JS SELF-PROFILING

Design overview and discussion @ TPAC 2018

acomminos@fb.com

OVERVIEW

WHAT

- A sampling profiler for client JS
- Introduces logical sampling profiler creation to `Performance` interface

```
[Exposed=(Window,Worker)]
partial interface Performance {
  Promise<Profiler> profile(ProfilerInitOptions options);
};
```

WHY

- Effectively find JS hotspots
- Enables production performance measurement
  - Different browsers, CPUs, network conditions result in drastically different execution
- Reduces instrumentation overhead
  - Measure execution time without breaking caches

USE CASES

- Complex web apps
  - Quantitative (aggregation) and qualitative (trace analysis for one-offs)
- A drop-in analytics library for instrumenting performance

DESIGN GOALS

- Expose very little information that isn't already polyfillable
- Grant the UA significant control over sampling behaviour
- Output trace objects in a structurally compressed format for efficient transmission

API OVERVIEW

IDIOMS

- JS may create "logical profilers"
  - Multiple per context supported, backed by the same profiling thread
  - Each has bounded buffer size
- Each profiler may select a set of categories to be traced (currently only JS)

TRACING A PAGE LOAD

```javascript
const profiler = await performance.profile({
  categories: ['js'],     // only 'js' supported in this spec
  sampleInterval: 5,      // milliseconds
  sampleBufferSize: 1024, // max # of sample objects
});

document.addEventListener('load', async () => {
  const trace = await profiler.stop();
  sendPayloadToTheBoss(JSON.stringify(trace));
});

// do work
```

CLIENT-SIDE TRACE TRAVERSING

```
let mathOMeter = 0;
for (const sample in trace.samples) {
  const stack = sample.getStack();
  for (const frame in stack.getFrames()) {
    if (frame.label.contains('Math')) {
      mathOMeter++;
    }
  }
}
```

TRACE FORMAT

```
{
  "frames": [{
    "category": "js",
    "label": "breakStuff @ hammer.js:140:5",
  }],
  "stacks": [{
    "parentId": null,
    "frameId": 0,
  }],
  "samples": [{
    "timestamp": 2718281828,
    "stackId": 0,
  }],
}
```

- Look familiar?
- Popular trie format used by Gecko and V8

PRIVACY AND SECURITY

SECURITY BY EQUIVALENCE

- Design aims to minimize data that cannot be exposed by an implementable polyfill
  - Equivalent concerns to `performance.now()`
  - Can be fuzzed appopriately
- New wants
  - Stack frames from third-party scripts

POLYFILL: OVERVIEW

Let "frames" refer to client JS stack frames.

`SlowProfiler` maintains a call stack in a SharedArrayBuffer.

The stack maintained by `SlowProfiler` is sampled periodically using a Web Worker.

POLYFILL: STACK OPERATIONS

Used to fetch stacks from instrumented functions.

- `SlowProfiler.beginFrame()` pushes the current call stack to the SAB
  - Backed by `(new Error()).stack`
- `SlowProfiler.endFrame()` pops the top call stack from the SAB

POLYFILL: FRAME OPERATIONS

These operate on the SlowProfiler's active call stack.

- `SlowProfiler.push(label: string)` adds a call frame to the SAB's top stack
- `SlowProfiler.pop()` removes the last call frame from the SAB's top stack

TACTIC: FIRST-PARTY JS AND ANCESTORS

First party JS frames can capture both 1P and 3P ancestors via instrumentation

```
function foo() {
  try {
    SlowProfiler.beginFrame();
    // body
  } finally {
    SlowProfiler.endFrame();
  }
}
```

TACTIC: IMMEDIATE CHILDREN

Top-level DOM and 3P JS entrypoints can be wrapped

```
SlowProfiler.push('Math.pow');
Math.pow(20, 1349);
SlowProfiler.pop();
```

THE MISSING LINK

Cannot capture 3P frames that are neither:

- An ancestor of a 1P frame
- An immediate descendant of a 1P frame

ENABLING CROSS-ORIGIN FRAMES

1. Require third party scripts to provide CORS header
    - Already done to provide stacks in `onerror` via `<script src="..." crossorigin>` attribute
2. Require third party scripts to provide a `Profiling-Allow-Origin` header
    - Offers more fine grained control over the level of information provided
    - Less likely for 3P scripts to provide user data embedded in stack frame labels

HIDING NON-CORS FRAMES

What if a script did not grant the appropriate header?

1. Fuzz with a "unknown" category
    - Cluster consecutive foreign frames in stacks
        - From any origin (foreign frame callees should be opaque)
        - Hide size of foreign substacks
2. Omit the frame entirely

IMPLEMENTABILITY

CONCERNS

- Implementations (if they already support profiling) have variable performance overhead
  - Need control over sample rate
- New timing source, similar concerns as HR-TIME

SAMPLE TIMING

- UA decides on a set of supported sample rates
  - User can request a rate, best-effort matching
  - Even low sample rates can be useful in aggregate
- UA takes best-effort approach to sampling
  - No SLA for sample delivery

VM FLEXIBILITY

- Optimizing out frames permitted
  - Minimal impact to trace analysis / attribution, will mostly be simple side-effect free functions removed
- Async start/stop to accommodate cost of spinning up and processing trace

OPEN QUESTIONS

CALLEES OF FOREIGN 3P FRAMES

- If a non-CORS 3P script calls a 1P script, should we include 1P frame data in the trace?
- Already exposed (see polyfill)
- May make traces difficult to reason about

SAMPLE BUFFERING

- Sample count based limiting
  - Client requests max sample count, clamped by implementation
  - `onsamplebufferfull` event fired once sample limit reached
- Is it enough?

CATEGORIES

- Spec defines the "js" frame category
- Any interest in sampling other categories in the future?
  - e.g. layout, rendering
  - Can share attribution types with long tasks?
- If not, should we kill?

MARKER SUPPORT

- Desirable for `Trace.samples` to store markers?
  - e.g. "GC" marker embedded in trace
- Uses same time space as `performance.mark`, may be redundant

STACK FORMAT

- Worth standardizing stack frame format?
- Pros
  - Aggregation is more straightforward
  - Makes WPT easier
- Cons
  - Spec coupling

THANK YOU!