Monday, December 4, 2023

Modular Servo(or the lack thereof)

Case study: Spidermonkey(SM) integration

- Integration points:
  - Rust bindings to SM API: https://github.com/servo/mozjs
  - Generated glue code for WebIDL: https://github.com/servo/servo/blob/master/components/script/dom/bindings/codegen/CodegenRust.py.
  - Various utilities in /components/script/dom/bindings, such as:
    - root.rs — integration with garbage collector for Rust types.
    - refcounted.rs — passing pointers to rooted object across threads: important for IPC callbacks via per process router thread.
    - structuredclone.rs — integration for safe passing of structured data. See also serializable.rs and transferable.rs (used only for Messageport and Blob).
  - Other bits and pieces:
    - Script interruption via background hang monitor(used for shutdown of hanging script)
    - Readable stream(PR to remove it).
    - Microtasks, script runtime, window proxy, …

```
};
use js::jsapi::{
    GCContext, Handle as RawHandle, HandleId as RawHandleId, HandleObject as RawHandleObject,
    HandleValue as RawHandleValue, JSAutoRealm, JSContext, JSErrNum, JSObject, JSTracer,
    JS_DefinePropertyById, JS_ForwardGetPropertyTo, JS_ForwardSetPropertyTo,
    JS_GetOwnPropertyDescriptorById, JS_HasOwnPropertyById, JS_HasPropertyById,
    JS_IsExceptionPending, MutableHandle as RawMutableHandle,
    MutableHandleObject as RawMutableHandleObject, MutableHandleValue as RawMutableHandleValue,
    ObjectOpResult, PropertyDescriptor, JSPROP_ENUMERATE, JSPROP_READONLY,
};
use js::jsval::{JSVal, NullValue, PrivateValue, UndefinedValue};
```

The joys of using SM in components/script

First, the easy part

Example of past move(2019) towards separation of concern in script => less dependence on SM: **refactoring of structured clone for Blob**.

Blob is a DOM object, and a standard Web API: a file-like object of immutable, raw data.

Implementation of Blob in Servo:

- **Before** refactoring:
  - components/script/dom/blob.rs contains DOM integration, **and** all file-like logic and data.
  - components/script/dom/bindings/structuredclone.rs does serialization using SM API—**unsafe and clunky**.
- **After** refactoring:
  - components/script/dom/blob.rs contains **only** the DOM integration part.
  - Components/shared/script/serializable contains BlobImpl, a pure Rust object that contains all the logic and data.
  - dom/globalscope links the two.
  - Light-touch integration with SM, data serialization done with Serde(popular Rust crate).

Result: separation of concern between implementation—safe and easy to use Rust—and the DOM struct. Only the DOM struct needs to be integrated with SM => smaller integration surface.

General pattern: Impl struct deals with logic and data. DOM struct provides JS integration. Globalscope links the two via Id.

Bonus: easier to do complicated multi-process stuff over IPC(see dom/messageport).

Bad example: ReadableStream: tight coupling between Dom and controller(ExternalUnderlyingSourceController).

Other Web APIs where the pattern may be useful:

- WebGPU: see AsyncWGPUListener trait implemented on GPUBuffer directly, mixing DOM logic(including use of SM raw API JSObject and NewExternalArrayBuffer) with IPC and shared mem type of logic.
- Response(use of DOM ReadableStream, doubling up as a native stream via FetchContext),
- HTMLMediaElement(Arc<Mutex<dyn Player>>, Arc<Mutex<dyn AudioRenderer>>)

Usually, the use of #[no_trace] around a complicated object is a hint.

Obviously, for simple stuff like DomRefCell<Option<ServoUrl>> it's convenient and ok.

Note: the irony is: when following this patterns, it's the global scope that ends-up full of "complicated objects with #[no_trace] around them", but at least it's all in one place and follows clear patterns?

For a comparison: look at FileListener(unfortunately using ReadableStream directly…), TimerListener, BroadcastListener, MessageListener in global scope, and then try to wrap your head around how a DOM response is tied-in with a fetch response.

(Opening issues if it sounds like a good idea…)

And now the hard part…

Modular ~~JS~~ execution engine in Servo?

What it would take:

1. Generalize interface to engine.
2. Rewrite code gen.

Positives: good abstractions already out there: see JSTraceable, Dom<T>. Most are either found in script/bindings, or in in the higher-level Rust part of mozjs.

Negatives: These abstraction internally are still tightly coupled with SM, and in script we have unsafe blocks using low-level SM bindings directly(see script_runtime, windowproxy, …).

Low hanging fruits(?):

1. Remove SM specific code outside of dom/bindings: replace use of `js::jsapi` **and** `js::rust` with `crate::bindings`.
2. Remove unsafe use of `js::jsapi` from bindings and codegen: use only safe `js::rust`

Harder:

- Rewrite codegen
- Complicated dom integrations:
  - Dom/Windowproxy
  - Microtask queue,
  - structured clone callbacks

Example: **dom/Windowproxy** contains mostly “normal” data, like **ServoUrl** or **TopLevelBrowsingContextId**. But, **Windowproxy::new** is unsafe, because of the use of js::jsapi like **JSAutoRealm**.

While part of the low hanging fruits could be moving the JS specific logic to dom/bindings, and move the unsafe part to js::rust, we could also start thinking about a general interface to the windowproxy(and other) concept?

RuntimeMethods::get_window_proxy_for_realm => WindowProxy

The endgame would be a generic interface to a runtime, which something like rust-mozjs would have to implement.

Something like dyn Compositor, but more complicated…

Potential benefit:

1. Leverage Rust ecosystem: Deno's V8 bindings (and codegen? Seems not as sophisticated as what Servo does now: https://github.com/denoland/deno/issues/11118).
2. Make Servo usable for other type of script execution(pure wasm?)
3. Remove unsafe use of SM specific API's, replace with use of safe interfaces to a generalized "execution engine" => easier for people to contribute.
4. Speculative: with clear interfaces and patterns for how to do things: use AI for codegen based on WebIDL? On can imagine generating not only the bindings, but also the Dom struct(with rooting code and default calls into globalscope), leaving only "real" business logic to be written(mostly outside of components/script) => much easier to contribute and make progress on Web APIs. (Note: we could do this with SM as well off-course)