



JavaScript

Memory Management Masterclass



@addyosmani
+AddyOsmani

DevTools Demos

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

Chrome Task Manager

Memory Timeline

Heap Profiler

Object Allocation Tracker

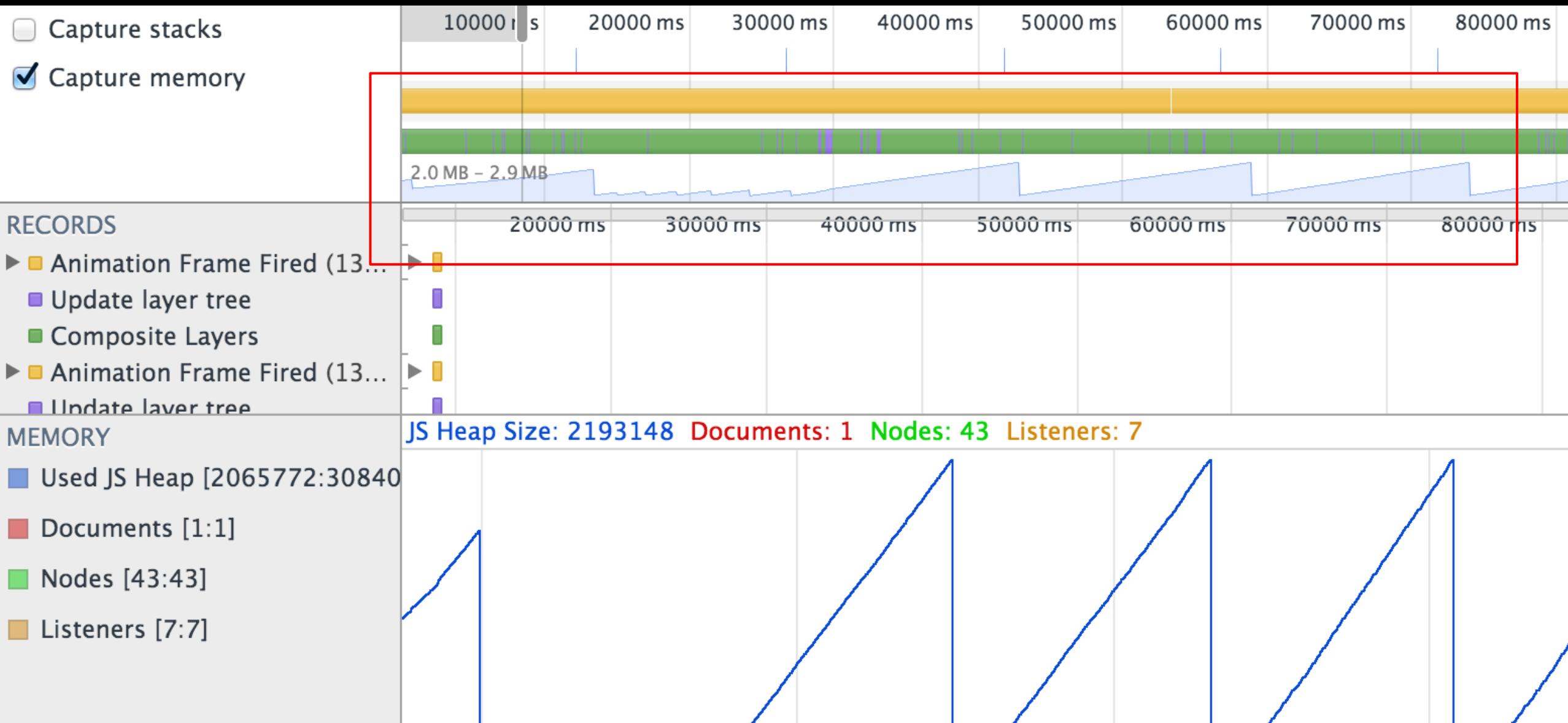
The Sawtooth Curve

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;  
    };  
}());
```

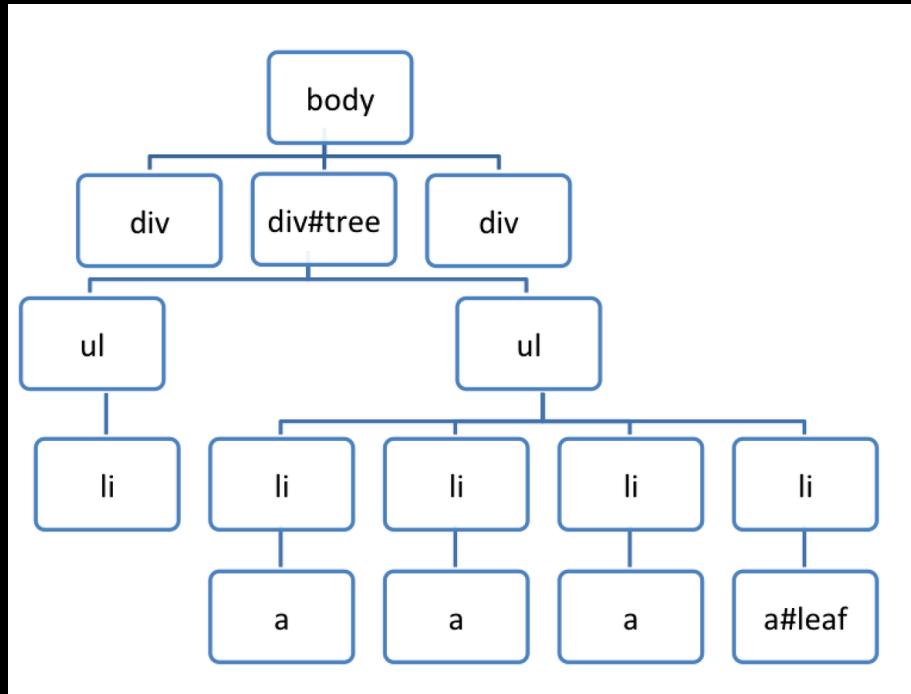
DOM Leaks

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  
        privatId = 0;  
  
    function Person(name) {  
        Object.defineProperty(this, "_id", { value:  
privatId++ });  
  
        privateData[this._id] = {  
            name: name  
        };  
    }  
  
    Person.prototype.getName = function() {  
        return privateData[this._id].name;  
    };  
  
    return Person;  
}());
```

```
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;  
}());
```

Avoid memory leaks
by holding refs to
properties weakly.

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

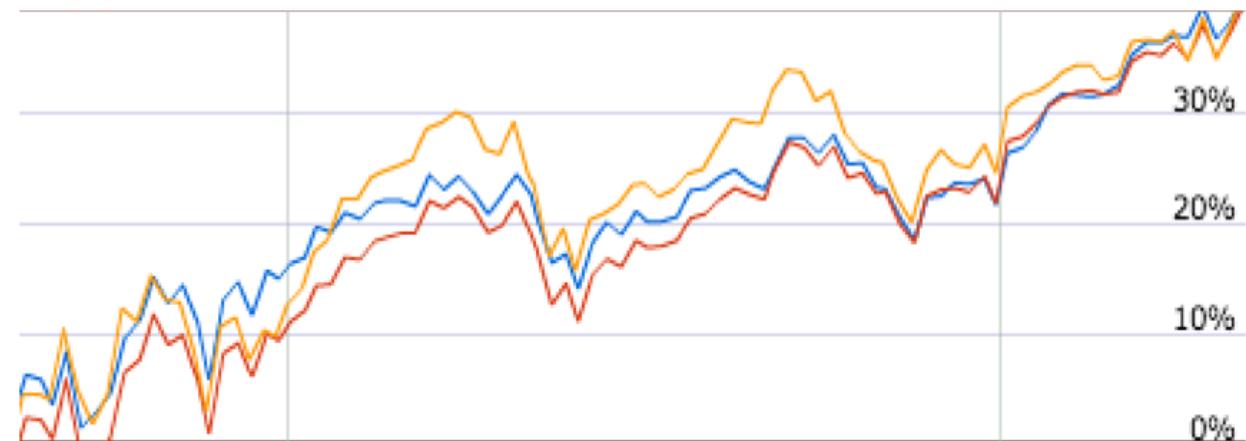
- Avoid long-lasting refs to **DOM elements** you no longer need
- Avoid **circular** object references
- Use appropriate **scope**
- Unbind **event listeners** that aren't needed anymore
- Manage **local cache** of data. Use an aging mechanism to get rid of old objects.

V8 Deep Dive.

Why does #perfmatter?

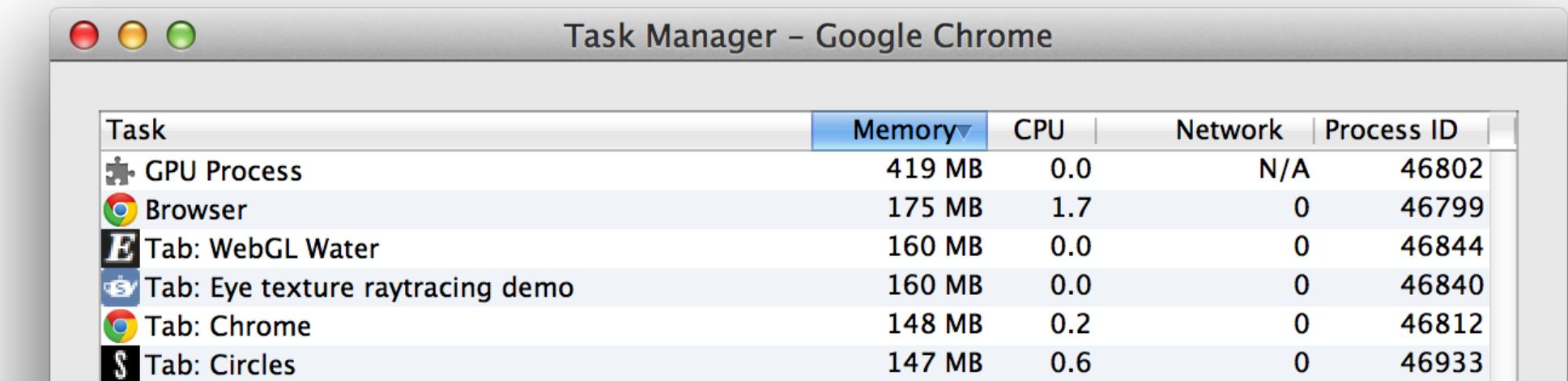
Silky smooth apps.

*Longer battery life
Smoother interactions
Apps can live longer*



Nothing is free.

Tools > Task Manager



A screenshot of the Google Chrome Task Manager window. The title bar reads "Task Manager – Google Chrome". The window contains a table with the following data:

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

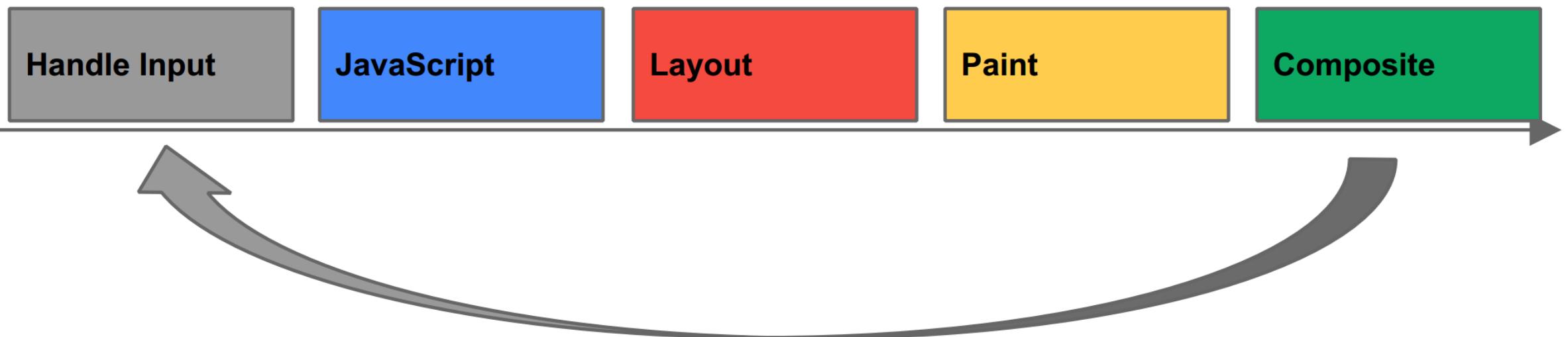
Popular sites

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.



Aw, Snap!

Something went wrong while displaying this webpage. To continue, reload or go to another page.

If you're seeing this frequently, try these [suggestions](#).



He's dead, Jim!

The process for the webpage was terminated for some other reason. To continue, reload or go to another page.

[Learn more](#)

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;
```

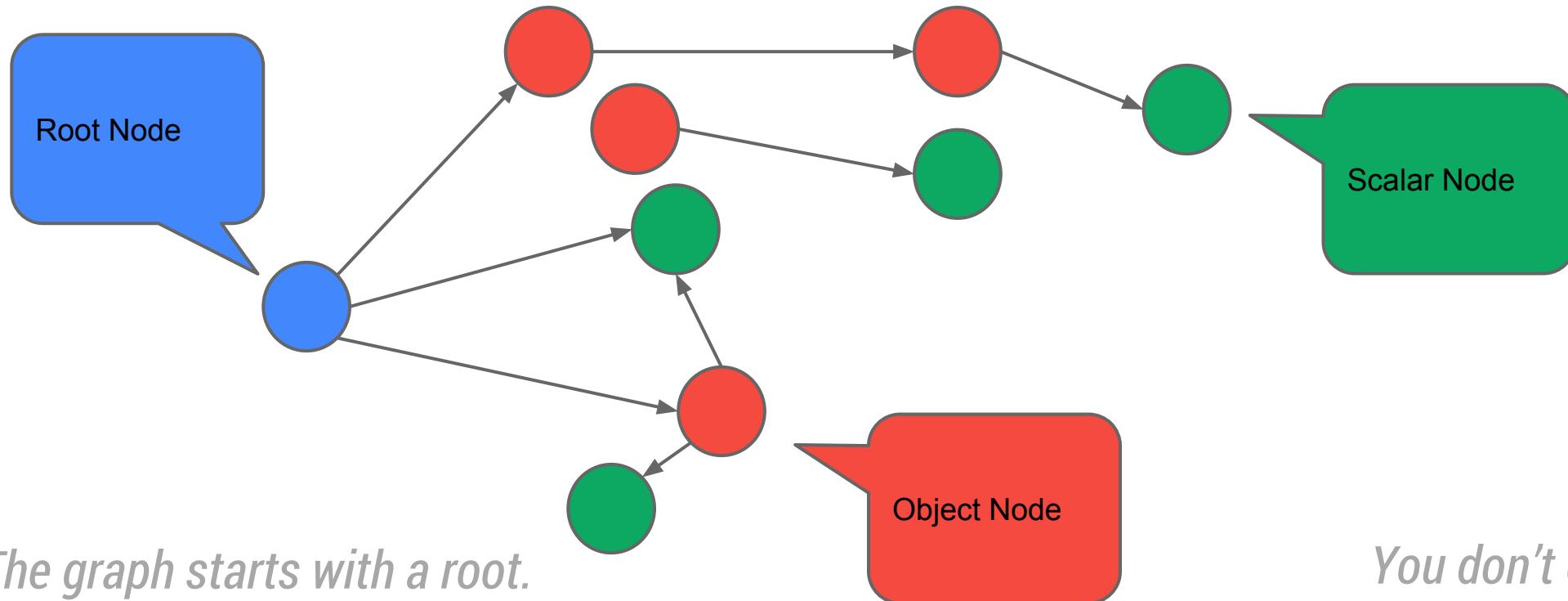
String only

Any variable type

Think of memory as a graph

The value graph

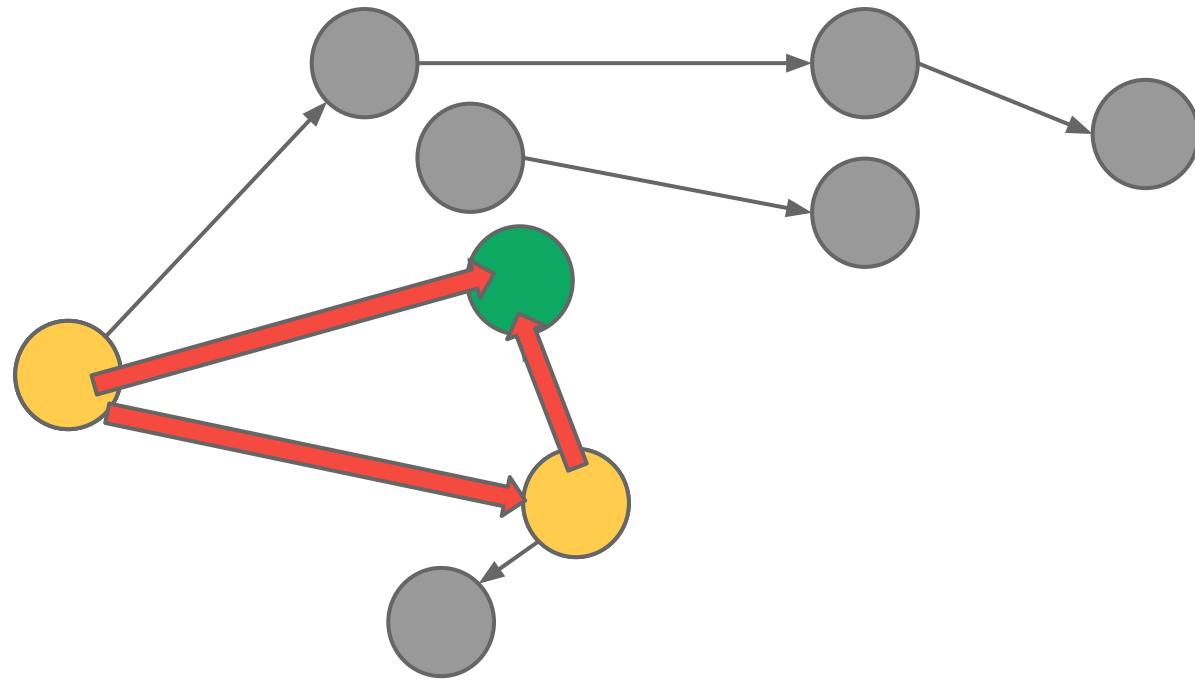
Root could be browser “window” or Global object of a Node module.



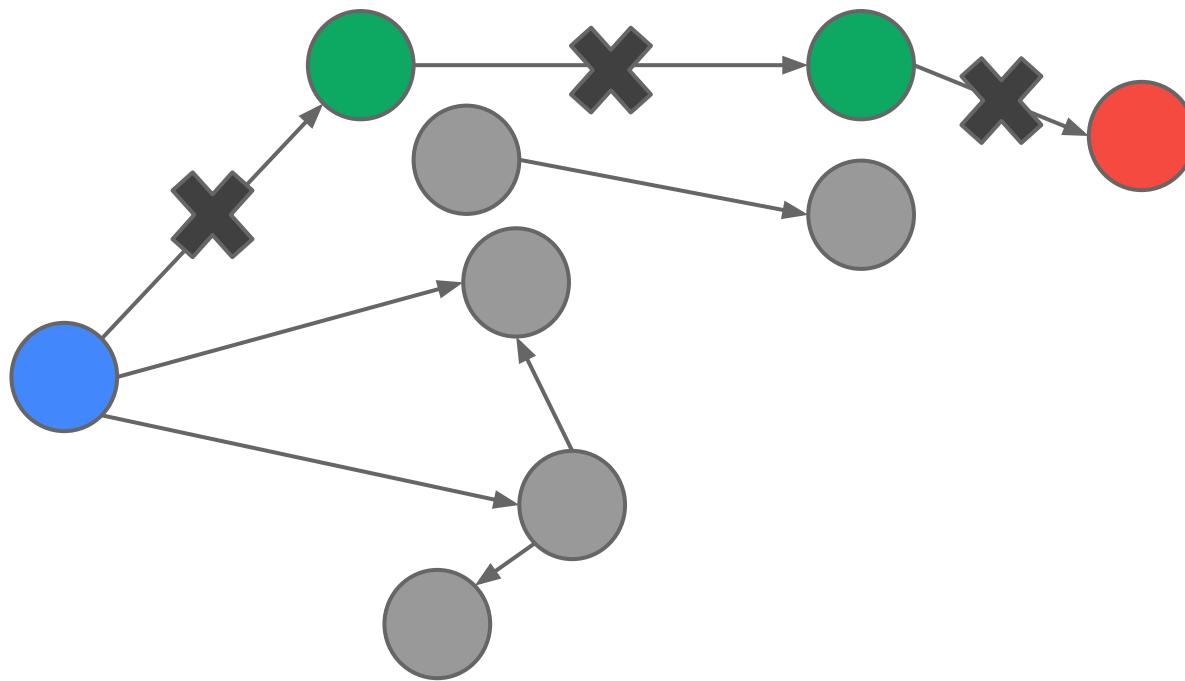
The graph starts with a root.

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

A value's retaining path(s)

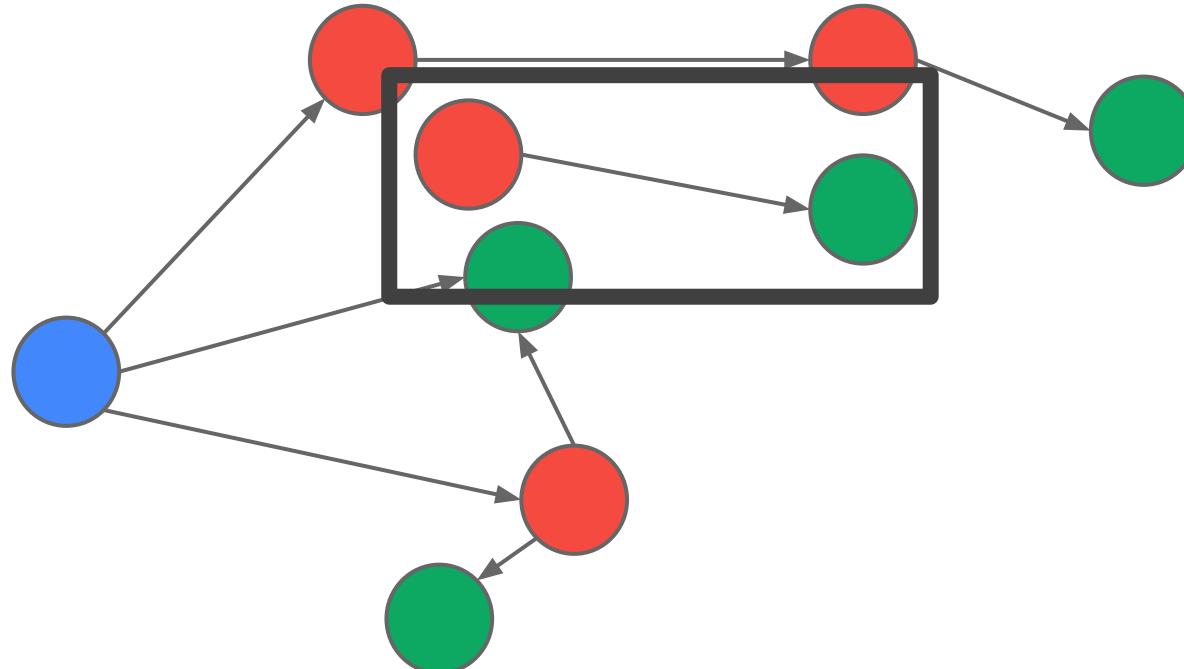


Removing a **value** from the graph



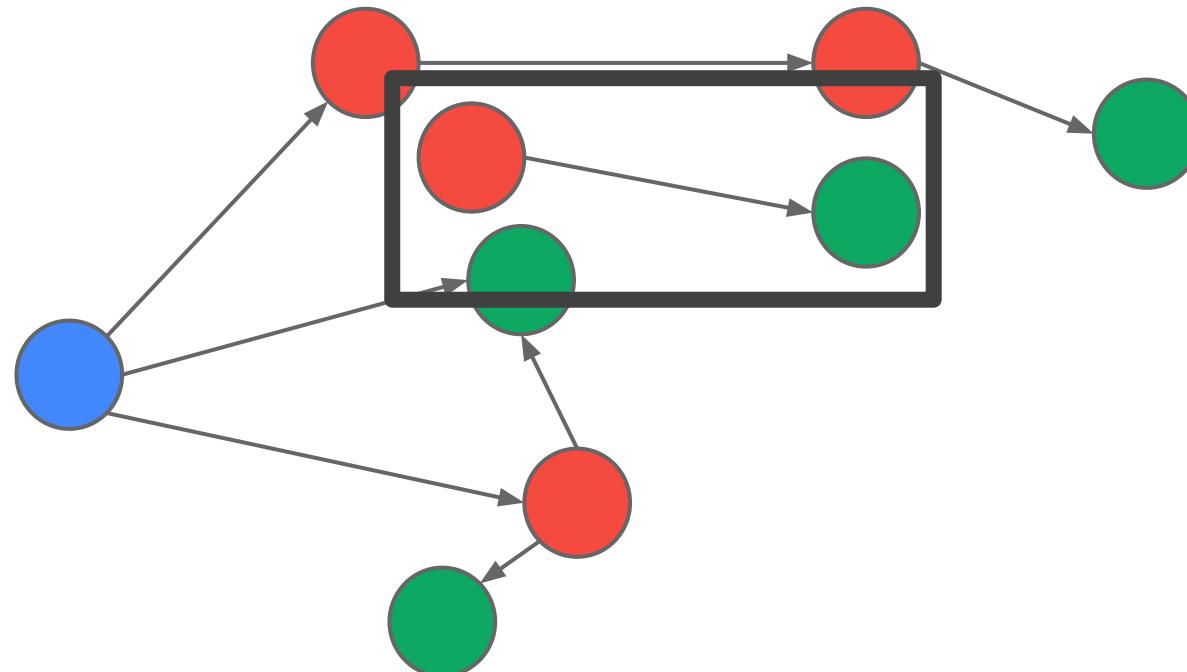
What is garbage?

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

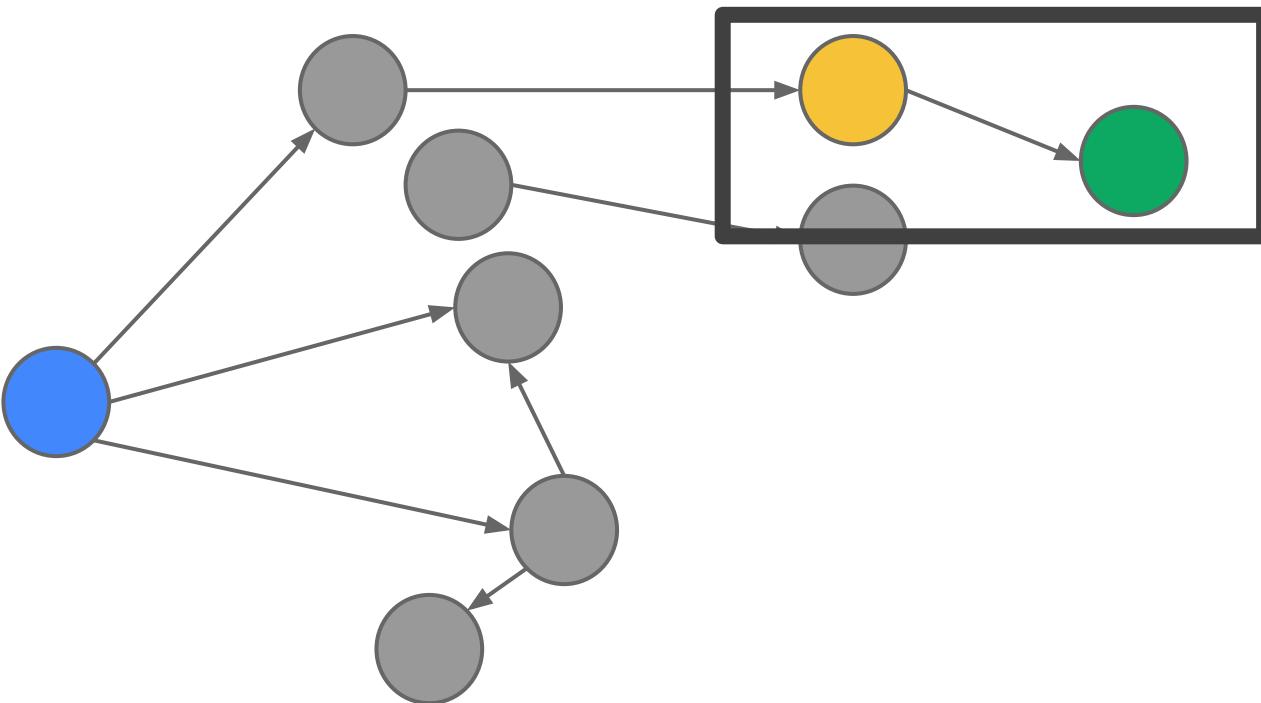


What is **garbage collection**?

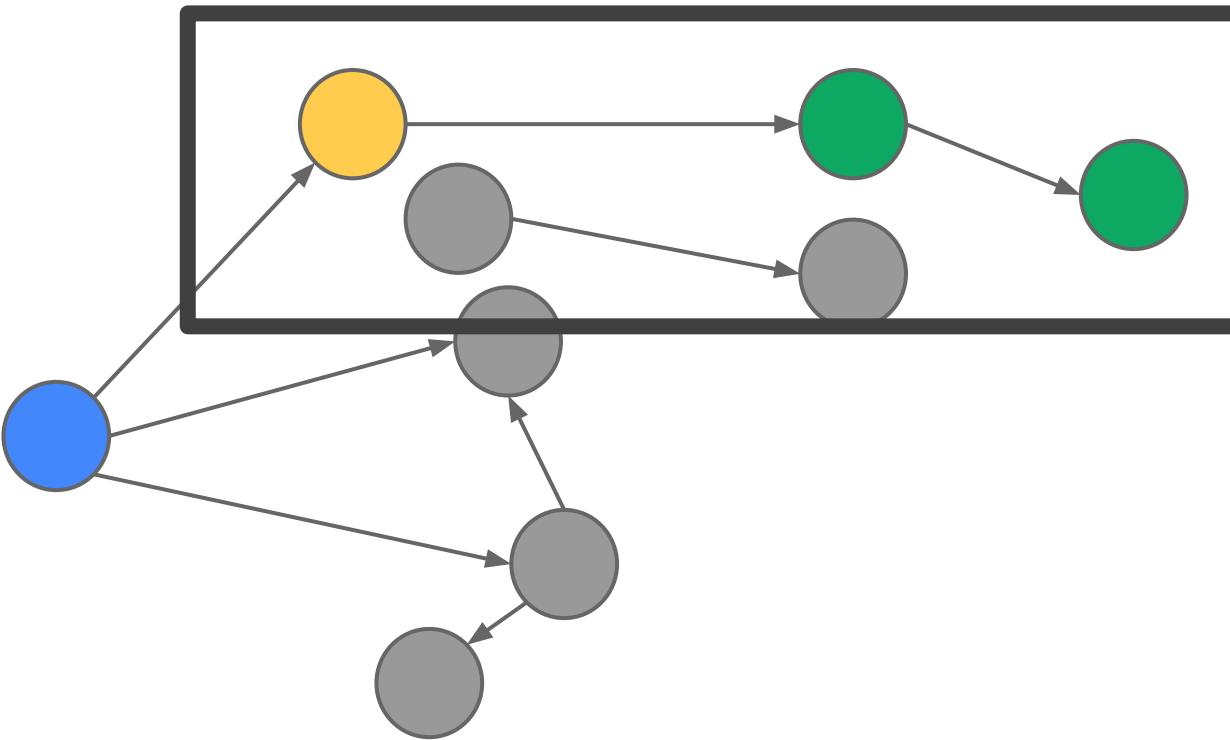
1. Find all live values
2. Return memory used by dead values to system



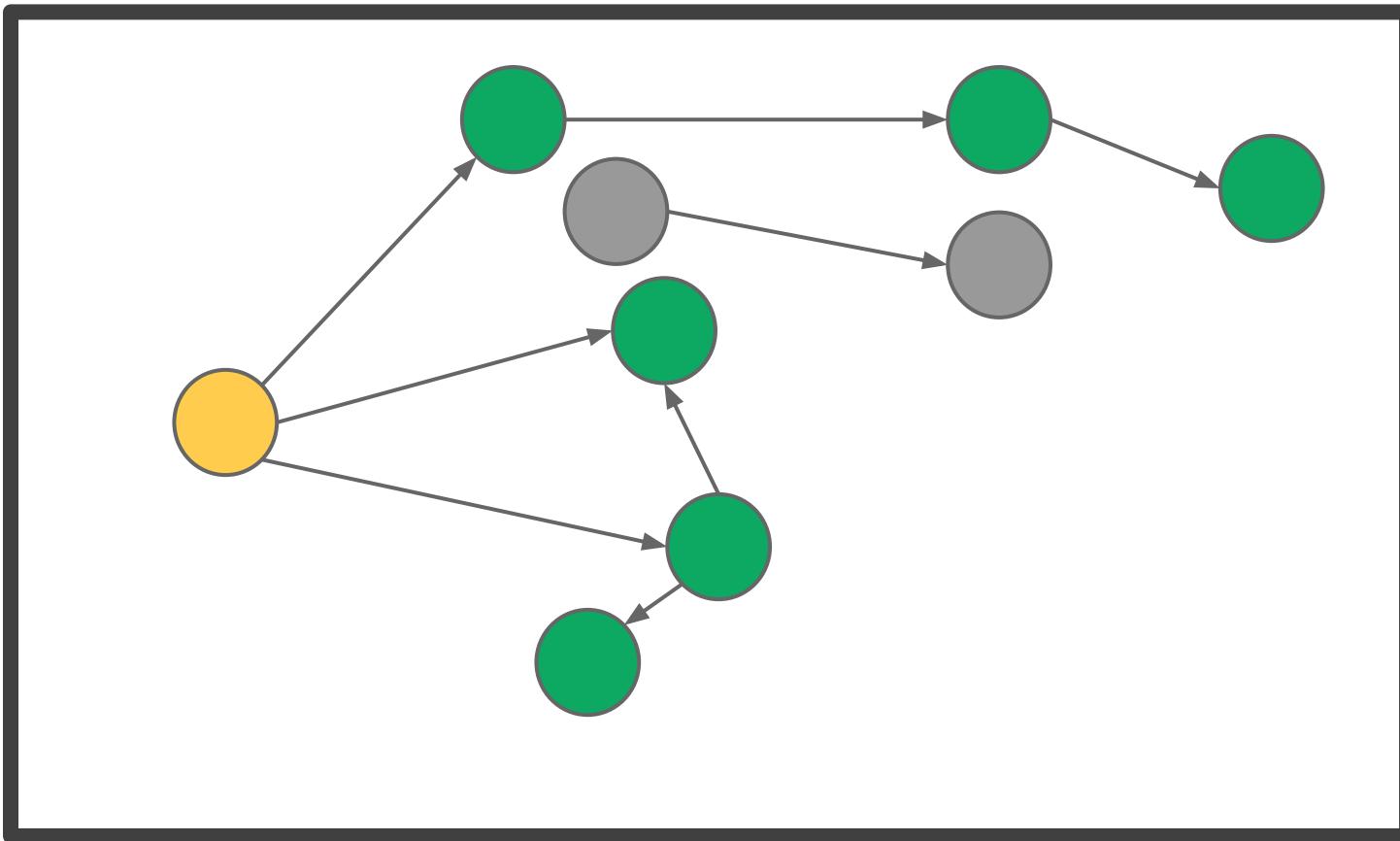
A **value's** retained size



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.

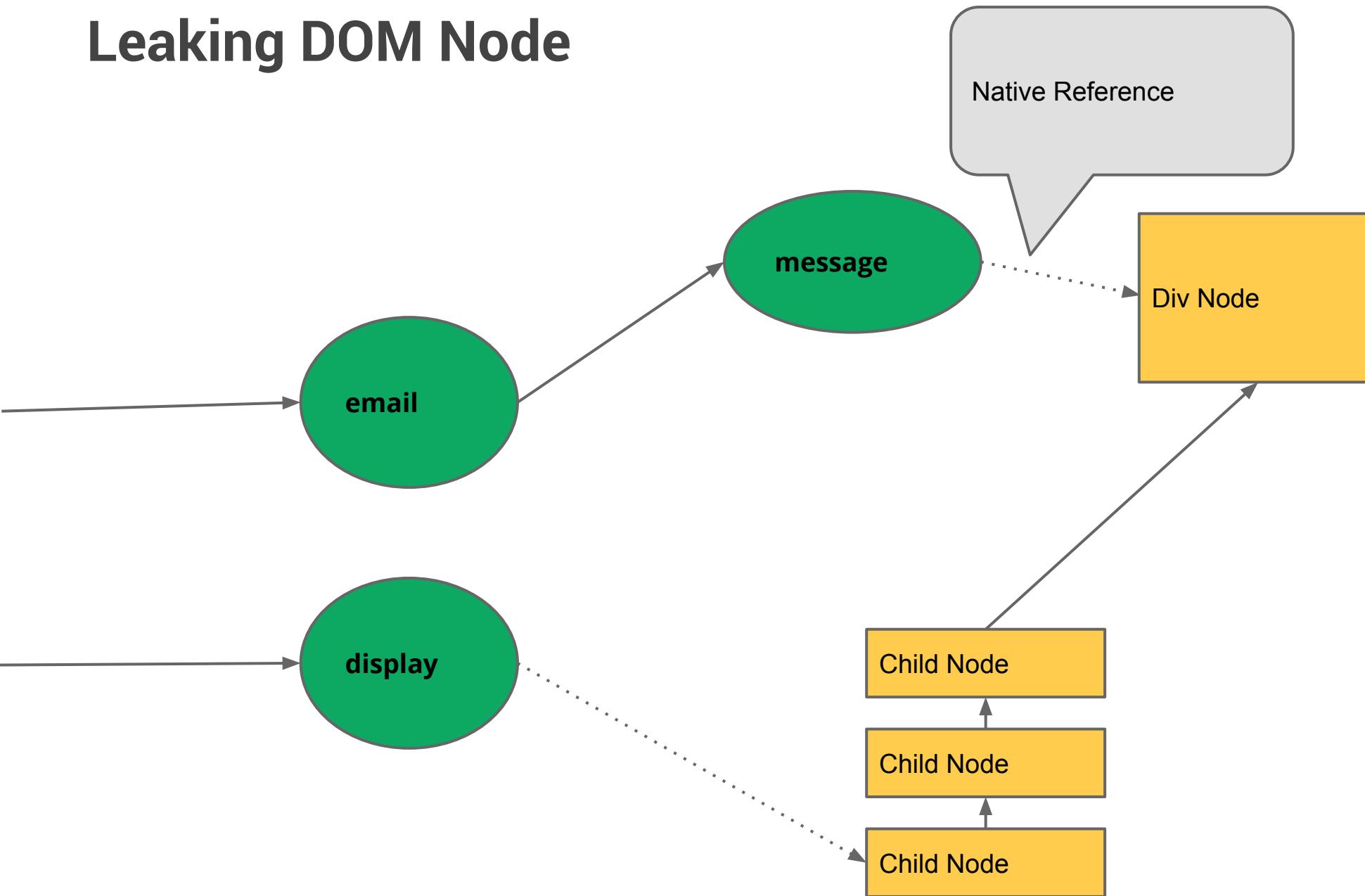
Leaks in JavaScript

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

JavaScript

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

Leaking DOM Node



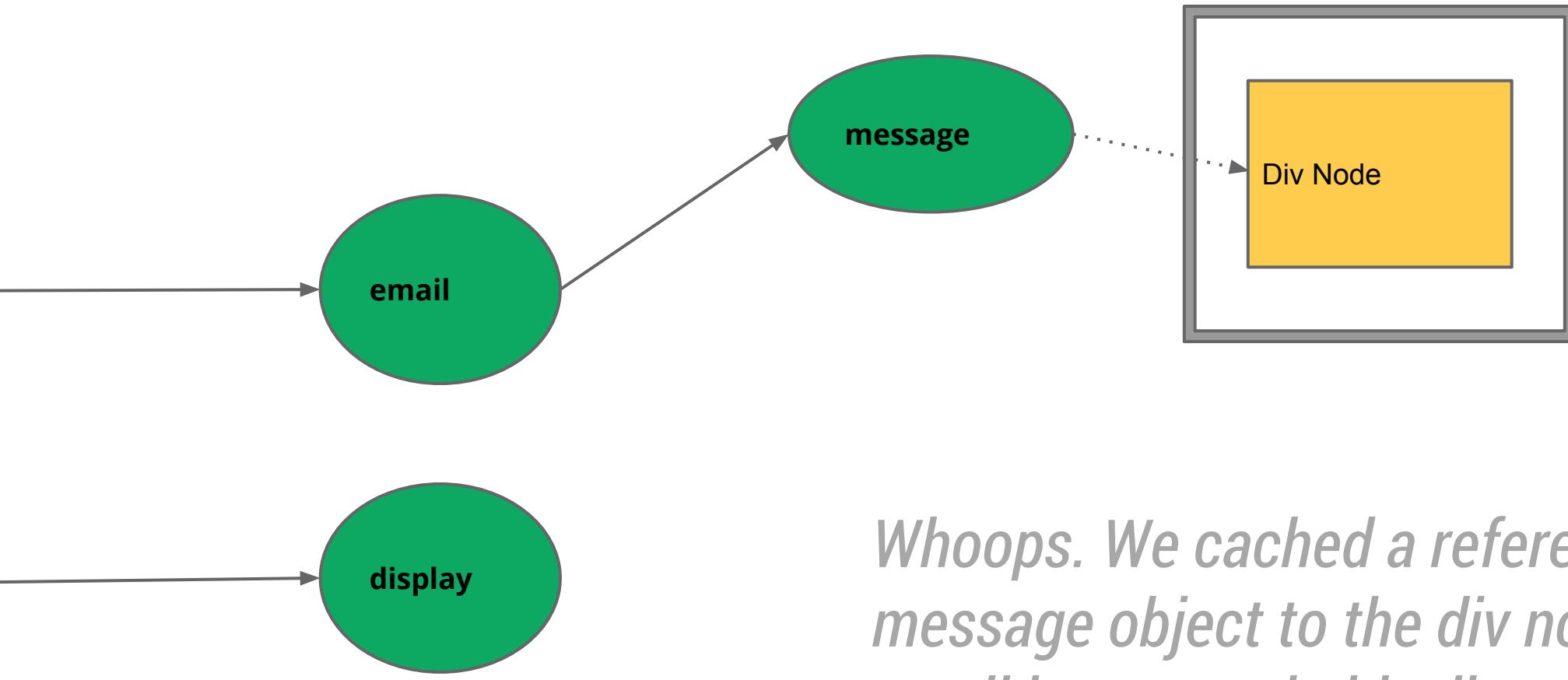
Leaks in JavaScript

*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)

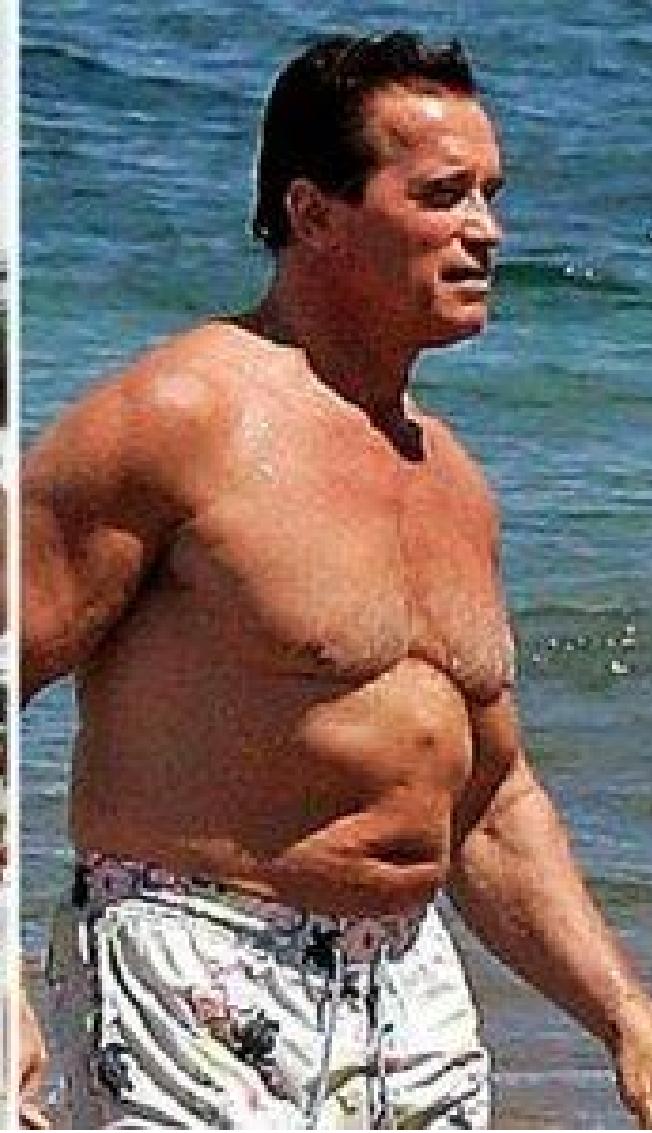
V8 memory management

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



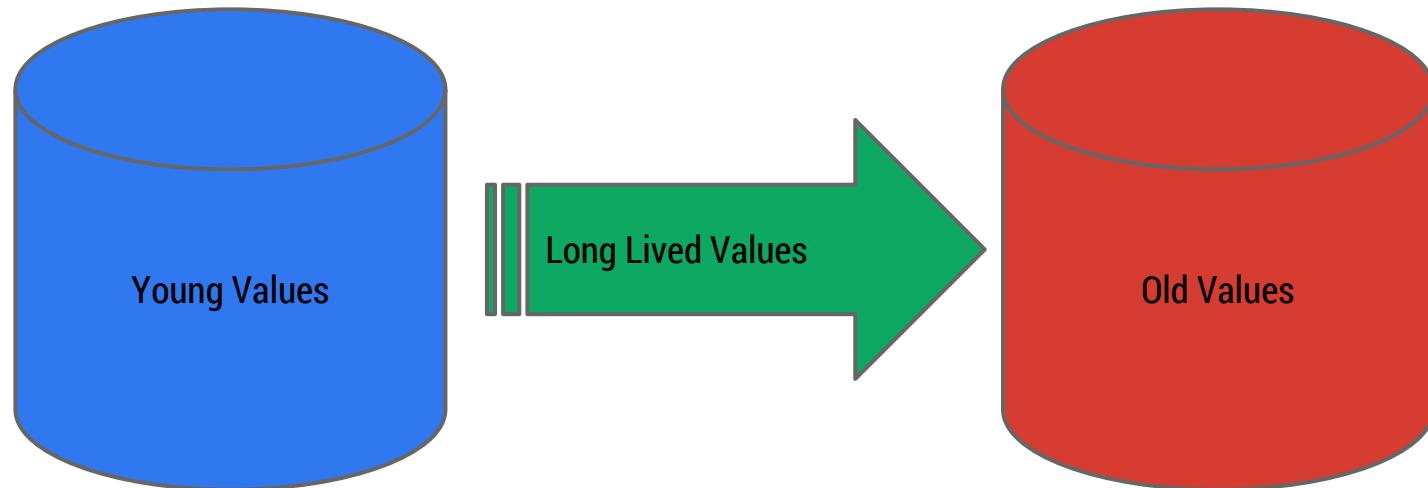
Old generation

How does V8 manage memory?

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

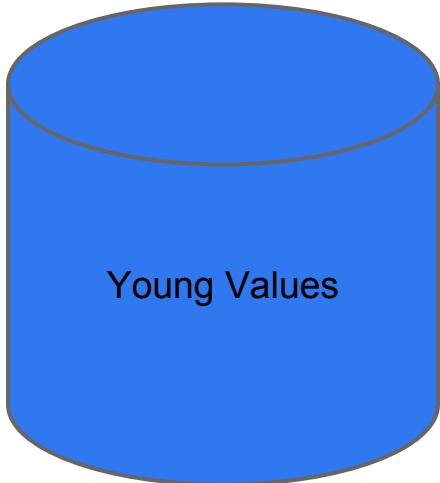
By young and old we mean how long has the JS value existed for.

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

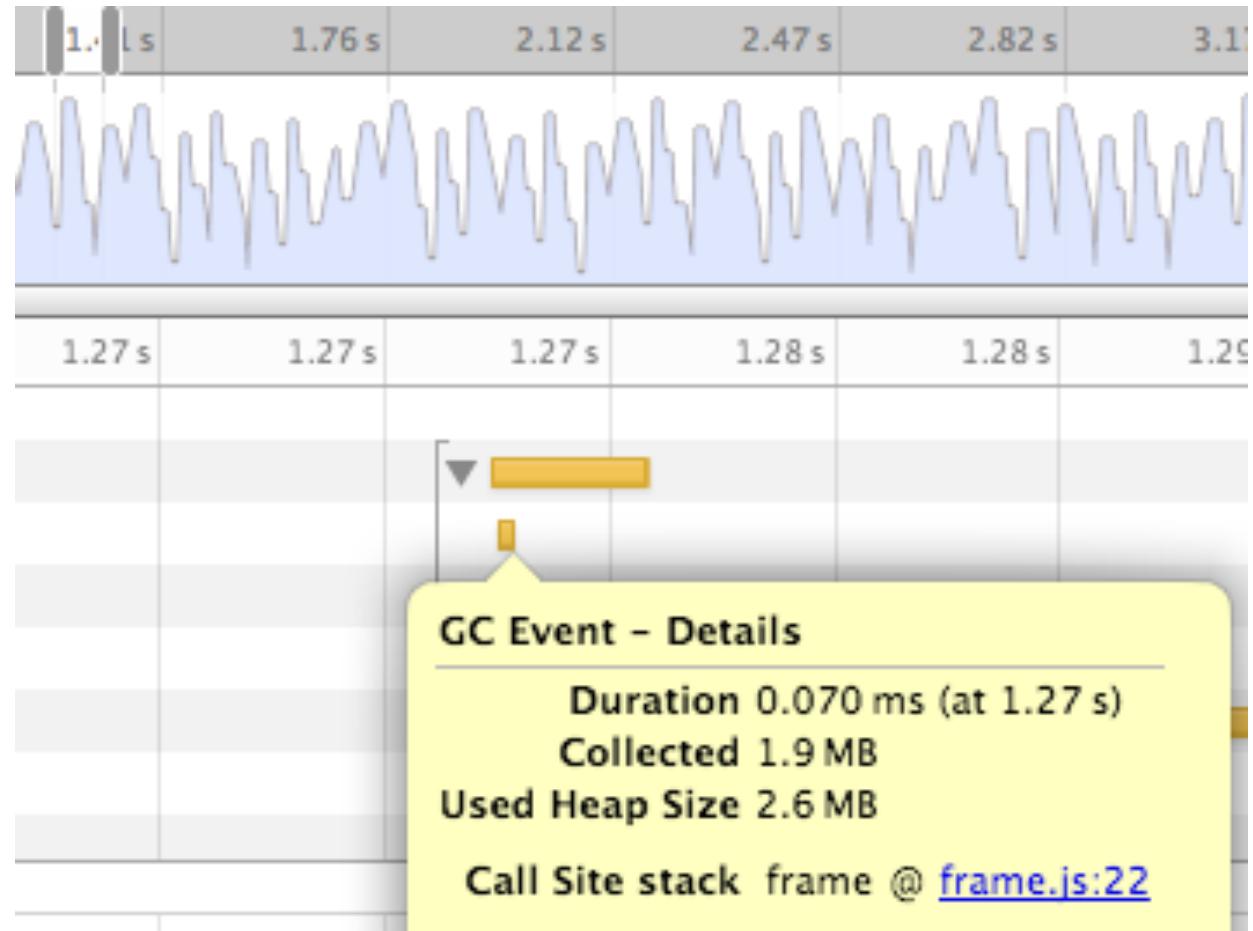


How does V8 manage memory?

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



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



How does V8 manage memory?

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



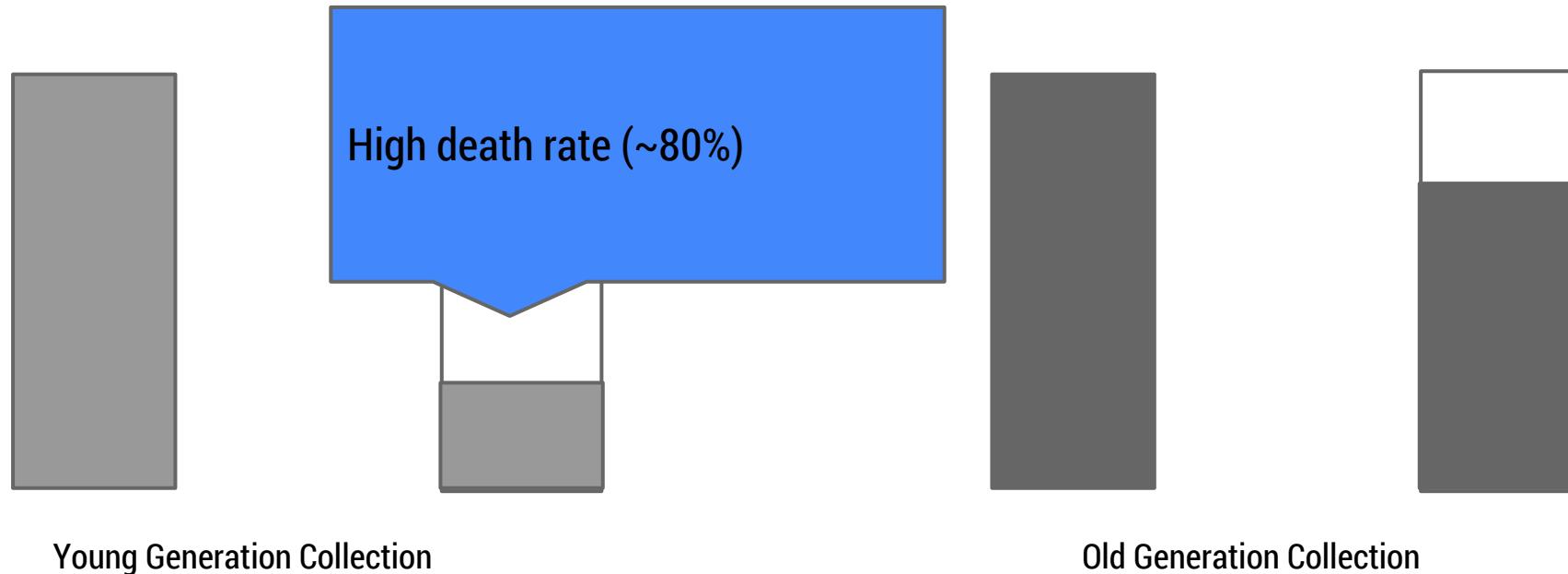
Some of the old generation's collection occurs in parallel with your page's execution.

- 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

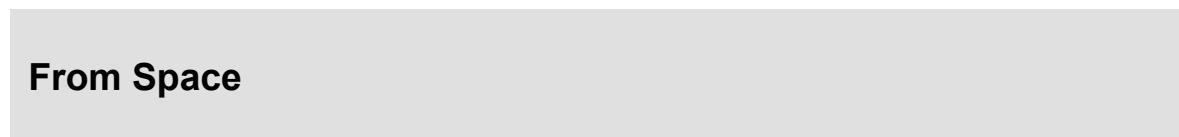
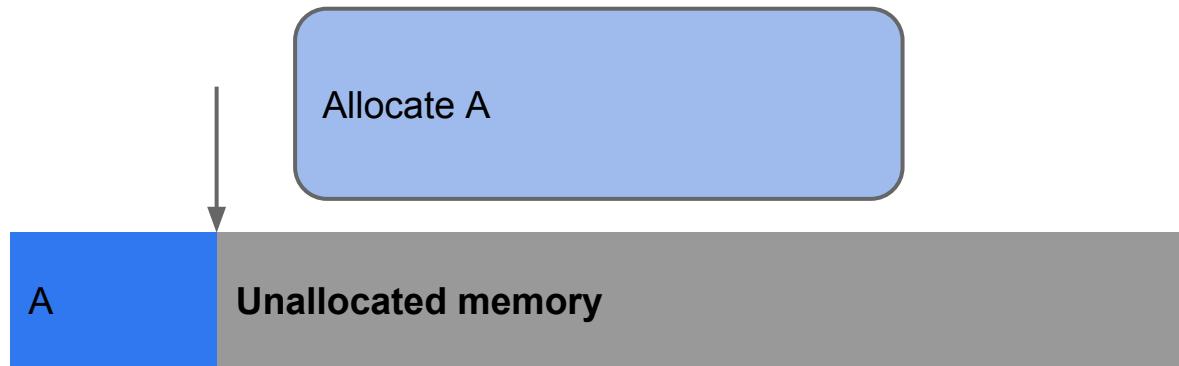
Assume the To Space started off empty and your page starts allocating objects..



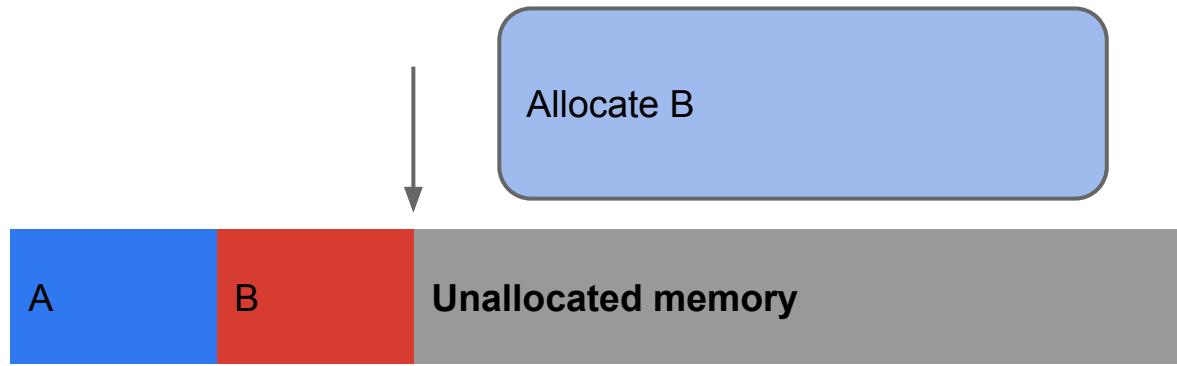
Unallocated memory

From Space

Young Generation In Action

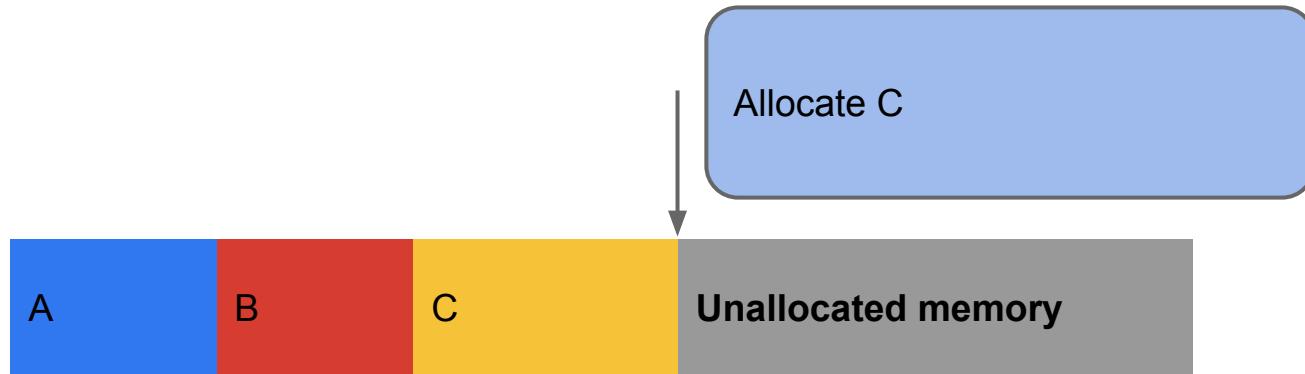


Young Generation In Action



From Space

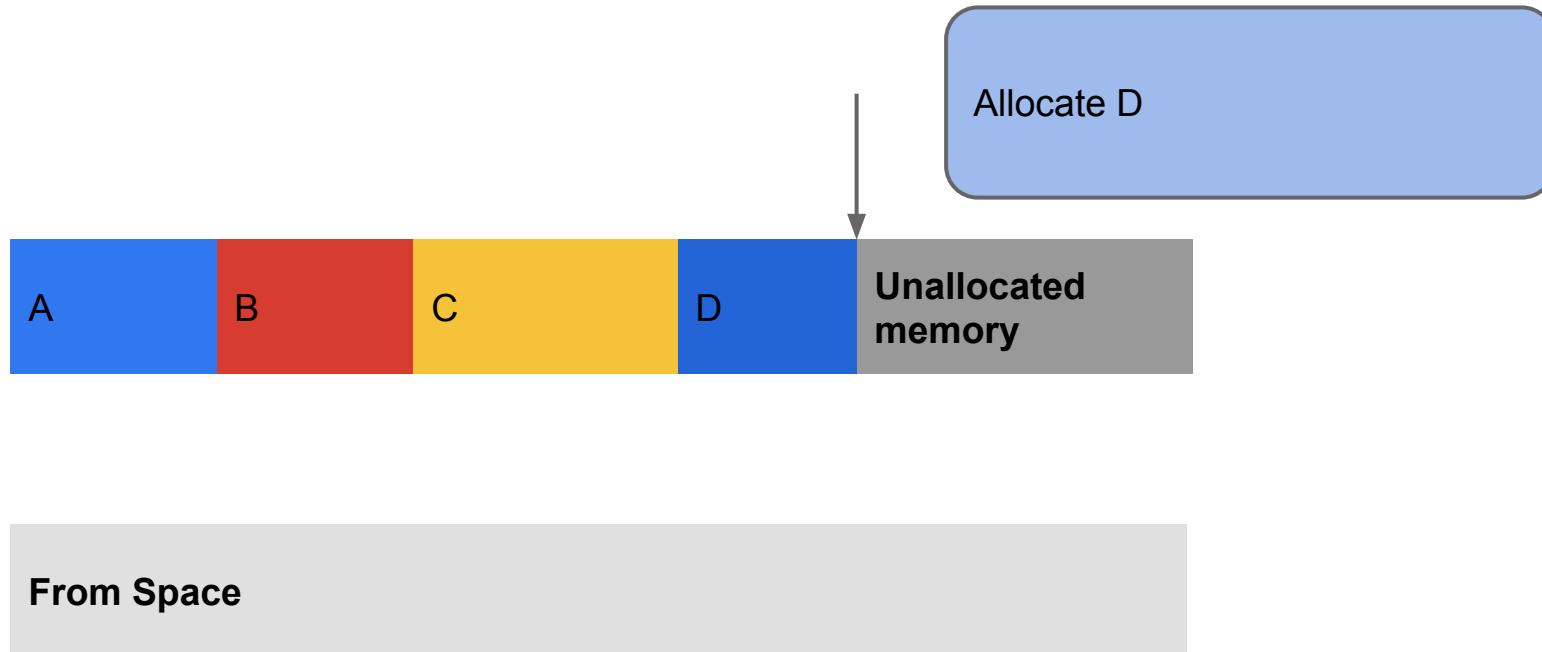
Young Generation In Action



From Space

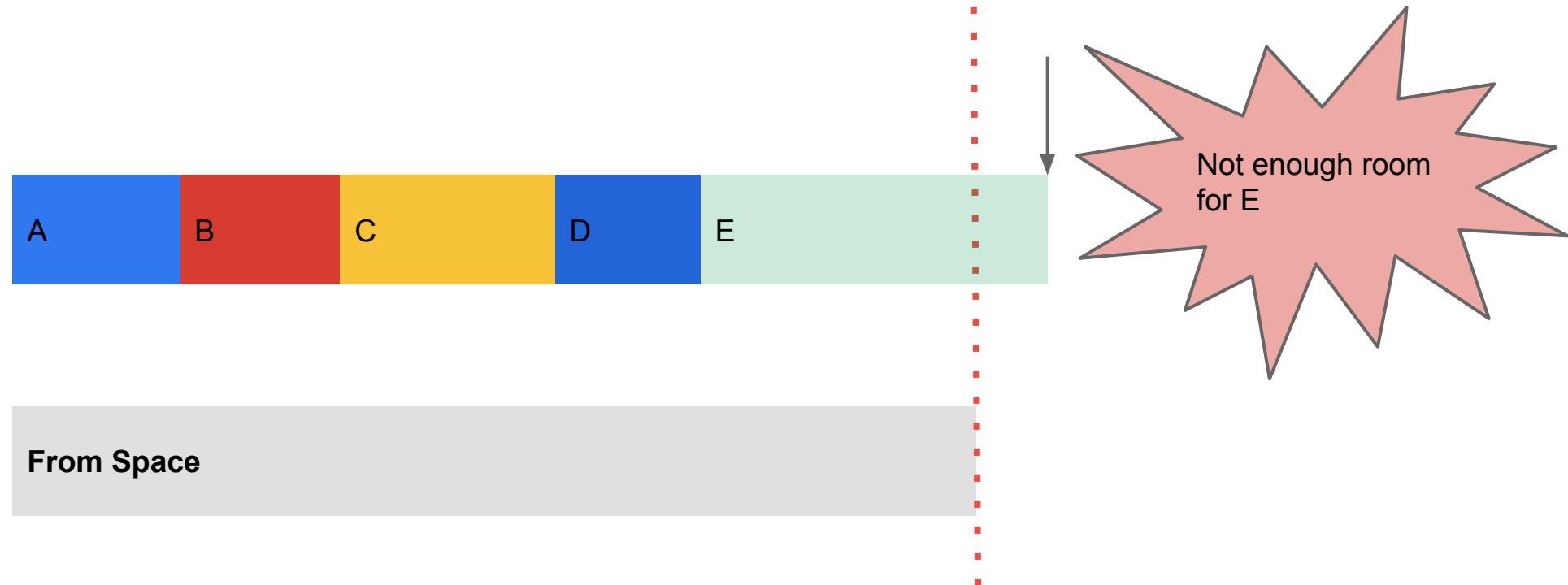
Young Generation In Action

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



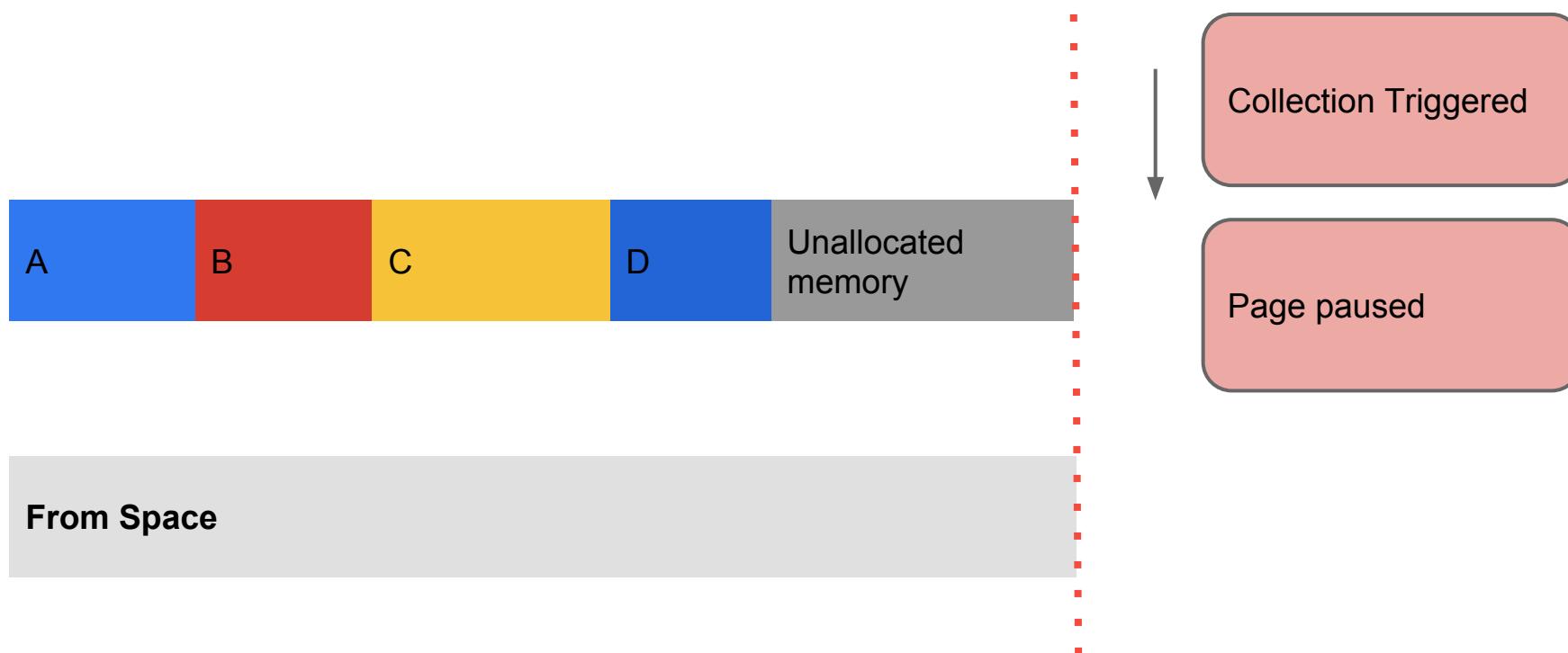
Young Generation In Action

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

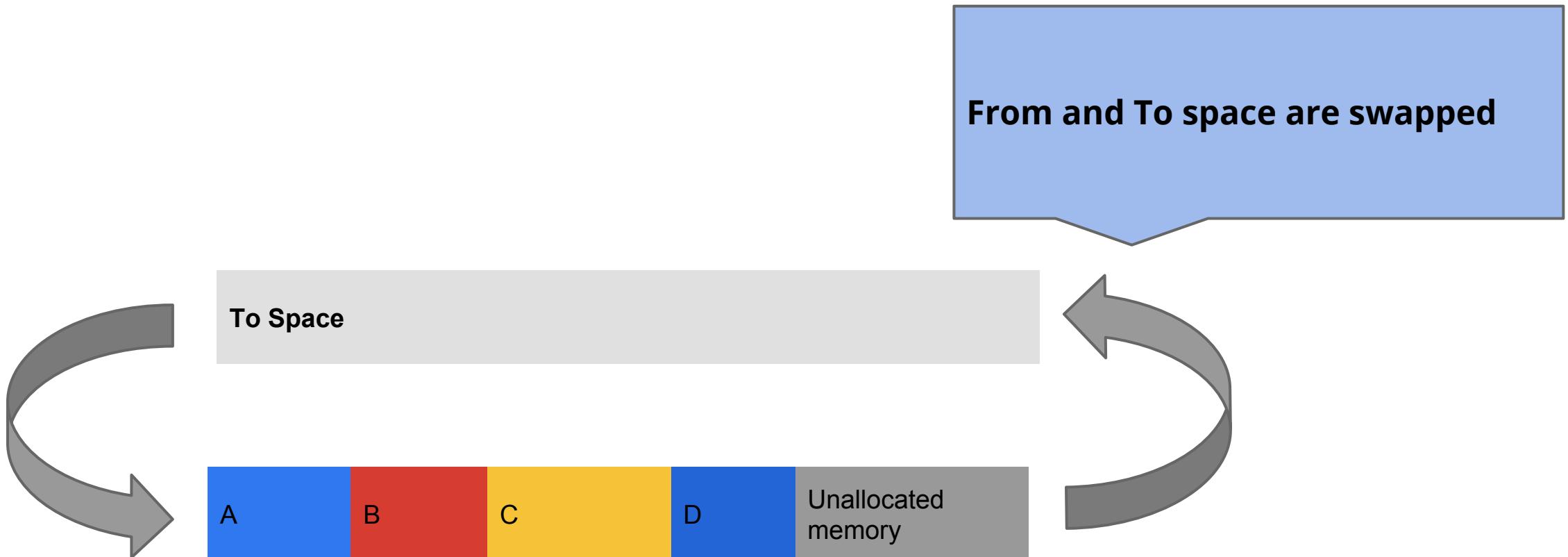


Young Generation In Action

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

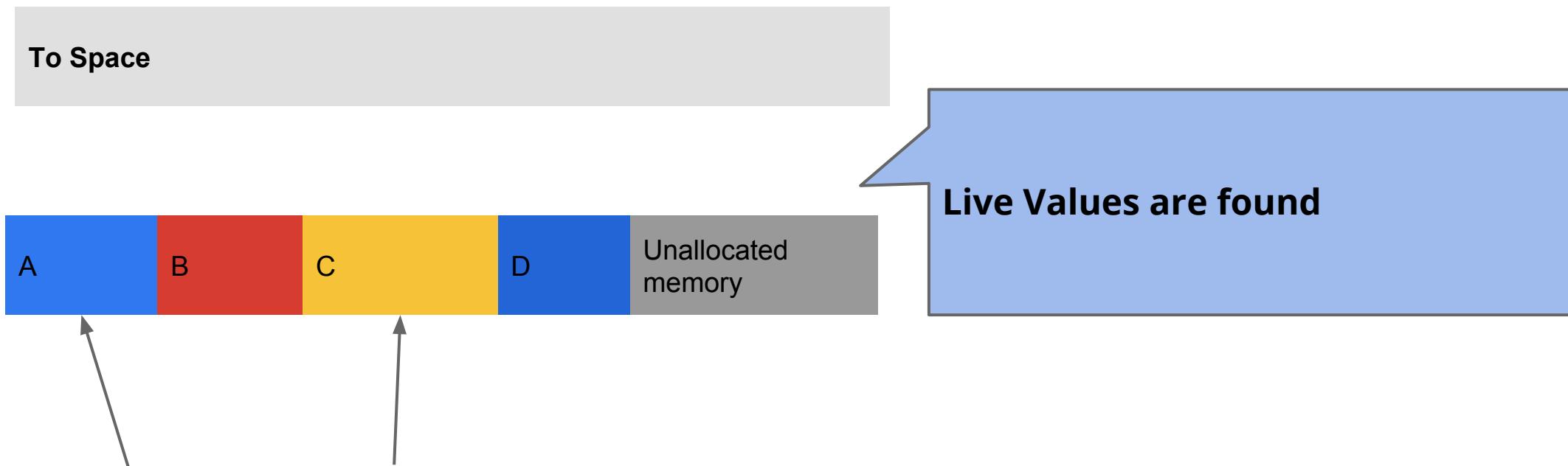


Young Generation In Action



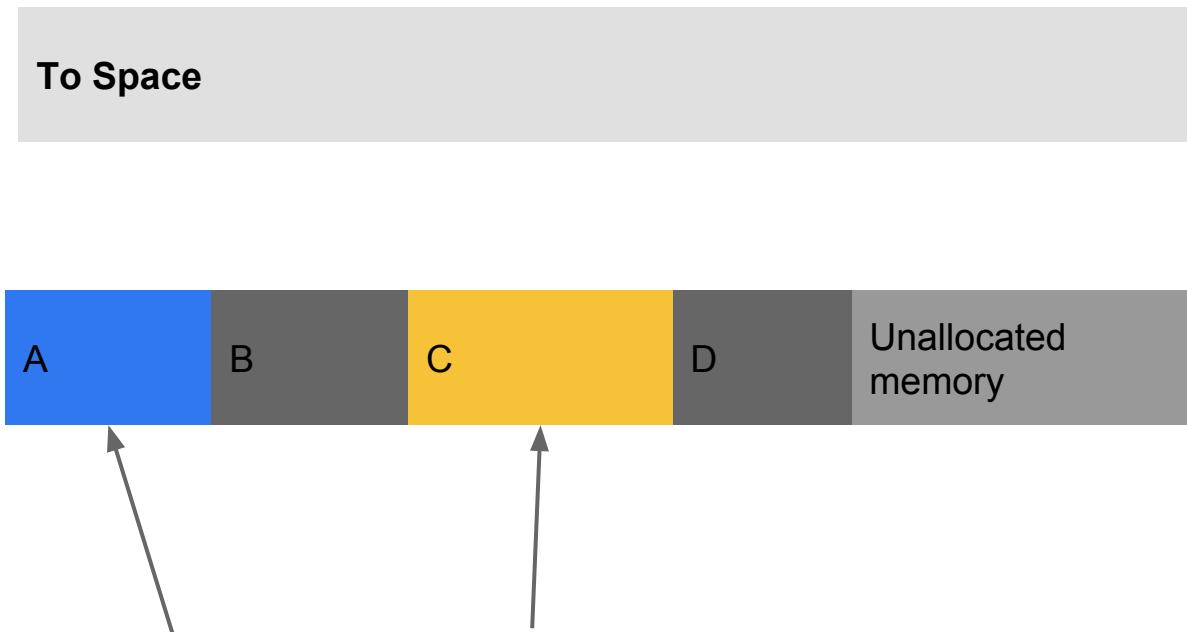
Young Generation In Action

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



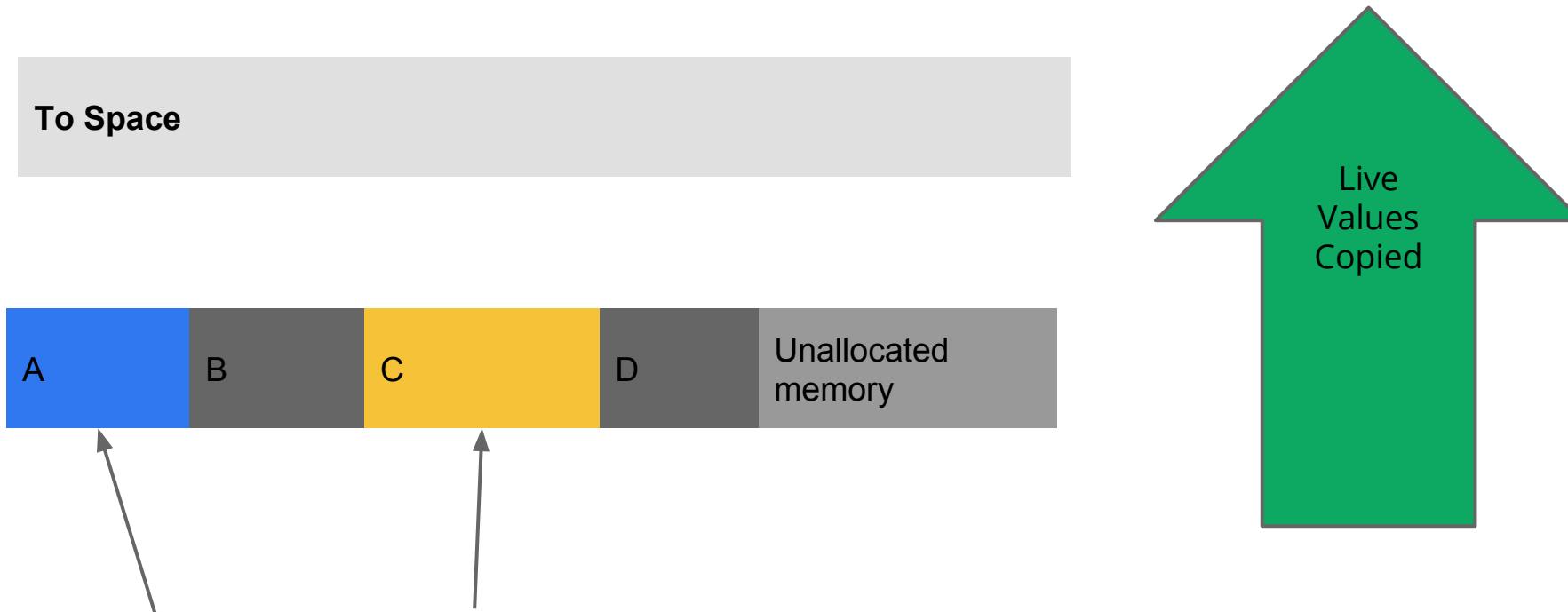
Young Generation In Action

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



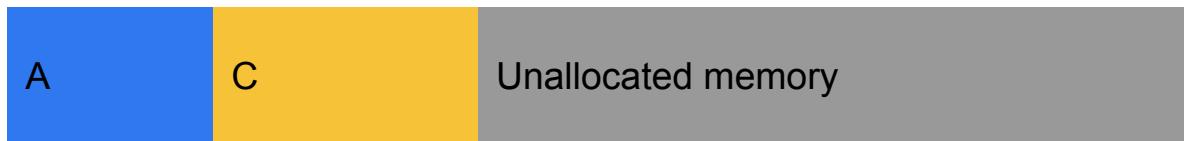
Young Generation In Action

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



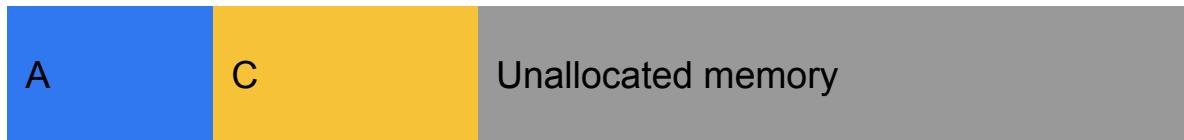
Young Generation In Action

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

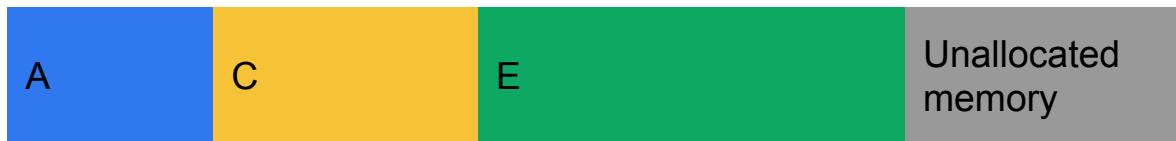
There's no other work done to it. It's just ready for use the next time there's a collection that needs to happen.



Young Generation In Action

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

Allocate E



From Space

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

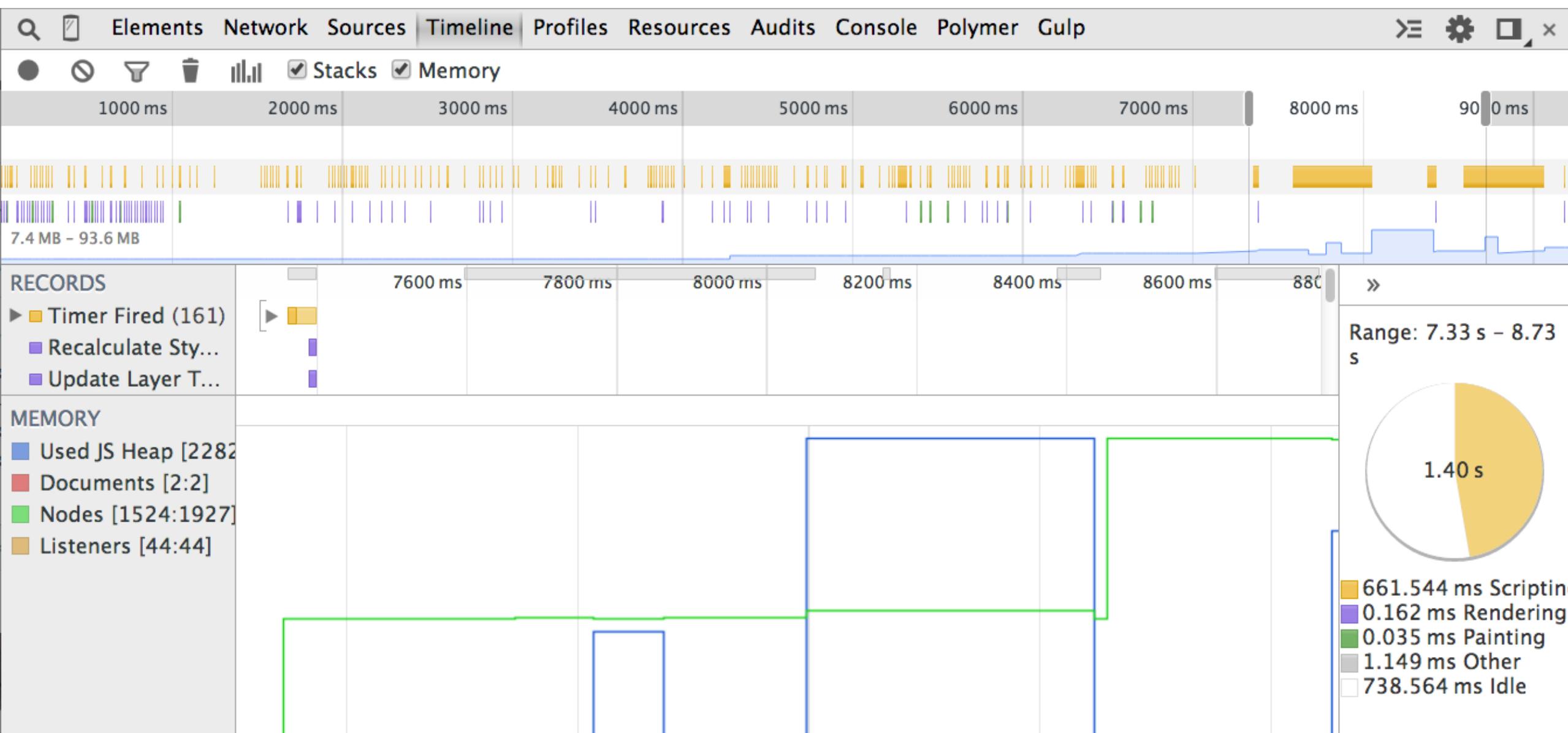
the amount of memory (in bytes) currently
being used

usedJSHeapSize

the amount of memory (in bytes) that the
JavaScript heap has allocated, including free
space

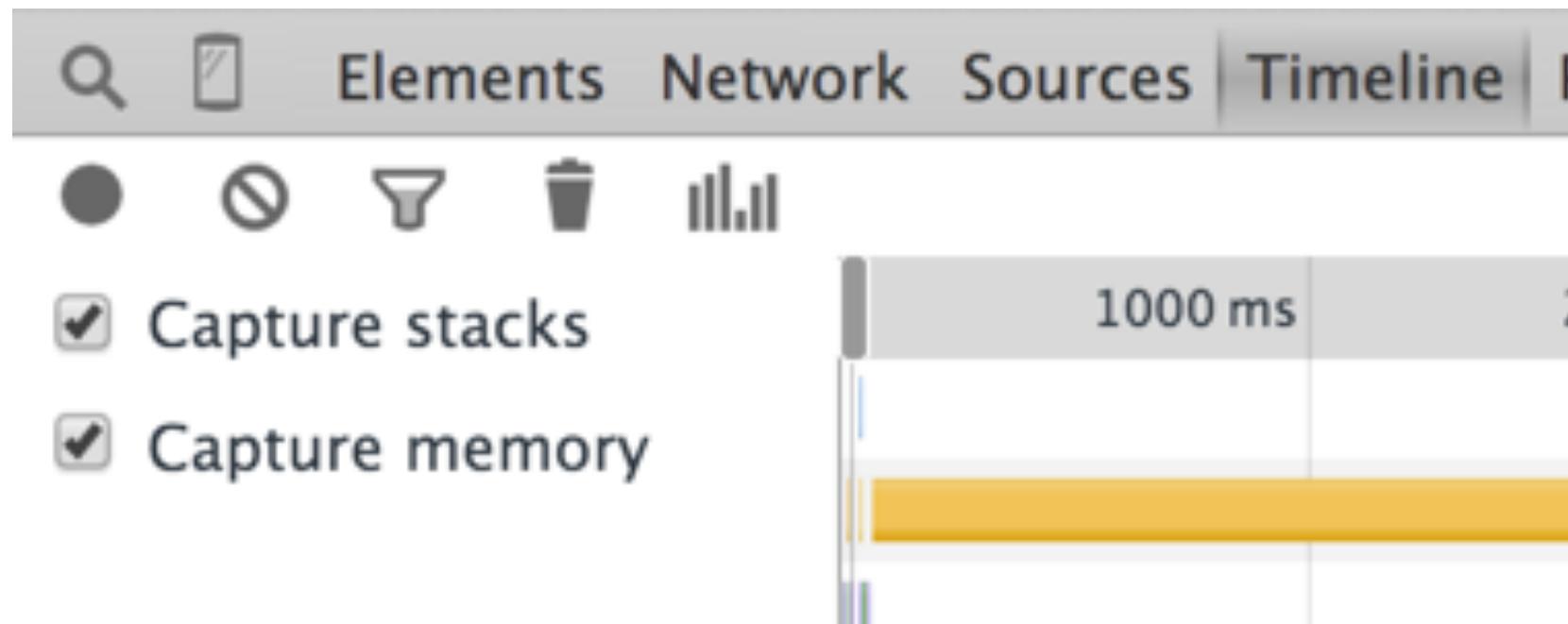
Chrome DevTools

DevTools Memory Timeline



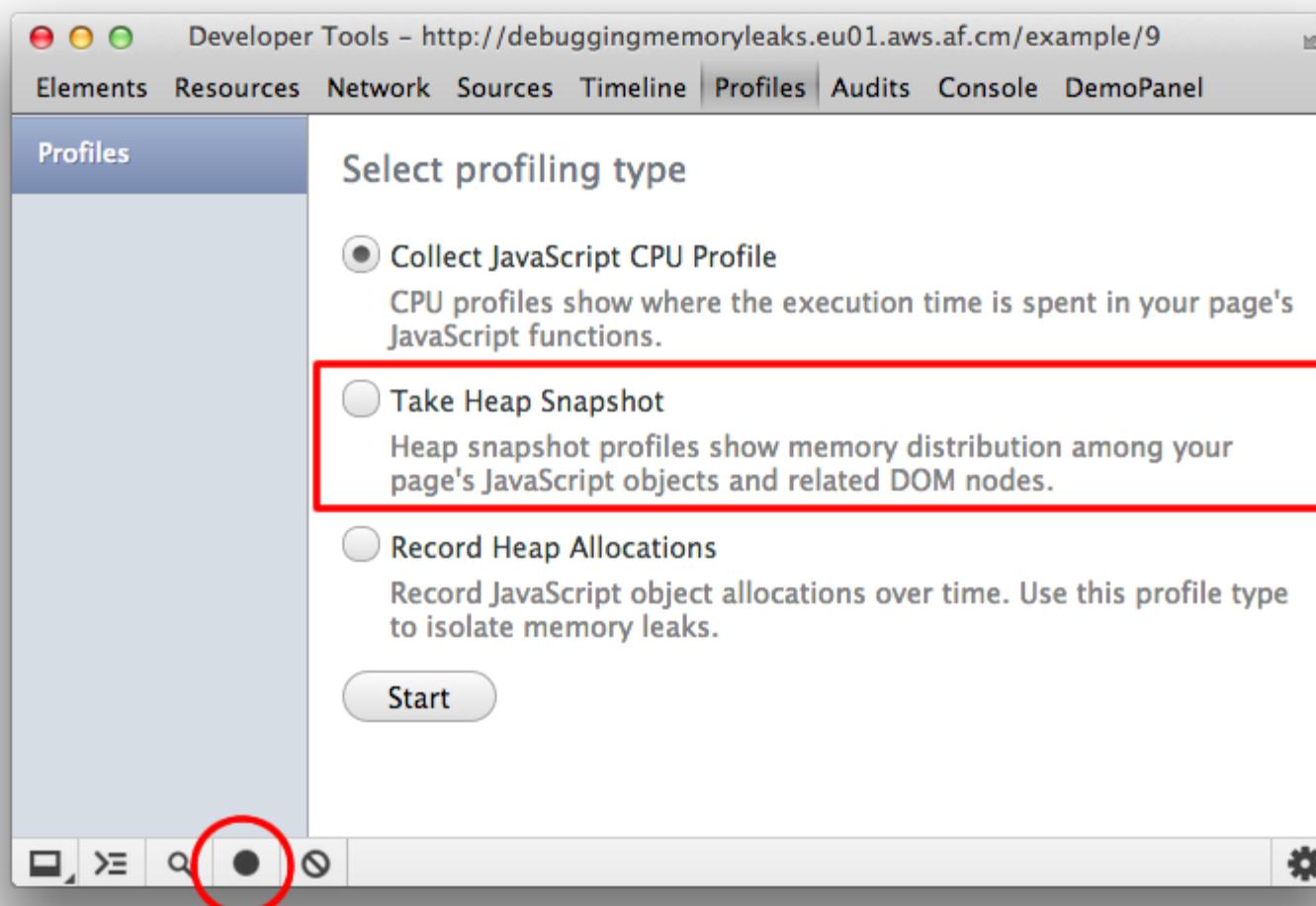
Force GC from DevTools

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



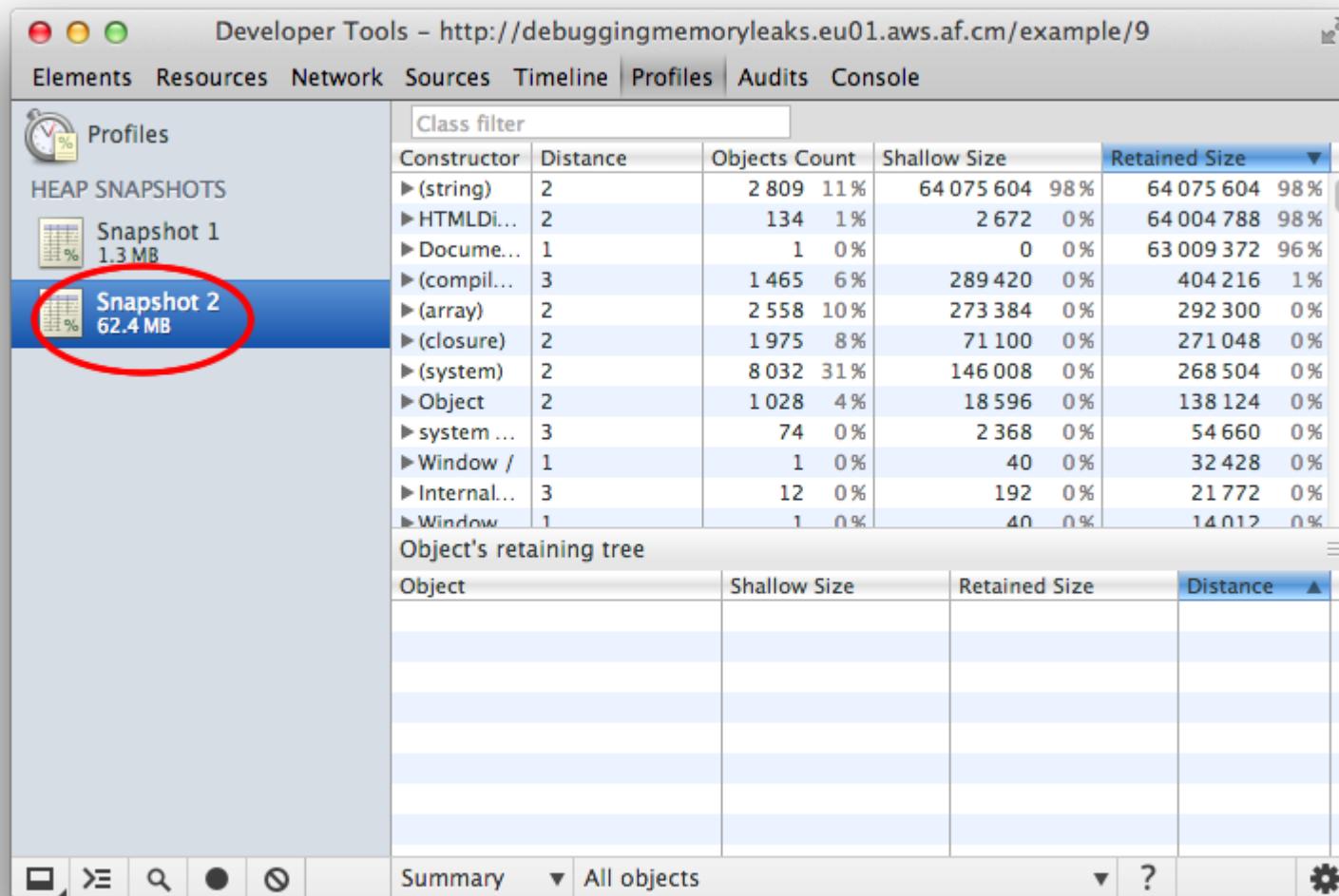
Memory distribution

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

The screenshot shows the Chrome DevTools interface with the 'Performance' tab selected. The top navigation bar includes 'Elements', 'Network', 'Sources', 'Timeline', 'Profiles', and 'Resources'. Below the navigation bar, there are two circular icons: a solid grey circle and an icon with a diagonal slash. A dropdown menu is open over the 'Profiles' button, listing three options: 'Summary' (which is highlighted with a blue background and a checked checkbox icon), 'Comparison', and 'Containment'. To the right of the dropdown, there is a 'Class filter' input field and a table with two columns: 'Distance' and 'Objects Count'. The table contains three rows of data:

Distance	Objects Count
2	112 791 23 %
2	65 435 14 %
1	25 760 5 %

At the bottom left, the text 'HEAP SNAPSHOTS' is visible.

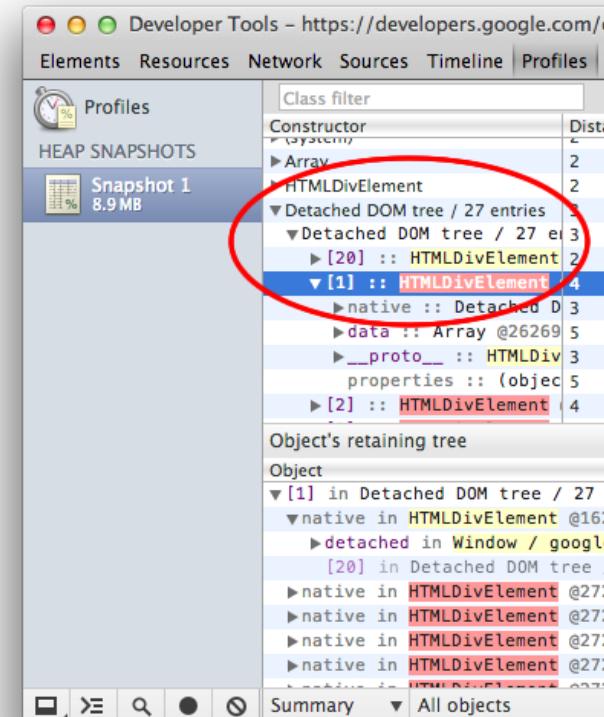
Understanding node colors

yellow

Object has a JavaScript reference on it

red

Detached node. Referenced from one with a yellow background.



Reading results

Summary

The screenshot shows the Chrome DevTools Profiles tab with a heap snapshot analysis. The left sidebar shows 'Profiles' and 'HEAP SNAPSHOTS'. A 'Snapshot 1' is selected, which is 4.6 MB in size.

Class filter: Constructor

	Distance	Objects Count	Shallow Size	Retained Size
▶ (compiled code)	3	5 678 5 %	1 290 600 27 %	1 801 128 38 %
▶ (array)	2	14 307 13 %	1 264 912 26 %	1 541 632 32 %
▶ (closure)	2	8 960 8 %	322 560 7 %	1 384 460 29 %
▶ (system)	2	28 965 26 %	597 784 13 %	1 338 092 28 %
▶ Object	2	4 740 4 %	82 988 2 %	1 117 748 23 %
▶ Window / http://localhost:3000/exam...	1	8 0 %	320 0 %	717 404 15 %
▶ Array	2	1 691 1 %	27 072 1 %	630 380 13 %
▼ Item	2	20 004 18 %	320 060 7 %	560 136 12 %
▶ Item @39957	2		16 0 %	359 880 8 %
▶ Item @39951	2		16 0 %	200 040 4 %
▶ Item @39953	2		16 0 %	112 0 %
▶ Item @65599	3		12 0 %	104 0 %
▶ Item @179537	4		16 0 %	32 0 %
▶ Item @179539	4		16 0 %	32 0 %

Object's retaining tree:

Object	Shallow...	Retaine...	Dis...
▼ stringCache in Window / localhost:3000/example/3 @36393	40 0 %	570 140 12 %	1
▶ global in @36587	276 0 %	30 860 1 %	2

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?

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 %
324	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 %
360	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.*

Objects 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 %

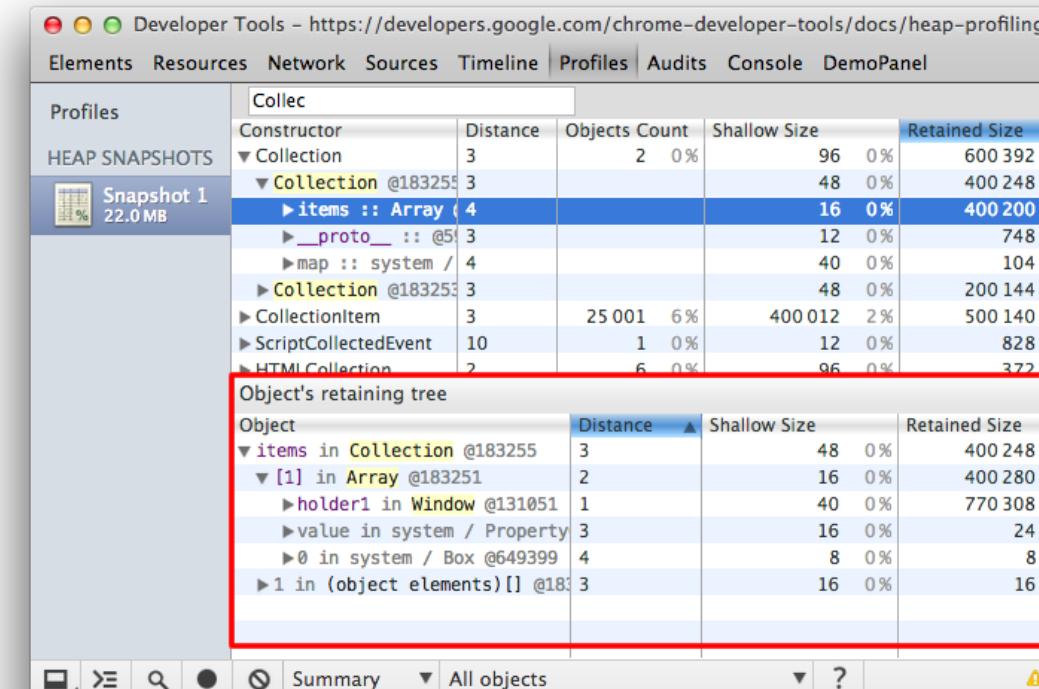
Constructor

All objects created with a specific constructor.

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	1
► Window / http...	1	1
► system / JSArr...	5	1
► JSONSchemaVa...	5	1
► Window / Jav...	1	1

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.

Class filter	Distance	Objects Co...	Shallow Size	
Constructor	2	22 371 14 %	805 356 6 %	
▼(closure)	3		36 0 %	
▶ function LC() @143221	3		36 0 %	
▶ function LC() @143225	3		36 0 %	
LCclosures.js:8	3		36 0 %	
function LC() {	3		36 0 %	
return largeStr;	3		36 0 %	
}	3		36 0 %	

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 SNAPSHTOS							
	Snapshot 1							
	Snapshot 2							
	Snapshot 3	1.4 MB						

Class filter

Constructor	Distance	Objects ...	Shallow Size	Retained 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 %
▶ HTMLDivElement+ @56553	5		20	0 %

Object's retaining tree

Object	Shallow Size	Retained Size
▼ [37] in Array @44265	16	0 %
▶ leakedNodes in Window @9191	40	0 %
▼ [3] in Detached DOM tree / 4 entries @2927992062	0	0 %
▶ native in HTMLDivElement @56535	20	0 %
▶ native in Text @56551	20	0 %
▶ native in HTMLDivElement @56549	20	0 %

Evolved memory profiling

Object Allocation Tracker

Record Heap Allocations



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.

StartLoad

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.



Profiles

HEAP TIMELINES

Snapshot 1
6.7 MB

5.00 s

10.00 s

5.00 s

20.00 s

25.00 s

500 KB

Class filter

Constructor

► (closure)

► system / Context

► (string)

► (compiled code)

Distance

Objects C...

Shallow Size

Retained

3 0 %

108 0 %

3 000 2

3 0 %

84 0 %

3 000 1

3 0 %

3 000 036 43 %

3 000 0

6 0 %

864 0 %

2 9

Object's retaining tree

Object

Shallow Size

Retained Size

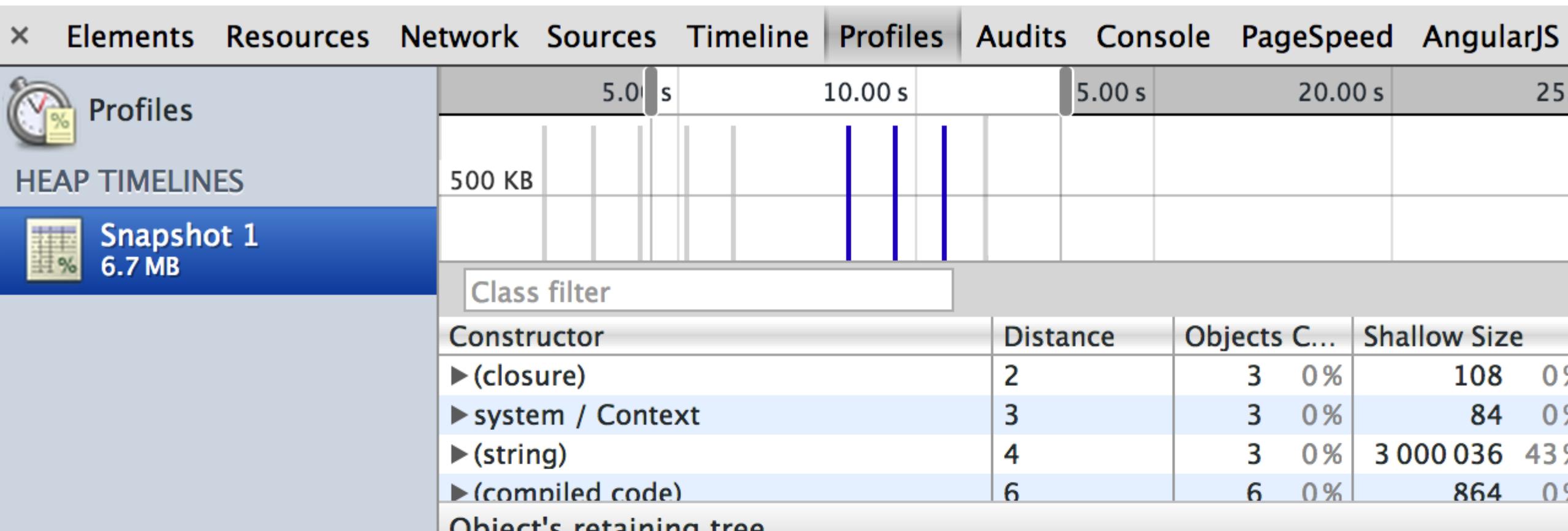
Di

blue bars

memory allocations. Taller = more memory.

grey bars

deallocated



Adjustable timeframe selector

Developer Tools - http://octane-benchmark.googlecode.com/svn/latest/index.html

Elements Network Sources Timeline Profiles Resources Audits Console Gulp

Profiles

HEAP TIMELINES

Snapshot 1 12.4 MB Save

Summary Class filter Selected size: 270 KB

Timeline (5.00 s to 25.00 s) | 100 KB

Constructor	Distance	Objects Count	Shallow Size	Retained Size
►(system)	4	187 0%	4 564 0%	13 840 0%
►(string)	4	50 0%	1 056 0%	1 056 0%
►(number)	4	54 0%	648 0%	648 0%
►Object	5	2 0%	96 0%	96 0%
►Array	6	4 0%	64 0%	76 0%
▼BenchmarkResult	4	1 0%	56 0%	68 0%
▼BenchmarkResult @4005	4		56 0%	68 0%
▼benchmark :: Ben	4		48 0%	120 0%
►__proto__ :: Ben	3		12 0%	160 0%
►Setup :: functio	5		36 0%	36 0%
►TearDown :: func	5		36 0%	36 0%

Retainers

Object	Distance	Shallow Size	Retained Size
▼ [0] in Array @47135	3	16 0%	28 0%
▼benchmarks in BenchmarkSuite @47145	2	24 0%	380 0%
►RayTrace in Window / octane-benchma	1	40 0%	2 257 452 17%
▼ [3] in Array @45239	3	16 0%	256 0%
▼suites in function BenchmarkSuite	2	36 0%	1 028 0%
►BenchmarkSuite in Window / octa	1	40 0%	2 257 452 17%
►constructor in BenchmarkSuite @	3	12 0%	36 0%
2 in [] @45185	4	20 0%	20 0%
4 in (map descriptors)[] @45293	5	124 0%	124 0%
►0 in (object properties)[] @28796	3	32 0%	32 0%

Heap
contents

Developer Tools - http://octane-benchmark.googlecode.com/svn/latest/index.html

Elements Network Sources Timeline Profiles Resources Audits Console Gulp

Profiles

HEAP TIMELINES

Snapshot 1 12.4 MB Save

Selected size: 270 KB

Constructor	Distance	Objects Count	Shallow Size	Retained Size
►(system)	4	187 0 %	4 564 0 %	13 840 0 %
►(string)	4	50 0 %	1 056 0 %	1 056 0 %
►(number)	4	54 0 %	648 0 %	648 0 %
►Object	5	2 0 %	96 0 %	96 0 %
►Array	6	4 0 %	64 0 %	76 0 %
►BenchmarkResult	4	1 0 %	56 0 %	68 0 %
►BenchmarkResult @4005	4		56 0 %	68 0 %
►benchmark :: BenchmarkResult @4005	4		48 0 %	120 0 %
►__proto__ :: Ben	3		12 0 %	160 0 %
►Setup :: functio	5		36 0 %	36 0 %
►TearDown :: func	5		36 0 %	36 0 %
►constructor	2		20 0 %	20 0 %
Retainers	Object	Distance	Shallow Size	Retained Size
▼ [0] in Array @47135	3	16 0 %	28 0 %	
►benchmarks in BenchmarkSuite @47145	2	24 0 %	380 0 %	
►RayTrace in Window / octane-benchmark	1	40 0 %	2 257 452 17 %	
▼ [3] in Array @45239	3	16 0 %	256 0 %	
►suites in function BenchmarkSuite	2	36 0 %	1 028 0 %	
►BenchmarkSuite in Window / octane-benchmark	1	40 0 %	2 257 452 17 %	
►constructor in BenchmarkSuite @47145	3	12 0 %	36 0 %	
2 in [] @45185	4	20 0 %	20 0 %	
4 in (map descriptors)[] @45293	5	124 0 %	124 0 %	
►0 in (object properties)[] @28796	3	32 0 %	32 0 %	

Allocation Stack Traces (New)

DevTools Settings > Profiler > Record Heap Allocation Stack Traces

Octane 2.0

Start Octane 2.0

Welcome to Octane 2.0, a JavaScript benchmark for the modern web. For more accurate results, [start the browser anew](#) before running the test.

[What's new in Octane 2.0](#) - [Documentation](#) - [Run Octane v1](#)

The screenshot shows the Chrome DevTools settings interface. At the top, there are tabs for 'General' and 'Profiler'. The 'Profiler' tab is active. On the left, there is a sidebar with 'General', 'Workspace', 'Experiments', and 'Shortcuts' sections. The 'Profiler' section contains three checkboxes: 'Show advanced heap snapshot properties' (checked), 'Record heap allocation stack traces' (checked), and 'High resolution CPU profiling' (unchecked). The background shows the Octane 2.0 benchmark page with its orange header and the 'Start Octane 2.0' button.

General

Profiler

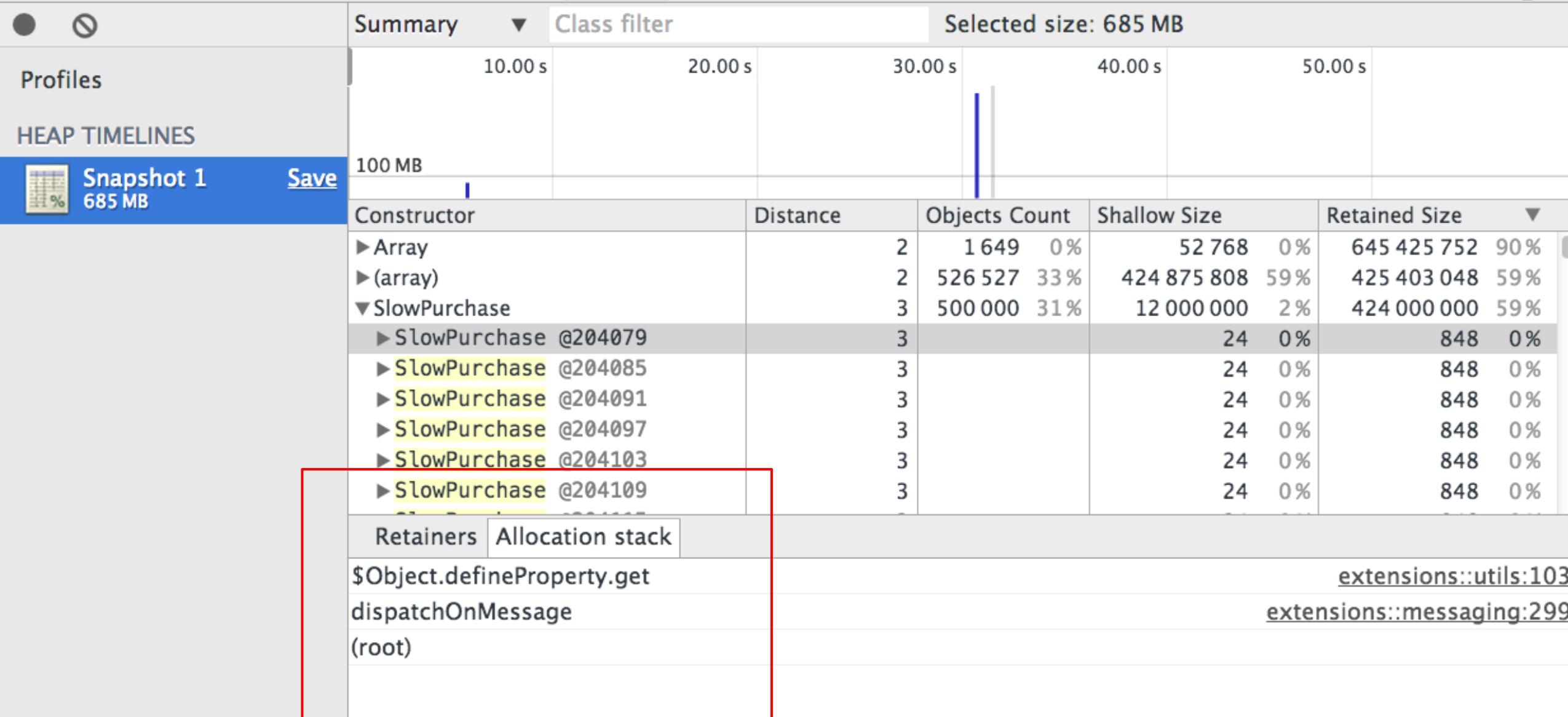
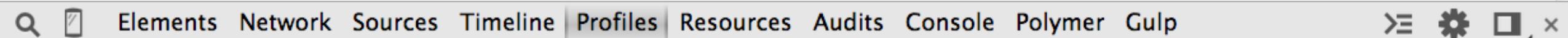
- Show advanced heap snapshot properties
- Record heap allocation stack traces
- High resolution CPU profiling

A small, square portrait of a man wearing glasses and a dark shirt, positioned in the top right corner of the slide.

Paul Lewis

Detached Nodes

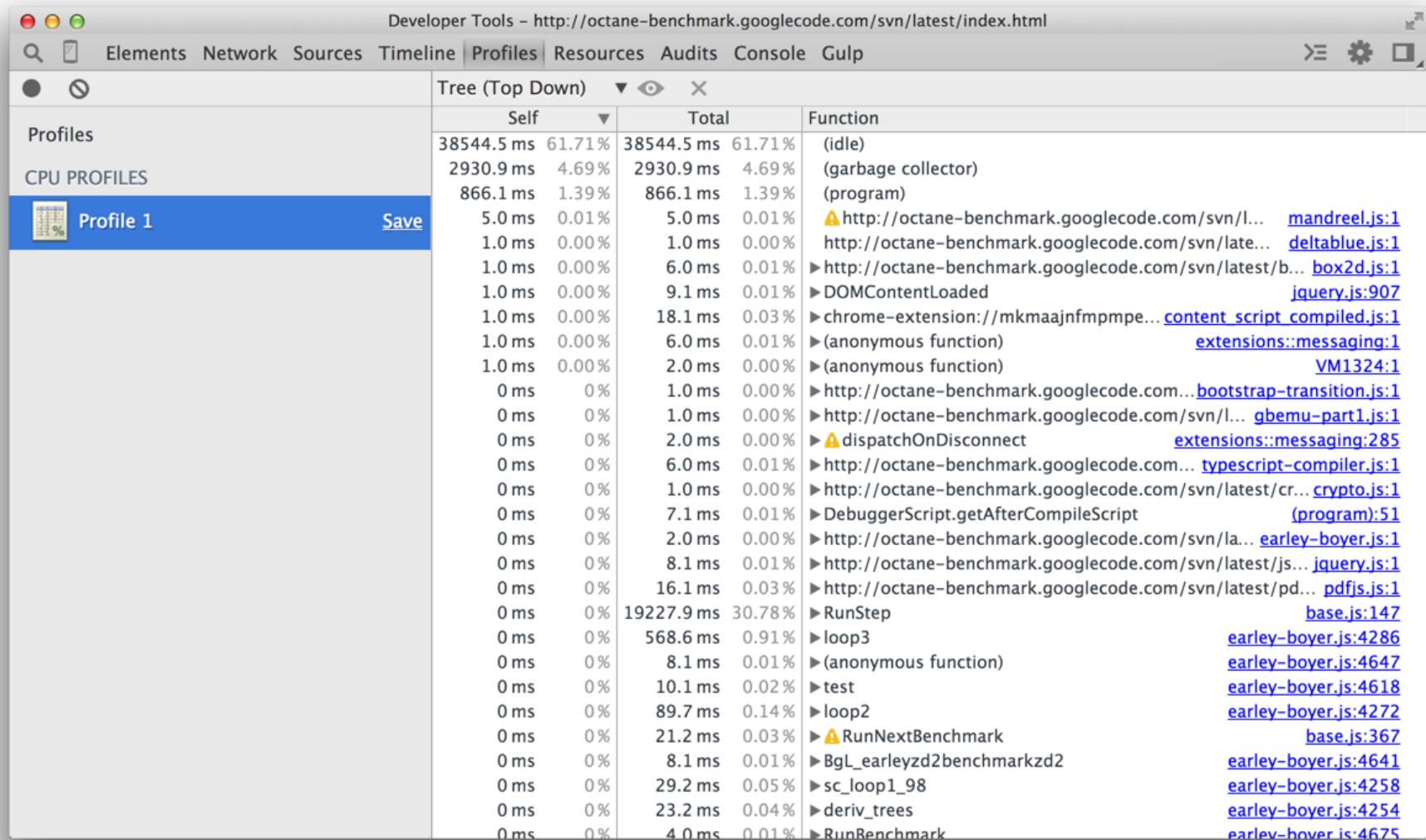
3:15PM



Visualize JS processing over time

JavaScript CPU Profile (top down)

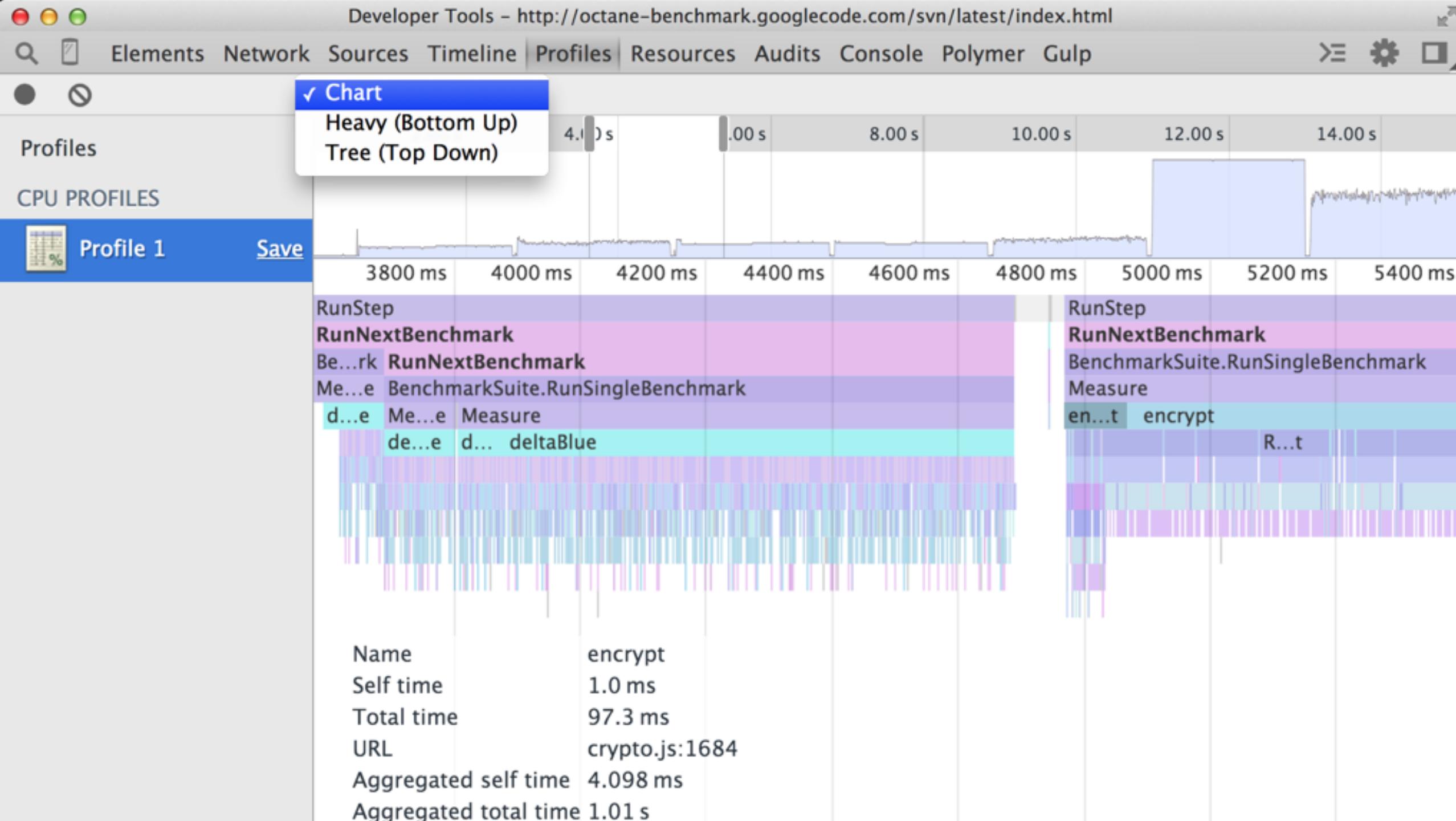
Shows where
CPU time is
statistically
spent on your
code.



Select “Chart” from the drop-down

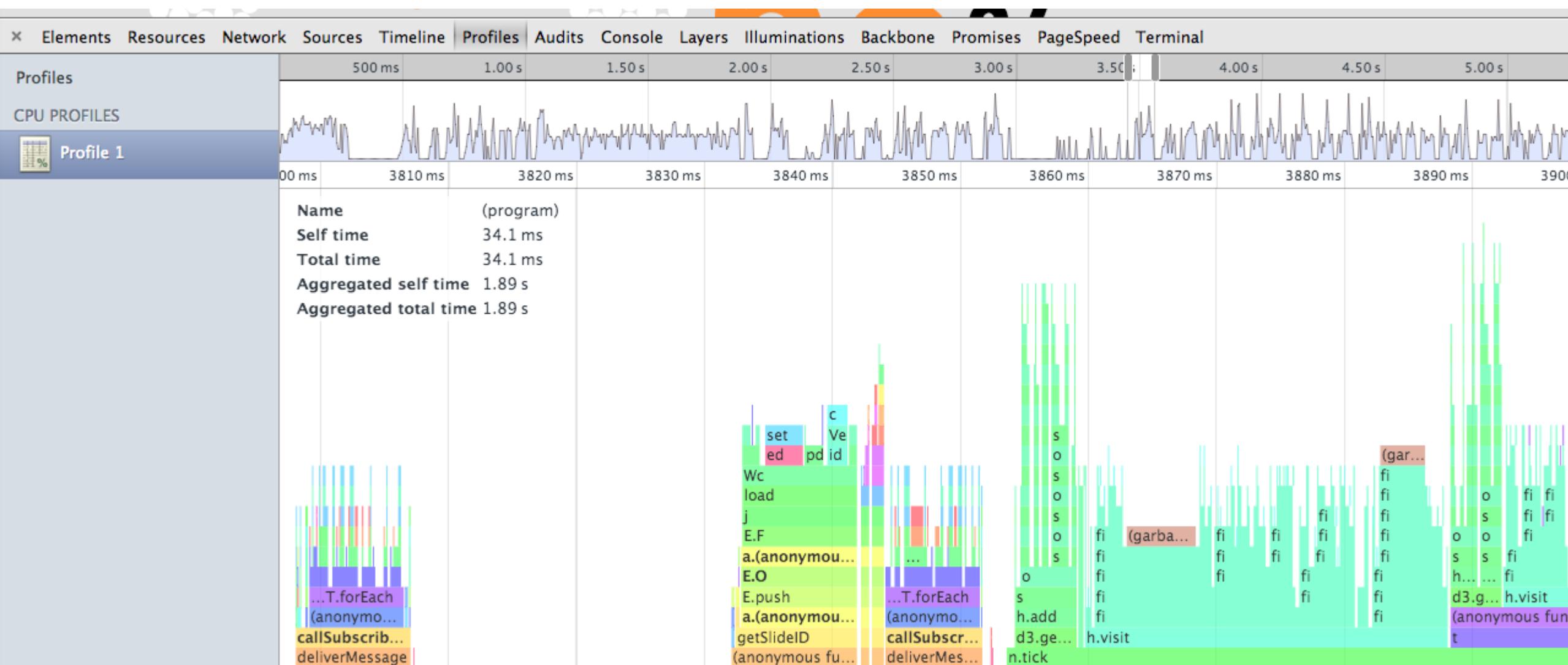
The screenshot shows the Chrome DevTools interface with the CPU Profiles tab selected. A context menu is open at the top, with the 'Chart' option highlighted in blue. Below the menu, there are two options: 'Heavy (Bottom Up)' and 'Tree (Top Down)', with 'Tree (Top Down)' having a checked checkbox next to it. The main table displays CPU profile data with columns for Self, Total, and Function. The first row shows '(idle)' with 2400.7 ms and 51.22%.

	Self	Total	Function
2400.7 ms	51.22 %	2400.7 ms	(idle)
190.8 ms	4.07 %	190.8 ms	(garbage collector)
50.2 ms	1.07 %	50.2 ms	(program)
1.0 ms	0.02 %	1.0 ms	⚠️ dispatchOnDisconnect
0 ms	0 %	1.0 ms	▶ Lazarus.Content.onBlur
0 ms	0 %	1.0 ms	▶ Lazarus.Mouse.onMouseMove
0 ms	0 %	1.0 ms	▶ Lazarus.Content.onFocus
0 ms	0 %	3.0 ms	▶ dispatchOnMessage
0 ms	0 %	1.0 ms	▶ Lazarus.Mouse.onMouseOut
0 ms	0 %	3.0 ms	▶ Lazarus.Content.onClick
0 ms	0 %	2031.2 ms	▶ onclick
0 ms	0 %	3.0 ms	▶ onclick



Flame Chart View

Visualize JavaScript execution paths

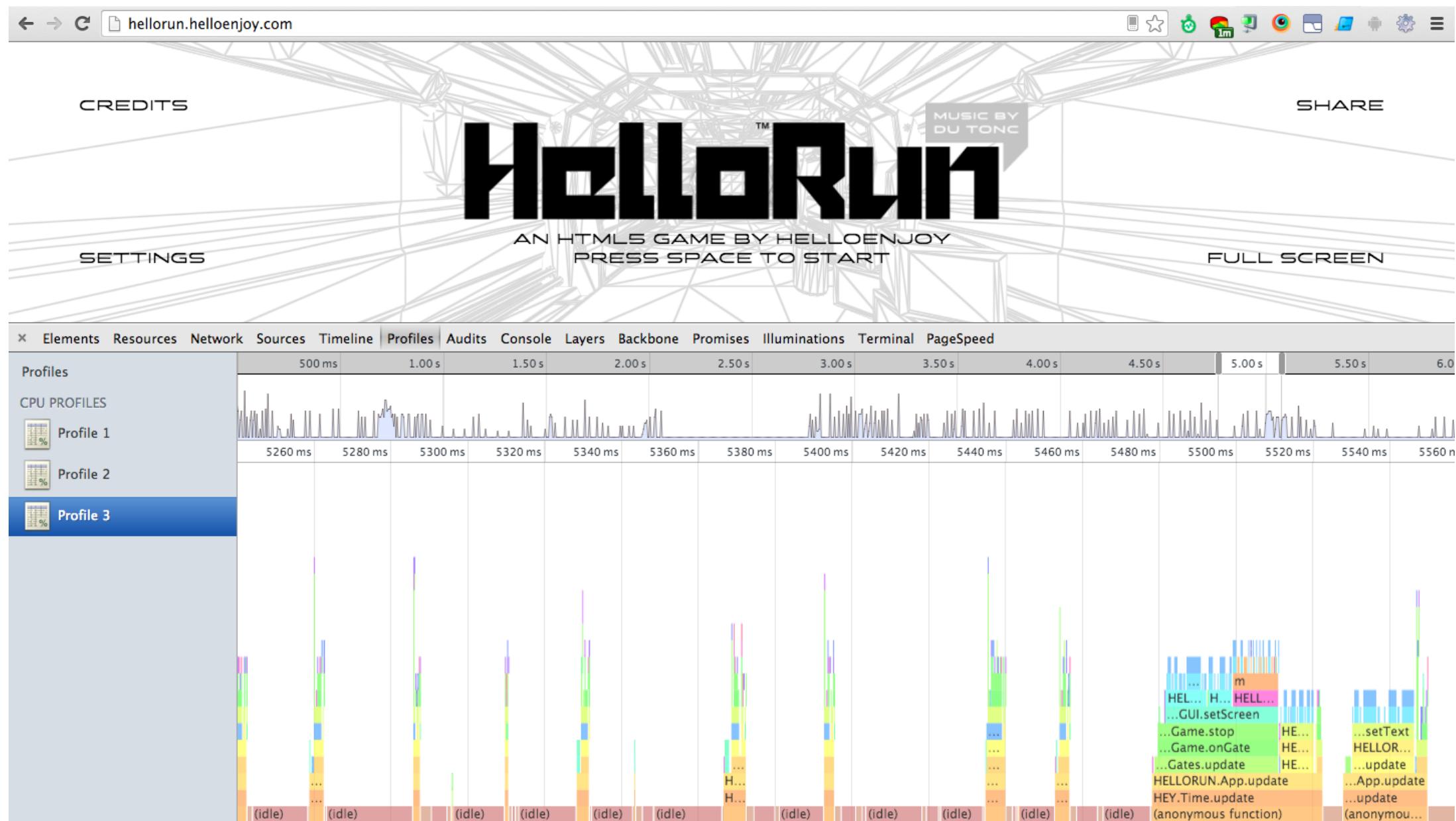


The Flame Chart

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.

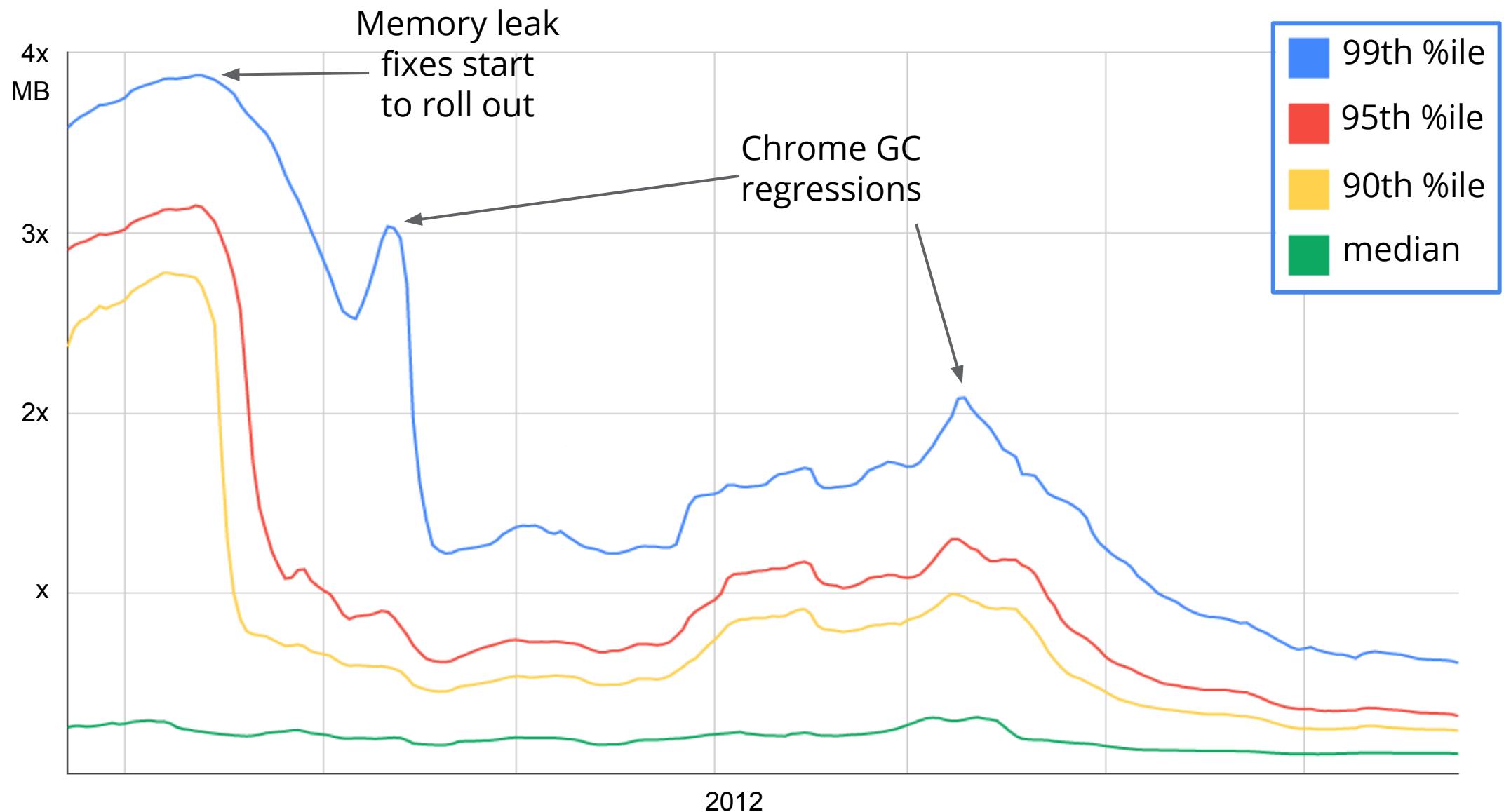
The height of all bars in a particular column is not significant, it simply represents each function call which occurred. What is important however is the width of a bar, as the length is related to the time that function took to execute.

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

[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.

Read the [Memory API](#) to learn more about memory management in JavaScript.

Note: Some of these features we will be using are currently only available in

Chrome Canary. We recommend using this environment 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

Questions to ask yourself

[Terminology and Fundamentals](#) +[Prerequisites and helpful tips](#) +[Heap Profiler](#) +[Views in detail](#) +[Object Location Tracker](#) +

Memory Profiling FAQ

[Supporting Demos](#) +[Community Resources](#) +

Notes and resources related to v8 and thus Node.js performance <https://thlorenz.github.io/v8-perf/>

10 commits

3 branches

0 releases

1 contributor



branch: master ▾

v8-perf / +



note about function closures

thlorenz authored 21 days ago

latest commit e5284ce271 ↗

test

adding boxing tests

2 months ago

.gitignore

dox after working through most referenced materials

2 months ago

.jshintrc

dox after working through most referenced materials

2 months ago

README.md

dox after working through most referenced materials

2 months ago

compiler.md

dox after working through most referenced materials

2 months ago

data-types.m

dox after working through most referenced materials

2 months ago

gc.md

dox after working through most referenced materials

2 months ago

memory-profiling.md

note about function closures

21 days ago

package.json

dox after working through most referenced materials

2 months ago

performance-profiling.m

dox after working through most referenced materials

2 months ago

runtime-functions.m

dox after working through most referenced materials

2 months ago

Code

Issues 6

Pull Requests 0

Pulse

Graphs



dependencies up to date

devDependencies up to date

HTTPS clone URL

<https://github.com/1> ↗

You can clone with HTTPS, SSH, or Subversion. ?

Clone in Desktop

Download ZIP

V8 Performance & Node.js

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



CODING

CSS

HTML

JavaScript

Techniques

DESIGN

Web Design

UI/UX

Inspiration

Business

MOBILE

iPhone & iPad

Android

Design Patterns

PERFORMANCE

Writing Fast, Memory-Efficient JavaScript

By Addy Osmani

November 5th, 2012

JavaScript, Optimization, Performance

Writing Memory-efficient JavaScript

As you develop, if you care about memory usage and performance, you should be aware of some as Google's V8 (Chrome, Node) are specifically designed for the fast execution of large JavaScript applications.



Not just websites

- ✓ 24/7 UK support
- ✓ Multi-site hosting

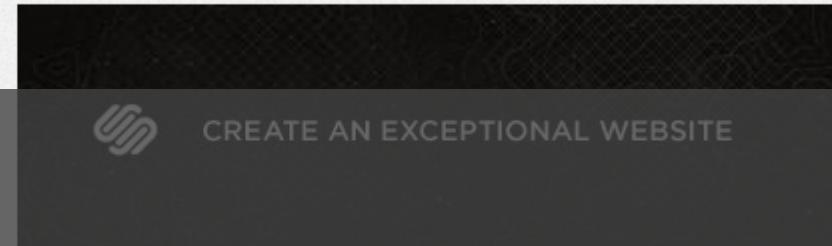
From only

£2.49/mo

heart internet

Advertisement

Advertise with us!



CREATE AN EXCEPTIONAL WEBSITE

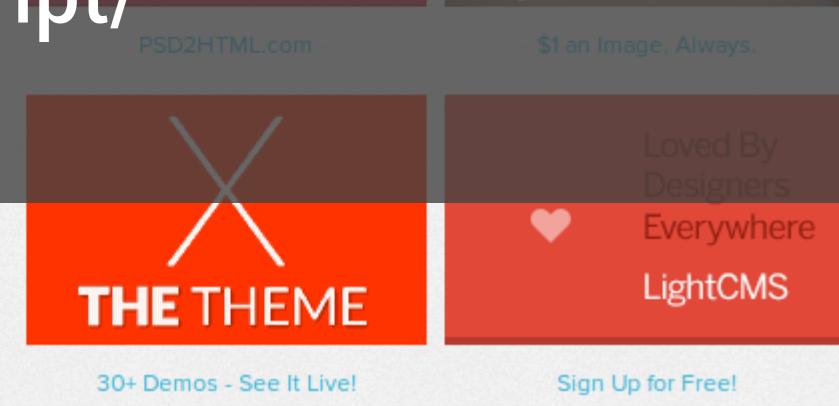


Better Websites For All.

Responsive HTML5 CSS3 + WordPress Development

PSD2HTML®

PSD2HTML.com



THE THEME

30+ Demos - See It Live!

LightCMS

Sign Up for Free!

With patch applied

Profiles

HEAP SNAPSHTOS

Snapshot 1 11.5 MB
Before quick edit

Snapshot 2 10.1 MB

Snapshot 3 11.4 MB

Snapshot 4 10.1 MB
After closing quick edit

Detached

Constructor	# New	# Deleted	# Delta	Alloc. Size ▾	Freed Size	Size Delta
▼ Detached DOM tree / 13 entries	1	0	+1	0	0	0
▼ Detached DOM tree / 13 entries @3519317322	.	.	.	0	0	0
▶ [1] :: Text @761201
▶ [2] :: NodeList @786121
▶ [3] :: Text @761203
▶ [4] :: HTMLDivElement @786125
▶ [5] :: NodeList @786123
▶ [6] :: HTMLDivElement @762117
▶ [7] :: HTMLElement @773727
▶ [8] :: HTMLDivElement @785535
▶ [9] :: HTMLDivElement @773731
▶ [10] :: HTMLButtonElement @761213
▶ [11] :: HTMLDivElement @761219
▶ [12] :: NodeList @785519
▶ [13] :: NodeList @785511
▶ [14] :: NodeList @785511
▶ [15] :: NodeList @785509
▶ [16] :: HTMLButtonElement @785574
▶ [17] :: HTMLDivElement @785574
▶ [18] :: NodeList @785711
▶ [19] :: HTMLDivElement @786191
▶ [20] :: NodeList @773725
▶ [21] :: HTMLDivElement @786185
▶ [22] :: NodeList @786189

Detached DOM tree

The toolbar is still in the DOM, but the toolbar fence is gone.

Fixing JS Memory leaks in Drupal's editor

<https://www.drupal.org/node/2159965>

DOM

JS

<button/>

Avoiding JS memory leaks in Imgur

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

Node.js Performance Tip of the Week: Memory Leak Diagnosis

02 May 2014 / 0 Comments / in How-To, Performance Tip, StrongOps / by Shubhra Kar

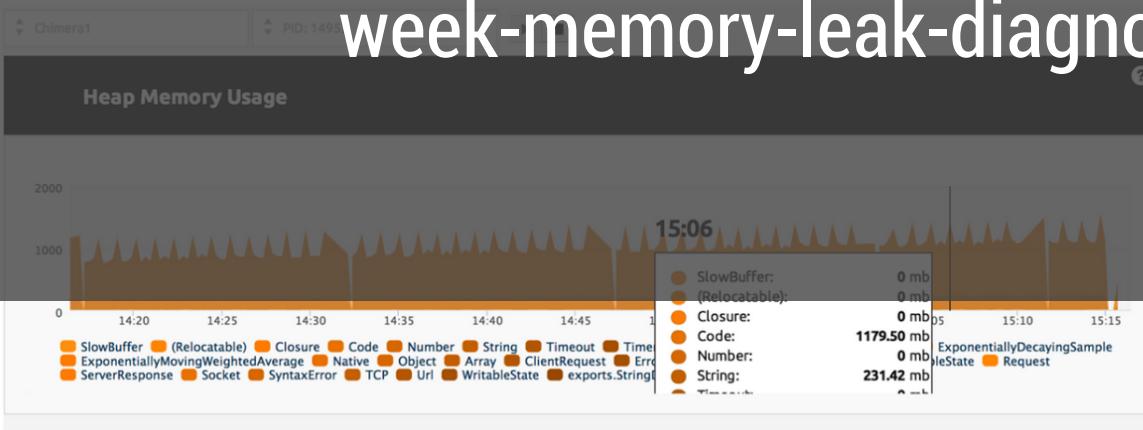


In last week's performance tip, we discussed in detail how to leverage Google V8's heap profiler to diagnose problems with Node applications. In this go around we look at different leak patterns and how a good diagnosis can lead us to find the root cause of a potentially troublesome production problem.

Identifying patterns

StrongLoop: Memory profiling with DevTools

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



 **StrongLoop Newsletter Archives**

Most Viewed

TJ Holowaychuk Passes Spo...

posted on 07/29/2014

Creating Desktop Applicat...

posted on 11/26/2013

What's New in LoopBack 2....

posted on 07/22/2014

Recent Posts

 Upcoming Breaking C++ API Changes in Node.js v0.12

August 29, 2014 - 7:47 am

 Express Documentation Roadmap

August 21, 2014 - 9:39 am

 Express 3.x to 4.x Migration Guide

August 20, 2014 - 12:20 pm

Categories

API Tip (1)

BACN (3)

Case Studies (3)

Cloud (9)

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

The screenshot shows the Chrome DevTools interface with the "Profiles" tab selected. In the "HEAP SNAPSHTOS" section, a snapshot labeled "Snapshot 1" (893 MB) is selected. The main area displays a table of memory usage data. The table has columns for Constructor, Distance, Objects Count, Shallow Size, and Retained Size. The "Retained Size" column is sorted in descending order, with the top item being an "HTMLDivElement" object at index [3] with a size of 40 MB. The "Object" column shows the memory address of each object, and the "Retainers" column lists the objects that are holding a reference to the selected item.

Constructor	Distance	Objects Count	Shallow Size	Retained Size
► Plugin	-	1	0 %	40 0 %
► TextMetrics	-	1	0 %	40 0 %
► XMLHttpRequestProgressEvent	-	1	0 %	40 0 %
▼ Detached DOM tree / 4 entries	4	20	0 %	0 0 %
▼ Detached DOM tree / 4 entries	4		0 0 %	0 0 %
► [1] :: Text @189931	5		40 0 %	40 0 %
► [2] :: HTMLDivElement @1898	5		40 0 %	40 0 %
► [3] :: HTMLDivElement @1898	3		40 0 %	40 0 %
► Detached DOM tree / 4 entries	4		0 0 %	0 0 %
► Detached DOM tree / 4 entries	4		0 0 %	0 0 %
► Detached DOM tree / 4 entries	4		0 0 %	0 0 %
► Detached DOM tree / 4 entries	4		0 0 %	0 0 %
► Detached DOM tree / 4 entries	4		0 0 %	0 0 %
► Detached DOM tree / 4 entries	4		0 0 %	0 0 %
► Detached DOM tree / 4 entries	4		0 0 %	0 0 %
► Detached DOM tree / 4 entries	4		0 0 %	0 0 %
► Detached DOM tree / 4 entries	4		0 0 %	0 0 %
► Detached DOM tree / 4 entries	4		0 0 %	0 0 %
▼ Retainers				
Object	Distance	▲	Shallow Size	Retained Size
▼ [0] in Array @84817	2		32 0 %	392 0 %
► leakedNodes in Window / localhost:3000/ @3951	1		80 0 %	35 576 0 %
► value in system / PropertyCell @177151	3		32 0 %	32 0 %
► 0 in (object elements)[] @190039	3		360 0 %	360 0 %
► [3] in Detached DOM tree / 4 entries @81546312	4		0 0 %	0 0 %





A man with dark hair and glasses, wearing a grey t-shirt with a logo, has a shocked or surprised expression with his mouth open. He is standing in what appears to be a backstage area with wooden walls and equipment in the background.

EASTWOOD
AND
CHAIR

OH MY GOD, I LOVED IT ALL!

Thank you!



+AddyOsmani
@addyosmani

#perfatters