Reusing existing Scripts in the isolate compilation cache

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

Contact: seth.brenith@microsoft.com
Status: Inception | **Draft** | Accepted | Done
Bug: v8:12808

LGTMs needed

Name	Write (not) LGTM in this row
leszeks	
<add yourself>	

Background

Isolate script cache

The Isolate script cache contains the top-level SharedFunctionInfo for previously executed scripts, so that if the same script is evaluated again, the new script evaluation can use the existing SharedFunctionInfo (as well as any other existing SharedFunctionInfos from that script, along with their bytecode, etc.). This reuse is nice for load speed and saves memory.

If the top-level SharedFunctionInfo has its bytecode flushed, then it is removed from the Isolate script cache. However, the Isolate script cache keeps a weak pointer to the Script object even after the top-level SharedFunctionInfo is cleared, as implemented recently. This means there are three possible outcomes when attempting to look up a script in the Isolate script cache:

1. Cache miss: there's no data about this script
2. Cache hit: there's a top-level SharedFunctionInfo which is compiled and ready to execute
3. Partial cache hit: there's a pre-existing Script, but it has no top-level SharedFunctionInfo, or the top-level SharedFunctionInfo isn't compiled

Ideally, all code paths would reuse the existing Script in the third case. Even though the top-level code needs recompilation, it is possible to save substantial memory by avoiding duplication of SharedFunctionInfos for other function literals within the script.

Ways to compile a script

There are four ways to generate bytecode for a script:

1. Parse the source text on the main thread.
2. Parse the source text on a background thread.
3. Deserialize code cache data on the main thread.
4. Deserialize code cache data on a background thread.

Both options involving background threads can start without the full script text being available. They must perform some finalization work on the main thread, but we generally try to keep that main-thread work as small as possible.

Handling a partial hit in the Isolate script cache is already implemented for the case of parsing on the main thread (which was very simple), but not yet for the other three cases.

Purpose of this document

This document describes some possible strategies for reusing an existing Script from the compilation cache when the new script instance is being parsed on a background thread or deserialized from code cache data, so that all four code paths listed above can benefit from the existing Script.

Implementation options

Option 1: Compile normally and then merge on the main thread

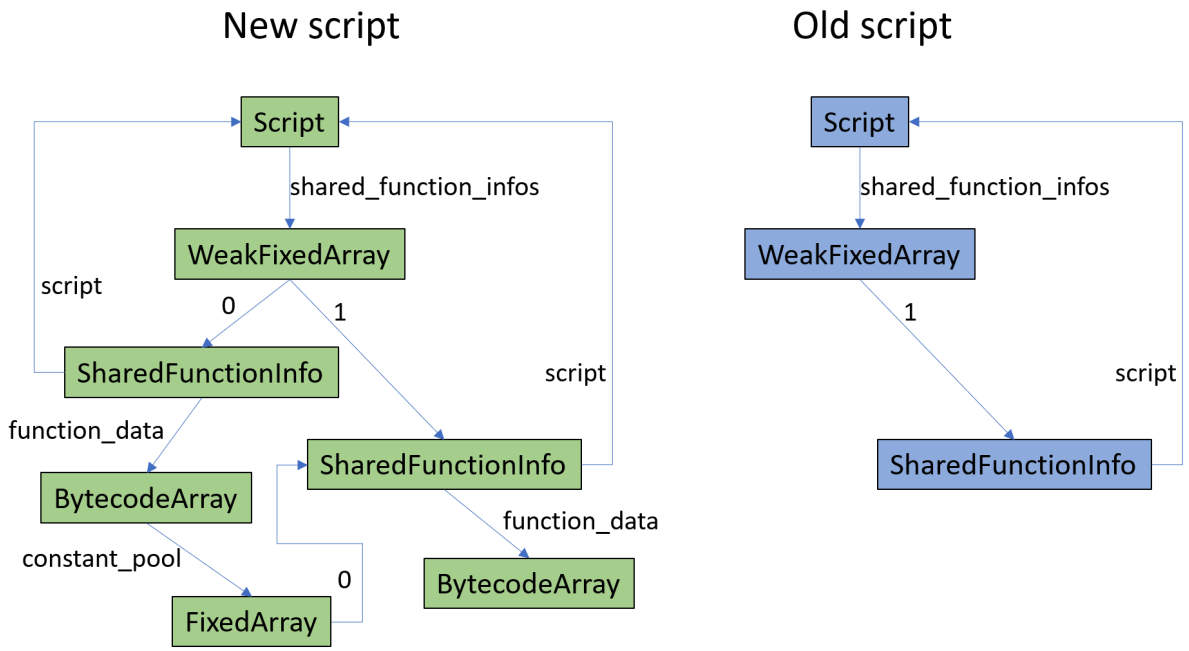
We can allow compilation to finish as usual, and then do a traversal of the newly created objects to merge their data into the existing Script. This option is appealing because of its simplicity: the same implementation works for any compilation method, the code is easy to read, and there are no threading risks to worry about. Here is a CL that implements this option.

When merging, we must reuse the pre-existing SharedFunctionInfos and bytecode wherever possible, since there might be pointers from elsewhere in the heap to those objects. If the newly compiled script contains a SharedFunctionInfo or bytecode that is lacking from the old script, we can update the old script to point to the new version of that object. Then we traverse the new

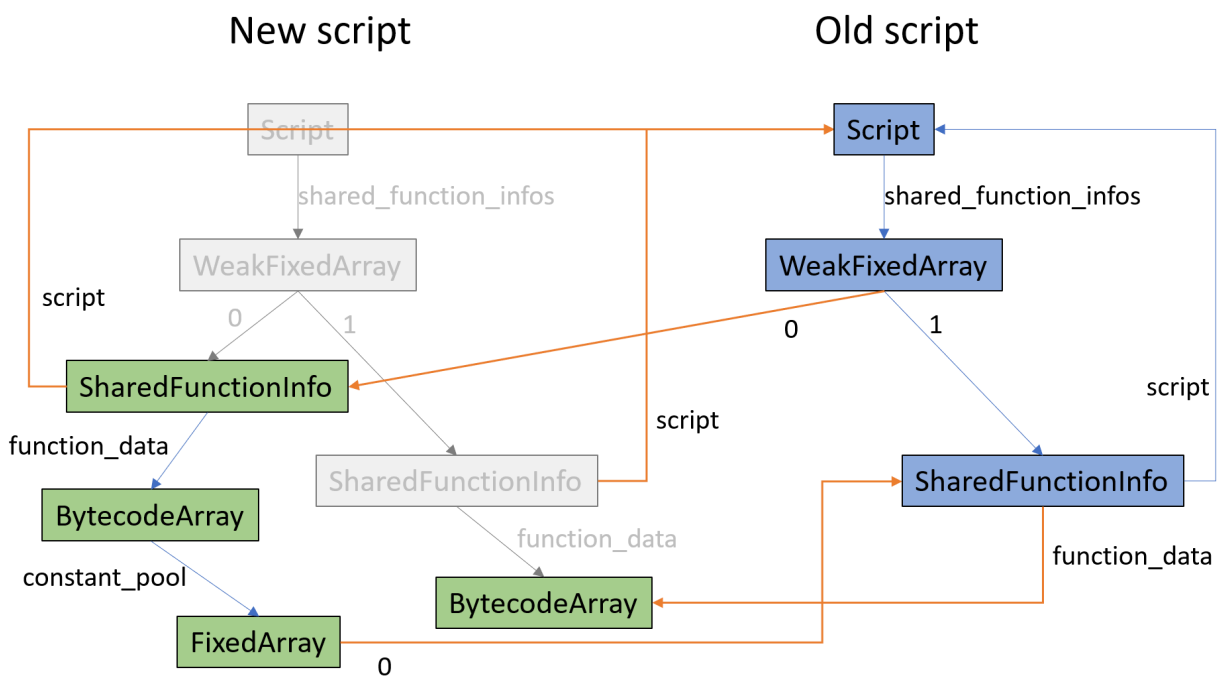
object graph in order to find pointers to SharedFunctionInfos or the Script, and update those pointers to refer to the corresponding pre-existing objects.

The following diagrams show what such a merge might look like for an extremely small script. On the right is the pre-existing script found in the Isolate script cache. It has no first SharedFunctionInfo, and its second SharedFunctionInfo has no bytecode. On the left is a newly compiled copy of the same script, where both SharedFunctionInfos were eagerly compiled. Note that the first SharedFunctionInfo has a reference to the second via its bytecode's constant pool; this reference must be found by traversal and updated. These diagrams omit feedback metadata for simplicity, but it would be handled like bytecode.

Before merging:



After merging:



Not every new object needs to be traversed when searching for pointers to update; the objects that will be removed (light gray) can easily be skipped.

The primary downside of this option is that it takes nontrivial time on the main thread. In some brief testing, I found that merging generally took 25-30 percent of the time it took to deserialize code cache data. Another downside is generation of objects which immediately become garbage (the light gray objects in the “after merging” diagram).

I believe that main-thread merging is the best option if partial cache hits are somewhat rare, because ease of maintenance outweighs the runtime costs. Furthermore, I suspect that partial cache hits are somewhat rare, based on the fact that nobody else has bothered to implement this feature before now. Perhaps we could start by getting some real-world telemetry on the partial cache hit rate (here is a CL to do so).

Option 2: Find slots on a background thread, then merge on the main thread

We could substantially cut down on the main thread merging time by splitting the merge algorithm into two parts: first, traverse the new object graph to find any slots that might need updating (anything that points to a Script or SharedFunctionInfo), and second, perform the merge iterating only those slots instead of every slot in the new object graph. This approach allows the object graph traversal to happen on a background thread, though putting that work in the background requires more GC awareness (handles, safepoints, rehashing the progress table after GC, etc.).

The tricky part is determining whether it is worthwhile to iterate the objects and find slots on the background thread, since performing the iteration on every background script compilation would be a waste of time. A background parsing task has the full source text by the time it completes, and thus could check the Isolate script cache in a careful thread-safe manner, but a background deserialization task never receives the full source text. So without further modifications, this option applies only to background parsing.

It is possible, with unlucky timing, that the main thread finalization logic finds a cached script even though the background thread did not, so occasionally the main thread will need to perform the object iteration.

Option 2a: Provide the full source text to a background deserialization task

I'm unsure of the specifics here, but perhaps we could extend the API and allow the embedder to provide the source text to a background deserialization task once it's available. Then the approach described above would work equally well for both background parsing and background deserialization.

Option 2b: allow Isolate script cache lookups by URL

The Isolate script cache allows reuse of scripts only when it is correct to do so: the source text is identical, and the script was loaded from the same origin with the same options. However, a new script loaded from the same URL as a previous script will *probably* have the same source text as the previous script. What if the Isolate script cache contained a second table where it was possible to look up scripts by just origin and options, not source text? I believe the embedder could easily provide this data when creating the background task. V8 could do a lookup in this second table when creating a background parsing or deserialization task, and use its result to guide what work the background task does. If the table indicates that the Script exists but doesn't have a compiled top-level SharedFunctionInfo, then the background thread should do the extra work to find slots in preparation for a merge.

This option allows another tangential improvement: if the table indicates that a compiled top-level SharedFunctionInfo is already available when creating a background task, then the background task should do nothing because any work will likely be thrown away.

Option 3: Track slots as they're created, then merge on the main thread

Instead of traversing the new object graph after parsing or deserialization is complete to find the slots that point to SharedFunctionInfos or the Script, we could build that list of slots as we build the objects. This change would be somewhat intrusive in both the code cache deserializer and the parser, but would probably be faster than a separate graph traversal. A background parsing

or deserialization task would need to know ahead of time whether to collect this list, so the imprecise script cache by URL as discussed in option 2b is probably a prerequisite. To mitigate the increased possibility of bugs, it might be valuable to extend the merge algorithm to replace the removable objects with FreeSpace; this way, the heap verifier could notice if something is still pointing to those objects.

— I'm pretty sure everything below this line is a bad idea —

Option 4: Perform the entire merge on a background thread

With the right locks in the right places, or a lot of deep thought, perhaps we could come up with a way to perform the merge on a background thread. This sounds very difficult and error-prone.

Option 5: ThinSharedFunctionInfo

Like ThinString, but for SharedFunctionInfo. This would speed up the merge at the cost of adding complexity to all the rest of V8.