Show navigation

Trash talk: the Orinoco garbage collector

Published 03 January 2019 · Tagged with internals memory presentations

Over the past years the V8 garbage collector (GC) has changed a lot. The Orinoco project has taken a sequential, stop-the-world garbage collector and transformed it into a mostly parallel and concurrent collector with incremental fallback.

Note: If you prefer watching a presentation over reading articles, then enjoy the video below! If not, skip the video and read on.

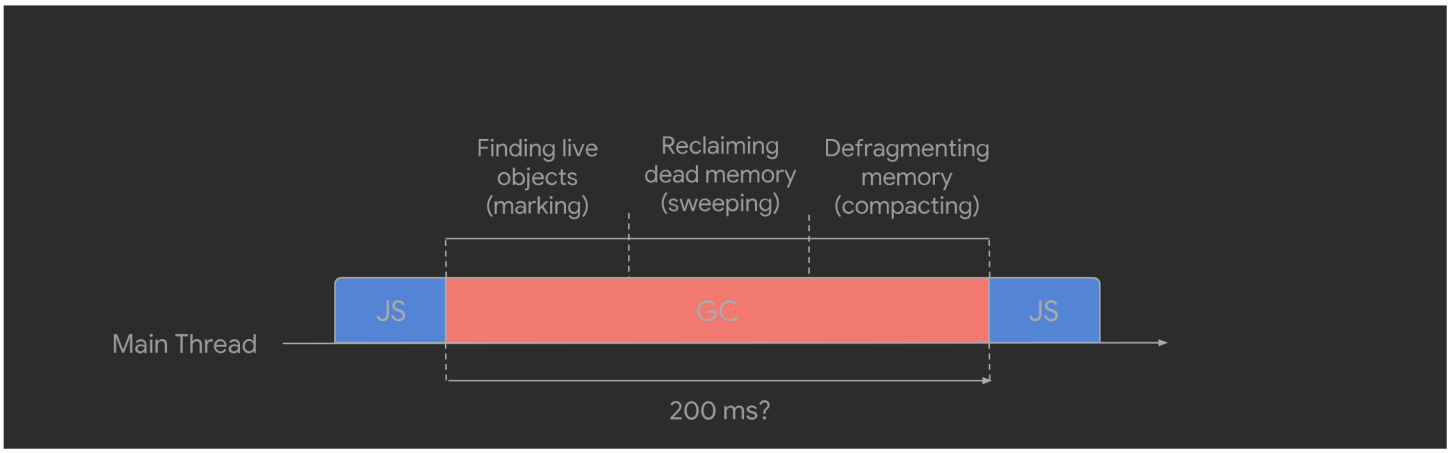
Any garbage collector has a few essential tasks that it has to do periodically:

1. Identify live/dead objects
2. Recycle/reuse the memory occupied by dead objects
3. Compact/defragment memory (optional)

These tasks can be performed in sequence or can be arbitrarily interleaved. A straight-forward approach is to pause JavaScript execution and perform each of these tasks in sequence on the main thread. This can cause jank and latency issues on the main thread, which we've talked about in previous blog posts, as well as reduced program throughput.

Major GC (Full Mark-Compact)

The major GC collects garbage from the entire heap.



Major GC happens in three phases: marking, sweeping and compacting.

Marking

Figuring out which objects can be collected is an essential part of garbage collection. Garbage collectors do this by using reachability as a proxy for 'liveness'. This means that any object currently reachable within the runtime must be kept, and any unreachable objects may be collected.

Marking is the process by which reachable objects are found. The GC starts at a set of known objects pointers, called the root set. This includes the execution stack and the global object. It then follows each pointer to a JavaScript object, and marks that object as reachable. The GC follows every pointer in that object, and continues this process recursively, until every object that is reachable in the runtime has been found and marked.

Sweeping

Sweeping is a process where gaps in memory left by dead objects are added to a data structure called a free-list. Once marking has completed, the GC finds contiguous gaps left by unreachable objects and adds them to the appropriate free-list. Free-lists are separated by the size of the memory chunk for quick lookup. In the future when we want to allocate memory, we just look at the free-list and find an appropriately sized chunk of memory.

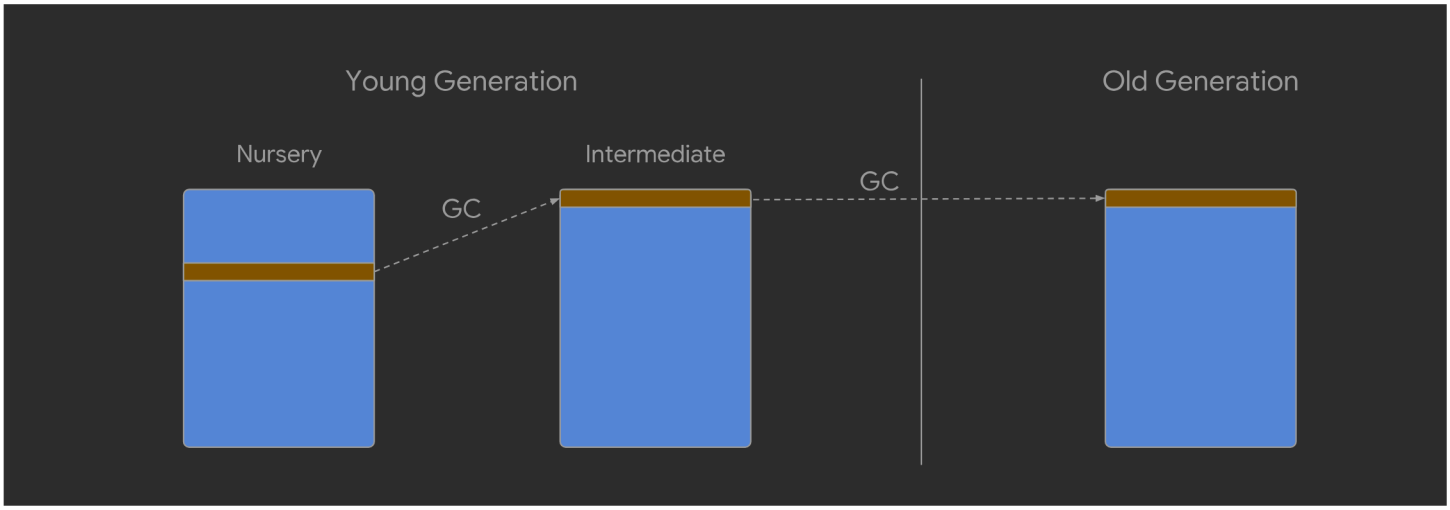
Compaction

The major GC also chooses to evacuate/compact some pages, based on a fragmentation heuristic. You can think of compaction sort of like hard-disk defragmentation on an old PC. We copy surviving objects into other pages that are not currently being compacted (using the free-list for that page). This way, we can make use of the small and scattered gaps within the memory left behind by dead objects.

One potential weakness of a garbage collector which copies surviving objects is that when we allocate a lot of long-living objects, we pay a high cost to copy these objects. This is why we choose to compact only some highly fragmented pages, and just perform sweeping on others, which does not copy surviving objects.

Generational layout

The heap in V8 is split into different regions called generations. There is a young generation (split further into 'nursery' and 'intermediate' sub-generations), and an old generation. Objects are first allocated into the nursery. If they survive the next GC, they remain in the young generation but are considered 'intermediate'. If they survive yet another GC, they are moved into the old generation.



The V8 heap is split into generations. Objects are moved through generations when they survive a GC.

In garbage collection there is an important term: "The Generational Hypothesis". This basically states that most objects die young. In other words, most objects are allocated and then almost immediately become unreachable, from the perspective of the GC. This holds not only for V8 or JavaScript, but for most dynamic languages.

V8's generational heap layout is designed to exploit this fact about object lifetimes. The GC is a compacting/moving GC, which means that it copies objects which survive garbage collection. This seems counterintuitive: copying objects is expensive at GC time. But we know that only a very small percentage of objects actually survive a garbage collection, according to the generational hypothesis. By moving only the objects which survive, every other allocation becomes 'implicit' garbage. This

means that we only pay a cost (for copying) proportional to the number of surviving objects, not the number of allocations.

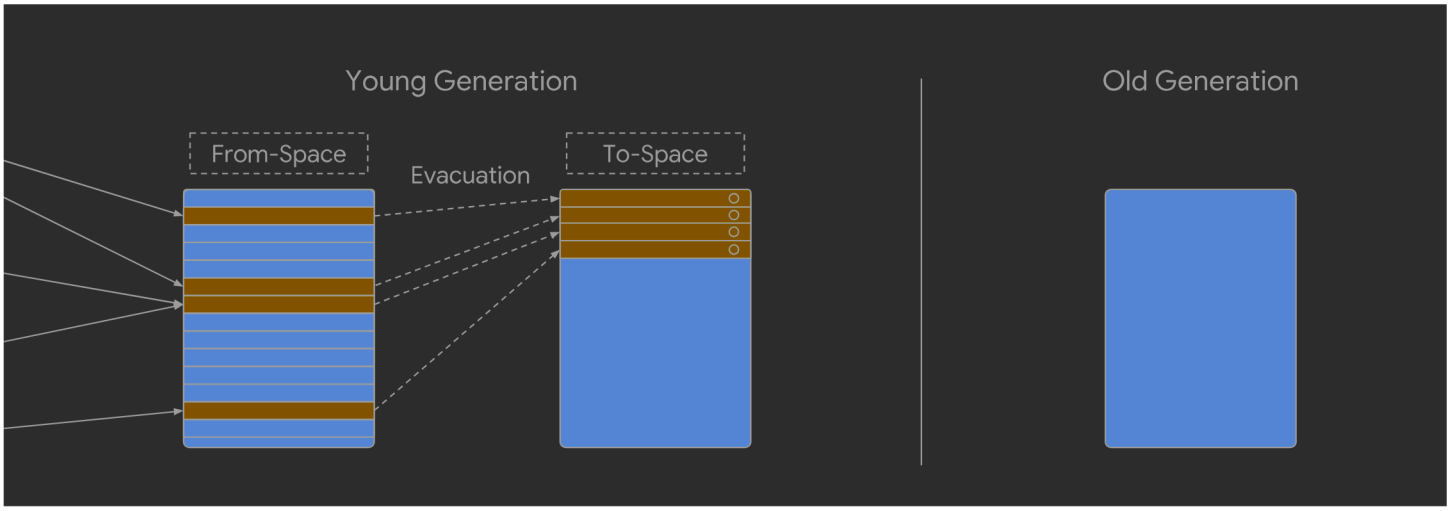
Minor GC (Scavenger)

There are two garbage collectors in V8. The **Major GC (Mark-Compact)** collects garbage from the whole heap. The **Minor GC (Scavenger)** collects garbage in the young generation. The major GC is effective at collecting garbage from the whole heap, but the generational hypothesis tells us that newly allocated objects are very likely to need garbage collection.

In the Scavenger, which only collects within the young generation, surviving objects are always evacuated to a new page. V8 uses a ‘semi-space’ design for the young generation. This means that half of the total space is always empty, to allow for this evacuation step. During a scavenge, this initially-empty area is called ‘To-Space’. The area we copy from is called ‘From-Space’. In the worst case, every object could survive the scavenge and we would need to copy every object.

For scavenging, we have an additional set of roots which are the old-to-new references. These are pointers in old-space that refer to objects in the young generation. Rather than tracing the entire heap graph for every scavenge, we use write barriers to maintain a list of old-to-new references. When combined with the stack and globals, we know every reference into the young generation, without the need to trace through the entire old generation.

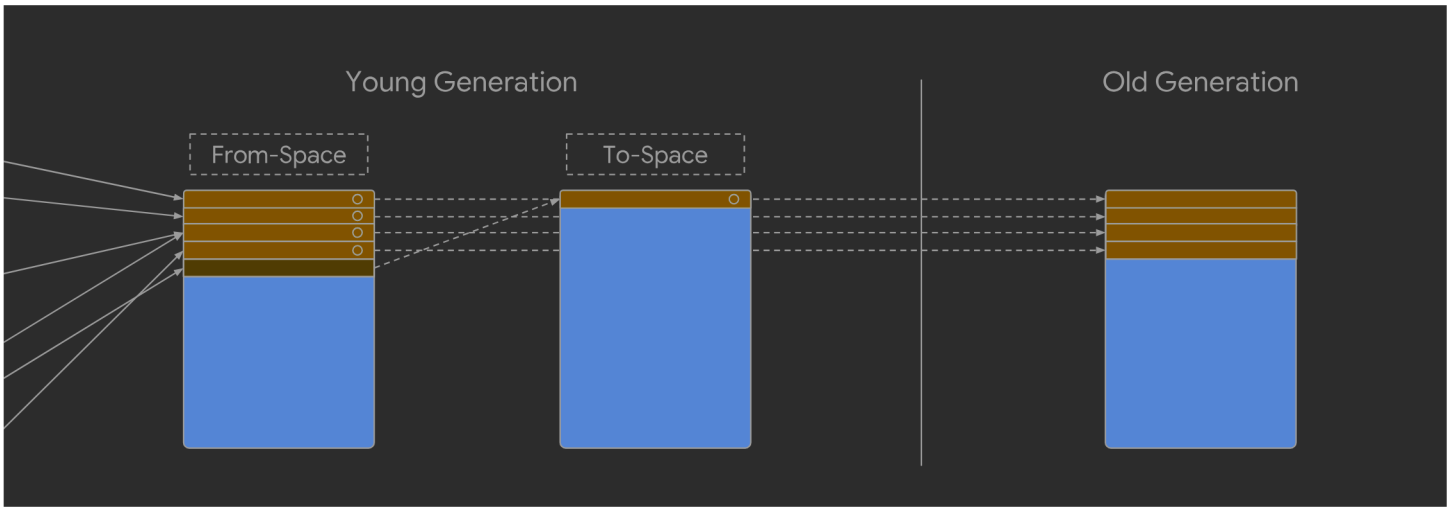
The evacuation step moves all surviving objects to a contiguous chunk of memory (within a page). This has the advantage of completing removing fragmentation - gaps left by dead objects. We then switch around the two spaces i.e. To-Space becomes From-Space and vice-versa. Once GC is completed, new allocations happen at the next free address in the From-Space.



The scavenger evacuates live objects to a fresh page.

We quickly run out of space in the young generation with this strategy alone. Objects that survive a second GC are evacuated into the old generation, rather than To-Space.

The final step of scavenging is to update the pointers that reference the original objects, which have been moved. Every copied object leaves a forwarding-address which is used to update the original pointer to point to the new location.

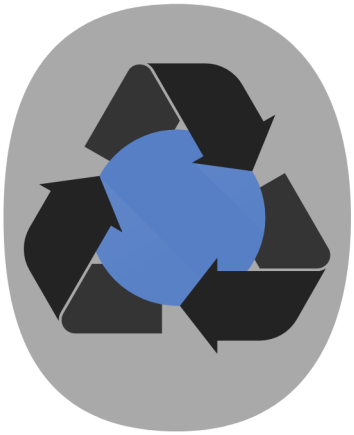


The scavenger evacuates ‘intermediate’ objects to the old generation, and ‘nursery’ objects to a fresh page.

In scavenging we actually do these three steps — marking, evacuating, and pointer-updating — all interleaved, rather than in distinct phases.

Orinoco

Most of these algorithms and optimizations are common in garbage collection literature and can be found in many garbage collected languages. But state-of-the-art garbage collection has come a long way. One important metric for measuring the time spent in garbage collection is the amount of time that the main thread spends paused while GC is performed. For traditional ‘stop-the-world’ garbage collectors, this time can really add up, and this time spent doing GC directly detracts from the user experience in the form of janky pages and poor rendering and latency.

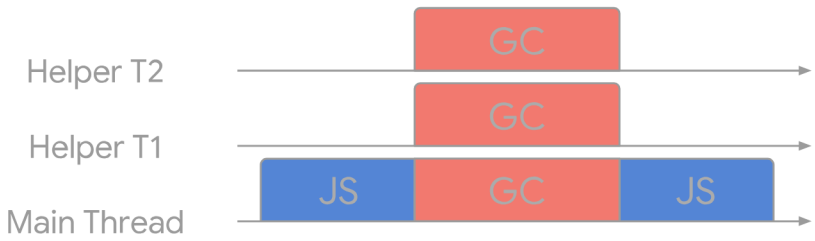


Logo for Orinoco, V8's garbage collector

Orinoco is the codename of the GC project to make use of the latest and greatest parallel, incremental and concurrent techniques for garbage collection, in order to free the main thread. There are some terms here that have a specific meaning in the GC context, and it's worth defining them in detail.

Parallel

Parallel is where the main thread and helper threads do a roughly equal amount of work at the same time. This is still a 'stop-the-world' approach, but the total pause time is now divided by the number of threads participating (plus some overhead for synchronization). This is the easiest of the three techniques. The JavaScript heap is paused as there is no JavaScript running, so each helper thread just needs to make sure it synchronizes access to any objects that another helper might also want to access.



The main thread and helper threads work on the same task at the same time.

Incremental

Incremental is where the main thread does a small amount of work intermittently. We don't do an entire GC in an incremental pause, just a small slice of the total work required for the GC. This is more difficult, because JavaScript executes between each incremental work segment, meaning that

the state of the heap has changed, which might invalidate previous work that was done incrementally. As you can see from the diagram, this does not reduce the amount of time spent on the main thread (in fact, it usually increases it slightly), it just spreads it out over time. This is still a good technique for solving one of our original problems: main thread latency. By allowing JavaScript to run intermittently, but also continue garbage collection tasks, the application can still respond to user input and make progress on animation.

Small chunks of the GC task are interleaved into the main thread execution.

Concurrent

Concurrent is when the main thread executes JavaScript constantly, and helper threads do GC work totally in the background. This is the most difficult of the three techniques: anything on the JavaScript heap can change at any time, invalidating work we have done previously. On top of that, there are now read/write races to worry about as helper threads and the main thread simultaneously read or modify the same objects. The advantage here is that the main thread is totally free to execute JavaScript — although there is minor overhead due to some synchronization with helper threads.

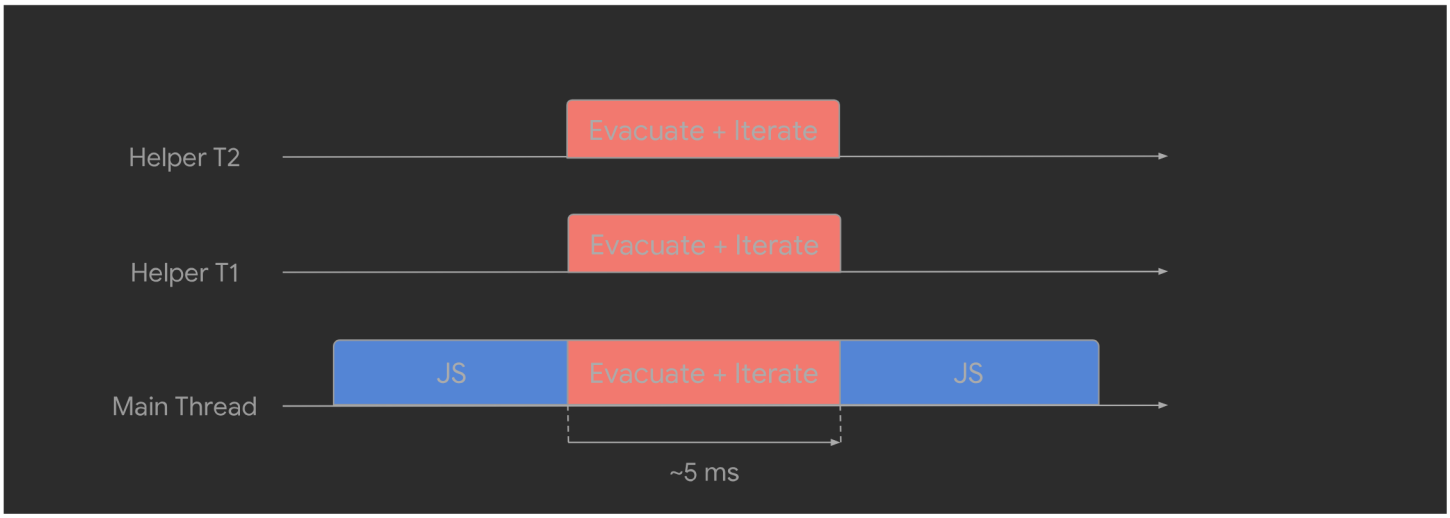
GC tasks happen entirely in the background. The main thread is free to run JavaScript.

State of GC in V8

Scavenging

Today, V8 uses parallel scavenging to distribute work across helper threads during the young generation GC. Each thread receives a number of pointers, which it follows, eagerly evacuating any live objects into To-Space. The scavenging tasks have to synchronize via atomic read/write/compare-and-swap operations when trying to evacuate an object; another scavenging task may have found the same object via a different path and also try to move it. Whichever helper moved the object

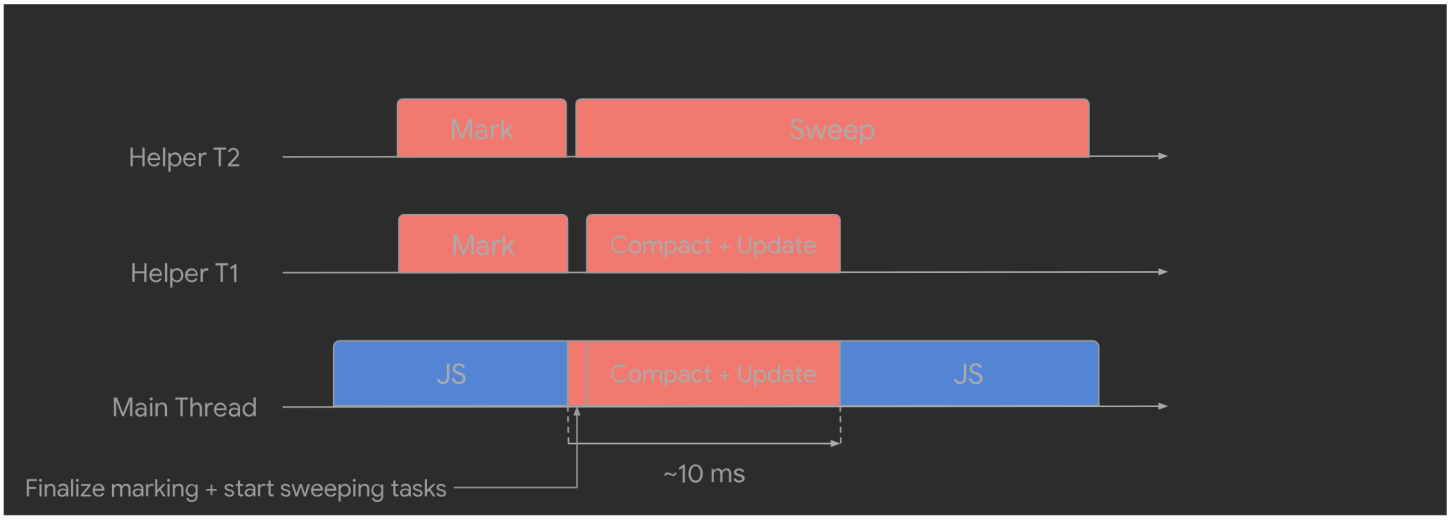
successfully then goes back and updates the pointer. It leaves a forwarding pointer so that other workers which reach the object can update other pointers as they find them. For fast synchronization-free allocation of surviving objects, the scavenging tasks use thread-local allocation buffers.



Parallel scavenging distributes scavenging work across multiple helper threads and the main thread.

Major GC

Major GC in V8 starts with concurrent marking. As the heap approaches a dynamically computed limit, concurrent marking tasks are started. The helpers are each given a number of pointers to follow, and they mark each object they find as they follow all references from discovered objects. Concurrent marking happens entirely in the background while JavaScript is executing on the main thread. Write barriers are used to keep track of new references between objects that JavaScript creates while the helpers are marking concurrently.



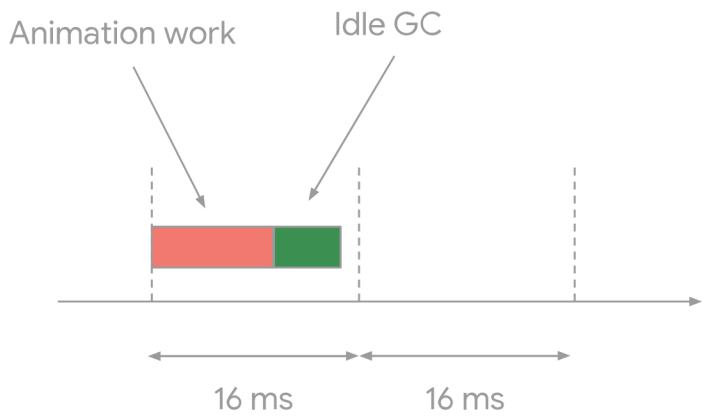
The major GC uses concurrent marking and sweeping, and parallel compaction and pointer updating.

When the concurrent marking is finished, or we reach the dynamic allocation limit, the main thread performs a quick marking finalization step. The main thread pause begins during this phase. This represents the total pause time of the major GC. The main thread scans the roots once again, to

ensure that all live objects are marked, and then along with a number of helpers, starts parallel compaction and pointer updating. Not all pages in old-space are eligible for compaction — those that aren't will be swept using the free-lists mentioned earlier. The main thread starts concurrent sweeping tasks during the pause. These run concurrently to the parallel compaction tasks and to the main thread itself — they can continue even when JavaScript is running on the main thread.

Idle-time GC

Users of JavaScript don't have direct access to the garbage collector; it is totally implementation-defined. V8 does however provide a mechanism for the embedder to trigger garbage collection, even if the JavaScript program itself can't. The GC can post 'Idle Tasks' which are optional work that would eventually be triggered anyway. Embedders like Chrome might have some notion of free or idle time. For example in Chrome, at 60 frames per second, the browser has approximately 16.6 ms to render each frame of an animation. If the animation work is completed early, Chrome can choose to run some of these idle tasks that the GC has created in the spare time before the next frame.



Idle GC makes use of free time on the main thread to perform GC work proactively.

For more details, refer to our in-depth publication on idle-time GC.

Takeaways

The garbage collector in V8 has come a long way since its inception. Adding parallel, incremental and concurrent techniques to the existing GC was a multi-year effort, but has paid off, moving a lot of work to background tasks. It has drastically improved pause times, latency, and page load, making animation, scrolling, and user interaction much smoother. The parallel Scavenger has reduced the main thread young generation garbage collection total time by about 20%–50%, depending on the workload. Idle-time GC can reduce Gmail's JavaScript heap memory by 45% when it is idle. Concurrent marking and sweeping has reduced pause times in heavy WebGL games by up to 50%.

But the work here is not finished. Reducing garbage collection pause times is still important for giving users the best experience on the web, and we are looking into even more advanced techniques. On top of that, Blink (the renderer in Chrome) also has a garbage collector (called Oilpan), and we are doing work to improve cooperation between the two collectors and to port some of the new techniques from Orinoco to Oilpan.

Most developers don't need to think about the GC when developing JavaScript programs, but understanding some of the internals can help you to think about memory usage and helpful programming patterns. For example, with the generational structure of the V8 heap, short-lived objects are actually very cheap from the garbage collector's perspective, as we only pay for objects that survive the collection. These sorts of patterns work well for many garbage-collected languages, not just JavaScript.

Posted by Peter 'the garbo' Marshall (@hooraybuffer).

Retweet this article!