Sliced string memory reduction

Attention - this doc is public and shared with the world!

Bug: v8:2869
Contributors: seth.brenith@microsoft.com
Status: Inception | Draft | Accepted | Done

Document overview

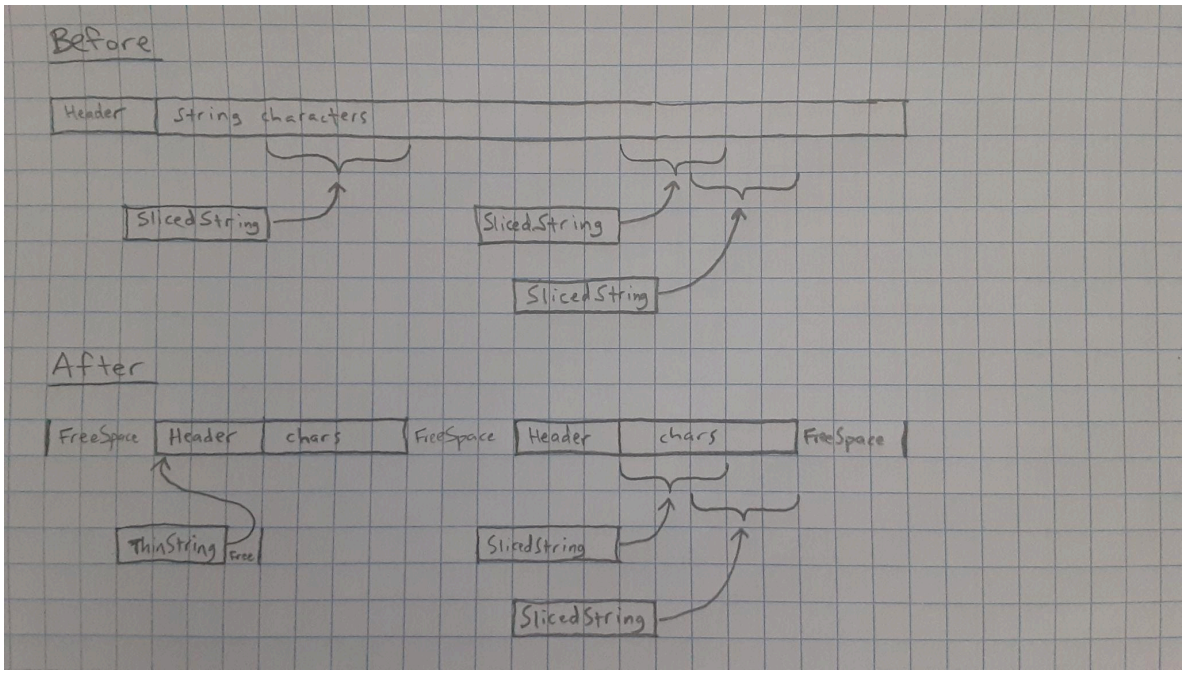
This document presents two orthogonal ideas related to reclaiming memory held by SlicedStrings, if the parent string is a SeqString. External strings are ignored.

Motivation

A long-standing bug in V8 (and some other JS engines) is that strings created by String.prototype.substr or similar point to, and keep alive, the original source string. In some cases this can mean a tiny substring keeping an enormous source string alive. This is a perennial source of problems for JS developers, who have no clear way to indicate their intended behavior when taking a substring and must resort to engine-specific hacks.

Idea 1: shrink-wrapping

If a SeqString is retained only by SlicedStrings, it would be nice to perform an in-place “shrink-wrapping” operation that keeps only the parts of the source string that are used. In the following example diagram, the “before” state depicts three SlicedString objects which refer to parts of a SeqString. After shrink-wrapping, the SeqString has been split into two smaller SeqStrings, without moving any of the character data, and the SlicedStrings are updated accordingly. One of them can even turn into a ThinString, if we relax the current rule that says ThinStrings only point to internalized strings. Otherwise, it could remain as a SlicedString.



This shrink-wrapping transformation is pretty simple and mechanical, and I hope that it would be fast enough to run during the GC's stop-the-world pause, because I'm not sure when else it could run safely. It requires writing a handful of object headers (both FreeSpace and SeqString), and updating the SlicedString data.

However, the same operation wouldn't work when the SeqString is in large-object space. We could imagine writing properly aligned MemoryChunk headers to split the space, but I think V8 expects each MemoryChunk to represent a separate OS-level allocation block, and I don't think OSes provide any way to split an allocation block. For large-object space, I think the only reasonable solution is to reallocate the string data as implemented by this proof-of-concept.

Idea 2: avoiding iterating every SlicedString

Leszek has stated that his previous attempt to address this problem did too much work during the interval where the GC must pause all execution, because it had to iterate every SlicedString in the heap. It seems clear that any solution needs to collect a set of SlicedStrings and iterate them, since those SlicedStrings must be updated to point to newly materialized SeqStrings, but iterating every SlicedString in the whole heap on every GC is too much work. What can be done? I propose a strategy that takes two GC cycles:

1. On the first GC, detect which SeqStrings are retained only by SlicedStrings, and, at a rough granularity, which parts of the SeqString are in use. Decide whether it is worthwhile to remove the SeqString.

2. On the second GC, for any SeqStrings which were chosen for removal, collect a list of all SlicedStrings that refer to them. Perform the appropriate transformation (either shrink-wrapping as described above, or reallocation).

This approach still requires some work during stop-the-world, but I think we declare that it is almost always *useful* work: we only iterate SlicedStrings when we've already decided that it's worth rewriting them. The only case where we would iterate and then not rewrite would require that between the first and second GCs, somebody used a weak reference to get the SeqString, made a new SlicedString referring to it, and then dropped their strong ref to the SeqString.

Tracking retained parts of SeqStrings

Step 1 above says we need to track "at a rough granularity, which parts of the SeqString are in use". That sounds like a bitmap. Do we have to allocate more space for this info? Conveniently, there's already a plenty-big bitmap allocated for us: the marking bits! In old space, there is a marking bit per four (or eight) bytes of data, and we only use the first two for tracking object reachability. In large-object space, there are thousands of marking bits in the MemoryChunk header, even though only two bits are needed for the resident object.

I'm not sure exactly what is the right granularity for the bitmap, and it should probably vary with the size of the SeqString so we don't need to set or check a ridiculous number of bits for very long strings.

We'll also need to designate a couple of bits as flags. I think this data could go in the third and fourth marking bits for the object, right after the usual color data. So the full list of data stored in the marking bits for a SeqString would look like this:

- First two marking bits: normal white/grey/black tracking
- Third bit: "RetainedBySlices": this SeqString is kept alive by SlicedStrings, even if its marking bits are white
- Fourth bit: "PendingDemolition": this SeqString, which is only used by SlicedStrings, has been selected for demolition, so we must collect a list of the SlicedStrings that point to it
- Fifth and subsequent bits: "CharactersNeeded": for each N bytes of SeqString data (where N is some TBD chunk size), does any SlicedString need any characters in that region?

First GC cycle

Any marking thread may encounter a reference from a SlicedString to a SeqString. If the SeqString is already grey or black, there's nothing to do: it's strongly referenced by something other than a SlicedString. Otherwise, the marking thread sets RetainedBySlices on the SeqString, and sets each of the CharactersNeeded bits corresponding to the parts of the string used by the SlicedString. This of course requires atomic CAS operations, but they can be done 32 bits at a time.

When the sweeper finds a SeqString that is marked white, it checks the RetainedBySlices bit. If that bit is set:

- The object is kept alive as if it were marked black, and
- The sweeper iterates through the rest of the CharactersNeeded bits. If enough of those bits are zero (based on a heuristic TBD), it sets the PendingDemolition bit. Otherwise it clears the CharactersNeeded bits.

Note that this work can happen in a deferred or background sweeping task, as long as the next marking cycle hasn't started yet.

Second GC cycle

When any marking thread encounters a reference from a SlicedString to a SeqString and the PendingDemolition flag is set on the SeqString, it adds that SlicedString to a list for follow-up work.

Finally, during stop-the-world, the garbage collector must iterate the SlicedStrings, determine the optimal cuts of the SeqString, and either shrink-wrap or queue a task to reallocate.