Streams C++ port design doc

This Document is Public

Authors: ricea@chromium.org

One-page overview

Summary

Blink’s implementation of the streams standard currently has the underlying logic implemented in JavaScript using V8 Extras, along with C++ wrappers to access them from other parts of Blink (particularly fetch). This effort will replace the JavaScript code with C++. The actual task of rewriting the JavaScript into C++ is mostly mechanical substitution, so this design doc largely focuses on the integration of the code into Blink, which is a lot more complex.

Platforms

All Blink platforms.

Team

blink-network-dev@chromium.org

Bug

https://bugs.chromium.org/p/chromium/issues/detail?id=902633

Code affected

Mostly core/streams and core/fetch.

Design

The JavaScript implementation has been a cause of unique problems too numerous to cover here. By porting to C++ we can replace these problems with the normal, well-understood issues of other Blink C++.

The Fetch API has deep dependencies on streams and will require significant changes to adapt.

Transition

The plan is to develop the port behind the flag, meaning that both the old and new implementations will live side-by-side for a while. This will increase short-term complexity but reduce the risk of the effort.

Preparatory work

Code in other modules needs to be isolated from the detail of which underlying implementation we are using. This mostly affects BodyStreamBuffer and ReadableStreamBytesConsumer, which directly hold references to JavaScript objects.

ReadableStream, WritableStream and TransformStream will be wrapped in C++ IDL objects. The implementation will delegate internally to the existing JavaScript implementation, but the “bare” JavaScript implementations will not be exposed to user code.

BodyStreamBuffer will be modified to hold a ReadableStream via a TraceWrapperMember. ReadableStreamBytesConsumer will be modified to hold a ReadHandle, a subset of the ReadableStreamDefaultReader interface with only the Read() method exposed.

The IDL wrapper will be changed to use virtual methods and the existing implementations will be moved to classes ending in “Wrapper” which the virtual methods delegate to. This will permit choosing to use “WritableStreamWrapper” or “WritableStreamNative” at runtime.

Adding the new implementation

The C++ implementation will use bindings generated from IDL. The standard is not WebIDL-based, but the IDL files will be mostly trivial. There are some slightly tricky points:

- Methods and accessors need to be marked with [NotEnumerable] to match ECMAScript class-definition semantics.

- The standard unpacks underlying object and strategy arguments in a specific order which doesn't match that specified by WebIDL, so they cannot be defined as dictionaries in IDL.
- Non-useful constructors like ReadableStreamDefaultController are not exposed in the standard, so will have to be labelled with [NoInterfaceObject]. Unfortunately this cannot be done for ReadableStreamDefaultReader and WritableStreamDefaultWriter as they are defined as having constructors, and that is not permitted for [NoInterfaceObject] interfaces.

We will temporarily use a naming convention of adding "Native" to the C++ class names, to make it easier to distinguish them from the IDL wrappers.

The Create() functions of the ReadableStream and WritableStream IDL wrappers will check the flag to determine which implementation to construct. The other IDL files are only used with the new implementation, and so don't need the check. The static C++ entry points will also check the flag to find which implementation to delegate to.

Porting

The goal in porting is to convert the existing code from JavaScript to C++ without introducing bugs and as efficiently as possible.

1. The initial porting can be done by performing search-and-replace on the existing JavaScript implementation to convert it to "pseudo-C++". For example, replacing "const" with "const auto".
2. The next step is to go through line-by-line adding missing type annotations and `script_state` and `exception_state` arguments.
3. Deciding the representation of data structures and algorithms will have to be done on a case-by-case basis.

Promises

Promises in the streams standard have slightly unusual requirements. It must be possible to:

1. store or return a resolved or rejected promise
2. resolve or reject a stored promise, including one that has already been returned to JavaScript
3. create a Promise resolved with a value, passing the value through unchanged if it is already a promise
4. Create a Promise rejected with a value
5. "Then" a promise with built-in algorithms at the fulfilled/rejected handlers
6. Determine if a promise is settled
7. Mark a promise as handled

The v8 API doesn't make it possible to do all these operations with a single type. In particular, you cannot take an existing Promise returned by JavaScript and convert it to a v8::Promise::Resolver. This means we need at least 2 different types to represent promises. Fortunately, the standard never directly stores a Promise that is returned by JavaScript: in other words, we never need to do 2. and 3. on the same type.

We end up using three types for Promises:

1. `StreamPromiseResolver` is a new type for Promises that are stored and returned to JavaScript. Internally it uses a `v8::Promise::Resolver` stored in a `TraceWrapperV8Reference`.
2. `v8::Local<v8::Promise>` is used for Promises that were returned from JavaScript, that we need to do the "Then" operation on.
3. `ScriptPromise` is used to wrap Promises for return to JavaScript. Since it creates a permanent reference to the Promise, it cannot be used more extensively.

Algorithms

To represent algorithms we'd like to use `base::Callback`, but as it is not traceable it can cause reference cycles. It would be over-the-top to implement a traceable equivalent to `base::Callback`, so instead we'll use a set of traceable base classes that cover our needs, and algorithms will be implemented by subclass these base classes. Specifically, we'll need

- `StrategySizeAlgorithm`, taking one `v8::Local<v8::Value>` argument and returning a double, potentially throwing an exception.
- `StreamAlgorithm`, taking an array of `v8::Local<v8::Value>` arguments and returning a `v8::Local<v8::Promise>`.
- `StreamStartAlgorithm`, taking a no arguments, returning a `v8::MaybeLocal<v8::Promise>`, and potentially thowing an exception.

The `Run()` method of these objects will also takea `ScriptState` argument to use internally.

Friends

The standard makes extensive use of access to the internal slots of other types. For example WritableStreamDefaultControllerAdvanceQueueIfNeeded accesses *stream*.[[state]]. When converted to C++, this means `WritableStreamDefaultController::AdvanceQueueIfNeeded()` will need to access `WritableStreamNative::state_`. It should be possible to handle all of these cases via accessors, but temporarily making `WritableStreamDefaultController` a friend of `WritableStreamNative` may save time.

Intermediate States

Until all the pieces are landed the streams implementation with the flag enabled will be quite broken. As long as it doesn't crash, this is fine.

Post-porting refactoring

This work doesn't need to be done until the new implementation is fully functional. However, it is important to leave the code in a maintainable state before we consider the port to be "finished".

Some of the ported code will be unnatural as C++. In particular, the way the operations directly modify the internals of other objects is not normal C++ practice. This code can be refactored to have better encapsulation, while being careful not to depart too far from the structure of the standard.

Most of the stream operations behave like static functions, taking the stream as an explicit parameter rather than via `this`. It is not clear whether modifying them to non-static methods would improve readability.

Extensive use of the auto keyword can make code hard to follow. The actual types will be substituted where needed for readability.

Post-launch cleanup

Once we are no longer shipping the V8 Extras implementation, we will delete it. We should also notify that V8 team that they should feel free to delete any V8 Extras functionality that has no remaining users.

Then we should rename ReadableStreamNative to simply "ReadableStream", mark it final, and make the methods non-virtual.

Memory Management

Reference cycles are a natural consequence of the API design. For example, the underlying source will often store a copy of the controller, which has a reference to the ReadableStream, which has a reference back to the underlying source. This is fine as long as everything in JavaScript, but crossing the V8/Blink boundary makes these cycles problematic. We are heavily dependent on unified Blink/V8 garbage collection to solve these problems for us.

JavaScript objects will be stored in a TraceWrapperV8Reference. TraceWrapperMember will be used until it is obsoleted by unified GC.

Testing

Both streams and fetch have extensive web platform tests, which should provide a high degree of confidence that we haven't broken anything. Existing unit tests should be ported to the new APIs where possible so that we don't lose coverage. New unit tests will be written for internal operations such as CreateAlgorithmFromUnderlyingMethod.

V8 internals

The JavaScript implementation uses various v8 extra internal functions. The C++ implementation will need replacements for some of them.

Internal	**Replacement**
`createPrivateSymbol`	Not needed. Will use member variables rather than private symbols.
`uncurryThis`	Not needed. Will call v8 API functions directly.
`createPromise`	`MakeGarbageCollected<StreamPromiseResolver>()`
`isPromise`	`v8::Value::IsPromise()`
`markPromiseAsHandled`	`v8::Promise::MarkAsHandled()`
`InternalPackedArray`	Not needed. Will use a `WTF::Deque` for queues.
`promiseState`	`v8::Promise::State()`
`rejectPromise`	`StreamPromiseResolver::Reject()`
`resolvePromise`	`StreamPromiseResolver::Resolve()`

Metrics

Success metrics

We can expect an improvement in the time for a ServiceWorker to service its first fetch, as this will no longer require parsing JavaScript. However, the primary purpose of this work is not performance, so we won't be actively looking for performance gains.

Regression metrics

- Crashes
- Standard "smoke test" metrics
  - PageLoad.PaintTiming.NavigationToFirstPaint
  - PageLoad.PaintTiming.NavigationToFirstContentfulPaint
  - PageLoad.Experimental.PaintTiming.NavigationToFirstMeaningfulPaint
  - PageLoad.InteractiveTiming.FirstInputDelay2
- ServiceWorkers:
  - PageLoad.Clients.ServiceWorker2.PaintTiming.NavigationToFirstContentfulPaint
  - PageLoad.Clients.ServiceWorker2.InteractiveTiming.FirstInputDelay3
  - PageLoad.Clients.ServiceWorker2.ParseTiming.NavigationToParseStart

Experiments

We will run an A/B experiment to measure stability, and ensure there are no performance regressions.

Rollout plan

Waterfall.

Core principle considerations

Speed

This change is generally expected to improve performance, particularly in common fetch-related cases. The exception is cases which currently benefit heavily from V8's inlining of JavaScript, for example extremely lightweight transforms.

Security

V8 extras have surprising security properties due to sharing the global object with untrusted code. The new implementation will use common Blink C++ coding patterns and should be much easier for general Blink coders to reason about the security properties.

Privacy considerations

None.

Testing plan

No specific work needed by the test team.

Followup work

Once the old implementation is removed, the extra indirection can be removed. ReadableStreamNative can be renamed just ReadableStream and clients can be modified to use it directly.

Many operations which in the old implementation could throw will now be reliable, so many of the methods can have ExceptionState arguments removed and base::Optional return types made non-optional. In many cases, the callers only have ExceptionState arguments because of streams, and so ExceptionState arguments (and error checking) can be removed several layers up the callstack.

It may be possible to simplify BodyStreamBuffer so that it can always shuffle data via the ReadableStream it owns, rather than bypassing it as it does now.

History

When	**Who**	**What**
25 October 2018	ricea	First version.
17 January 2019	ricea	Added “Promises” and “Algorithms” sections.
22 January 2019	ricea	Extensively updated with insights gained from porting

		miscellaneous operations.
30 January 2019	ricea	Updated “Promises” and “Algorithms” sections.
31 Janury 2019	ricea	Rename StreamPromise to StreamPromiseResolver.
20 March 2019	ricea	Update to match latest implementation.