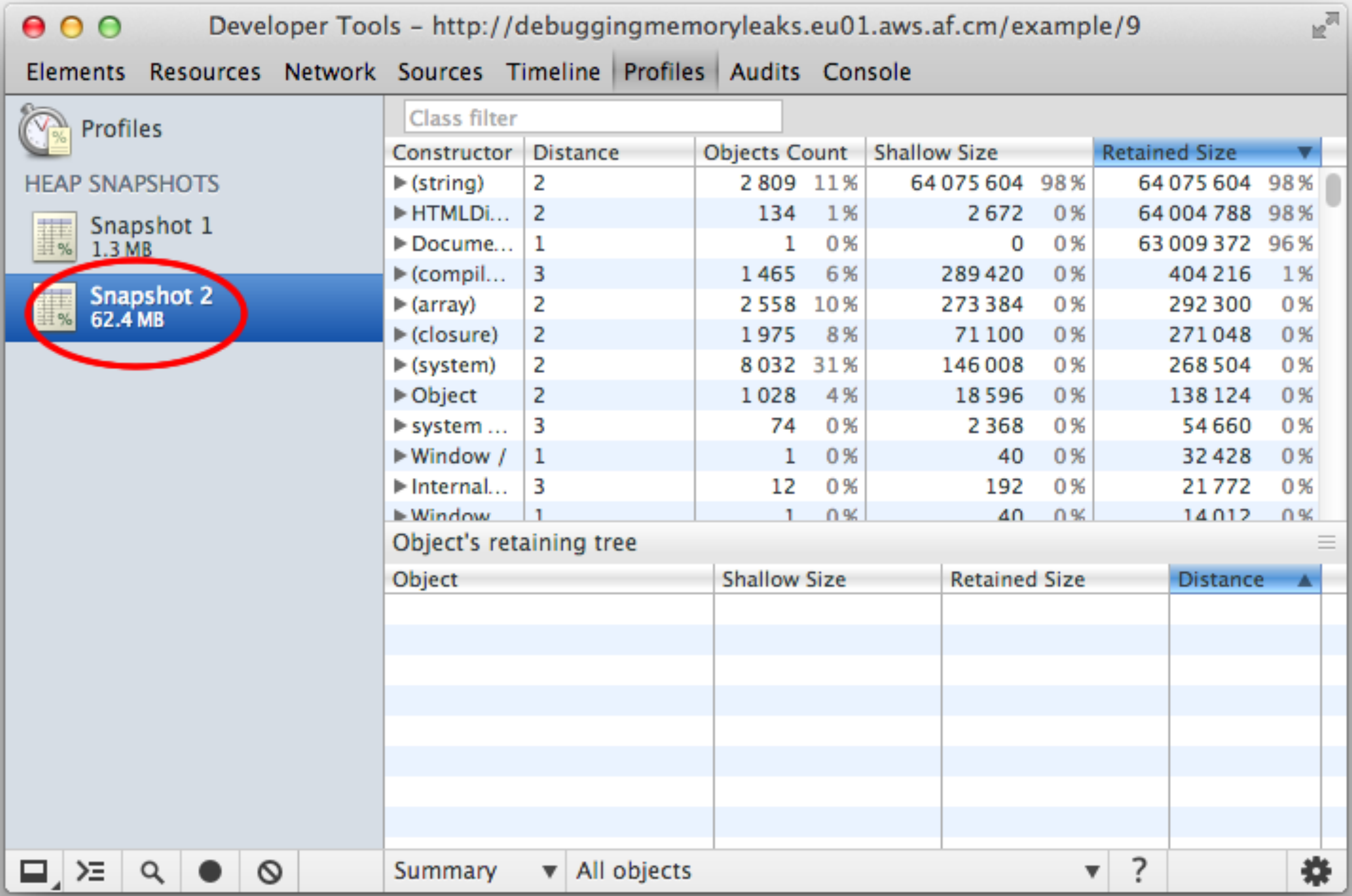
Memory distribution

Taking heap snapshots



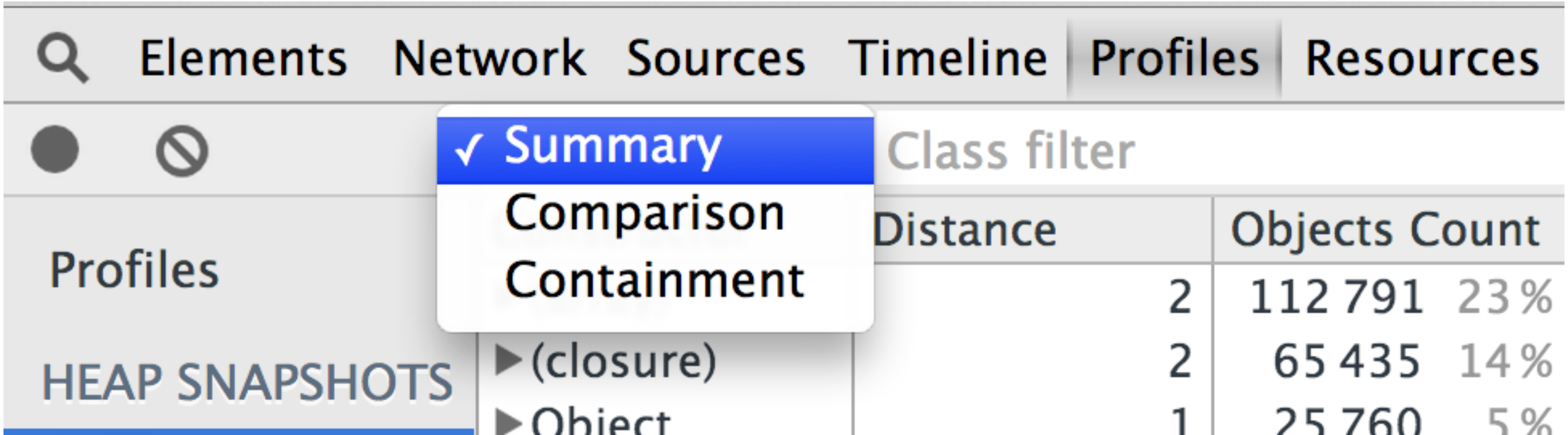
Results

Reachable JavaScript Objects



Switching between views

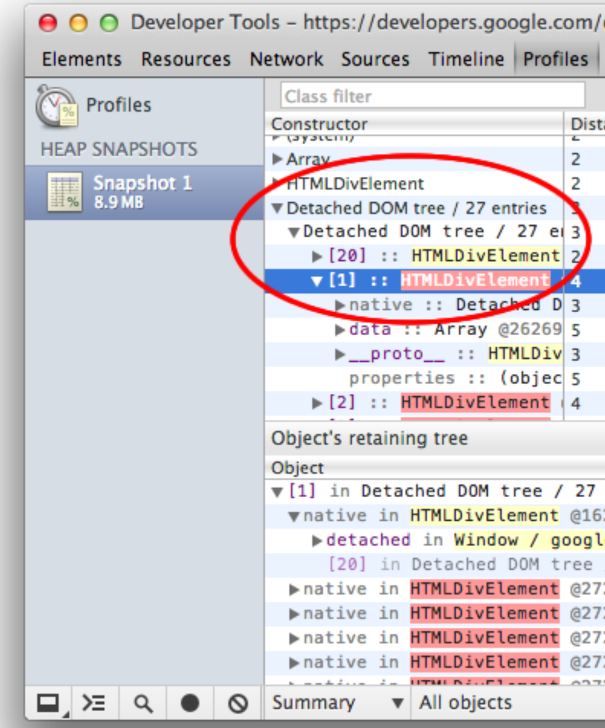
Summary groups by constructor name
Comparison compares two snapshots
Containment bird's eye view of the object structure



Understanding node colors

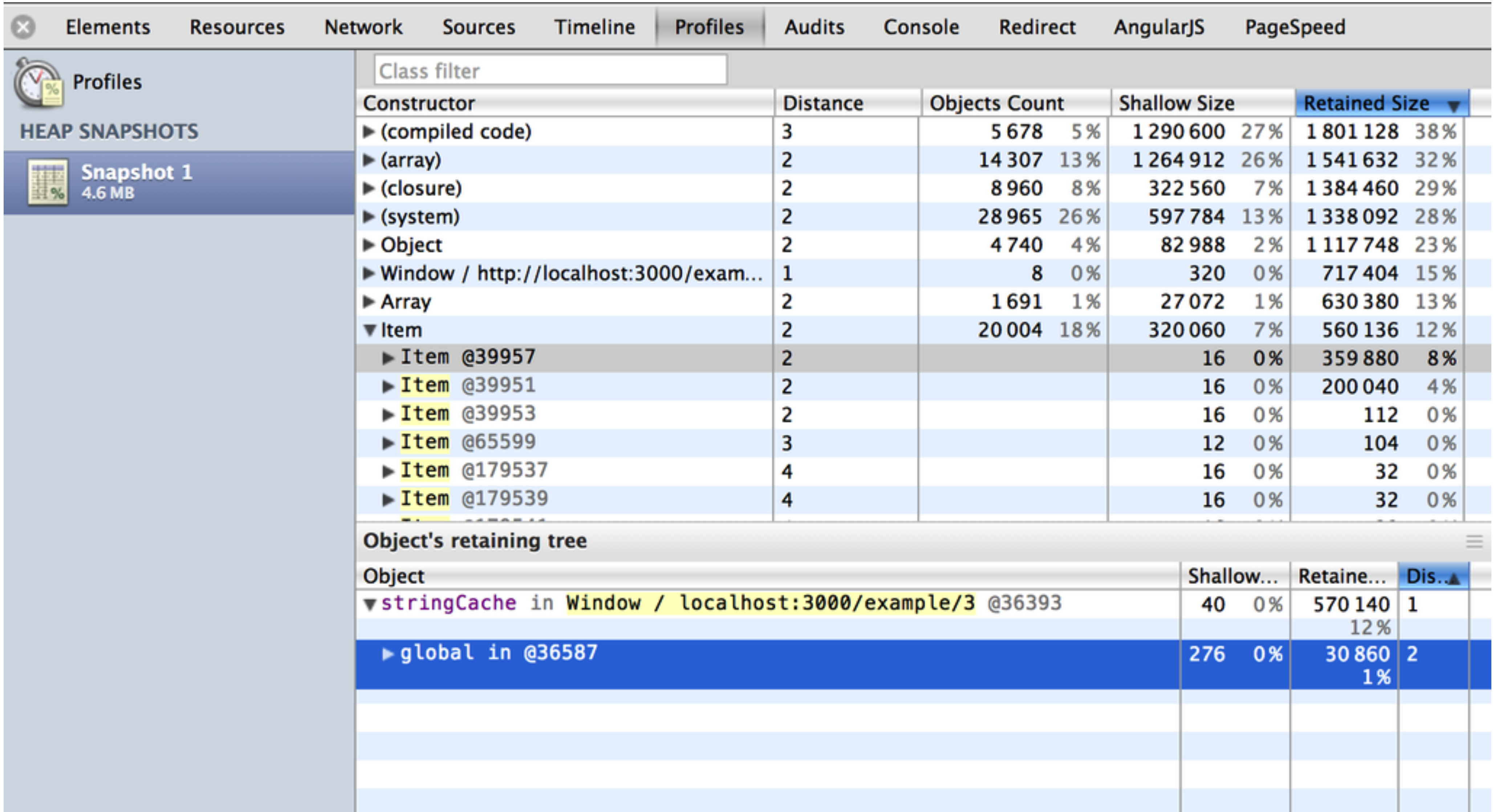
yellow — Object has a JavaScript reference on it

red — Detached node. Referenced from one with a yellow background.



Reading results

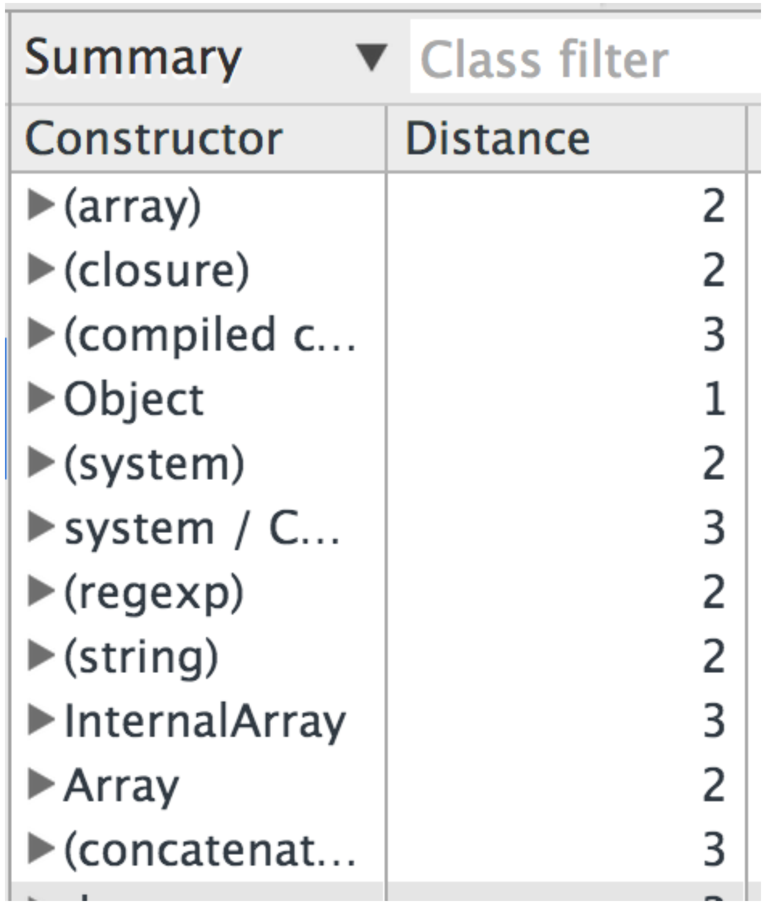
Summary



Distance

Distance from the GC root.

If all objects of the same type are at the same distance and a few are at a bigger distance, it’s worth investigating. Are you leaking the latter ones?



Summary ▼ Class filter

Constructor	Distance
▶(array)	2
▶(closure)	2
▶(compiled c…	3
▶Object	1
▶(system)	2
▶system / C…	3
▶(regexp)	2
▶(string)	2
▶InternalArray	3
▶Array	2
▶(concatenat…	3

Retained memory

Memory used by objects and the objects they are referencing.

		Retained Size	
392	34 %	4 327 728	47 %
924	10 %	3 515 640	38 %
280	22 %	2 875 108	31 %
396	2 %	2 632 980	28 %
376	15 %	2 474 832	27 %
536	1 %	1 410 864	15 %
000	0 %	434 488	5 %
272	4 %	417 272	4 %
024	0 %	343 496	4 %
512	1 %	288 264	3 %
540	1 %	92 384	1 %
548	0 %	68 616	1 %
980	0 %	51 756	1 %
500	0 %	43 892	0 %
176	0 %	41 252	0 %
176	0 %	32 908	0 %
960	0 %	28 960	0 %
384	0 %	28 788	0 %
176	0 %	27 424	0 %
584	0 %	25 000	0 %
192	0 %	20 228	0 %

Shallow size

Size of memory held by object

Even small objects can hold large amounts of memory indirectly by preventing other objects from being disposed.

ts Count		Shallow Size	
0 490	21 %	3 114 392	34 %
4 609	13 %	885 924	10 %
1 699	6 %	2 047 280	22 %
9 999	5 %	208 896	2 %
4 998	34 %	1 396 876	15 %
2 028	1 %	90 636	1 %
750	0 %	27 000	0 %
4 018	7 %	417 272	4 %
64	0 %	1 024	0 %
4 530	2 %	72 512	1 %
3 277	2 %	65 540	1 %
32	0 %	648	0 %
40	0 %	980	0 %
37	0 %	500	0 %
4	0 %	176	0 %
4	0 %	176	0 %

Constructor

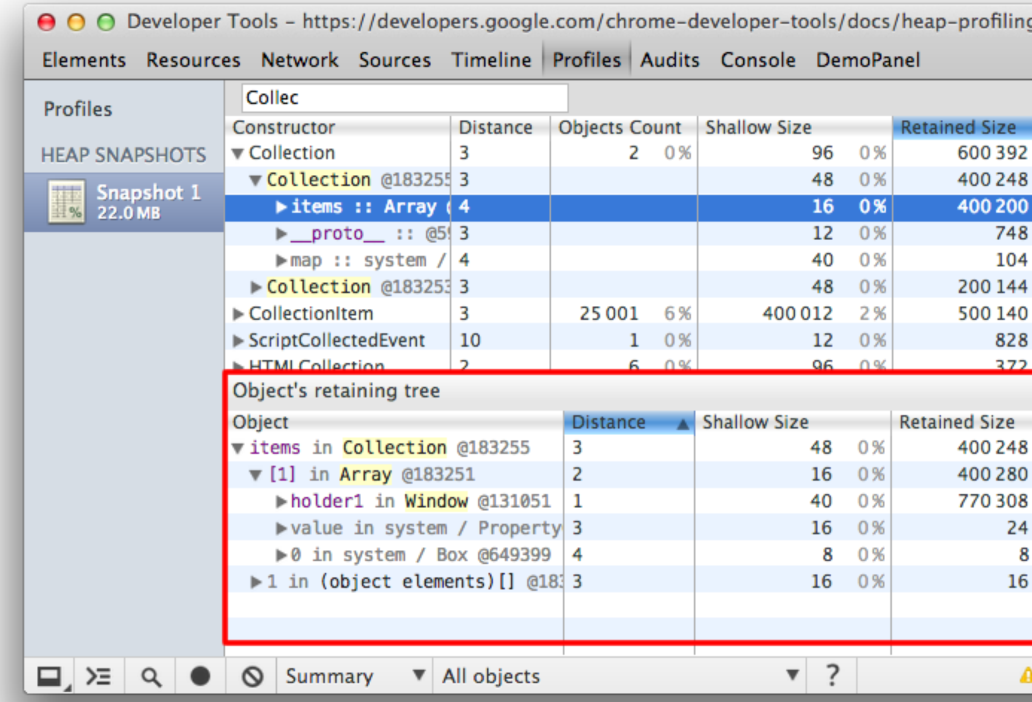
All objects created with a specific constructor.

Summary ▼ Class filter

Constructor	Distance	Objects C
(array)	2	40 49
(closure)	2	24 60
(compiled code)	3	11 69
Object	1	9 99
(system)	2	64 99
system / Cont…	3	2 02
(regexp)	2	75
(string)	2	14 01
InternalArray	3	6
Array	2	4 53
(concatenated …	3	3 27
d	3	3
Window	1	4
c	3	3
Window / http…	1	
Window / http…	1	
system / JSArr…	5	1
JSONSchemaVa…	5	2

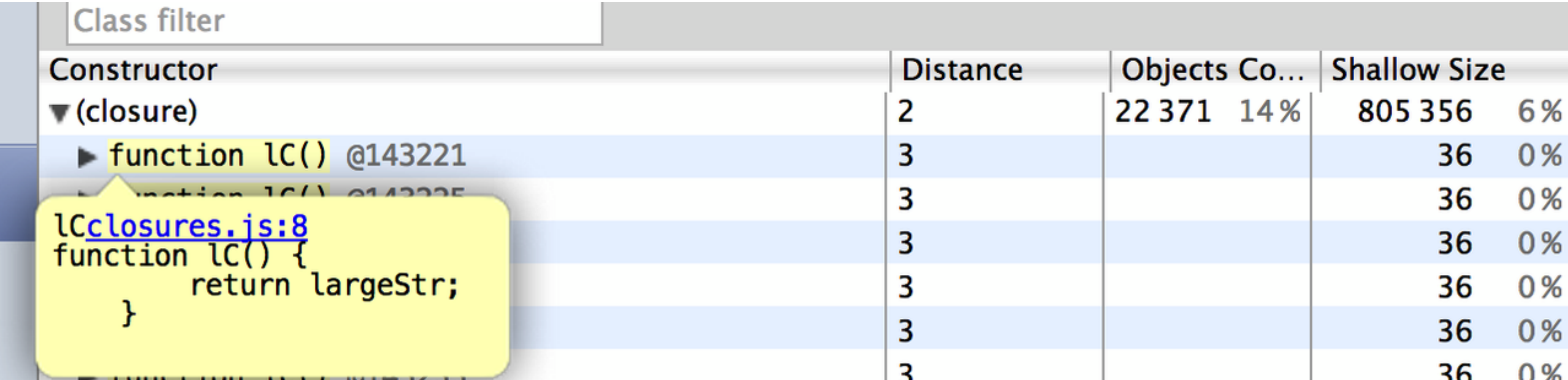
Object's retaining tree

Information to understand why the object was not collected.



Closures

Tip: It helps to name functions so you can easily find them in the snapshot.



app.js

```
function createLargeClosure() {
    var largeStr = new Array(1000000).join('x');
    var lC =  function() { //this IS NOT a named function
        return largeStr;
    };
    return lC;
}

function createLargeClosure() {
    var largeStr = new Array(1000000).join('x');
    var lC = function lC() { //this IS a named function
        return largeStr;
    };
    return lC;
}
```

Profiling Memory Leaks

Three snapshot technique

retired

What do we expect?

New objects to be constantly and consistently collected.

Start from a steady state.

Checkpoint 1

We do some stuff.

Checkpoint 2

We repeat the same stuff.

Checkpoint 3

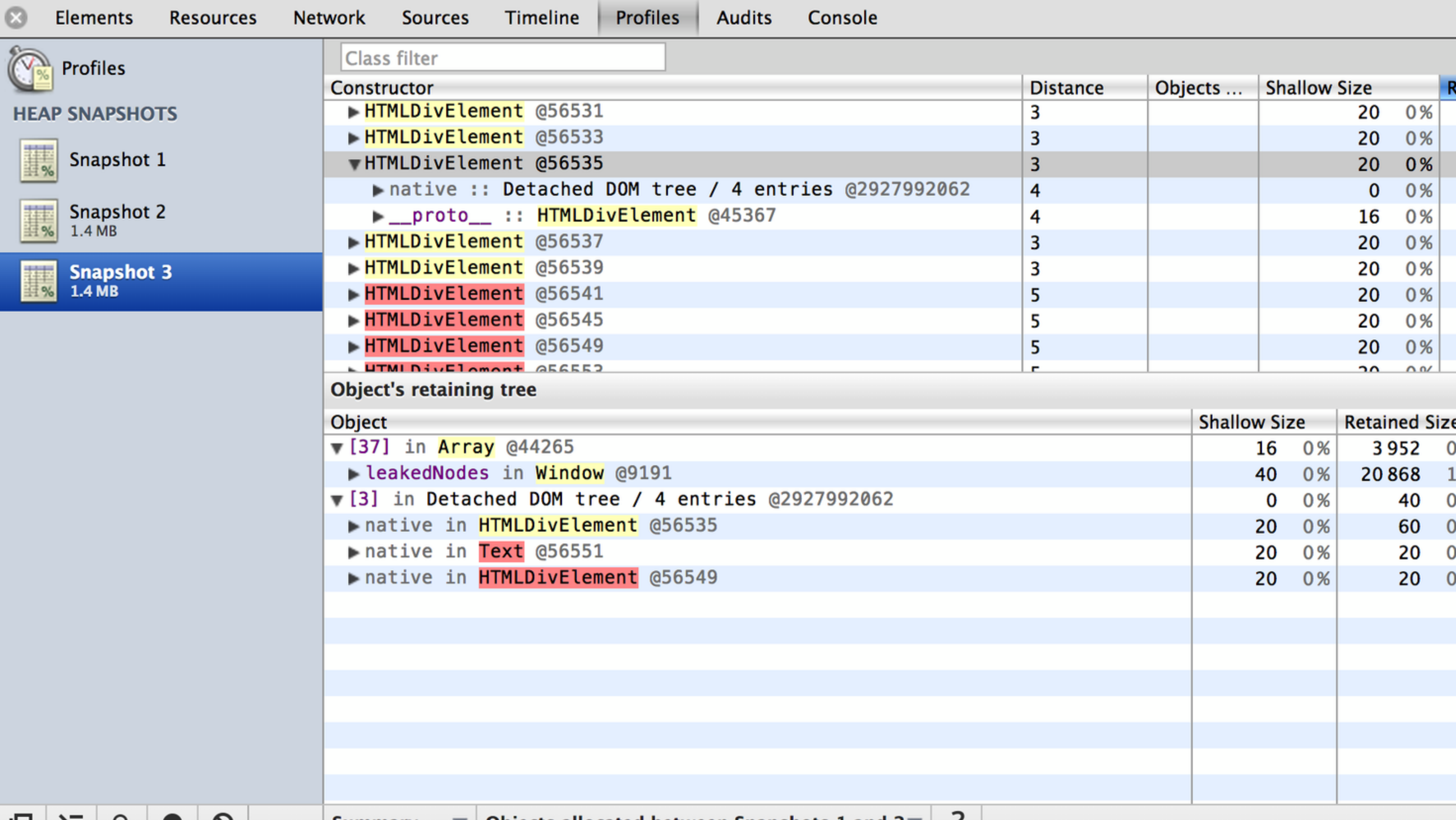
Again, what do we expect?

All new memory used between Checkpoint 1 and Checkpoint 2 has been collected.

New memory used between Checkpoint 2 and Checkpoint 3 may still be in use in Checkpoint 3.

The Steps

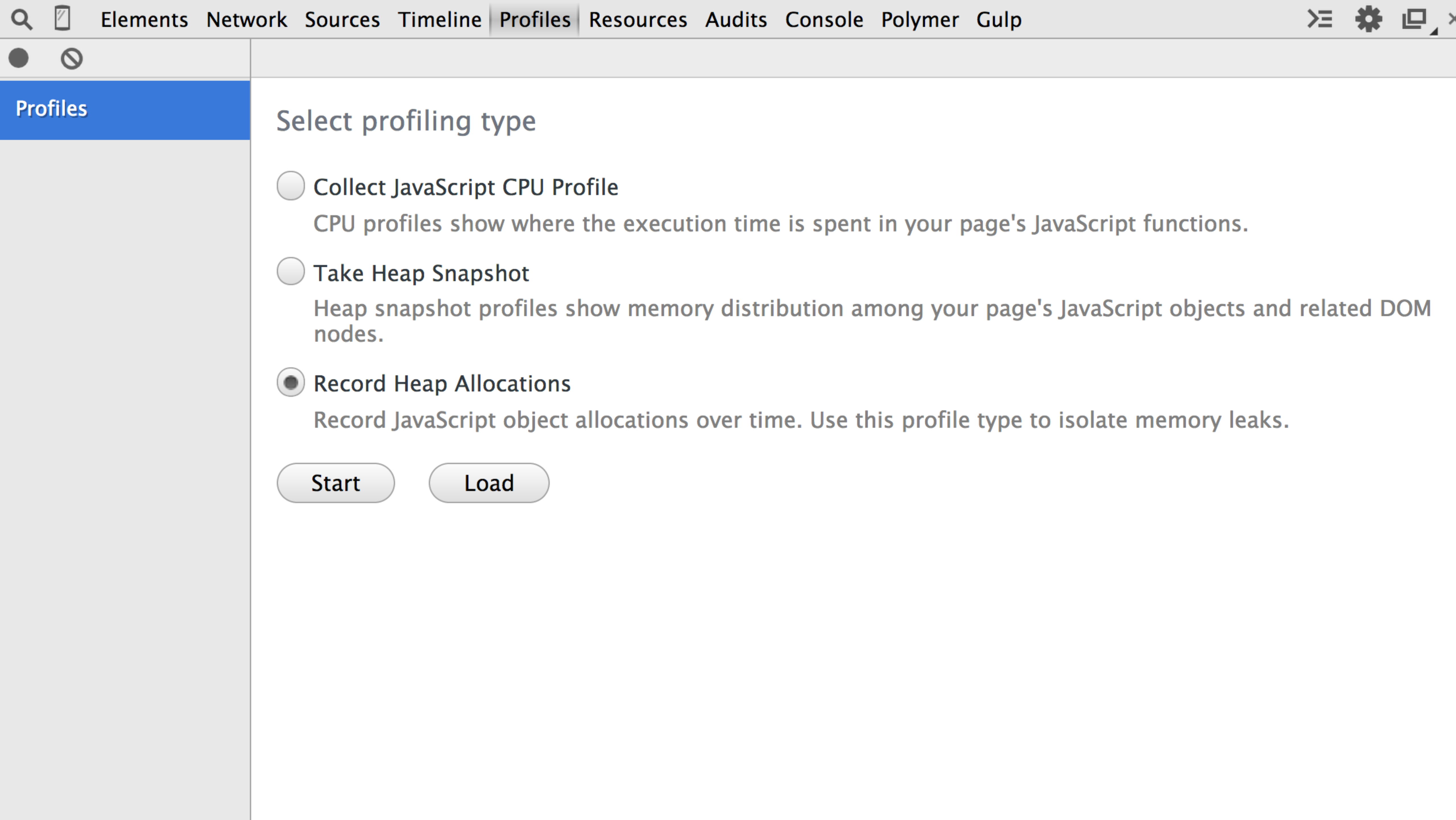
- Open DevTools
- Take a heap snapshot #1
- Perform suspicious actions
- Take a heap snapshot #2
- Perform same actions again
- Take a third heap snapshot #3
- Select this snapshot, and select
- "Objects allocated between Snapshots 1 and 2"


Elements
Resources
Network
Sources
Timeline
Profiles
Audits
Console
Profiles
HEAP SNAPSHOTS
Snapshot 1
Snapshot 2
1.4 MB
Snapshot 3
1.4 MB
Class filter
Constructor
Distance
Objects ...
Shallow Size
HTMLDivElement @56531 3 20 0%
HTMLDivElement @56533 3 20 0%
HTMLDivElement @56535 3 20 0%
native :: Detached DOM tree / 4 entries @2927992062 4 0 0%
__proto__ :: HTMLDivElement @45367 4 16 0%
HTMLDivElement @56537 3 20 0%
HTMLDivElement @56539 3 20 0%
HTMLDivElement @56541 5 20 0%
HTMLDivElement @56545 5 20 0%
HTMLDivElement @56549 5 20 0%
Object's retaining tree
Object
Shallow Size
Retained Size
[37] in Array @44265 16 0% 3 952
leakedNodes in Window @9191 40 0% 20 868
[3] in Detached DOM tree / 4 entries @2927992062 0 0% 40
native in HTMLDivElement @56535 20 0% 60
native in Text @56551 20 0% 20
native in HTMLDivElement @56549 20 0% 20

Evolved memory profiling

Object Allocation Tracker

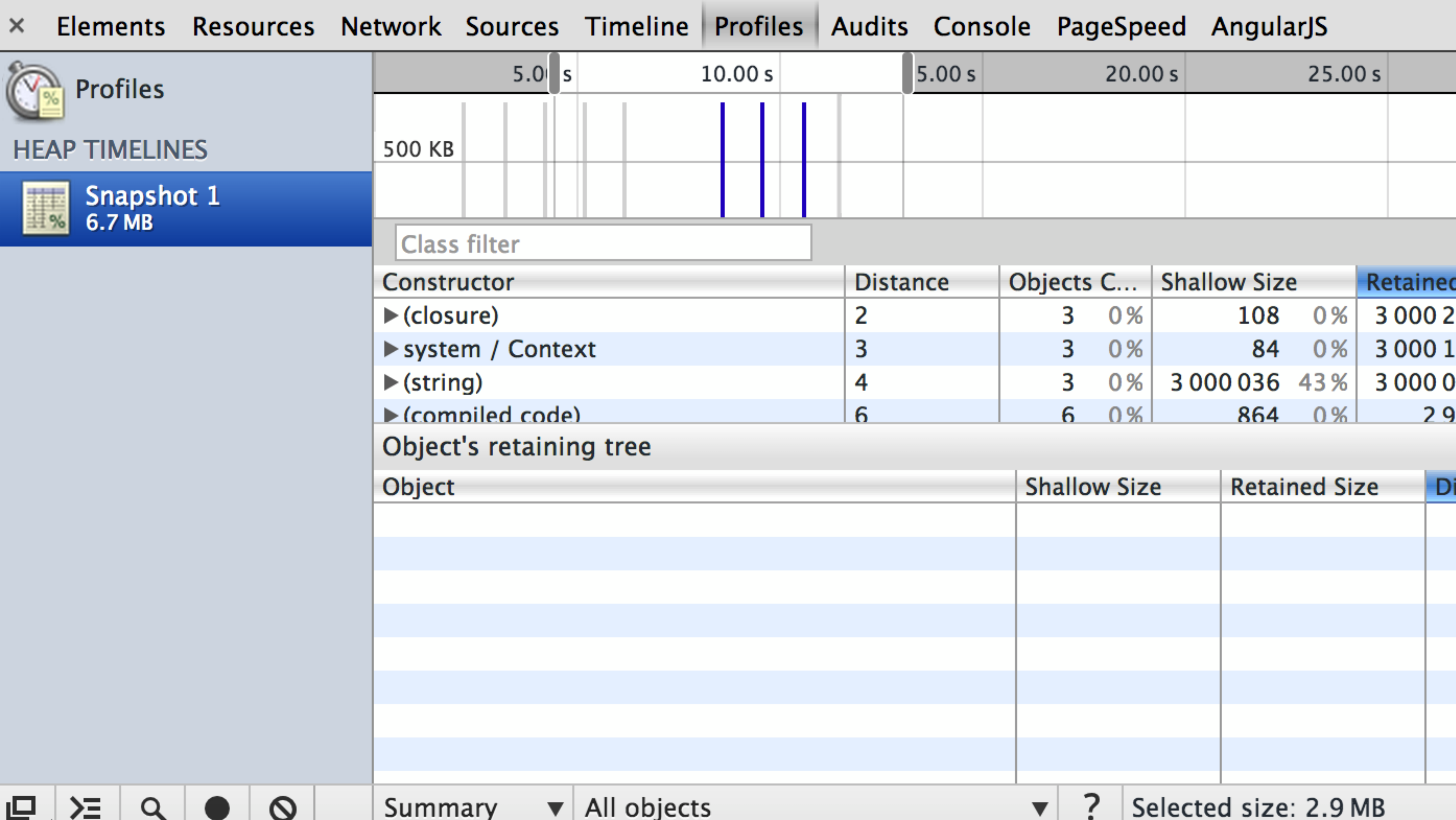
Record Heap Allocations


Elements
Network
Sources
Timeline
Profiles
Resources
Audits
Console
Polymer
Gulp
Profiles
Select profiling type
Collect JavaScript CPU Profile
CPU profiles show where the execution time is spent in your page's JavaScript functions.
Take Heap Snapshot
Heap snapshot profiles show memory distribution among your page's JavaScript objects and related DOM nodes.
Record Heap Allocations
Record JavaScript object allocations over time. Use this profile type to isolate memory leaks.
Start
Load

Object Allocation Tracker

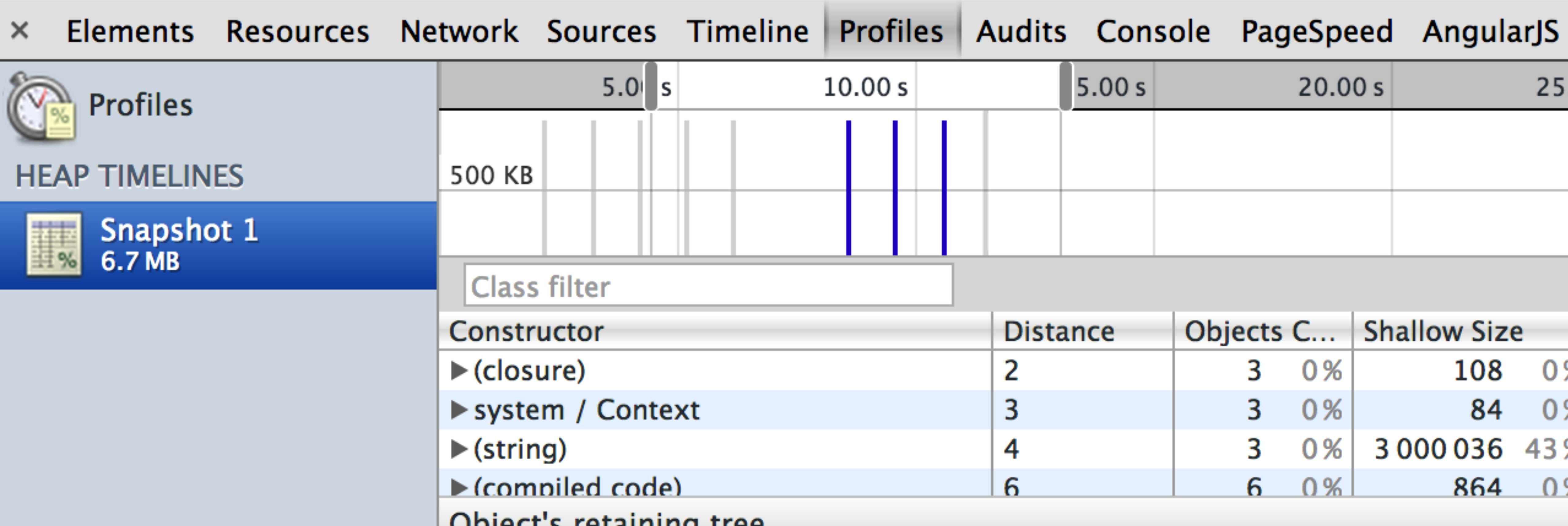
The object tracker combines the detailed snapshot information of the heap profiler with the incremental updating and tracking of the Timeline panel. Similar to these tools, tracking objects' heap allocation involves starting a recording, performing a sequence of actions, then stopping the recording for analysis.

The object tracker takes heap snapshots periodically throughout the recording and one final snapshot at the end of the recording. The heap allocation profile shows where objects are being created and identifies the retaining path.

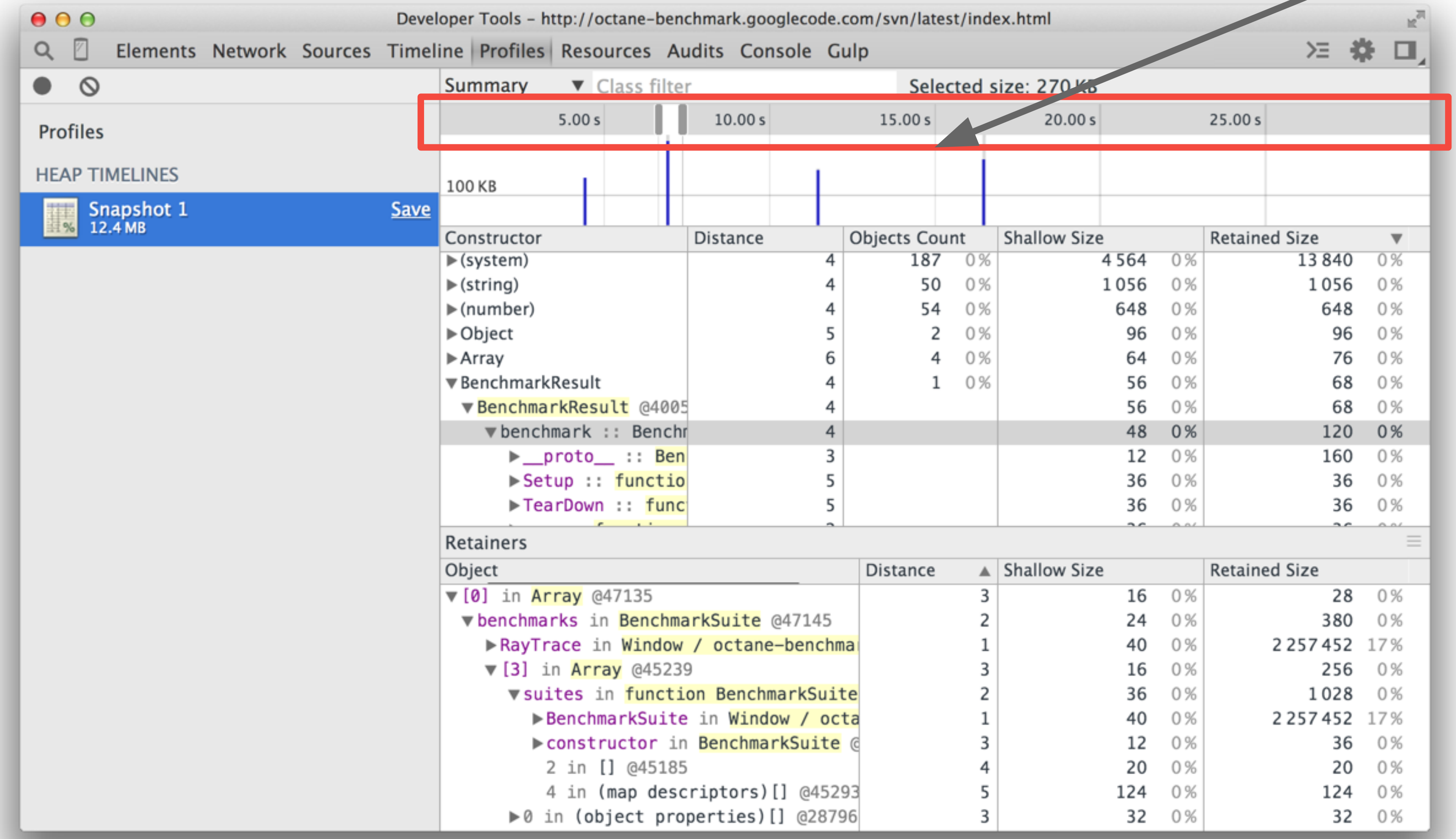

Elements
Resources
Network
Sources
Timeline
Profiles
Audits
Console
PageSpeed
AngularJS
Profiles
HEAP TIMELINES
Snapshot 1
6.7 MB
10.00 s
20.00 s
25.00 s
500 KB
Class filter
Constructor
Distance
Objects C...
Shallow Size
(closure)
system / Context
(string)
(compiled code)
Object's retaining tree
Object
Shallow Size
Retained Size
Summary
All objects
Selected size: 2.9 MB

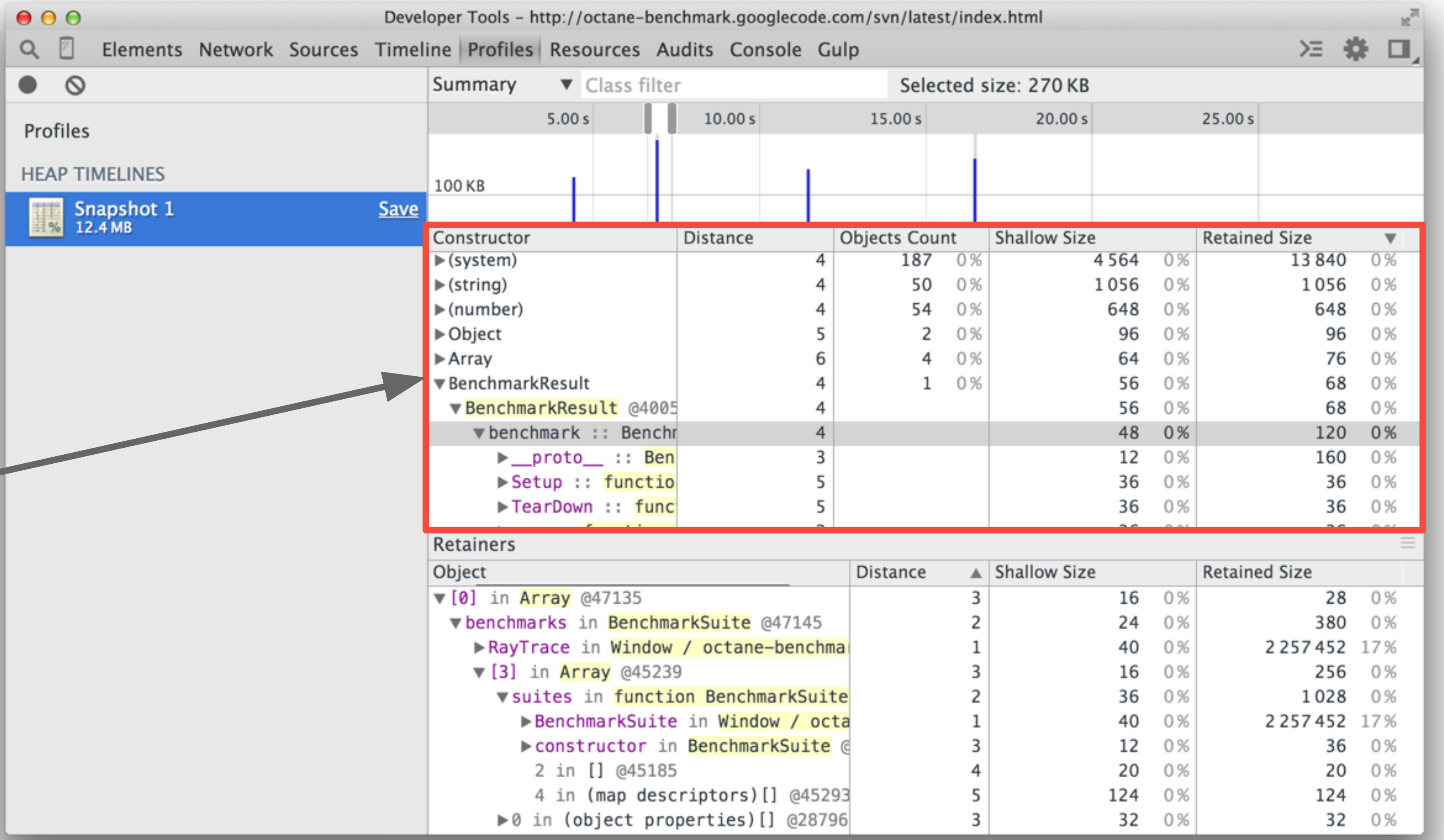
blue bars — memory allocations. Taller = more memory.

grey bars — deallocated



Adjustable timeframe selector

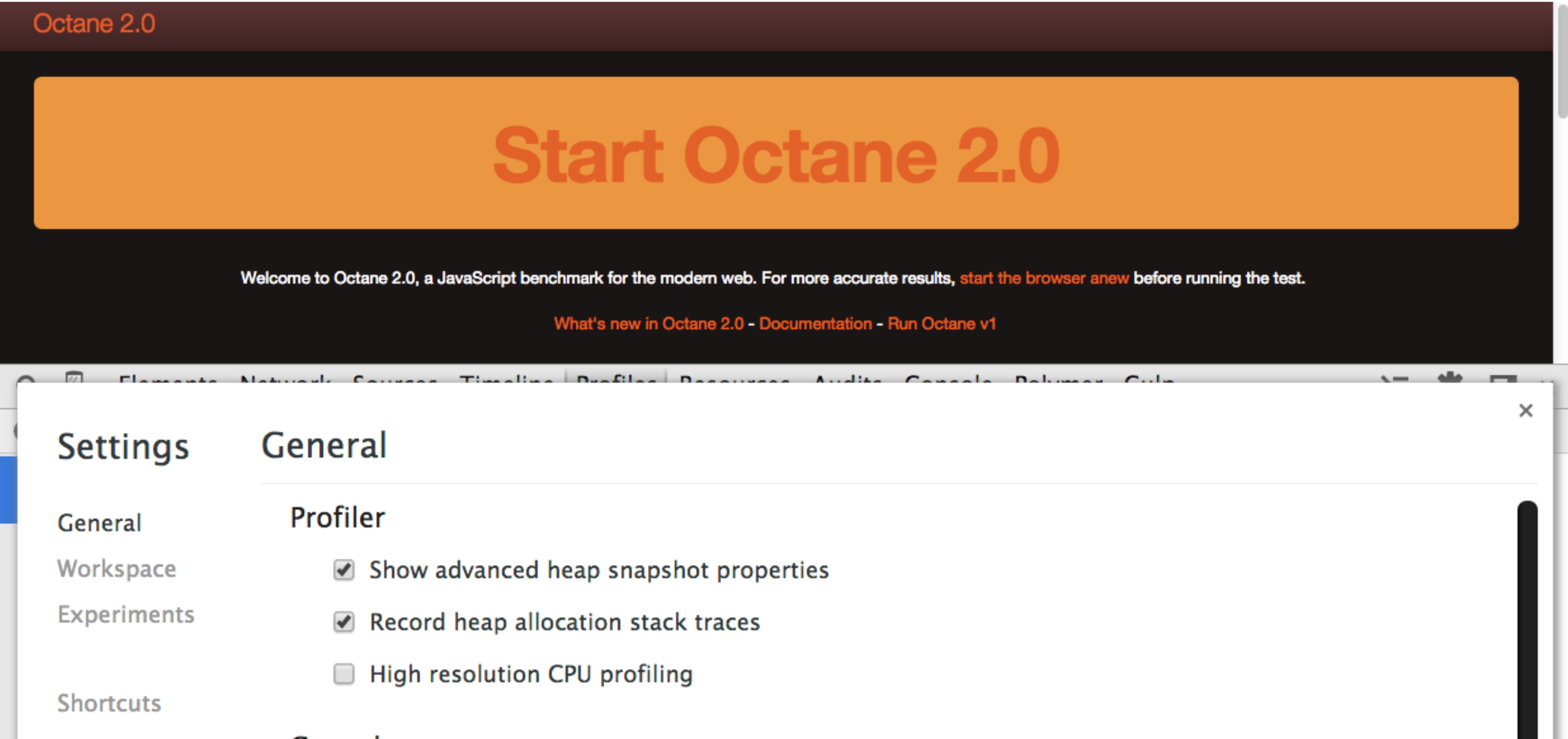


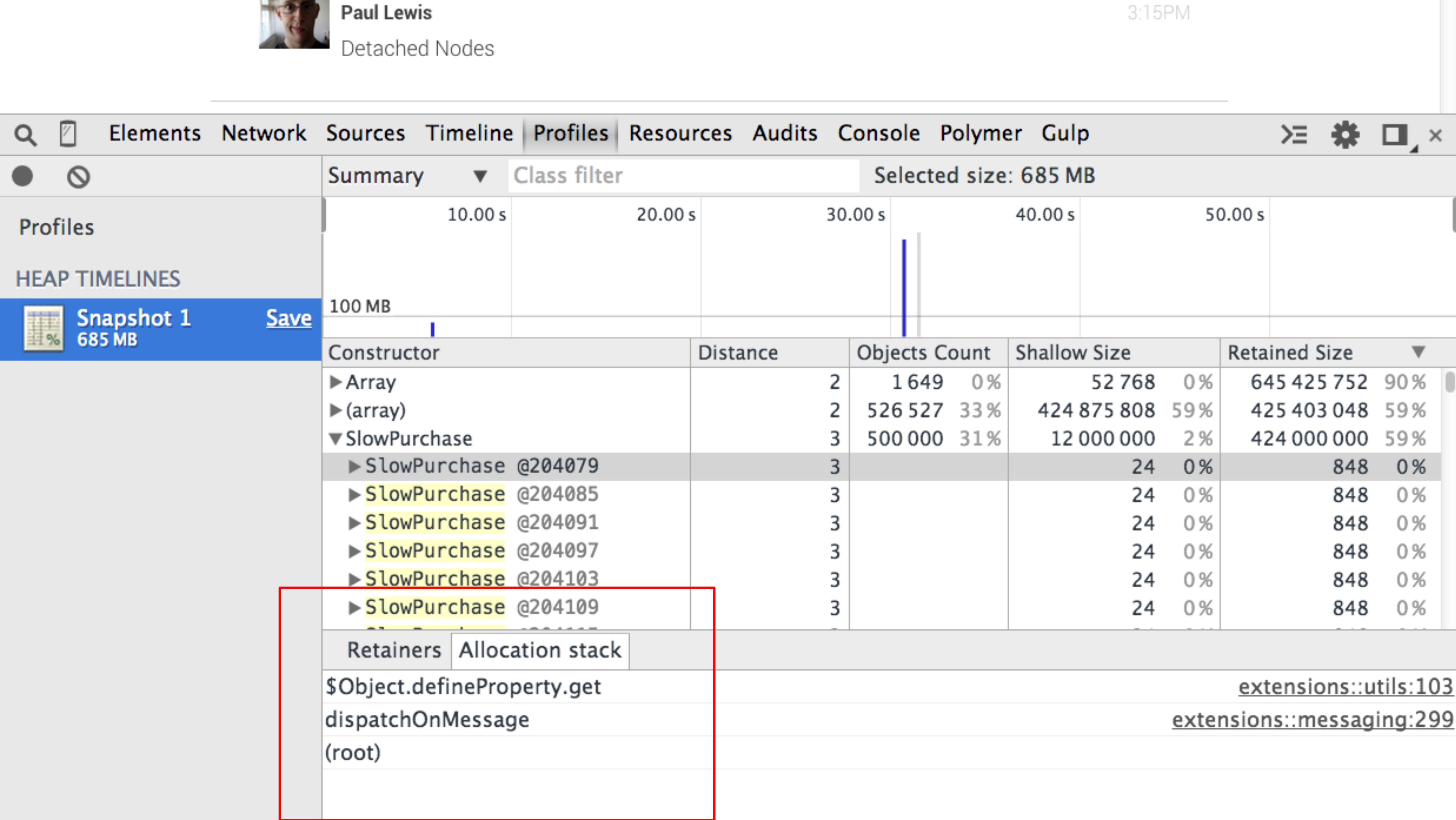


Heap contents

Allocation Stack Traces (New)

DevTools Settings > Profiler > Record Heap Allocation Stack Traces

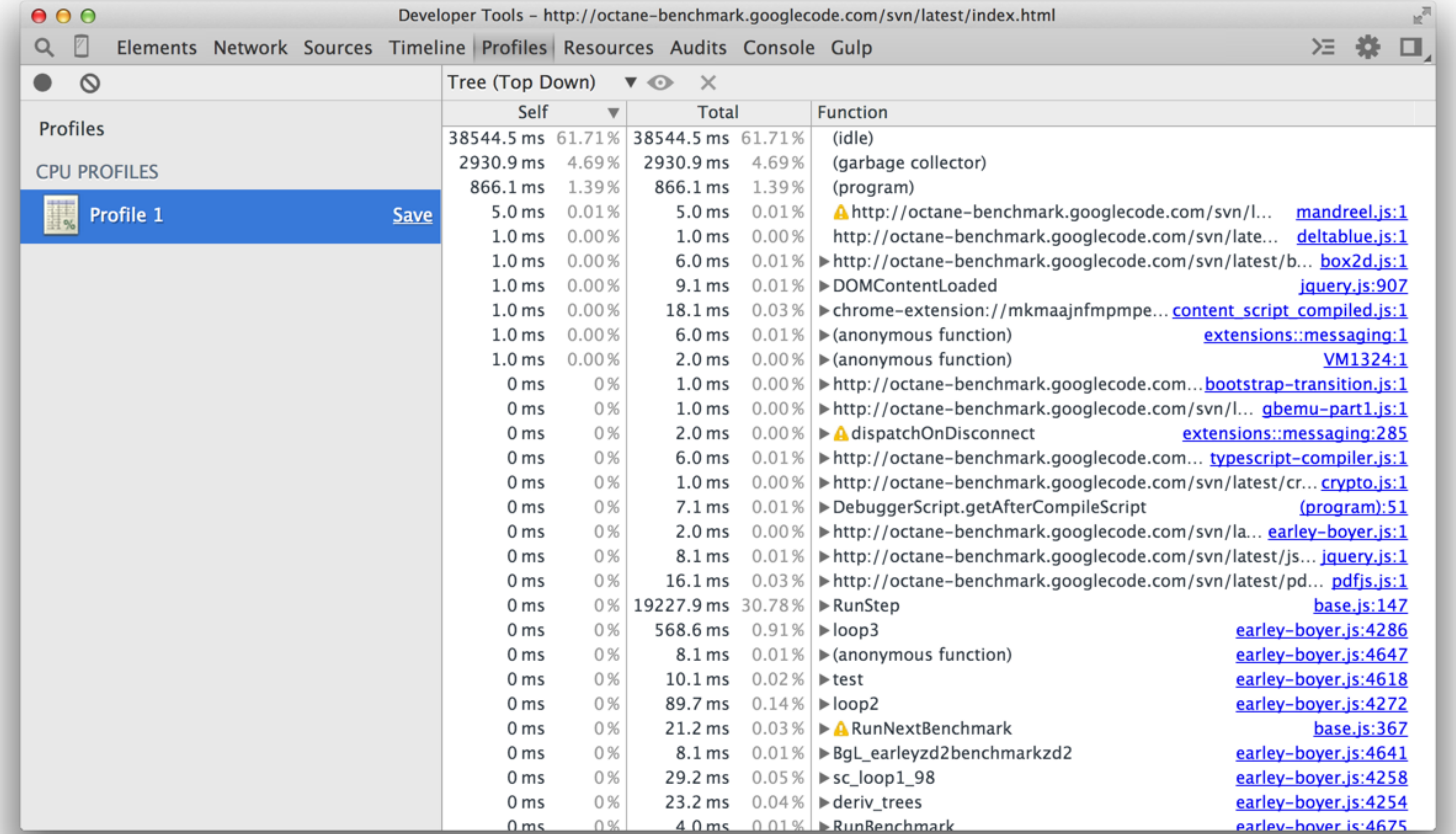



Paul Lewis
3:15PM
Detached Nodes
Elements
Network
Sources
Timeline
Profiles
Resources
Audits
Console
Polymer
Gulp
Summary
Class filter
Selected size: 685 MB
10.00 s
20.00 s
30.00 s
40.00 s
50.00 s
100 MB
Profiles
HEAP TIMELINES
Snapshot 1
685 MB
Save
Constructor
Distance
Objects Count
Shallow Size
Retained Size
Array 2 1 649 0 % 52 768 0 % 645 425 752 90 %
(array) 2 526 527 33 % 424 875 808 59 % 425 403 048 59 %
SlowPurchase 3 500 000 31 % 12 000 000 2 % 424 000 000 59 %
SlowPurchase @204079 3 24 0 % 848 0 %
SlowPurchase @204085 3 24 0 % 848 0 %
SlowPurchase @204091 3 24 0 % 848 0 %
SlowPurchase @204097 3 24 0 % 848 0 %
SlowPurchase @204103 3 24 0 % 848 0 %
SlowPurchase @204109 3 24 0 % 848 0 %
Retainers
Allocation stack
$Object.defineProperty.get
extensions::utils:103
dispatchOnMessage
extensions::messaging:299
(root)

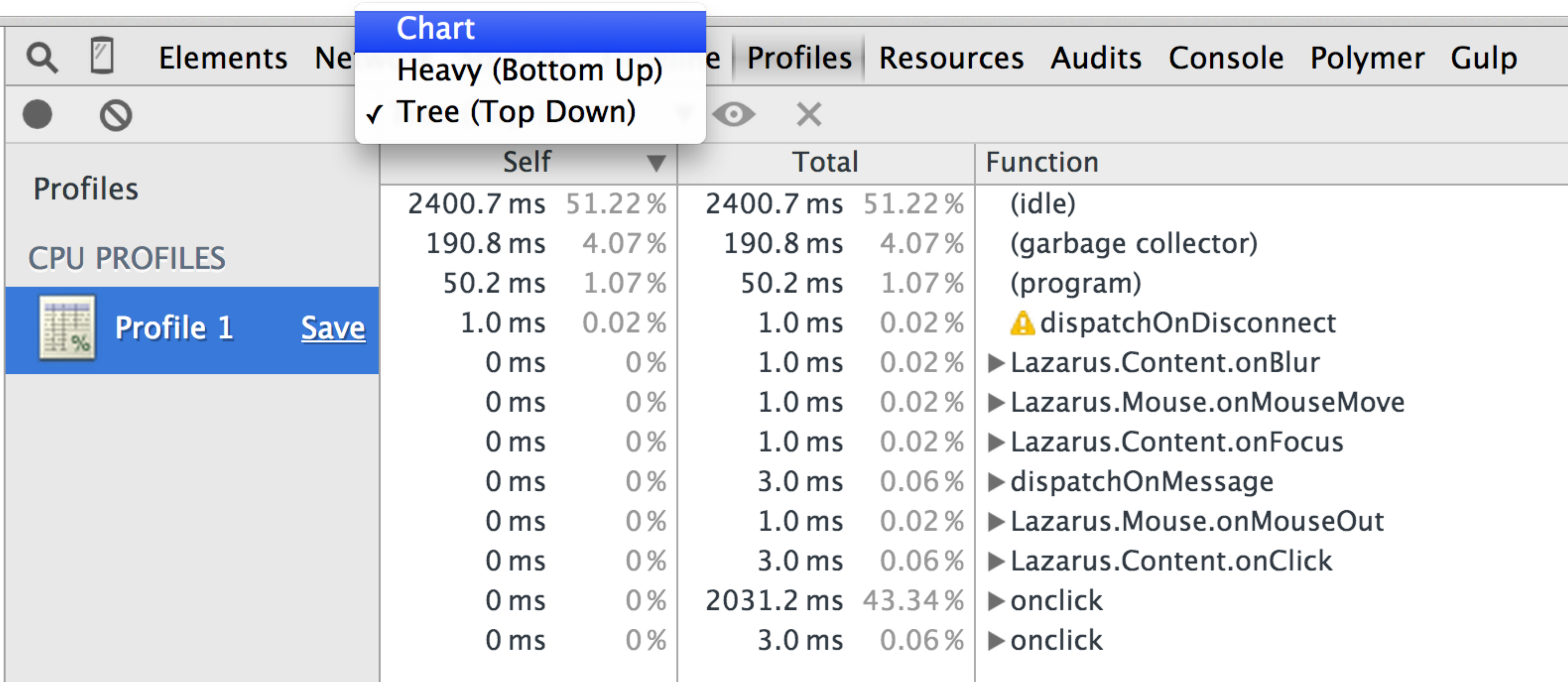
Visualize JS processing over time

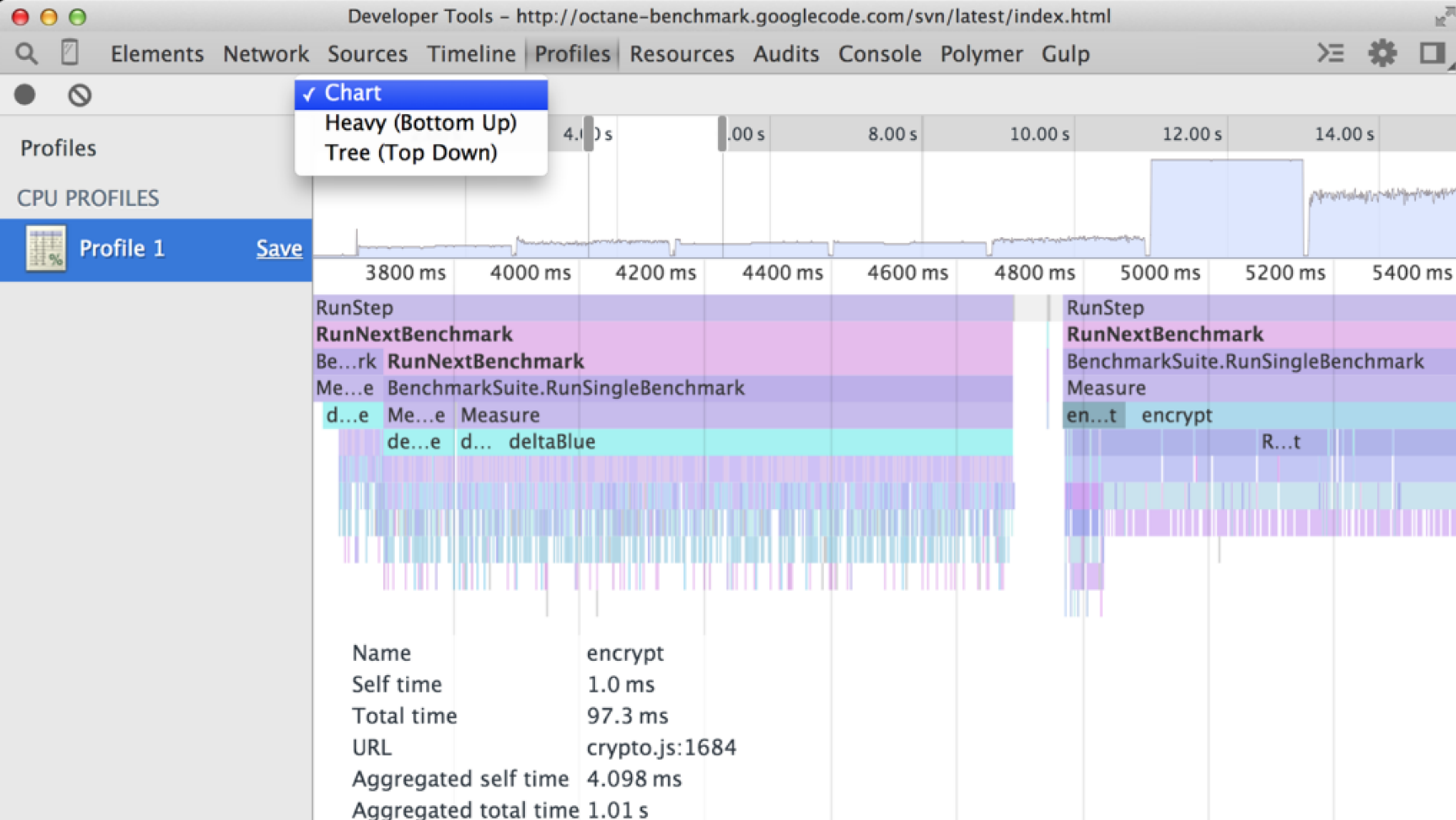
JavaScript CPU Profile (top down)

Shows where CPU time is statistically spent on your code.



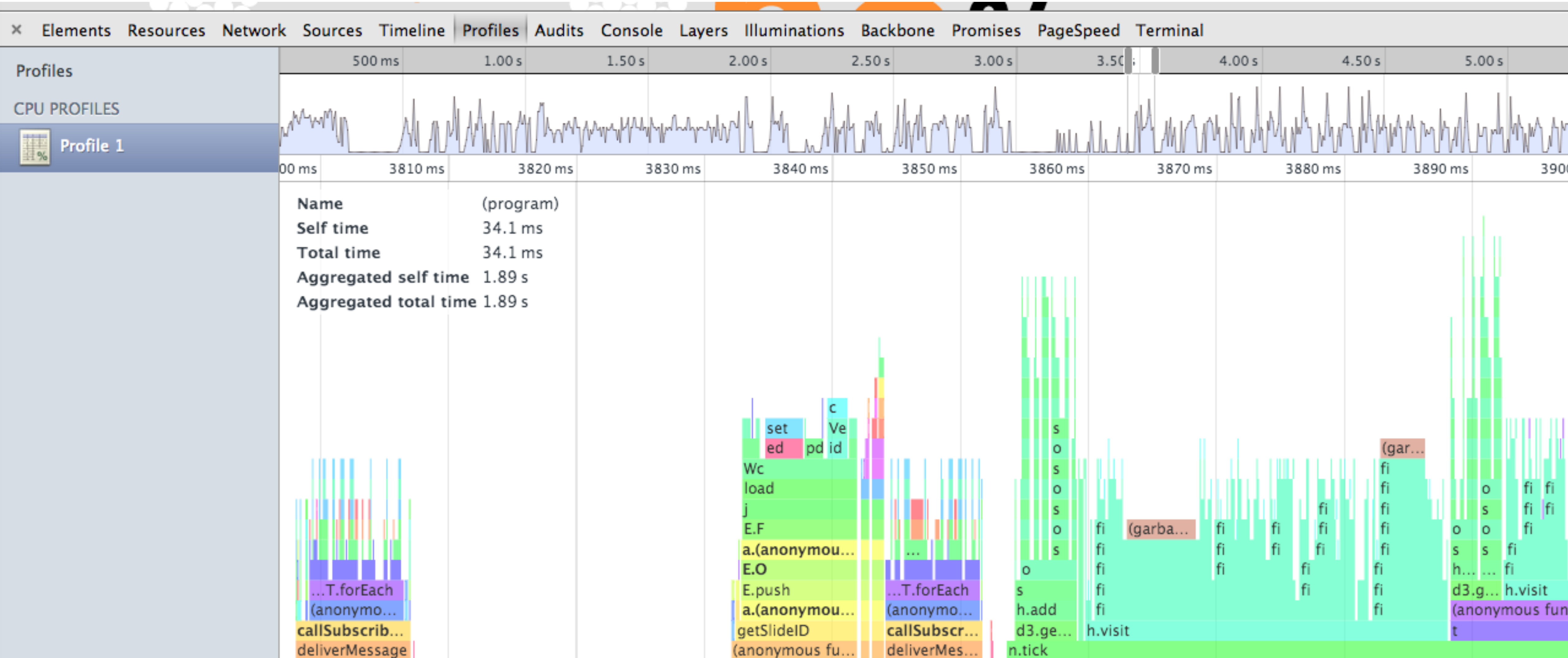
Select “Chart” from the drop-down




Developer Tools – http://octane-benchmark.googlecode.com/svn/latest/index.html
Elements
Network
Sources
Timeline
Profiles
Resources
Audits
Console
Polymer
Gulp
✓ Chart
Heavy (Bottom Up)
Tree (Top Down)
Profiles
CPU PROFILES
Profile 1
Save
8.00 s
10.00 s
12.00 s
14.00 s
3800 ms
4000 ms
4200 ms
4400 ms
4600 ms
4800 ms
5000 ms
5200 ms
5400 ms
RunStep
RunNextBenchmark
Be…rk
RunNextBenchmark
Me…e
BenchmarkSuite.RunSingleBenchmark
d…e
Me…e
Measure
de…e
d…
deltaBlue
RunStep
RunNextBenchmark
BenchmarkSuite.RunSingleBenchmark
Measure
en…t
encrypt
R…t
Name
encrypt
Self time
1.0 ms
Total time
97.3 ms
URL
crypto.js:1684
Aggregated self time
4.098 ms
Aggregated total time
1.01 s

Flame Chart View

Visualize JavaScript execution paths

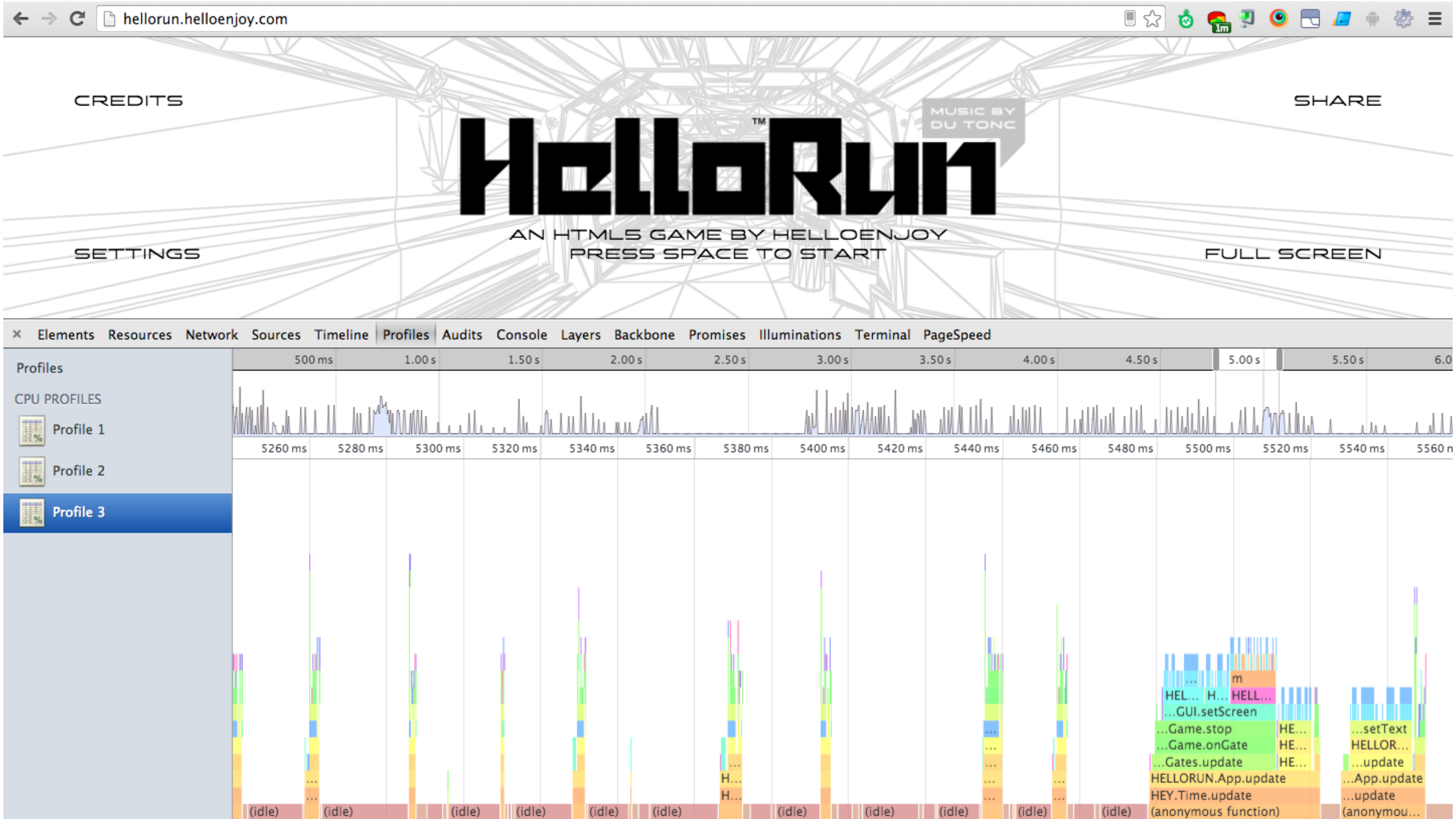


The Flame Chart

The Flame Chart provides a visual representation of JavaScript processing over time, similar to those found in the Timeline and Network panels. By analyzing and understanding function call progression visually you can gain a better understanding of the execution paths within your app.

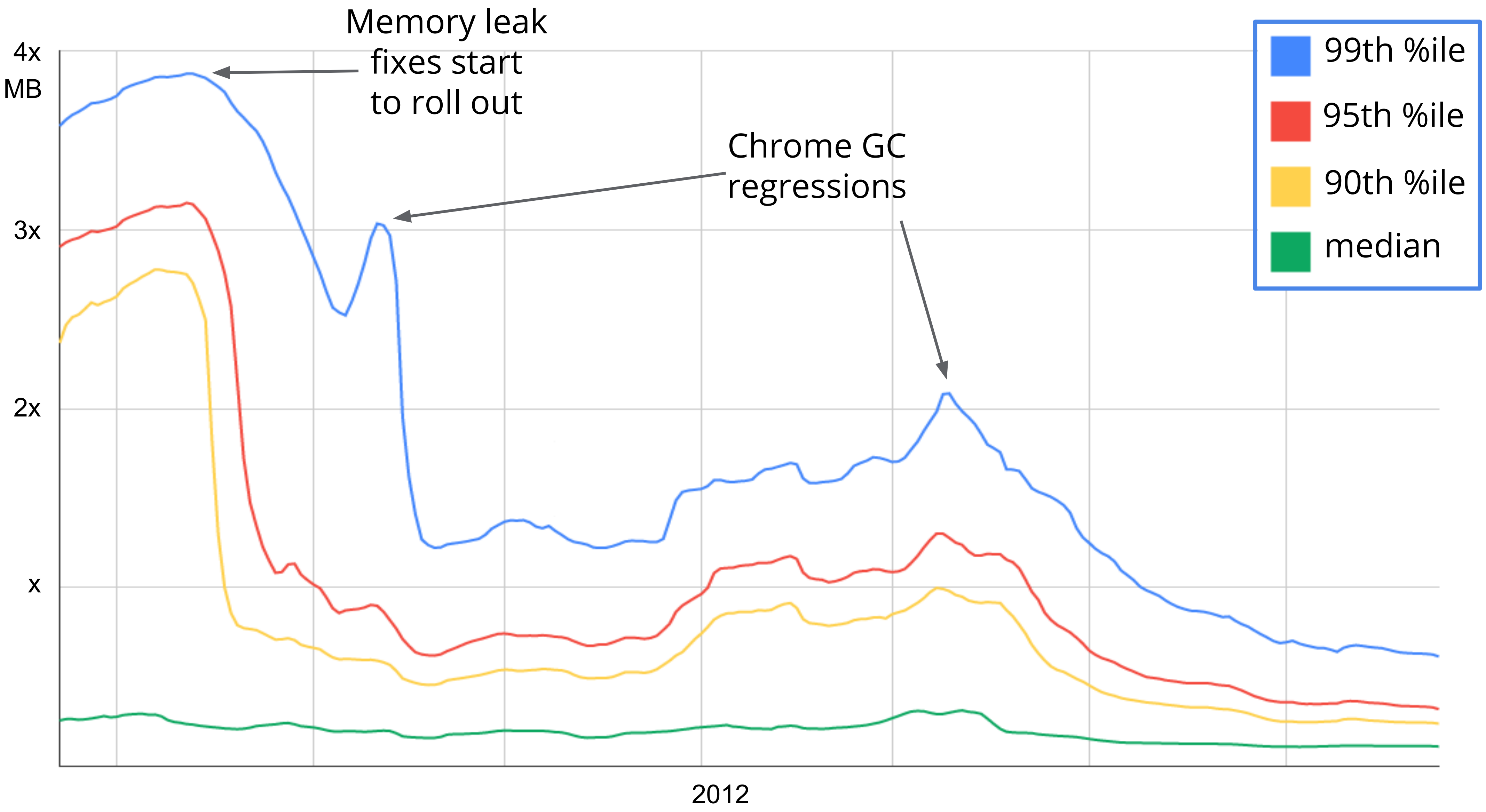
The height of all bars in a particular column is not significant, it simply represents each function call which occurred. What is important however is the width of a bar, as the length is related to the time that function took to execute.

Visualize profiler data against a time scale



Is optimization worth the effort?

GMail's memory usage (taken over a 10 month period)



"Through optimization, we reduced our memory footprint by 80% or more for power-users and 50% for average users."

Loreena Lee, GMail

Resources

Official Chrome DevTools docs

devtools.chrome.com

chrome

DEVTOOLS MULTI-DEVICE PLATFORM

JavaScript Memory Profiling

A **memory leak** is a gradual loss of available computer memory. It occurs when a program repeatedly fails to return memory it has obtained for temporary use. JavaScript web apps can often suffer from similar memory related issues that native applications do, such as **leaks** and bloat but they also have to deal with **garbage collection pauses.**

Although JavaScript uses garbage collection for automatic memory management, **effective** memory management is still important. In this guide we will walk through profiling memory issues in JavaScript web apps. Be sure to try the **supporting demos** when learning about features to improve your awareness of how the tools work in practice.

> **Note:** Some of these features we will be using are currently only available in **Chrome Canary**. We recommend using this version to get the best memory profiling tooling for your applications.

Questions to ask yourself

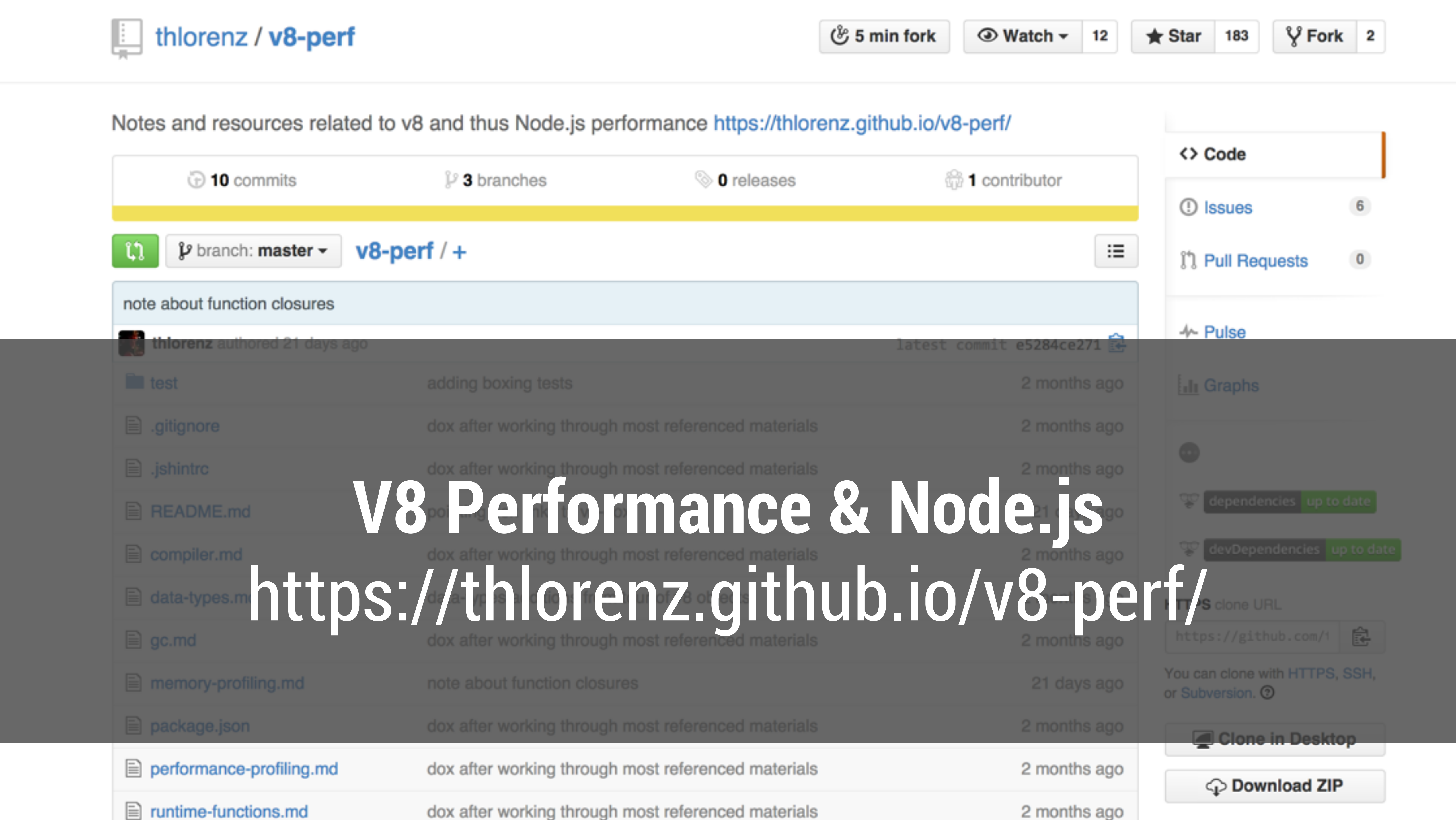
In general, there are three questions you will want to answer when you think you have a memory leak:

JavaScript Memory Profiling

JavaScript Memory Profiling

Demos

Contents

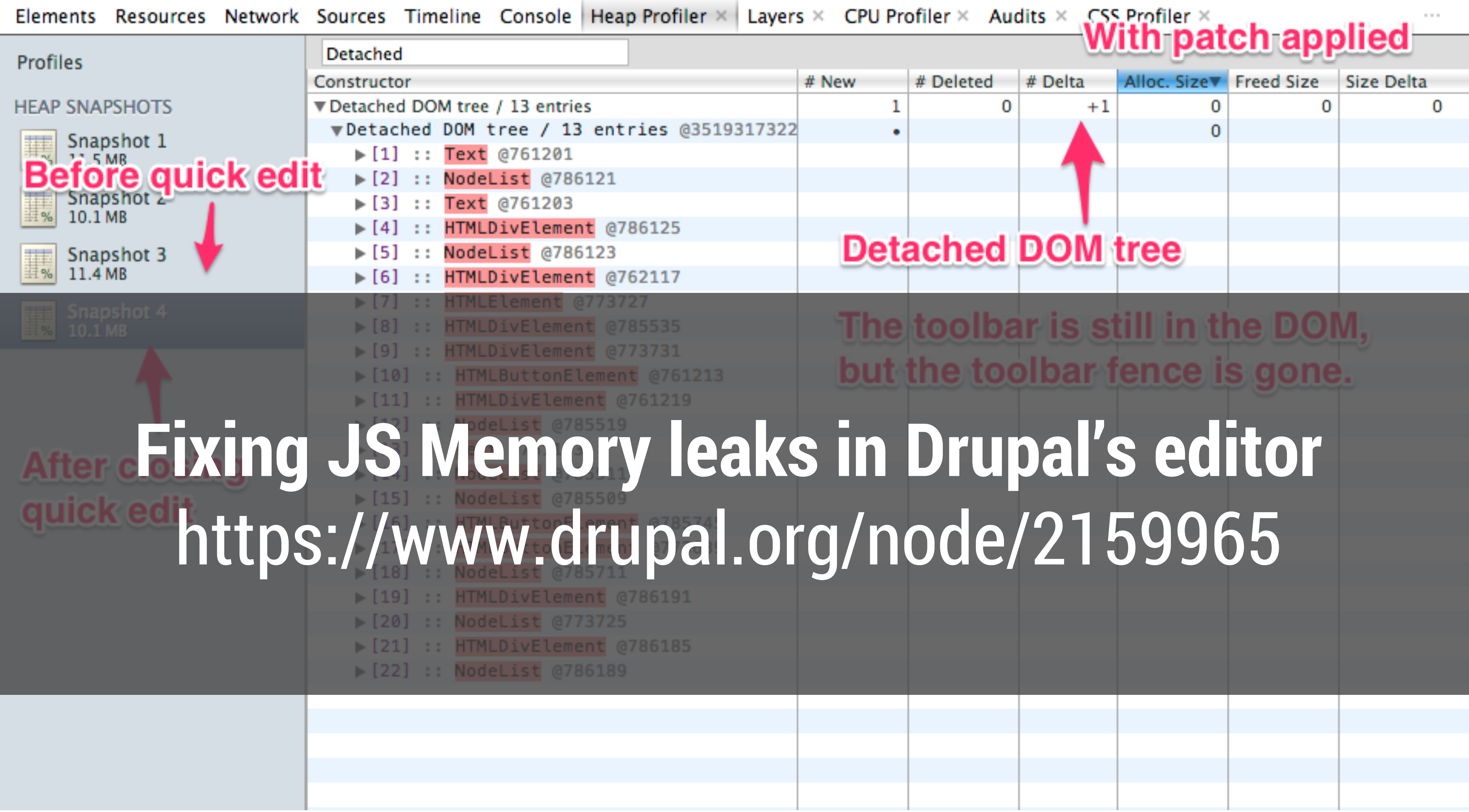


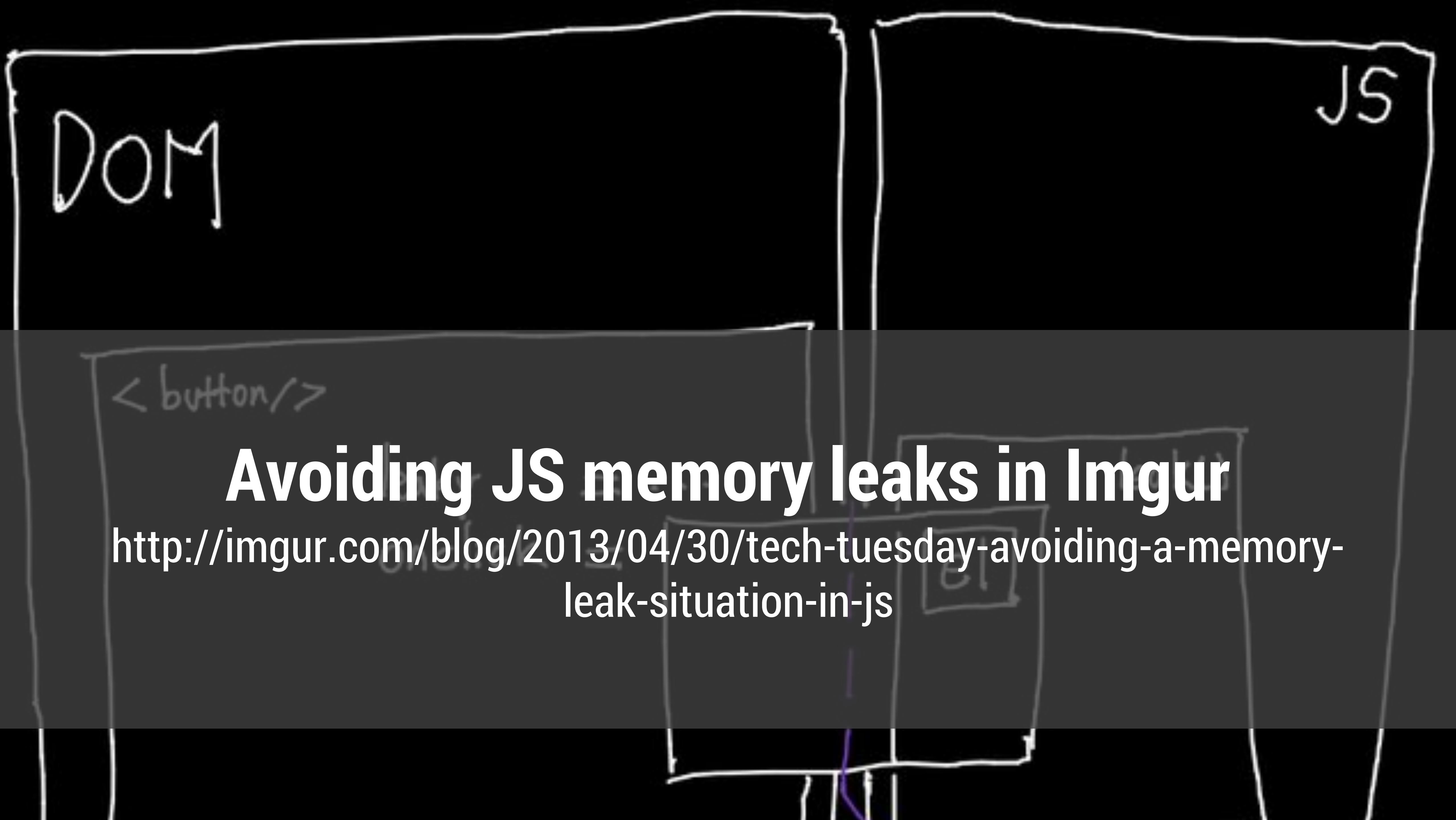
V8 Performance & Node.js

https://thlorenz.github.io/v8-perf/

Writing Memory-efficient JavaScript

http://www.smashingmagazine.com/2012/11/05/writing-fast-memory-efficient-javascript/


Fixing JS Memory leaks in Drupal's editor
https://www.drupal.org/node/2159965
With patch applied
Before quick edit
After closing quick edit
Detached DOM tree
The toolbar is still in the DOM, but the toolbar fence is gone.
Elements Resources Network Sources Timeline Console Heap Profiler Layers CPU Profiler Audits CSS Profiler
Profiles
HEAP SNAPSHOTS
Snapshot 2
10.1 MB
Snapshot 3
11.4 MB
Snapshot 4
10.1 MB
Detached
Constructor
New
Deleted
Delta
Alloc. Size
Freed Size
Size Delta
Detached DOM tree / 13 entries
Detached DOM tree / 13 entries @3519317322
[1] :: Text @761201
[2] :: NodeList @786121
[3] :: Text @761203
[4] :: HTMLDivElement @786125
[5] :: NodeList @786123
[6] :: HTMLDivElement @762117



Avoiding JS memory leaks in Imgur

http://imgur.com/blog/2013/04/30/tech-tuesday-avoiding-a-memory-leak-situation-in-js

StrongLoop: Memory profiling with DevTools

http://strongloop.com/strongblog/node-js-performance-tip-of-the-week-memory-leak-diagnosis/

Checklist

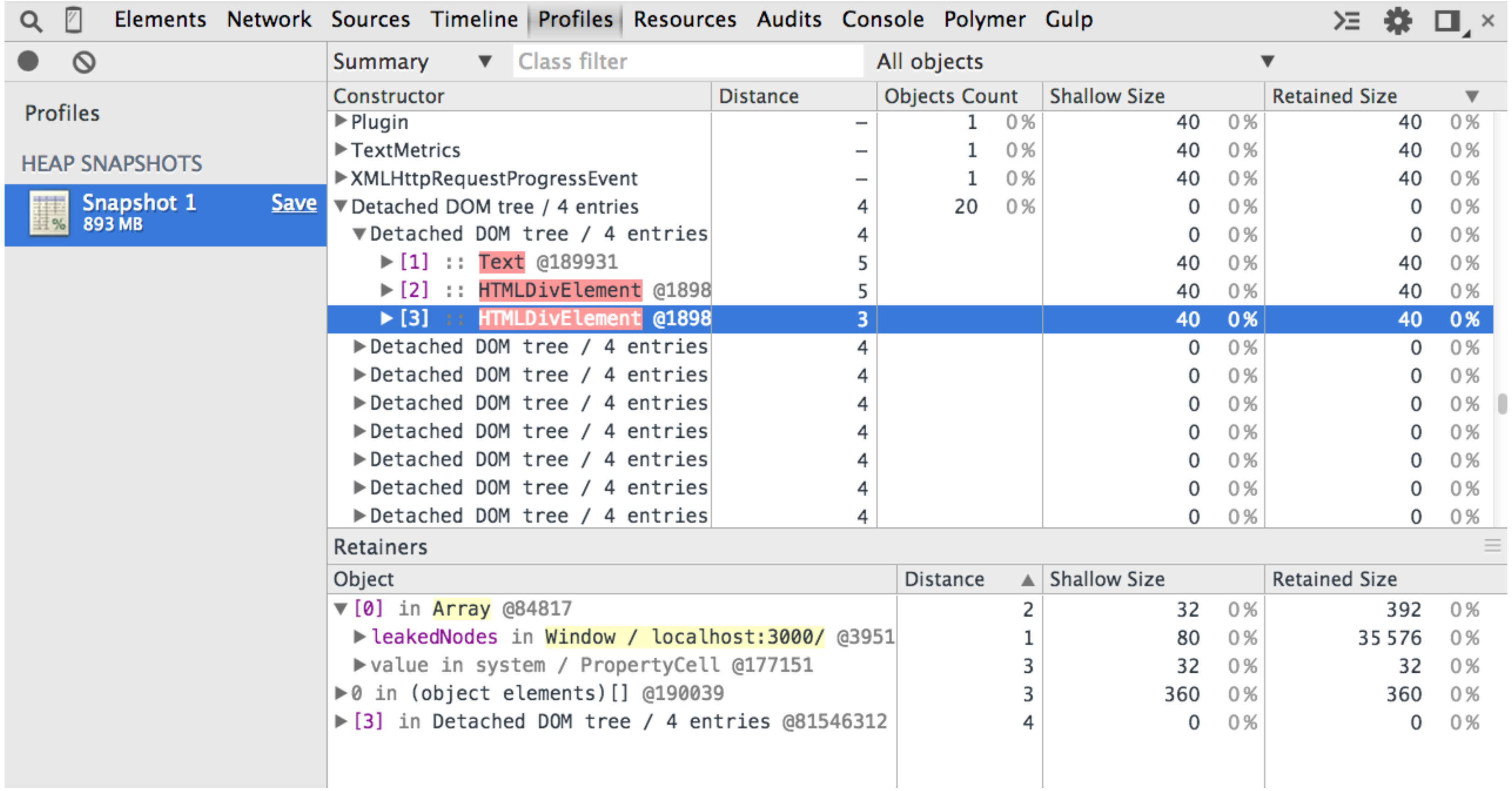
Ask yourself these questions:

- How much memory is your page using?
- Is your page leak free?
- How frequently are you GCing?

Know Your Arsenal.

Chrome DevTools

- window.performance.memory
- Timeline Memory view
- Heap Profiler
- Object Allocation Tracker




EASTWOOD AND CHAIR
OH MY GOD, I LOVED IT ALL!

Thank you!

+AddyOsmani
@addyosmani

#perfmatters