DOM Profiling in Blink/V8

Overview

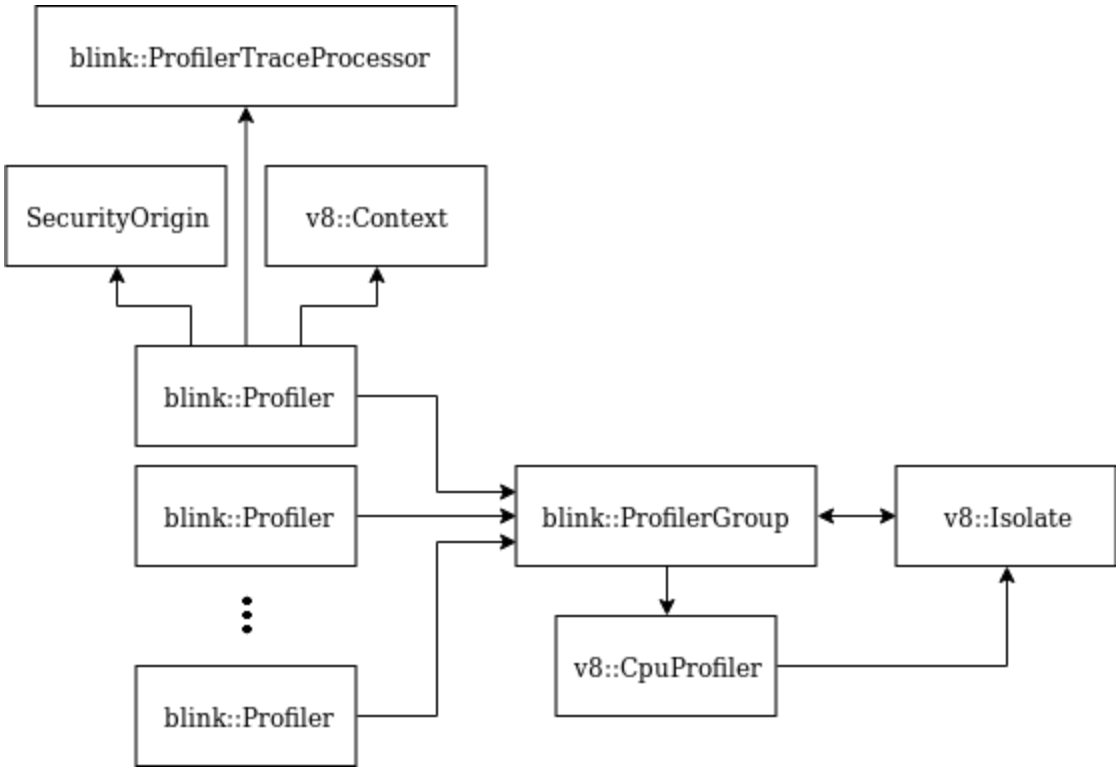
We propose the implementation of a sampling profiler exposed to JavaScript that meets the security and performance concerns of the web, while providing new insight into JS execution characteristics in the wild.

Explainer
Spec

Design principles

- Ensure compliance with the web's security model: do not expose any new cross-origin data or implementation details
- Attempt to leverage existing v8::CpuProfiler infrastructure and extend it in a generic way suitable for a variety of consumers (such as node's profiling)
- Avoid significant CPU and memory overhead from well-intentioned API usage

Overview



The blink::Profiler interface

A blink::Profiler is 1:1 with the concept of a profiling session, as defined in the JS Self-Profiling API spec. Each blink::Profiler contains:

- A v8::Context from the invoker
- A security origin from the invoker
- A sample interval
- A sample buffer capacity
- A "sample-buffer filled" callback
- A handle from v8::CpuProfiler::StartProfiling

Through the DOM bindings, users provide a requested sample interval, buffer capacity, and callback. This sample rate is transformed by the profiler group to satisfy sampling constraints (as described below).

The blink::ProfilerGroup interface

A blink::ProfilerGroup is a per-isolate construct that manages a set of blink::Profilers that share a common sampling thread / v8::CpuProfiler. It is responsible for choosing an appropriate sample interval that can accomodate all of its dependent blink::Profiler instances, as well as spinning up and stopping blink::Profilers.

For the initial implementation, we propose requiring that profiler groups enforce a **base sample interval of ~20ms** for their profilers. To ensure that all profilers in a group use only a single sampling thread, we plan on **rounding up all requested sample intervals for profilers to multiples of the base value**. This will allow us to remove extraneous frames with ease (see "dynamic subsampling" for v8::CpuProfiler below), and upsample appropriately to accommodate multiple concurrent blink::Profiler instances.

On Windows, the base sample interval may be chosen to accommodate the periodicity standard-resolution timer.

Trace processing

When blink::Profiler::stop is called or the sample buffer is filled, the resulting trace (as a v8::CpuProfileNode) is dispatched for processing. This logic is performed by blink::ProfilerTraceProcessor, which produces the ProfilerTrace object described in the spec. blink::ProfilerTraceProcessor is also responsible for performing the necessary origin-based filtering on all frames, a part of Appendix A.

This result object is either returned directly if the default trace format is specified, or serialized and returned as a GZIP blob in an ArrayBuffer if the "json-gzip" format is specified.

Extensions to v8::CpuProfiler

In order to efficiently satisfy the privacy, security, and potential memory concerns of the profiler, certain changes to v8::CpuProfiler and its accompanying implementation classes will be required.

Stack filtering

There are two types of stack frame filtering that our implementation needs (see Appendix A):

1. Filtering stack frames from contexts other than the blink::Profiler's instantiated context (e.g. other blink frames or content scripts from other worlds)
2. Filtering stack frames from scripts that are from a different origin than the blink::Profiler's instantiated origin, and do not have the crossorigin bit set (a la Error.stack)

For case 1, we suggest recording the v8::NativeContext (which maps to the incumbent context) of each stack frame object obtained during stack unwinding, and having each CpuProfile accept an optional context parameter to filter by. During dispatch of a sample to a CpuProfile, stack frames whose context is not equal to the profiler-provided value will be discarded. This check should be relatively inexpensive, and save on the amount of post-processing work required. Details here.

For case 2, we require public API exposure of the crossorigin bit associated with the script's v8::ScriptOriginOptions, as well as the script's location from which to derive the security origin of (which is already present). We think that origin-based filtering makes sense in blink::ProfilerTraceProcessor as a post-processing step, as V8 should not concern itself with Blink's notion of security origins.

Dynamic subsampling

In order to leverage multiple profiler instances utilizing the same sampling thread, we propose adding support for subsampling to profilers created by v8::CpuProfiler::StartRecording. That is, given a v8::CpuProfiler that samples at 50ms, we would like to only accumulate samples for some consumers every 100ms - easily accomplished by skipping every second sample.

Let the **target sample interval** be equal to the client's requested sample interval, rounded up to the nearest multiple of the base sample interval.

Let the **common sample interval** be equal to the sample interval currently being used by the sampling thread.

Observe that the GCD (greatest common divisor) of both the common and target sample intervals is a multiple of the base sample interval by the invariant that all blink::Profilers have their sample intervals locked to multiples of the base sample interval. By changing the common sample interval to the aforementioned GCD and configuring all profilers attached to the sampling thread to subsample every (target sample interval)/(common sample interval)th sample, we can effectively multiplex a dynamic range of supported sample intervals.

There are two parts that we propose to implement:

1. Add support for subsampling to individual v8::CpuProfile instances.
    - Each profile handle can be configured to select every n-th sample it obtains from the sampling thread.
2. Add support for pausing and changing the interval of a v8::CpuProfiler's underlying sampling thread.

When the GCD of all profilers in a profiler group changes due to addition or removal of a profiler, the following steps are executed:

- The profiling thread is spun down.
- The sampling interval of the v8::CpuProfiler is adjusted to the new GCD interval.
- All attached profilers are adjusted to subsample accordingly based on the new GCD.
- The profiling thread is spun back up with the new sample interval.

Sample limiting

To prevent OOMs when a profiler is not stopped, each blink::Profiler possesses a "sample buffer capacity" parameter, in addition to a callback when the capacity limit is reached.
We suggest that v8::CpuProfiler be augmented to stop an active profiler and dispatch an callback asynchronously when an associated sample limit is reached. This functionality is likely to prove useful for other implementations, such as node profiling consumers.

Frame formatting

As the specification aims to leverage the ECMA-262 concept of a "function instance name" for each stack frame recorded, minimal changes should be necessary in order to record a normative label for each stack frame. We intend on implementing a variant of ProfilerListener to record the normative name rather than the debug name for the web-exposed profiler, and triage any cases where this does not align with the ECMA-262 spec.

Additionally, we'd like to surface whether or not a CpuProfileNode is considered to contain V8 metadata (e.g. "(idle)" or "(gc)"), in which case the blink logic should filter these out. This can be accomplished by attaching an enumerated value to each CpuProfileNode indicating the type of frame captured (e.g. js/wasm/gc/idle/program/native).

Reduce CPU usage of sampling on Windows < 100ms

Currently, sampling with an interval less than 100ms on Windows is done by busy-waiting on the profiler thread. We aim to avoid these wasted cycles by adding support for sampling aligned to the tick frequency of Windows' clock interrupt. This will be controlled by a parameter, as this precision should be retained for the DevTools profiler, etc.

Source

Appendix A: Security model

The scope of a blink::Profiler is fixed to the browsing context origin and v8::Context in which it originated in. Sampled stack frames must pass the following algorithm in order to be included in a trace. Stack frames that fail this check will be omitted from the trace.

1. If the stack frame's associated v8::internal::NativeContext is not equal to the profiler's associated v8::internal::NativeContext, return **filter**.
2. Let *stack frame origin* be the origin of the resource that contained the function associated with the frame.
3. If the stack frame origin is not equal to the profiler's origin:
    1. If the script containing the source for the stack frame is a classic script, the crossorigin attribute is set, and the source script passes a CORS check, return **pass**.
    2. If the script containing the source for the stack frame is a module script, and the script passes a CORS check, return **pass**.
    3. Otherwise, return **filter**.
4. Return **pass**.