Web Snapshots 0.1

marja@ - March 2021 - Attention: Shared Google-externally

Info about web snapshots

Orig doc: Browser-independent Web Snapshots

Version 0.1

What would be the minimal thing which we can run and measure?

Example (based on https://github.com/corbanbrook/dsp.js/blob/master/dsp.js.): Below we show the JS equivalent of what will be in the snapshot:

```
// Object with integer-valued properties
var DSP = {
  LEFT:           0,
  RIGHT:          1,
};

// Properties with function values
DSP.invert = function(buffer) { ... };

// Creating a function with an IIFE - the end result is an inner
// function + context
DSP.deinterleave = (function() {
  var left, right, mix, deinterleaveChannel = [];
  ...
  return function(channel, buffer) {
    left; right; mix; deinterleaveChannel;
  };
}());

// Plain functions
function FourierTransform(bufferSize, sampleRate) { ... }

// Constructor-like functions with properties in the .prototype
function DFT(bufferSize, sampleRate) { ... }
DFT.prototype.forward = function(buffer) {  ... };

// Properties directly in the function object and in the .prototype
```

```
function Oscillator(type, frequency, amplitude, bufferSize,
sampleRate) { ... }
Oscillator.prototype.setAmp = function(amplitude) { ... }
Oscillator.Sine = function(step) { ... }
```

Features needed:

Types
- Minimal type system for expressing: object, function, string, number, oddballs (undefined, true, false)

Objects
- Keys: only strings
- Values: primitives, objects (belonging to the snapshot), functions (belonging to the snapshot)
    - Not supported: built-in objects like {‘a’: Array.prototype}
- Cycles allowed

Maps
- Objects can share a map (but then the objects must contain the same type of value in the location; this can be relaxed in a later version)
- Deserializing doesn’t create map transitions

Functions
- Functions as properties: o.a = function() { … }
- Top-level functions and inner functions
    - Minimal ad-hoc context representation for inner functions
- Function source as string
- .prototype
- Functions can contain properties too, like objects
- No classes
- No methods (even in object literals)
- No parsing (no even preparsing) when they’re deserialized; parsed only when called. We create lazy functions when we deserialize.

Contexts
- Expressed as a list of “variable name, variable value” pairs.
- One parent context
    - Open Q: to allow optimizing contexts and avoid false sharing (a function doesn’t need a variable to be context-allocated but it’s context-allowed because of something else), do we need to allow multiple parent contexts in the future? Does any JS engine need them?

Other features:

- Ad-hoc (but as minimal as possible) binary snapshot format
  - Will most probably be changed later
  - Using the existing Serializer / Deserializer code for simple parts (such as writing numbers), but not using the same format per se. (The format is probably impossible to sell to TC39.)
  - Refactoring the serializers in such a way that maximises code sharing between Web Snapshots and the existing serializers is a TODO for later.
- A d8-based tool for creating the snapshot
  - Configuration file which tells what to include in the snapshot
  - Warnings for (the most obvious) unsupported cases (but we probably can't warn about everything)
- Deserializing the snapshot in d8

d8 usage example

Creating a snapshot can be initiated through the --web-snapshot-config flag, which expects a relevant configuration file. This file currently holds the list of the exports.
Once a snapshot is created, it can be used through the --web-snapshot flag in d8.

```
$ cat produce-snapshot.js
var foo = {};
foo.bar = function() { print('hi'); };

$ cat web-snapshot.conf
foo

$ out/x64.optdebug/d8 produce-snapshot.js
--web-snapshot-config=web-snapshot.conf

$ cat consume-snapshot.js
foo.bar();

$ out/x64.optdebug/d8 --web-snapshot web.snap consume-snapshot.js
--allow-natives-syntax
hi
```

Later versions (immediate future)

- Decide how to handle strictness

- Should we implicitly assume the snapshot source is strict? Or encode strictness?
    - Forbid non-simple parameter lists?
- Property attributes (writable, enumerable, configurable)
- Arrays
- Modules
    - Being able to use module imports?
- Classes incl. methods
    - Home object needs special handling
- Support more function types
- __proto__
- Design the format to be less ad-hoc
    - “Binary format best practices”
    - Maybe take inspiration from WebAssembly (sections, required sections etc)
- Setup code included in the snapshot (for post-processing the objects created from the snapshot) (if needed)
    - Per-object reviver?
- Blink integration
    - Code cache integration
- mjsunit integration in a way that is fuzzable. Maybe with Realms?
- Machinery for using web snapshot as a JSON replacement (allow web snapshots in the APIs which are used for requesting / processing data).
- Support isolate cache (deserializing the same snapshot in the same Isolate multiple times should give back the same SFIs)
- Lazy deserialization (TBD)
    - Example: special syntax for functions coming from a snapshot: if(foo) { obj = %deserialize(#1) } else { obj = %deserialize(#2) }
- TODO: pre-allocate the memory when deserializnig

Later versions (later)

- References to built-in objects (does anybody need them?)
    - Subclassing built-ins
- Do we need cons strings?
- References to things in other scripts
    - Do we use the revivers for setting them up?
- DOM nodes
    - For this, we’d need to snapshot in Chrome?
    - An explicit way to express that an object in the snapshot points to a DOM node and that connection will be set up during deserializing?
    -
- Extension interface for the snapshot format
    - DOM as an extension?
- Dev tools for creating a snapshot?
- Dev tools integration

    - Displaying objects / functions coming from the snapshot even though there's no actual source code
- Typed annotations in the maps for 32bit smi and unboxed doubles?
- Limits of the "inspecting with d8" approach; maybe we need a different tool? Or at least some annotations to help
    - We can't figure out a function was imported from another module after the fact; foo.bar = <func_imported_from_module> will look just like any function.
    - How far can we get with the d8 inspecting approach + using annotations + using revivers?
- Once the format is more stable, we could add a json based tool for converting to and from the snapshot.
    - Snapshot-to-json, useful for displaying a human-readable version of the snapshot when developing
    - Json-to-snapshot, useful for constructing the snapshot when testing
- Streaming the snapshot, possible other multithreading improvements
- Web API for taking a snapshot and storing it in the cache
- "Compile this function on the background" hint for the functions

More complicated example:
https://gist.github.com/developit/049887ec8e18cd1b5ca61bccae83f5e3

Design (prototype)

The following design principles will help to make fast progress towards the MVP:
- Serialization:
    - Doesn't need to be particularly efficient, since the serialization tool is not on the critical path. (But let's of course not do anything overly silly.)
- Deserialization:
    - Deserialization speed matters.
    - "Keep on trucking" approach to error handling to reduce branches.
        - If the snapshot is invalid, set an error state and proceed with the deserialization. Don't check the error state repeatedly but only at the end.
        - Caveat: need to ensure we don't crash.
        - Caveat: need to check the error state before we produce anything observable.
        - Security-wise, it's not any worse than handling the errors early: We cannot trust the data anyway. Every upcoming byte can have an arbitrary value, not depending on whether or not we've encountered an error before.

V8 specific quirks

- 1-to-1 correspondence between prototypes and maps: two objects can't share a map if at least one of them is a prototype. The snapshot might contain objects which violate this invariant, and we need to deal with it at deserialization time.
- We'd need to create map transition trees (even if the snapshot only contains the final map) and connect them to the outside maps
  - Map deprecation (a field goes from SMI to double)
  - Slack tracking
  - If we deserialize two snapshots, we need to get the same map for two objects if they're coming from the same constructor.
- De-duping SharedFunctionInfos
  - We probably don't want to force adding the "number of inner functions" data element in the snapshot, unless it's useful for other browsers too.

More information about field representation tracking:
https://docs.google.com/document/d/10N7EYJeGWAlSROrQYH3m5qIPX8VjhBPU5--7uu6FMMo/edit?usp=sharing

Organizing and deduplicating SharedFunctionInfos

Problem:

```
// Snapshot (exported = [foo])
let foo = {};
foo.outer = function(a) {
  return function() {
    return a;
  }
}
foo.inner = foo.outer('hi');

// After deserializing:
const inner2 = foo.outer();
// Now we expect the SFI for inner2 to be the same as the SFI
// for foo.inner.
```

Additionally, V8 numbers SFIs in textual order. Many parts rely on finding the functions based on their SFI order number and also rely on the inner functions' SFIs being numbered immediately after the parent.

This is so that even if the bytecode of a function is discarded, when we compile it again, we get the same SFIs for the inner functions (if they're still around) as the previous time.

Alternative SFI numbering solutions:

1. Require the “how many inner funcs” information in the snapshot. Not great, since it’s a V8-specific quirk. (Not sure if other browsers need it.)
2. Preparse functions when deserializing the snapshot. Not great, since we don’t want to spend that time for no good reason (internal V8 compilations are not a good reason). Instead, we should wait until a function is called and at that point we should parse it (never preparse).
3. Relax the numbering so that the inner functions are allowed to be numbered in such a way that their IDs don’t immediately follow the parent’s ID. Add a hash map to Script (start position -> index in the SFI array) for finding the relevant SFI for de-duping.

Suggested solution: 3.

Alternative de-duping solutions:

1. Add some kind of “SFI” to the snapshot data, allowing the snapshot to define that “foo.inner” is actually the same function as “inner” inside “foo.outer”. It’s questionable how this solves the problem.
2. Add function position information (positions in the original script) to the snapshot and de-dupe based on the position. Requires adding the “original start position” to the SFI. (Both it and the “inner function ID start” can be added to a WebSnapshotData object so that we only waste one more slot.)
3. Add a “script snippet” type to the snapshot. A function source code is expressed as (snippet id, position inside the snippet). When deserializing, we can create one big script by concatenating all the snippets, and set SFI positions so that they point to the right functions. De-dupe SFIs based on positions. The right inner function SFIs are magically found.

Suggested solution: 3.

Other engines don’t have the same SFI numbering problems as V8. SpiderMonkey doesn’t discard the bytecode of functions which have inner functions, so it doesn’t need to deduplicate the inner function “SharedFunctionInfos” after recompilation.

FIXME: jsc?

Another related problem: same function, different context. We need something in the snapshot for expressing that - otherwise we’ll just duplicate the code.

```
// Snapshot (exported = [foo])
function func_generator(p) {
  return function() { return p; }
}
```

```
let foo = {};
foo.func1 = func_generator('hello');
foo.func2 = func_generator('world');
```

Solution: Separate Function (source code string + original position) and FunctionInstance (Function + Context).

Background deserialization

TBD

See also Off-thread compilation finalization and Allocation during deserialization.

Thoughts

Serializing the full Realm vs. serializing the non-builtin parts

There are two design choices here:

1. Serialize a whole realm, including all things reachable from its global scope like all the intrinsics.
   - Bigger snapshot size.

2. Serialize a subset of non-builtin things. A snapshot deserializes into an existing global and doesn't create its own.
   - More freedom to the developer, e.g., setting up a realm ahead of time (potentially containing things that are hard to serialize), loading multiple snapshots into a single realm.
   - Smaller snapshot size
   - Snapshot behaves the same way than loading plain JS
   - Need to think of a solution for the snapshot to point to intrinsics (or disallow it)

Going with choice 2.

"Corresponding JS"

Invariant: every snapshot should have a “corresponding js” which more or less “does the same” than deserializing the snapshot.

This way we could get an automatic fallback to the “corresponding js”:

```
<script src=”fallback.js” snapshot=”snapshot.ws”></script>
```

(Edit: Counterpoint: apparently getting people to update their script tags (which load 3rd party content) is painful. We'd like to do this transparently via a Content-access header, so that the server can transparently send a web snapshot file instead of a JS file for browsers that support it.)

But let’s not verify correspondence to the “corresponding js”. 1) We don’t want the browser to download it just for the purposes of verification. 2) We want the snapshot to contain the minimal amount of information, and not add anything just for the purposes of verification.

Example:

```
let foo = {};
foo.f = function() {
  let a = 0;
  function inner1() { return a; }
  function inner2() { ++a; }
  return [inner1, inner2];
}
[foo.inner1, foo.inner2] = foo.f();
```

What if the user now gives us the following snapshot, claiming that it corresponds to that js:

c1 = a context with variable ‘a’, value 0
c2 = a context with variable ‘a’, value 0
foo.f = function with this source code: `() {`
```
  let a = 0;
  function inner1() { return a; }
  function inner2() { ++a; }
  return [inner1, inner2];
}
```
foo.inner1 = an inner function with source code which is a substring of the previous source code, context = c1
foo.inner2 = an inner function with source code which is a substring of the previous source code, context = c2

That’d break the “validity” of the contexts; foo.inner1 and foo.inner2 would no longer refer to the same context and access the same variable.

But that is fine! We only know about the discrepancy because we see that foo.inner1's source code is inside foo.f's source code and foo.inner2's source code likewise, and that we only know because foo.f is included in the snapshot.

But consider this:

```
let foo = {};
function func_generator() {
  let a = 0;
  function inner1() { return a; }
  function inner2() { ++a; }
  return [inner1, inner2];
}
[foo.inner1, foo.inner2] = func_generator();
```

Now func_generator wouldn't be included in the snapshot, so we'd only see this:

foo.inner1 = an inner function with source code `() { return a; }`, context = ...
foo.inner2 = an inner function with source code `() { ++a; }`, context = ...

And we wouldn't have any way to know that foo.inner1 and foo.inner2 came from the same outer function, thus no way to verify whether the contexts are set up correctly or not.

This means that the "source is a substring of another source string" optimization can be used even if the functions are not related. For example:

```
let foo = {};
foo.f = function() {
  let a = 0;
  function inner() { return a; }
  return inner;
}
foo.inner1 = foo.f1();
function func_generator() {
  let a = 0;
  function inner() { return a; }
  return inner;
}
foo.inner2 = func_generator;
```

Now inner1's source code and inner2's source code can be both represented as substrings of f's source code, even though inner2 is not an inner function of f.

In addition, the snapshot might contain a more efficient context chain than what a real JS VM would create if it runs the code, and we should allow that.

Example:

```
function outer() {
  let a = <something big>;
  let b = <something big>;
  let c = <something big>;
  function f1() { a; c; }
  function f2() { b; c; }
  return [f1, f2];
}

// Contexts created by V8:
// c0: a, b, c
// f1 -> c0, f2 -> c0

// More efficient:
// c0: c
// c1: a
// c2: b
// f1 -> c1 -> c0
// f2 -> c2 -> c0
```

Versioning & backwards compatibility

Historically, selling anything versioned to TC39 has been difficult (impossible?). Can we come up with a format which maintains backwards compatibility - even if we modify the snapshot format, 1) every old snapshot is also a valid new snapshot 2) disregarding the added information doesn't make the interpretation incorrect?

Ideally, we should decide whether to download the snapshot or the corresponding JS upfront. Let's try to avoid the design where we first download the snapshot, figure out we cannot handle it, and download the corresponding JS instead.

Versioning & backwards compatibility in JavaScript:

- The website author is responsible for not shipping "too new" JavaScript to the browser
- Old features are supported forever by the browser
- The language generally accepts only backwards compatible changes: every valid JavaScript file will be valid forever

Versioning in WebAssembly: The header of a Wasm file contains a version number. But there's only one version (1.0) with no plans to switch to a newer version, and also no plans regarding how many different versions a browser would support.

However, as the Wasm format is section-based, it's possible to do backwards compatible changes without changing the version number. E.g., if the function section would need to be extended in a backwards-incompatible way, there could now be two sections, "function_section" and "function_section _new". Interestingly, the version number doesn't play a role here.

Versioning & backwards compatibility in Wasm:

- The website author is responsible for not shipping "too new" Wasm files to the browser
- Old features are supported forever by the browser
- The language generally accepts only backwards compatible changes: every valid Wasm file will be valid forever

---

Backwards compatibility & extensible formats: Will there ever be a situation where we could safely process only a subset of the data in a web snapshot and ignore the rest? That doesn't seem likely. It would only make sense if there "additional data" is metadata (e.g., for Dev Tools).

Lazy deserialization

Basic level (supporting what JavaScript currently supports):

- Functions are shipped as source text (which we don't even preparse at snapshot deserialization time)
- We create a JSFunction (and a Context if needed) to hold all the data we need

Laziness case 1: lazy object graphs

- Support deserializing objects lazily too; dereferencing such an object will trigger deserialization of the object graph
- Lifetime problems: we might deserialize an object and then lose the reference to it, even though something to be deserialized later would've needed the reference.

```
// Snapshot contents
let exported_lazy_object1 = { a: <obj_in_snapshot> };
let exported_lazy_object2 = { a: <obj_in_snapshot> };

// After deserializing:

// Access exported_lazy_object1. This causes deserialization of the
// subgraph, in particular <obj_in_snapshot>
exported_lazy_object1.foo();

// Get rid of references to <obj_in_snapshot>
exported_lazy_object1 = null; // or
exported_lazy_object1.a = null;

// Access exported_lazy_object2
exported_lazy_object2.foo();

// Expectation: we kept <obj_in_snapshot> alive the whole time.
(It's
// not sufficient to deserialize it again at this point, since it
// might have been modified, and those modifications need to be
// visible.
```

    - Use case: module internal data which is referred by 2 lazy exports; it will get created when the first lazy export is deserialized, and after that it has to be kept alive until the second lazy export has been deserialized.
    - Potential solution: scan all lazy parts beforehand to discover references, for each object in the snapshot, track how many lazy clusters refer to it. If there are unserialized lazy clusters referring to it, the object must be kept alive.
    - Alternative to scanning: Lazy clusters could tell which objects (from the main table or lazy tables) they refer to, so that we can keep them alive.
- The object references can also come from function contexts.
- We might allow the user to specify for each reference, whether the object should be deserialized eagerly or lazily. Or, only add the laziness hint to top-level objects. Or, add no hints but decide where the laziness boundary is according to some heuristic.
    - Users might do the wrong thing with the hints and we might need to ignore them anyway.
    - Making every link lazy is too much work, assuming we'll need to call into runtime at the lazy boundary. If we implement deserialization in CSA, it might become feasible.
    - What do other browsers want?

Laziness case 2: functions declaring deserialized members

- Lazy functions can declare that some of its local variables (incl inner functions) should be deserialized from the snapshot
- We need to deserialize before parsing the function code (so that references to those variables work properly). (At least we need to know the variable names at parse time.)
- We need to re-create them every time the function is called.
    - At least objects from literals, and contexts for inner functions.

Naming

syg@ pointed out anything which contains the word “snapshot” might not be a good name. Portable Object Graphs (POGs)?

Fuzzing

Fuzzing the snapshot

The snapshot is a binary format and it’s security critical that we process it in a robust way. We need a fuzzer that generates snapshots which V8 then consumes.

Especially invalid snapshots (including “almost valid” snapshots) and snapshots that are valid but which our tools wouldn’t produce are interesting.

Fuzzing the objects produced from the snapshot

We’d also like to verify the objects we produce based on the snapshot are “sound” and meet the (often unwritten) V8-specific invariants.

To this end, we can have some JS mechanism for defining and reading in a snapshot, and then we’d just use that with the standard JS-mutating fuzzers, to use the objects read from a snapshot in various ways (pass them to functions etc) to detect potential inconsistencies.

Correctness fuzzing

We could use the correctness fuzzing for comparing the deserialized snapshot to the “corresponding JS”.

Invariant: Deserializing the snapshot + <test js code> produces the same result as loading the snapshot’s “corresponding JS” + <test js code>. (Here the “test js code” will be fuzzed.)

Concurrency

In later versions, we can…

- Deserialize the snapshot off-thread in a streaming manner
- Compile functions in the snapshot lazily off-thread

Streaming deserialization

- Unlike normal parsing, the web snapshot data format is not (necessarily) deeply nested. We could create small deserialization tasks which could be executed by the main thread or by any background thread (though, only one at a time) and we could yield between tasks.
- Thus, the streaming deserialization architecture doesn't need to be similar to streaming parsing (where the limiting factor is that we cannot give up the thread when we run out of data, since we might be deep in parsing some nested structure).
- We could even split the snapshot into items (objects etc) in advance (if we design the format so that it's easy to know the size of an item just after starting to read it) and have multiple threads work on the items.
    - String table likely to become a bottleneck in that design

Integration to Blink

How to integrate with code caching? Caching anything else related to the deserialized snapshot might be hard, but we could cache the code of the functions in the snapshot. For that we'd need to extend the code cache to be able to cache individual functions (unless already done?).

DX

How would the debugging experience for objects / functions from the snapshot look like?

Dev tools could download the "corresponding js", but it might be non-trivial to map it to the snapshot.

Functions are expressed as source text, so at the minimum, Dev Tools should be able to display the function source code and do the normal actions (such as setting breakpoints).

Possible complication: the object structure V8 generates based on the snapshot might not match what V8 would create based on plain JS (the exact differences still TDB). This might complicate debugging.

Measuring performance

We do parts of parsing / compilation on background thread. JS execution is always on the main thread. Also parts of the snapshot deserialization can be on the background thread.

A (pessimistic) measurement would be: Snapshot deserialization time vs. the corresponding JS execution time (not taking into account parsing / compiling). If we can prove good performance in that measurement, that guarantees good performance overall.

Investigations

- Run real world web pages to measure which amount of code is “startup code”. (Code that is run only once slightly overapproximates it but probably good enough).
- Investigate type systems in other compilers; what can we encode in the map?

Spec side of things

- Lazy error reporting
    - We don’t want to preparse functions upfront, since there’s no need to. The spec would need to allow thoroughly lazy error reporting (only when the function is called, we’ll parse it).
- In general, we’d like to do as little work as possible regarding “verifying the snapshot”.

Prior art

Differences to Structured serialization

- Web Snapshot supports functions and contexts.
- The format for Web Snapshots is not internal to each browser, but standardized.

That said, Structured Clone could be implemented in terms of Web Snapshots (but Web Snapshots can’t be implemented with the current implementation of Structured Clone).

Minor differences:

- Implementation detail: Web Snapshot supports encoding maps; thus encoding a set of objects with the same map is particularly efficient.

Q: Can’t we just augment Structured Serialization?

A1: The Web Platform most probably doesn’t want to add functions to it.
A2: Currently, the format for Structured Serialization is not standardized. Web Platform most probably doesn’t want to standardize it.

That said, Web Snapshots can be seen as a “Standardized format for Structured Serialization augmented with functions and contexts” - in a very hand-wavy way.

FIXME: expand this section

Structured clone

Startup snapshots in V8

Stencil / Mozilla
https://searchfox.org/mozilla-central/source/js/src/frontend/Stencil.h#74

Web Shared Libraries

Binary AST

Use cases

FB modules

(not ES6 modules but self-baked ones)

Example 1:

```
__d("MyModule1", [], (function(a,b,c,d,e) {
  "use strict";
  a = { // Huge object literal
      prop1: [
        {
          prop2: ...,
          prop3: ...,
        },
        ...
      ]
    };
  };
  e.exports = a;
}));
// We don't actually call the parenthesized function
```

Example 2:

```
__d("MyModule2", [], (function(a,b,c,d,e) {
  "use strict";
  a = function() {
    var a = [{
      prop1: "...",
      prop2: "...",
    }];
    return { // Huge object literal
      prop1: [
        {
          prop2: ...,
          prop3: ...,
        },
        ...
      ]
    };
  }(); // a is now that huge object
  e.exports = a;
}));
// We don't actually call the parenthesized function
```

How to support this with web snapshots?

- We don't want to create the huge object literal eagerly; it should be in the web snapshot but deserialized lazily if the function is called.