Promise Hooks Proposal

Note that this document is publicly accessible.

Author: yangguo@google.com, bmeurer@google.com, ofrobots@google.com

Issue: https://github.com/nodejs/diagnostics/issues/188

Last updated: 2018-05-02

Motivation

Promise hooks in Node.js is part of Async hooks, a way to track the context across asynchronous operations.

In its current implementation, Promise hooks can cause significant performance degradations. In some cases enabling Promise hooks can regress performance by up to 70%. This is not acceptable in use cases where Promise hooks are enabled in production.

In this document we want to discuss ways to reduce the overhead for Promise hooks.

Status quo

When enabling Promise hooks, Node uses v8::Isolate::SetPromiseHook to add a C++ callback to V8. For every Promise created in that V8 Isolate, V8 will call the callback for the following events in the Promise lifecycle:

- kInit: when the Promise is created. Maps to the init hook.
- kResolve: when the Promise is resolved. Maps to the promiseResolve hook.
- kBefore: before the Promise reaction job is executed. Maps to the before hook.
- kAfter: after the Promise reaction job is executed. Maps to the after hook.

In addition, Node also implemented a destroy hook for when the Promise object has been garbage collected. To implement this destroy hook, Node creates a weak global handle for every Promise in kInit.

For each of these five hooks, Node calls JavaScript callbacks to dispatch to hooks installed by the user.

Performance Issues

This design has several issues that contribute to bad performance. Also read this document for details.

- According to spec, every await call creates four Promises, some of which are not observable from JavaScript, but are now through Promise hooks.
- For every Promise, Node creates a C++ wrapper object.
- Each wrapper object also creates a weak global handle. This adds additional overhead for the garbage collector and additional callbacks to perform the destroy hook.
- During the lifetime of every Promise V8 triggers the callback four times.
- For every callback, V8 enters C++, and from there re-enters JavaScript. Both are not particularly cheap to perform.

Other Issues

The fact that the Promise hook is installed on the Isolate rather than on a native context means that Promises created in every native context will trigger the same Promise hook callback. That results in issues such as this one, where an unexpected native context causes the Promise hook callback to crash.

Assumptions

We made some assumptions as basis for our proposal to lessen the performance impact.

- Use cases that use async hooks only use the destroy hook to dispose metadata (see these examples).
- Use cases that use the destroy hook to perform more work than just disposing metadata do not need to track the destroy event for every Promise created during await call.
- Promises that are not observable from JavaScript are not interesting to async hooks.

Proposals

Separate destroy hook

We propose to remove the destroy hook and to add the async resource to the remaining async hooks. (Or not create weak global handle if the destroy hooks is not installed).

One class of uses keep metadata for every async operation and map the async id to the metadata. The destroy hook is necessary to clean up the metadata. Relying on the after hook is not sufficient, since Promise objects could be garbage collected before they are resolved.

For every four PromiseHook types, V8 provides the affected Promise object to the callback. Of the corresponding async hooks, each one provides the async id, but only the init hook provides

the async resource object (a wrapper object for the affected Promise). Users of these hooks that keep additional metadata need to map the id to the metadata. The destroy hook is required to remove this mapping and release the metadata.

If the async resource is passed not only to init, but also to the other hooks, user code could use a WeakMap to map the resource to the metadata, or store the metadata on the resource directly. (Also an executionAsyncResource next to executionAsyncId). This removes the need for a destroy hook to release the metadata. Internally, Node.js would use a WeakMap to map the Promise provided by V8's PromiseHook callback to the resource.

The benefits of this change is that we no longer have to create a weak global handle for every Promise. If the async resource is created in JavaScript too, then we no longer have to enter C++ for the init hook, and no longer need to call a C++ finalizer for every Promise.

For the second class of uses, the destroy hook is used to perform actions that go beyond keeping track of metadata. A finalizer has to run. We suggest to use finalizers in the WeakRef language proposal instead. Until WeakRef is implemented in V8, a polyfill based on weak global handles can serve as a stop gap solution.

We find that it is reasonable to pay the price in form of performance for this second class of uses for the provided flexibility.

JavaScript instead of C++ callbacks

Assuming that we no longer have to call into C++ to wrap the Promise object and install a weak global handle to implement the destroy hook, we can now directly call into JavaScript.

We propose to change V8's PromiseHook API to take four v8::Function objects as arguments, one for every Promise hook type. On Promise events, V8 can directly call these functions without paying the cost for entering C++ and re-entering JavaScript.

Since v8::Function objects are context-dependent, it is natural that Promise hooks are installed onto the native context instead of applying Isolate-wide. This also solves the previously mentioned issue with unexpected native context.

The benefit is that we can avoid crossing the C++/JavaScript boundary, which enables further optimizations. Having Promise hooks in JavaScript may enable optimizations such as inlining in the future.

Only trigger hooks for observable Promises

Some Promises that are only created as part of implementation detail prescribed by the spec and are not usually interesting to users of async hooks. E.g. the throwAwayCapability at step 10 of the specification of await.

We propose to not trigger hooks for these unobservable Promises.

Add specification and tests

The current test coverage for Promise hooks is very sparse. Given the long turnaround time between landing a V8 change and propagating it to Node.js, and users wanting to rely on async hooks, this is a very bad situation.

We would welcome to see a spec text based on the ECMA262 spec regarding when hooks should be fired, and how async IDs should be assigned. Tests are essential not only to catch regressions, but also to provide help for implementations in other VMs.

Timeline

These proposals are breaking changes that need to be implemented before async hooks move out of experimental. V8 will implement the new API based on v8::Function callbacks, as well as behavior changes as discussed above.

V8 6.9 goes stable on 2018-09-04, and branches on 2018-07-19.
V8 7.0 goes stable on 2018-10-16, and branches on 2018-08-30.

Depending on which V8 version the initial release of Node 11 chooses, we have 11 to 17 weeks from now, if we want to be ready for Node 11.

Changes on Node's side have time until the actual release of Node 11, which includes using the new API provided by V8, removing the destroy hook and Promise wrapping, and offering a stop gap solution for weak callbacks.

Passing the resource through the hooks can be implemented for Node 10 and maybe even backported to Node 8.

Alternatives considered

Do nothing

Promise hooks are a powerful and intrusive instrumentation into V8. There have already been some efforts to improve Promise performance, some of which help with Promise hooks, but most do not, since it prevents certain shortcuts.

We could argue that the impact on performance is a fair price to pay for this level of intrusiveness.

A subset of the above

While each of the three proposals are breaking changes, they are somewhat orthogonal to each other and can be implemented independently from each other.

Make Promise constructor monkey-patchable

This has been discussed here.

This moves the instrumentation entirely to Node.js, and avoids some of the performance costs paid to enter and leave C++. The API to enable this does not have to be exposed to JavaScript in order to avoid violating the ECMA262 spec.

However, the potential for performance improvement through this seems limited.

This does not solve the performance issues around the destroy hook.

Revive Zones proposal

Zones as proposed here are designed to solve the async context tracking issue, but at a higher level and is less powerful than async hooks. If integrated into the ECMA262 spec, performance overhead could be reduced a lot.

However, it is unclear whether Zones will have a chance at TC39. And even if it does, how long it will take to reach stage 4. In the best case, it will take at least a year until it becomes ready.

Zones also do not offer a destroy hook.