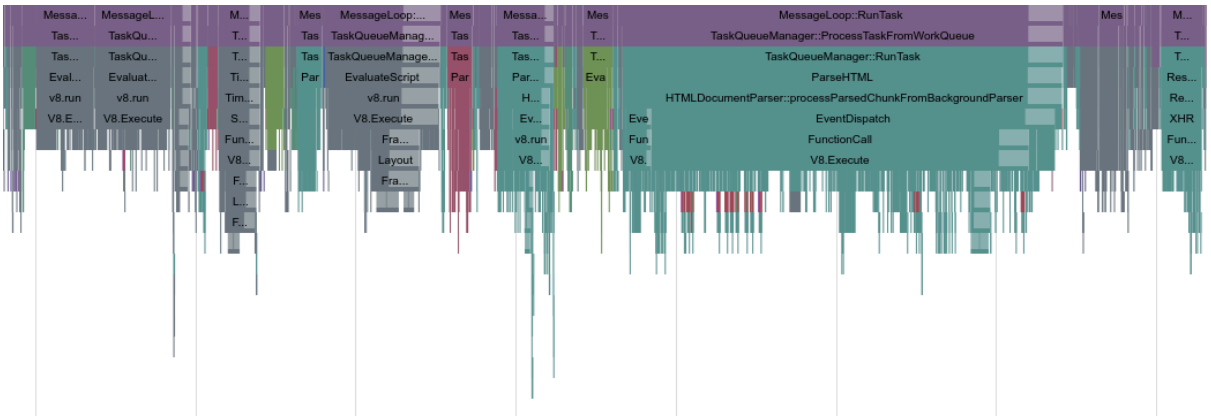
FrameBlamer

skyostil@
December 1st, 2015
Tracking bug: crbug.com/546021



FrameBlamer is about attributing events in the Trace Event model to different *contexts* or logical units of the browser or the web site. This allows us to associate costs such as execution time or memory usage more closely to specific actors in the system. Example contexts include:

- a frame in a web document (the motivating example for this project),
- a window of the browser,
- a browser tab,
- a javascript file,
- an input event, or
- one graphics frame in an animation.

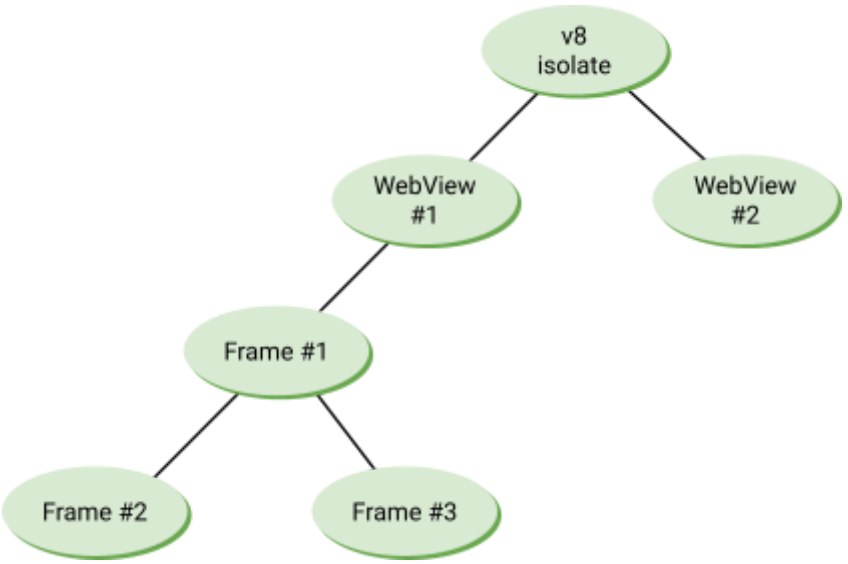
New *context trace events* are used to annotate code doing work for a particular context:

```
// The current thread is now performing work for |context|. Similarly to normal trace
// events, |category_group| defines the category for the context trace event. |name|
// declares the type of the context and allows disambiguating two contexts with the same
// identifier.
TRACE_EVENT_ENTER_CONTEXT(category_group, name, context)

// The current thread is no longer performing work for |context|.
TRACE_EVENT_LEAVE_CONTEXT(category_group, name, context)

// A scoped helper that combines the two macros above.
TRACE_EVENT_SCOPED_CONTEXT(category_group, name, context)
```

The context is uniquely identified within a single process by its type (the |name| parameter) and id (the |context| parameter, an opaque 64-bit identifier – usually the memory address of the object that represents the context).

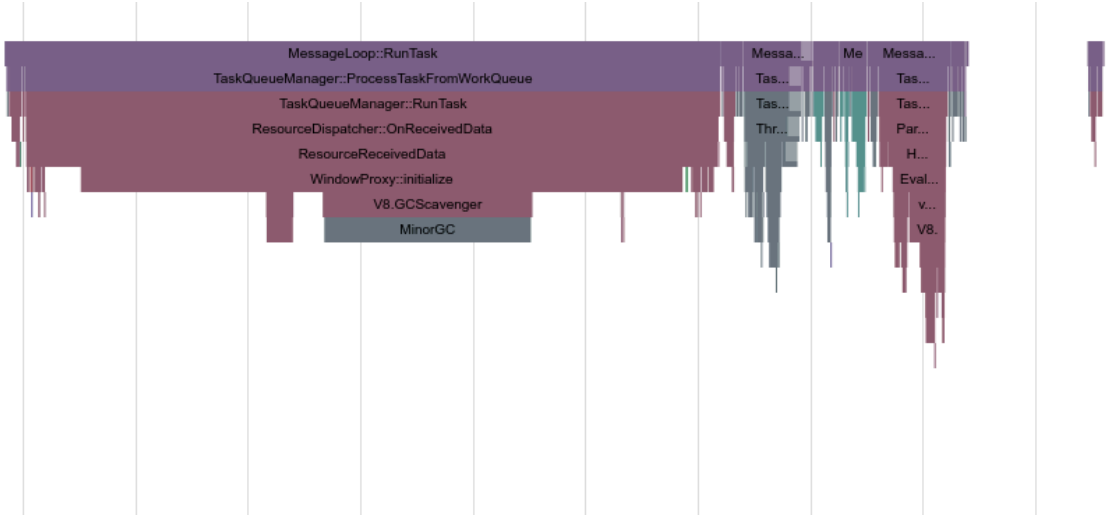
Because it is not always possible to exactly attribute work to a single context, we also introduce the concept of a *context tree*. When executing work that spans multiple contexts belonging to the same tree, we attribute the work as belonging to the common ancestor of each related context.

For example the frame context tree might look like this:



As a practical example, let's assume we are executing a requestAnimationFrame handler belonging to Frame #1 shown above. In the middle of this we need to perform a garbage collection, so we temporarily enter the v8 isolate context for the duration of the GC work and after this return to the context of Frame #1. The reason is that garbage collection may need to touch all objects within the isolate, so it would be unfair to attribute that cost just to Frame #1.

This is illustrated in the figure below where the active context is represented by a color. Note how the thread temporarily activates a different context during the "MinorGC" garbage collection task.



Context selection

There are two operations for selecting the set of active contexts for a thread: **entering** and **leaving**. Conceptually, at any point in time each thread has a set of entered contexts called the *active context set*.

Entering and leaving contexts of different types works as you might expect: the respective context is simply added to or removed from the thread's active context set.

However after entering a *related context* (which has the same type as an existing context represented in the active set) we want to be able to later leave that context and go back the previous related context once the new context is left. In the garbage collector example this corresponds to temporarily entering the v8 isolate context and then returning to the context of Frame #1.

This sequence is achieved by having a *context stack* data structure for each context type and processing the enter and leave operations as follows:

- **Entering** a context **replaces** any other related context in the active set and **pushes** that context into that context type's stack.
- **Leaving** any context **removes** that context from the active set and **pops** a context (if any) from the context type's stack, **inserting** it into the active set.

For example the sequence of events for the garbage collection case would be:

1. Enter(Frame #1) ⇒ active_set = **{Frame #1}**
frame_context_stack = **[]**
2. Perform work for Frame #1.
3. Enter(v8::Isolate) ⇒ active_set = **{v8::Isolate}**
frame_context_stack = **[Frame #1]**
4. Perform garbage collection.
5. Leave(v8::Isolate) ⇒ active_set = **{Frame #1}**
frame_context_stack = **[]**
6. Resume work for Frame #1.

Trace file format

The context trace macros emit the following types of events into the trace file:

- Entering a context with id 0x1000:

```
{"ph": "(", "id": "0x1000", "cat": "c", "name": "n", "ts": 20, "pid": 1, "tid": 1}
```

- Leaving a context with id 0x1000:

```
{"ph": ")", "id": "0x1000", "cat": "c", "name": "n", "ts": 20, "pid": 1, "tid": 1}
```

Each context object is represented as a standard object snapshot with any number of custom properties. The only defined property is a parent context pointer which is used to establish the context tree (if any):

```
{
  "name": "LocalFrame", "id": "0x1000", "ph": "O", "ts": 20, "pid": 1, "tid": 1,
  "cat": "example",
  "args": {
    "snapshot": {"parent": {"idRef": "0x2000"}}
   }
}
```

Trace model

While importing the trace, trace viewer will process all encountered context entering and leaving events and maintain the active set of contexts for each thread at any given time. The current value for this set is recorded into a new **contexts** property for each event in the trace model.

Each context object corresponds to the decoded object snapshot and can be used for reading context properties as well for traversing the context tree.

Frame context implementation

In addition to the generic context tracking mechanism, this project is about implementing a frame context tree for attributing work to individual <iframes> or the top level frame. This is done as follows:

- The following blink objects will be turned into context objects. This means snapshotting them into the trace when their properties change and recording a parent pointer for the next level above in the hierarchy:
  - v8::Isolate (done in blink::V8PerIsolateData)
  - WebViewImpl (property: visibility)
  - LocalFrame and RemoteFrame (property: url)
- Any scheduler task queue that is permanently associated with one of the above objects will enter and leave the respective context before and after executing a task.
- V8 garbage collection will enter and leave the isolate context (done in blink::V8GCController).
- Operations within BeginMainFrame and Blink's input handling code will be changed to enter and leave a particular frame's context where possible. The reason for this is that the compositor task queue is shared by all WebViews, so it cannot be permanently associated with any of them.
- DisplayItemLists should be annotated with context markers and processed while rastering (TODO: how?).
- As a follow-up, the default timer and loading task queues for the scheduler will be replaced with WebView/Frame-specific ones.

Cross process contexts

By default contexts are specific to a given process, because they are normally identified by their memory address. However in some use cases like tracing resource requests it is desirable to associate contexts across process boundaries. This is made possible by letting contexts refer to a context which belongs to a different process:

```
{
  "name": "LocalFrame", "id": "0x1000", "ph": "O", "ts": 20, "pid": 1, "tid": 1,
  "cat": "example",
  "args": {
    "snapshot": {"sibling": {"idRef": "0x2000", "pidRef": "1234"}}
   }
}
```

In this case the context id may be a pointer in the destination process or simply an opaque identifier. Different contexts may share the same id since the context type is used as the primary key.

A pid of -1 is reserved for referring to the main process in the case where the actual pid is unknown, e.g., because of sandboxing.

Computing per-context cost

TBD.

Future work: Runtime contexts

The context attribution mechanism described here is only applicable for offline analysis. If it becomes necessary to have access to context information at runtime (e.g., for attaching contextual data to memory allocations), we can extend the system as follows:

- Each context trace event (enter/leave) maintains a TLS pointer keyed by the |name| parameter. This points to a fixed size stack that maintains a per-context type and per-thread history of visited contexts.
- Any time a context is entered, the previous context for that type is pushed on the stack. Similarly leaving a context does a pop.
- A new macro is added to get the active a list of all the active contexts:

```
TRACE_EVENT_FOR_EACH_CONTEXT(callable)
```

e.g.:

```
TRACE_EVENT_FOR_EACH_CONTEXT([](const char* name, TraceContext context) {
  fprintf(stderr, "%s: %p\n", name, context);
});
```

- Another macro can be used to get the current context of a given type:

```
TRACE_EVENT_GET_CURRENT_CONTEXT(name, result)
```

- Alternatively there could be a pair of macros for (cheaply) saving and snapshotting the set of active contexts:

```
TraceEventContextSet contexts;
TRACE_EVENT_SAVE_CONTEXTS(&contexts)
contexts.AsValue();
```

Open issues

- Do we also need to represent v8::Context in the frame tree?
- How do we deal with async slices?
    - A simple starting point is for async slices to inherit the context set of the thread that started them.
- What about context propagation?
    - Most likely this should be done using flow events, but this needs more study. Again we could simply inherit the context set from where the flow started.
- How do we annotate network requests?
    - Using the async rule above, the resource request layer in the browser could just enter the context of the requesting frame when starting the async slice (modulo handwaving about translating context ids between processes).

      However this would only give frame attribution but other context types would not be represented unless they are explicitly communicated to the browser when starting the request.
- How about input event contexts?
    - One way to do this would be to define an input event context with no tree hierarchy and the following properties:
        - type: {“TouchStart”, “ScrollUpdate”, “MouseWheel”, etc.}
    - The browser starts a flow, which will automatically inherit that context (could even reuse the id).
    - Instead of passing around the latency info object, we pass around the flow id. To make it possible to pass the id into another process, it should not be a memory address but a generated unique id (maybe negative to distinguish it from valid memory addresses).
    - Whenever there is an incoming flow with an associated context set, **enter** each context on that thread.
    - Whenever there is an outgoing flow with an associated context set, **leave** each context on that thread.

Resources

- Prototype patches

    - Chromium: https://codereview.chromium.org/1447563002/
    - Trace viewer: https://codereview.chromium.org/1451143003/
- Patches:
    - Add trace context macros: https://codereview.chromium.org/1499683002/
- Prerequisite: Multi-process object references
- Tracing frame trees (how to represent the global tree of frames that are spread into different renderers)
    - FrameBlamer frame tree - Google Drawings
- Task traits (previous approach)