A tour of V8: Garbage Collection

Published on 2013-09-23
Edited on 2014-01-26
Tagged: garbage-collection javascript v8 virtual-machines

This article is part of the series "A tour of V8".

This article has been translated into Chinese. Thanks to liuyanghejerry!

In the last few articles, we dived deep into the implementation of the V8 JavaScript engine. We discussed the non-optimizing full compiler, the optimizing Crankshaft compiler, and the representation of objects. In this article, we'll examine V8's garbage collector.

Garbage collection is really a double-edged sword. On the positive side, it allows for a massive simplification in languages that use it, since memory no longer needs to be managed explicitly by the programmer. It reduces (but does not eliminate!) a large class of errors, memory leaks, which plague large long-running applications. For some programs, it can even improve performance.

On the other hand, using a garbage collected language means relinquishing a great deal of control over how memory is managed in your program, which is an especially big concern for mobile applications. In JavaScript's case, you relinquish *all* control over how memory is managed: the ECMAScript specification doesn't expose any interface to the garbage collector. There's no way for a web app to measure its memory usage or provide any hints to the garbage collector.

Performance of garbage collected languages in not strictly better or worse than languages without managed memory. In C, allocating and freeing objects can be costly, since heap bookkeeping tends to be more complicated if you need to free objects later. With managed memory, allocation usually means just incrementing a pointer, but as we'll see, you have to pay for it eventually when you run out of memory and the garbage collector kicks in. Poorly implemented garbage collection systems can lead to long, unpredictable and embarrassing pauses, which make interactive systems (especially those with animations) frustrating to use. Reference counting is frequently touted as an alternative to garbage collection, but these systems can have the same unpredictable pauses when the last reference to a large sub-graph of objects is removed. Reference counting systems can also have a fairly large overhead for executing numerous load/increment/store instructions.

For better or worse, JavaScript requires garbage collection. V8's implementation has improved and matured significantly in the last year. Performance is pretty good, pauses are short, and overhead is pretty manageable.

Let's dig in!

The basics

The fundamental problem garbage collection solves is to identify dead regions of memory. Once identified, these regions can be re-used for new allocations or released back to the operating system. An object is *dead* if it is not live (duh). An object is *live* if it is pointed to by a root object or another live object. *Root objects* are live by definition. They are objects pointed to directly by V8 or the web browser. For example, objects pointed to by local variables are root objects, since the stack is scanned for roots. Global objects are roots, since they are always accessible. Browser objects, such as DOM elements, are also roots, though these may be weakly referenced in some cases.

As an aside, you should note that the above definition of *live* is fairly relaxed. We could have said that an object is *live* if it will be referenced again by the program. For example:

```
function f() {
  var obj = {x: 12};
  g();   // might contain an infinite loop.
  return obj.x;
}
```

Unfortunately, we can't solve this problem precisely, because it is equivalent to the Halting Problem, which is undecidable. So we make a convenient approximation: an object is live if it is reachable through some chain of pointers from an object which is live by definition. Everything else is garbage.

Heap organization

Before we dive into the internal workings of the garbage collector, let's talk about how the heap itself is organized. V8 divides the heap into several different *spaces*:

- **New-space:** Most objects are allocated here. New-space is small and is designed to be garbage collected very quickly, independent of other spaces.
- **Old-pointer-space:** Contains most objects which may have pointers to other objects. Most objects are moved here after surviving in new-space for a while.
- **Old-data-space:** Contains objects which just contain raw data (no pointers to other objects). Strings, boxed numbers, and arrays of unboxed doubles are moved here after surviving in new-space for a while.
- **Large-object-space:** This space contains objects which are larger than the size limits of other spaces. Each object gets its own `mmap`'d region of memory. Large objects are never moved by the garbage collector.
- **Code-space:** `Code` objects, which contain JITed instructions, are allocated here. This is the only space with executable memory (although `Codes` may be allocated in large-object-space, and those are executable, too).
- **Cell-space, property-cell-space and map-space:** These spaces contain `Cells`, `PropertyCells`, and `Maps`, respectively. Each of these spaces contains objects which are all the same size and has some constraints on what kind of objects they point to, which simplifies collection.

Each space is composed of a set of pages. A `Page` is a contiguous chunk of memory, allocated from the operating system with `mmap` (or whatever the Windows equivalent is). Pages are always 1 MB in size and 1 MB aligned, except in large-object-space, where they may be larger. In addition to storing objects, pages also contain a header (with various flags and meta-data) and a marking bitmap (used to indicate which objects are live). Each page also has a *slots buffer*, allocated in separate memory, which forms a list of objects which may point to objects stored on the page. This is commonly known as a *remembered set*. More on this later.

With the background out of the way, let's dive into the garbage collector.

Discovering pointers

Distinguishing pointers and data on the heap is the first problem any garbage collector needs to solve. The GC needs to follow pointers in order to discover live objects. Most garbage collection algorithms can migrate objects from one part of memory to another (to reduce fragmentation and increase locality), so we also need to be able to rewrite pointers without disturbing plain old data.

There are three popular approaches to identifying pointers:

- **Conservative** - This is necessary for implementations which get no support from the compiler. Basically, we treat all aligned words on the heap as if they were pointers. This means some data will be treated as pointers. Consequently, we might get weird memory leaks when an integer that kind of looks like a pointer keeps a large subgraph of objects alive. We can't move any objects around in memory, since we might accidentally change data that we thought was a pointer. As a result, we don't see the benefits of compacting garbage collection (simpler allocation, lower memory footprint, better cache locality). Garbage collectors for C/C++ such as the Boehm-Demers-Weiser garbage collector take this approach.
- **Compiler hints** - If we are working in a statically-typed language, the compiler can tell us the offsets of pointers within each class. As long as we can identify what class an object comes from, we can find all of its pointers. The Java Virtual Machine takes this approach. Unfortunately, this does not work well for dynamically typed languages like JavaScript, since any field in an object can contain pointers or data.
- **Tagged pointers** - With this approach, we reserve a bit at the end of each word to indicate whether it is pointer or data. This approach requires limited compiler support, but it's simple to implement while being fairly efficient. V8 takes this approach. Some statically typed languages, such as OCaml, also take this approach.

V8 represents small integers (called Smis internally) in the range -230..230-1 with a 32-bit word with the low bit 0. Pointers have low bits 01. This works because objects are always at least 4-byte aligned, and we can never have a pointer to the middle of an object in JavaScript. Most objects on the heap just contain a list of tagged words, so the garbage collector can quickly scan them, following the pointers and ignoring the integers. Some kinds of objects, such as strings, are known to contain only data (no pointers), so their contents do not have to be tagged.

Generational collection

In the vast majority of programs, objects tend to die young: most objects have a very short lifetime, while a small minority of objects tend to live much longer. To take advantage of this behavior, V8 divides the heap into two *generations*. Objects are allocated in *new-space*, which is fairly small (between 1 and 8 MB, depending on behavior heuristics). Allocation in new space is very cheap: we just have an allocation pointer which we increment whenever we want to reserve space for a new object. When the allocation pointer reaches the end of new space, a *scavenge* (minor garbage collection cycle) is triggered, which quickly removes the dead objects from new space. Objects which have survived two minor garbage collections are *promoted* to old-space. Old-space is garbage collected during a *mark-sweep* or *mark-compact* (major garbage collection cycle), which is much less frequent. A major garbage collection cycle is triggered when we have promoted a certain amount of memory to old space. This threshold shifts over time depending on the size of old space and the behavior of the program.

Since scavenges occur frequently, they must be very fast. Scavenge is an implementation of Cheney's algorithm. Basically, new-space is divided into two equal sized semi-spaces: to-space and from-space. Most allocations are made in to-space (although there are certain kinds of objects, such as executable `Codes` which are always allocated in old-space). When to-space fills up, we swap to-space and from-space (so all the objects are in from-space). Then, we copy the live objects out of from-space, moving them to to-space or promoting them to old-space. The objects are compacted in to-space in the process, so this improves cache locality, and allocation remains fast and simple.

Here is a pseudo-code version of the algorithm:

```
def scavenge():
  swap(fromSpace, toSpace)
  allocationPtr = toSpace.bottom
  scanPtr = toSpace.bottom

  for i = 0..len(roots):
    root = roots[i]
    if inFromSpace(root):
      rootCopy = copyObject(&allocationPtr, root)
      setForwardingAddress(root, rootCopy)
      roots[i] = rootCopy

  while scanPtr < allocationPtr:
    obj = object at scanPtr
    scanPtr += size(obj)
    n = sizeInWords(obj)
    for i = 0..n:
      if isPointer(obj[i]) and not inOldSpace(obj[i]):
        fromNeighbor = obj[i]
        if hasForwardingAddress(fromNeighbor):
          toNeighbor = getForwardingAddress(fromNeighbor)
        else:
          toNeighbor = copyObject(&allocationPtr, fromNeighbor)
          setForwardingAddress(fromNeighbor, toNeighbor)
        obj[i] = toNeighbor

def copyObject(*allocationPtr, object):
  copy = *allocationPtr
  *allocationPtr += size(object)
  memcpy(copy, object, size(object))
  return copy
```

At all times in this algorithm, we maintain two pointers into to-space: `allocationPtr` points to where we will allocate the next object. `scanPtr` points to the next object we will scan for live pointers. Objects below `scanPtr` have been completely processed: they and all their neighbors are in to-space, and all their pointers are updated. Objects between `scanPtr` and `allocationPtr` have been copied to to-space, but their internal pointers still point to objects in from-space which may not have been copied yet. You can think of `scanPtr` and `allocationPtr` as the front and back of a queue of objects in a breadth-first-search, since that's logically what's happening.

We initialize the algorithm by copying new-space objects reachable from the roots, then we enter the main loop. In each step of the loop, we remove an object from the queue by incrementing `scanPtr`, and then we follow each of its internal pointers. If we find a pointer that does not go to from-space, we ignore it, since it must point to old-space, which we are not collecting. If we find a pointer to an object in from-space that has not been copied yet (no forwarding address), we copy it to the end of to-space by incrementing `allocationPtr` (adding it to the queue). We also set a forwarding address which points to the new copy. The forwarding address is stored in the first word of the object in from-space, replacing the map pointer. The garbage collector can distinguish forwarding addresses and map pointers by checking the low bit: map pointers are tagged (low bit is set), while forwarding addresses are not (low bit is clear). If we find a pointer to an object in from-space which has already been copied (has a forwarding address), we simply update the pointer to point to the to-space copy.

The algorithm terminates when there are no more objects to process (`scanPtr` reaches `allocationPtr`). At that point, the contents of from-space are considered garbage and may be freed or reused for other purposes.

Write barriers: the secret ingredient

There was one detail I glossed over above: what if a live object in old-space contains the only pointer to an object in new-space? How do we know the new object is still alive? We certainly don't want to scan all of old-space for pointers to new-space; that would make scavenges very expensive.

To get around this problem, we actually maintain a list of pointers from old-space to new-space in the *store buffer*. When an object is newly allocated, we know no other objects point to it. Whenever a pointer to an object in new-space is written to a field of an object in old-space, we record the location of that field in the store buffer. In order to accomplish this, after most stores, we execute a bit of code called a *write barrier*, which detects and records these pointers.

You are probably thinking: isn't it expensive to execute a bunch of extra instructions every time we write a pointer into an object? Well, yes. Yes it is. This is one of the ways we pay for the convenience of garbage collection. But it's not as expensive as you might think. Writes are not nearly as common as reads. Some garbage collection algorithms (not V8's) use read barriers, and they require hardware assistance in order to have acceptably low overhead. There are also a number of optimizations implemented to lower the cost:

- Most execution time is spent in optimized code generated by Crankshaft. Frequently, Crankshaft can statically prove an object is in new-space. Write barriers are omitted for stores to these objects.
- A new optimization in Crankshaft is to allocate objects on the stack when no non-local references are made to them. Stores to stack-allocated objects obviously don't need write barriers.
- Oldâ†’new pointers are relatively rare, so we can optimize for the common case by detecting newâ†’new and oldâ†’old pointers quickly. Every page is aligned on a 1 MB boundary, so given an object address, we can find its page by masking off the low 20 bits. Page headers have flags indicating whether they are in old or new space. So we can check which space both objects are in with just a few instructions.
- Once we've found an oldâ†’new pointer, we just record the location of the pointer at the end of the store buffer. Periodically (which the store buffer fills up), we sort and de-duplicate the entries in the store buffer and remove entries which have been overwritten and no longer point to new-space.

Mark-sweep and Mark-compact

The Scavenge algorithm is great for quickly collecting and compacting a small amount of memory, but it has large space overhead, since we need physical memory backing both to-space and from-space. This is acceptable as long as we keep new-space small, but it's impractical to use this approach for more than a few megabytes. To collect old space, which may contain several hundred megabytes of data, we use two closely related algorithms, *Mark-sweep* and *Mark-compact*.

Both of these algorithms operate in two phases: a *marking phase* and a *sweeping phase* or a *compacting phase* (as you might have guessed from the names).

During the marking phase, all live objects on the heap are discovered and marked. Each page contains a marking bitmap with one bit per allocatable word on that page. This is necessary since objects can start at any word-aligned offset. Obviously, this incurs some memory overhead (3.1% on 32-bit systems, 1.6% on 64-bit systems), but all memory management systems have some overhead, and this is pretty reasonable. Pairs of bits are used to represent the marking state of an object; objects are at least two words long, so these pairs never overlap. There are three marking states. If an object is *white*, it has not yet been discovered by the garbage collector. If an object is *grey*, it has been discovered by the garbage collector, but not all of its neighbors have been processed yet. If an object is *black*, it has been discovered, and all of its neighbors have been fully processed.

The marking algorithm is essentially a depth-first-search, which makes sense if you think of the heap as a directed graph of objects connected by pointers. At the beginning of the marking cycle, the marking bitmap is clear, and all objects are white. Objects reachable from the roots are colored grey and pushed onto the marking deque, a separately allocated buffer used to store objects being processed. At each step, the GC pops an object from the deque, marks it black, marks neighboring white objects as grey, and pushes them onto the deque. The algorithm terminates when the deque is empty and all discovered objects have been marked black. Very large objects, such as long arrays, may be processed in pieces to reduce the chance of the deque overflowing. If the deque does overflow, objects are still colored grey but are not pushed onto the deque (so their neighbors are not discovered). When the deque is empty, the GC must scan the heap for grey objects, push them back onto the deque, and resume marking.

Here's a pseudocode version of the marking algorithm:

```
markingDeque = []
overflow = false

def markHeap():
  for root in roots:
    mark(root)

  do:
    if overflow:
      overflow = false
      refillMarkingDeque()

    while !markingDeque.isEmpty():
      obj = markingDeque.pop()
      setMarkBits(obj, BLACK)
      for neighbor in neighbors(obj):
        mark(neighbor)
  while overflow


def mark(obj):
  if markBits(obj) == WHITE:
    setMarkBits(obj, GREY)
    if markingDeque.isFull():
      overflow = true
    else:
      markingDeque.push(obj)

def refillMarkingDeque():
  for each obj on heap:
    if markBits(obj) == GREY:
      markingDeque.push(obj)
      if markingDeque.isFull():
        overflow = true
        return
```

When the marking algorithm terminates, all live objects are marked black, and all dead objects are left white. This information is used by either the sweeping phase or the compacting phase, depending on which algorithm is being used.

Once marking is complete, we can reclaim memory by either sweeping or compacting. Both algorithms work at a page level (remember, V8 pages are 1 MB contiguous chunks, not the same as virtual memory pages).

The sweeping algorithm scans for contiguous ranges of dead objects, converts them to free spaces, and adds them to free lists. Each page maintains separate free lists for small regions (< 256 words), medium regions (< 2048 words), large regions (< 16384 words), and huge regions (anything larger). The sweeping algorithm is extremely simple: it just iterates across the page's marking bitmap, looking for ranges of unmarked objects. Free lists are mostly used by the scavenge algorithm for promoting surviving objects to old-space, but they are also used by the compacting algorithm to relocate objects. Some kinds of objects can only be allocated in old-space, so free lists are used for those, too.

The compacting algorithm attempts to reduce actual memory usage by migrating objects from fragmented pages (containing a lot of small free spaces) to free spaces on other pages. New pages may be allocated, if necessary. Once a page is evacuated, it can be released back to the operating system. The migration process is actually pretty complicated, so I'll gloss over some of the details here. Basically, for each live object on an evacuation candidate page, space is allocated from a free list on another page. The object is copied into the freshly allocated space, and a forwarding address is left in the first word of the original object. During the evacuation, the locations of pointers between evacuated objects are recorded. Once the evacuation is complete, V8 iterates over the list of recorded pointer locations and updates them to point to the new copies. Locations of pointers between different pages are recorded during the marking process, so pointers from other pages are also updated at this time. Note that if a page becomes too "popular", i.e., there are too many pointers to be recorded to its objects from other pages, pointer location recording is disabled for that page, and the page cannot be evacuated until the next garbage collection cycle.

Incremental marking and lazy sweeping

As you may have guessed, mark-sweep and mark-compact can be pretty time-consuming if they are applied to a large heap with a lot of live data. When I first started working on V8, it was not unusual to see garbage collection pauses in the 500-1000 ms range. This is obviously unacceptable, even for mobile devices.

In mid-2012, Google introduced two improvements that reduced garbage collection pauses significantly: incremental marking and lazy sweeping.

Incremental marking allows the heap to be marked in a series of small pauses, on other order of 5-10 ms each (on mobile). Incremental marking begins when the heap reaches a certain threshold size. After being activated, every time a certain amount of memory is allocated, execution is paused to perform an incremental marking *step*. Incremental marking, like regular marking, is basically a depth-first-search, and it uses the same white-grey-black system for classifying objects.

The difference between incremental and regular marking is that during incremental marking, the object graph can change! The main problem we need to watch out for is for pointers being created from black to white objects. As you'll recall, black objects have been completely scanned by the garbage collector, and won't be scanned again, so if a blackâ†’white pointer is created, we could end up classifying the live white object as dead. Once again, write barriers come to our rescue. Not only do they record oldâ†’new pointers, but they also detect blackâ†’white pointers. When such a pointer is detected, the black object is changed to grey and pushed back onto the marking deque. When the marking algorithm pops the object later, its pointers will be re-scanned, and the white object will be discovered. Order is preserved.

Once incremental marking is complete, lazy sweeping begins. All objects have been marked live or dead, and the heap knows exactly how much memory could be freed by sweeping. All this memory doesn't necessarily have to be freed up right away though, and delaying the sweeping won't really hurt anything. So rather than sweeping all pages at the same time, the garbage collector sweeps pages on an as-needed basis until all pages have been swept. At that point, the garbage collection cycle is complete, and incremental marking is free to start again.

Google has also recently added support for parallel sweeping. Since the main execution thread won't touch dead objects, pages can be swept by separate threads with a minimal amount of synchronization. There is some activity happening with parallel marking, too, but this seems to be in the early experimental stages right now.

Conclusion

Garbage collection is really complicated. I glossed over a lot of details in this article, and it still ended up being really long. One of my colleagues told me he considered the garbage collector to be scarier to work on than the register allocator, and I completely agree. That said, I would much rather have the complexity of memory management hidden inside the runtime than force all application programmers to deal with it. While garbage collection may have some overhead and occasionally odd behavior, it frees us from worrying about a lot of messy details so we can focus on more important things.

If you're interested in learning more about garbage collection, I would encourage you to read *Garbage Collection* by Richard Jones and Rafael Lins. It's an excellent reference on the subject, and it covers pretty much everything you need to know. You might also be interested in *Garbage First Garbage-Collection*, a research paper describing the garbage collection algorithm used by the JVM.