Fetch API Cancellation Design Doc (public version)

This Document is Public

Authors: ricea@chromium.org

One-page overview

Summary

Adding support for the AbortController and AbortSignal interfaces, and the "signal" attribute on `Request` and `RequestInit`. This will make it possible to cancel an ongoing fetch operation. This will permit code like:

```
const controller = new AbortController();
const signal = controller.signal;
const promise = fetch(request, { signal });
// Give up if the fetch takes more than a second.
setTimeout(() => controller.abort(), 1000);
```

Platforms

All Blink Platforms.

Team

blink-network-dev@chromium.org

Bug

750599

Code affected

Fetch API implementation in Blink.

Design

Fetch cancellation is an often requested feature. It was standardised this year, and is implemented and shipping in Firefox and Edge.

`AbortController` and `AbortSignal` are part of the DOM Standard:
https://dom.spec.whatwg.org/#aborting-ongoing-activities

The integration with fetch is described in the Fetch Standard:
https://fetch.spec.whatwg.org/#dom-request-signal

Although `AbortController` and `AbortSignal` are scheduled to be used in other APIs in the future (Streams and Web Locks are expected to use them), this work is scoped as an incremental improvement to Blink's `fetch()` implementation.

Blink's `fetch()` implementation is built on top of `blink::ThreadableLoader`, which already supports cancellation. As a result, the changes for this work will be limited to the `fetch()` implementation. In particular, no additional IPC or inter-thread messaging is required.

AbortController

`AbortController` has minimal functionality. It owns an `AbortSignal` object and has an `abort()` method which triggers the "signal abort" algorithm. From a C++ point-of-view, it is a class with one member and one trivial method.

A use counter will be created for `AbortController`.

AbortSignal

`AbortSignal` inherits from `EventTarget`. It has a boolean indicating whether it has been aborted yet, and a set of abort algorithms. The DOM standard supports removing an algorithm from the set, but the Fetch standard doesn't require that, so in the interests of simplicity we won't implement that until we need it. For the purposes of this work we'll use a vector of closures, and implement an `AddAlgorithm(OnceClosure)` method to be used by the fetch implementation.
`AbortSignal` will need a `SignalAbort()` method which is not web-exposed but is available for use by the `Request` constructor and `clone()` method, and `AbortController`'s `abort()` method.

Since `AbortSignal` cannot be constructed by user code, it will not have its own use counter.

Request

In the Fetch Standard, a `Request` always has an associated `AbortSignal` object. Since most requests don't need it, as a simple optimisation we can save some memory and CPU by initialising it lazily when it is needed. The same applies to `RequestInit`.

To match the behaviour of the standard, the `AbortSignal` is not stored in `FetchRequestData`, but as a separate member of the `Request` object.

RequestInit

The `RequestInit` constructor explicitly unpacks the options parameter rather than using the bindings to do this. The case of "signal" is difficult because `undefined` and `null` need to be treated differently.[1] For this reason "signal" must first be unpacked as an `IDLPassThrough` type to check if it is `null`, and then explicitly converted to an `AbortSignal`.

The object is stored as an `Optional<Member<AbortSignal>>` to preserve the distinction between `null` (indicated by an explicit `nullptr`) and "not present" (indicated by no value being stored in the Optional).

FetchManager

`FetchManager::Loader` gains an `Abort()` method which is used to perform the actual abort. It implements the Abort fetch algorithm from the standard.

The `Fetch()` method gains an extra `AbortSignal` parameter. When `GlobalFetch::Fetch()` calls `FetchManager::Fetch()` it passes a pointer to the `AbortSignal` object.

`Fetch()` stops immediately if the signal is already aborted. Otherwise it calls `AddAlgorithm()` on the signal with a Closure that calls the `Loader::Abort` method.

Body Cancellation

The following step in the standard creates a lot of complexity in our implementation:

> 7. If response's body is not null and is readable, then error response's body with error.

Simply cancelling the request results in attempts to access the body being rejected with a TypeError, which is non-compliant. The `AbortSignal` object needs to be passed through several layers to get the correct behaviour.

---

[1] Setting `signal: null` will remove any existing signal from the Request, whereas `signal: undefined` will leave it alone.

In Blink, the `Response` object owns `FetchResponseData`, which is created with a `BodyStreamBuffer` when the response is received. Upon an attempt to access the response body via one of the methods on `Body`, a promise is created and passed to a `FetchDataLoader::Client` subclass which is passed to `BodyStreamBuffer::StartLoading`. It is this promise that we need to reject with an `AbortError` when abort is signalled.

`BodyStreamBuffer` doesn't keep a reference to the `FetchDataLoader::Client` subclass; it wraps it in a `BodyStreamBuffer::LoaderClient` object and passes it to the `FetchDataLoader::Start()` method. `BodyStreamBuffer` keeps a reference to `FetchDataLoader` and `FetchDataLoader` keeps a reference to the `BodyStreamBuffer::LoaderClient` wrapper which has a reference to the original `FetchDataLoader::Client` which has a reference to the promise that we need to reject with an AbortError.

This will be fixed in three stages:

1. **Cancellation before the body has been read:** a reference to the `AbortSignal` object will be stored on the `FetchManager::Loader` class so that it can pass it to the `BodyStreamBuffer` when it creates it. `Body::RejectInvalidConsumption()` will be modified to reject attempts to read the body when the `AbortSignal` has been signalled.
2. **Cancellation of the Response body ReadableStream:** Since this stream is attached directly to the `BodyStreamBuffer` object, it can cancel the stream easily. A new `BodyStreamBuffer::Abort()` method will be added which cancels the `ReadableStream` with an `AbortError`. This can be called synchronously in the constructor if the abort has already happened, and asynchronously later by adding an algorithm with `AddAlgorithm`.
3. **Cancellation of the body while reading is pending:** A new virtual method, `Abort()`, will be added to `FetchDataLoader::Client` and implemented in `BodyConsumerBase` to reject the promise. `BodyStreamBuffer::StartLoading()` will add an algorithm will be added to the `AbortSignal` to call `Abort()` on `FetchDataLoader::Client`.

Lifetime Management

`AbortController`, `Request` and `RequestInit` should keep their `AbortSignal` alive. An `AbortSignal` shouldn't keep any of those things alive. What this means in practice is that the closure passed to `AddAlgorithm()` should use a weak pointer.

Metrics

Success metrics

In the long-term this change is expected to increase the usage counter for the fetch API as it removes a blocker to adoption. However, this is conflated by many other factors (the usage counter is going up anyway). So there are no direct metrics to measure success.

The use counter for `AbortController` will naturally start at 0 and go up, so while in the long-term it will be useful to measure adoption in the short-term it will not be very meaningful.

Regression metrics

Crashes and memory usage are the main areas where regressions would be seen.

TBD: Are there any other metrics that might regress in a meaningful way?

Experiments

The feature won't do anything until web developers use it, so an experiment is unlikely to yield useful information.

Rollout plan

Waterfall.

Core principle considerations

Everything we do should be aligned with and consider Chrome's core principles. If there are any specific stability concerns, be sure to address them with appropriate experiments.

Speed

When not used, the feature will increase the size of `blink::FetchRequestData` by one pointer, and add a few extra branches. Both of these changes will be lost in the noise. There will also be the normal increase in binary size associated with a small new feature, and the two extra symbols on the window object will cause a minor increase in runtime memory usage.

When in use, the feature will in general reduce network bandwidth, memory and power consumption by permitting unwanted resources to be cancelled. The degree of saving is specific to the way it is used by web developers.

Security

This is very similar to the existing XHR `abort()` method. The request cancellation paths are extremely well tested. There are no known security issues with this functionality.

Privacy considerations

No privacy considerations. No additional data is exposed to the page.

Testing plan

Web platform tests already exist: https://wpt.fyi/dom/abort for `AbortController` and `AbortSignal`, and https://wpt.fyi/fetch/api/abort for integration with fetch. Some testing gaps will be addressed during implementation.[2]

Followup work

In a service worker, `fetchEvent.request.signal` should reflect the status of the original fetch. This requires changes outside the implementation of the Fetch API and so will be implemented separately.

`AbortController` and `AbortSignal` classes will be moved out of modules/fetch and into core once other web platform features need them.

"Remove an algorithm" algorithm

Something will have to be done if this algorithm gets a user. It's specified at https://dom.spec.whatwg.org/#abortsignal-remove.

Possible implementation strategies:

1. Return an id from AddAlgorithm() which can be passed to RemoveAlgorithm(). Change the data structure to a map keyed on id. Id increments monotonically so "in

[2] Known missing:

- Test that `AbortController`'s signal always returns the same value.
- Test that `abort()` doesn't do anything the second and subsequent times it is called.
- Test that adding an event listener after abort is called behaves correctly.

order" semantics are preserved.

Advantages:

- No changes to existing callers

Disadvantages:

- Callers that need "remove" have to store the id
- Extra memory usage and CPU overhead for the map
- Id will have to be int64 as int could be trivially overflowed

2. Change callers to inherit from a common base class AbortSignal::Algorithm and pass |this| to RemoveAlgorithm(). Change the data structure to an ordered set.

   Advantages:

   - Callers do not need to store an ID

   Disadvantages:

   - All callers have to be changed
   - Callers grow in size by an additional vtable
   - Extra memory usage for the set
   - Multiple inheritance

3. No RemoveAlgorithm() method added. Callers have to keep track of whether they have "removed" the algorithm and arrange for it to be a no-op if they have.

   Advantages:

   - No changes and no overhead for existing callers
   - No changes to AbortSignal

   Disadvantages:

   - If there are many algorithms removed the overhead of running all those no-op callbacks could be significant.
   - If a single caller adds and removes algorithms multiple times over its lifetime the complexity of keeping track of which algorithms haven't been removed could be significant.