Oilpan in Node.js

Contact: Joyee Cheung <joyee@igalia.com>
Contributors: Joyee Cheung <joyee@igalia.com>
Status: Draft | **In review** | Accepted | Done
Last updated: 2024 Aug 22

Abstract

This design doc describes a plan to integrate Oilpan (cppgc) into Node.js and migrate the Node.js native objects to the new unified heap.

Background

In Node.js, most of the native objects are built on top of the BaseObject class. A BaseObject holds a v8::Persistent referencing its JS-land counterpart. The memory management of a BaseObject can be classified into the following categories:

1. In many cases, the native BaseObject's lifetime depends on the JS-land object - whenever there are no longer references to the JS-land object, it's fine to release the BaseObject. In this case, the v8::Persistent is made weak, and in the weak callback we delete the BaseObject.
2. In some cases, the native BaseObject's lifetime depends on a certain native resource, so the v8::Persistent remains strong, and we will only delete the BaseObject when the native resources are no longer alive.

For 2, there is also a cleanup queue that keeps track of all the BaseObjects allocated in a Node.js instance and it is drained to delete all of them when the Node.js instance shuts down. This is a safety net that ensures that long-lived objects are always deleted when the Node.js instance is shutting down, even if the native resource is still alive.

This has been working fine for the main context, because we always have a clear idea about when the Node.js instance is shutting down and can delete all the BaseObjects before we dispose the context. But it does not work well in the ShadowRealms, because the lifetime of the native ShadowRealm depends on the lifetime of the context, and any strong reference to a JS-land object would keep the context alive forever, which in turn keeps the ShadowRealm alive forever, resulting in a leak (see #47353).

Another issue that comes with 1 is that this kind of lifetime relationship is not understandable by V8 - obviously V8 cannot understand the semantics of the C++ code of the weak callback

passed to it as a pointer. Whenever there is a cycle in the memory graph with a v8::Persistent link that is only broken when a weak callback is called, it's very easy to introduce a leak - but if we break that link too early, it can then result in a use-after-free (#44211 is a good example of this). This is also why V8 explicitly warns against using the weak callback for this purpose, still this has been a practice that exists in Node.js for quite some time.

Motivation

Better memory safety

With the integration of Oilpan/CppHeap, we can use a few new tools to deal with the issues (on what Oilpan is, see the V8 blog post):

1. We can use v8::TracedReference instead of v8::Persistent to create C++-to-JS references. This avoids creating global strong references unnecessarily. V8's GC understands the lifetime relationship expressed through this mechanism, making the handling of cycles less leak-prone.
2. We can allocate native objects as cppgc::GarbageCollected in the CppHeap, and create JS-to-C++ references using an internal field. Then instead of being notified via the weak callback, we will directly know that the native object is no longer reachable from the unified heap when the destructor is called. It prevents use-after-free for these objects, since the native object's destructor would not be called by the V8 GC while the JS object is still alive to use it.

Better performance

A unified heap with references known to V8 helps improve garbage collection schedules, which may reduce memory consumption and decrease latency.

The current BaseObject management incurs a significant overhead during object creation, partly from the book-keeping of BaseObjects in the cleanup queue, partly from global handle creation. A local prototype shows that creating a cppgc-managed object can be 2.5x faster than creating a BaseObject:

```
misc/object-wrap.js method="ExampleCppgcObject" n=1000000: 8,113,612.185256452
misc/object-wrap.js method="ExampleBaseObject" n=1000000: 3,218,022.6465318813
```

Another prototype shows that migrating from BaseObject to cppgc-managed objects in the crypto Hash implementation can speed up the creation of Hash objects by 27% and a one-shot `createHash().update().digest()` operation by 10% (numbers from a microbenchmark where there's significant impact from scavenge GC).

A different prototype shows that ContextifyScript creation (on smaller scripts) can be around 6% faster after migration.

However, some classes took a performance hit after migration - especially SerializerContext/DeserializerContext which involves frequent allocation of array buffers. However the performance impact is positive again with some ad-hoc `AdjustAmountOfExternalAllocatedMemory()` calls, so this may have to do with pre-existing under-reporting bugs.

Design

Reference: Universal Garbage Collection for V8 in Blink

Embedder ID configuration

In the current CppHeap design, embedders who wants to make V8 aware of cppgc-managed objects referenced by the JS objects need to configure the JS objects to point to an embedder ID configured during the initialization of CppHeap. #43521 has configured this ID to be a hard-coded value of 0x90de for Node.js's BaseObjects so that it does not clash with Blink's embedder ID and accidentally enable cppgc on non-cppgc-managed objects.

To allow Node.js to make use of CppHeap, we need two embedder IDs - one for cppgc-managed objects and one for non-cppgc-managed objects. In the case of Blink, the embedder ID for cppgc-managed objects is 0x1, but hard-coding the ID to a value in an external codebase isn't ideal. So the plan is:

1. Add an V8 API to expose the effective cppgc embedder ID: V8:4598833
2. When Node.js is initialized, deduce the non-cppgc embedder ID from the effective cppgc embedder ID, and use different IDs for objects managed in different ways: #48660

CppHeap initialization

To enable cppgc tracing, Node.js needs to run on an isolate with an initialized CppHeap. In a standalone Node.js executable, Node.js is usually the only owner/creator of the isolate and it can just create/tear down the CppHeap at appropriate times, with full control over what the wrapper ID of the CppHeap is. But when Node.js is embedded, for example, together with Blink (in Electron), the isolate may be owned/created by another V8 embedder, so Node.js needs to

play along with any existing owner of CppHeap to create its own cppgc-managed objects. The plan is:

1. At initialization time, detect if the isolate already has a CppHeap attached (which is the case in unsandboxed processes in Electron)
2. Create a CppHeap if there isn't one already, or just save the embedder ID if there is already a CppHeap.

Who should own the CppHeap?

The CppHeap needs to live somewhere, and there are a few alternatives:

1. In NodePlatform, and create it / tear it down in NodePlatform::RegisterIsolate()/NodePlatform::RegisterIsolate()
2. In node::IsolateData

2 may be more preferable because node::IsolateData is already where per-isolate data is supposed to go (implemented in 45704).

Helper Mixin for creating cppgc-based wrappers

To ease migration from BaseObject to cppgc-managed wrappers, it would be nice to have a helper cppgc::GarbageCollectedMixin to abstract away the differences.

This has been implemented in https://github.com/nodejs/node/pull/52295. See the merged documentation for details.

The high-level overview of this mixin looks like this:

```
class MyWrap final : CPPGC_MIXIN(MyWrap) {
 public:
  SET_CPPGC_NAME(MyWrap)  // Sets the heap snapshot name to "Node / MyWrap"

  // The constructor can only be called by `cppgc::MakeGarbageCollected()`.
  MyWrap(Environment* env, v8::Local<v8::Object> object) {
    // This cannot invoke the mixin constructor and has to invoke via a static
    // method from it, per cppgc rules.
    CppgcMixin::Wrap(this, env, object);
  }

  // Helper for constructing MyWrap via `cppgc::MakeGarbageCollected()`.
  // Can be invoked by other C++ code outside of this class if necessary.
  // In that case the raw pointer returned may need to be managed by
  // cppgc::Persistent<> or cppgc::Member<> with corresponding tracing code.
```

```
  static MyWrap* New(Environment* env, v8::Local<v8::Object> object) {
    return cppgc::MakeGarbageCollected<MyWrap>(
        env->isolate()->GetCppHeap()->GetAllocationHandle(), env, object);
  }

  // Binding method to help constructing MyWrap in JavaScript.
  static void New(const v8::FunctionCallbackInfo<v8::Value>& args) {
    Environment* env = Environment::GetCurrent(args);
    Isolate* isolate = env->isolate();
    Local<Context> context = env->context();

    CHECK(args.IsConstructCall());

    // Get more arguments from JavaScript land if necessary.
    New(env, args.This());
  }

  void Trace(cppgc::Visitor* visitor) const final {
    CppgcMixin::Trace(visitor);
    visitor->Trace(...);  // Trace any additional data MyWrap has
  }

  // Method to be invoked on the wrapper
  static void Method(const v8::FunctionCallbackInfo<v8::Value>& args) {
    MyWrap* wrap;
    ASSIGN_OR_RETURN_UNWRAP_CPPGC(&wrap, args.This());

    // Usually, this should be the same as wrap->env().
    Environment* env = Environment::GetCurrent(args);
    Isolate* isolate = env->isolate();
    Local<Context> context = env->context();
  }
};
```

Blocker: Startup Snapshot & Heap Snapshot support

Background: how it currently works

Both startup snapshot integration and heap snapshot integration requires a way to iterate over the embdder objects. Currently Node.js tracks all the embedder objects with a `CleanupQueue` (essentially a std::unordered_set) stored indirectly in `node::Environment`. Whenever an embedder object is constructed it's added to this set and when it's destructed it's removed from this set.

In the case of startup snapshot, this set is important for a two-pass process to avoid altering the heap during the serialization/deserialization of the context:

1. At serialization time, Node.js iterates over all the embedder objects that it tracks (the first pass), if the embedder object needs to persist reference to some JavaScript value, use `SnapshotCreator::AddData()`to record an ID for it , and save this ID in a `StartupData` blob.
2. When the context is serialized, V8 iterates over all the objects in the heap again and call `SerializeInternalFieldsCallback` on embedder objects found (second pass): In this callback Node.js return a `StartupData` containing ID it obtains in the first pass.
3. When the context is deserialized, whenever V8 finds an embedder object with additional data it calls `DeserializeInternalFieldsCallback.` In this V8 API callback Node.js enqueue another callback internally to do the actual deserialization when the context is fully deserialized.
4. After the context is fully deserialized, Node.js processes the queued callbacks and use `Context::GetDataFromSnapshotOnce()` and the ID extracted from `StartupData` to restore the references from the embedder object to other JavaScript values.
5. In some cases, the JavaScript values referenced by the embedder objects are only temporary and don't need to be persisted in the snapshot. For these references, Node.js simply releases them in 1, and in 2, re-initialize the temporary objects and reset the references (e.g. `AliasedBuffers`)
6. To support certain embedder objects that are not currently supported, Node.js may need to alter the reference graph in the first pass of the serialization e.g. adding a symbol property.

Most embedders that hold references to other embedder objects are requests that are not supported to be persisted in the startup snapshot. However we do need to show them properly in heap snapshots used for memory diagnostics. To this end, all the embedder objects that know how to properly track their own size and their relationship with other nodes implement the `node::MemoryRetainer` interface. When a heap snapshot is taken, V8 invokes a `BuildEmbedderGraphCallback`. From there, Node.js iterates over the `CleanupQueue` and invoke `MemoryRetainer::MemoryInfo()` . recursively on the embedder objects found. To build the embedder graph, for each embedder objects:

1. Construct a `node::MemoryRetainerNode` which implements `v8::EmbedderGraph::Node`, representing the native embedder object
2. Look up the V8 node corresponding to the wrapper object using `v8::EmbedderGraph::V8Node`
3. Create "native_to_javascript" and "javascript_to_native" edges between them
4. Call `MemoryRetainer::MemoryInfo()` recursively for all its members that are also `MemoryRetainers`, and add an edge between the current

`node::MemoryRetainerNode` and the `node::MemoryRetainerNode` that each of these members corresponds to.

And after this recursive process V8 would have a `v8::EmbedderGraph` with all the embedder objects and their relationships recorded to convert into part of the heap snapshot.

Dropping the cleanup queue

This cleanup queue was primarily invented for disposing undisposed objects during environment shutdown though it then was multi-purposed for snapshot integrations. In the case of cppgc-managed objects, the final disposition would also be taken care of by V8 (in `CppHeap::Terminate()`), so for them a cleanup queue would only serve snapshot integration.

The current cleanup queue-based tracking relies on `v8::Global` handles to the wrapper objects of the embedder objects and this comes with a non-trivial overhead. It would be nice if we could drop this heavy-weight tracking during the normal operation of the application. This would mean that we need to do a heavier weight heap iteration when the snapshots are taken, but since snapshot integration is only needed on-demand (when users actively takes a snapshot), and users normally care less about performance in those scenarios (startup snapshot is only taken at build time and in heap snapshot users generally care more about accuracy and completeness than performance), this should be acceptable.

Implementation idea 1: special heap-iteration before snapshot is taken

To enable snapshot support without the cleanup queue for cppgc-managed objects, we may need an API from V8 that does:

1. Perform a thorough garbage collection (e.g. `CollectAllAvailableGarbage()`)
2. Iterate over the heap to find all live cppgc-managed objects (even some false positives are okay as long as valid references are passed to the embedder).
3. Pass either a list of `v8::Global` to the wrapper objects or `cppgc::Persistent` to the native objects to the embedder so that we could continue to do what we do with the cleanup-queue-based iteration and prepare for snapshot serialization.

We could also prototype by using/abusing `v8::HeapProfiler::QueryObjects()` which essentially can achieve the same thing.

In progress: implementation idea 2: special interface in cppgc

See the spin-off design doc, V8 prototype CL and Node.js integration prototype.

Another possible solution is to extend the cppgc API so that we can implement an interface for this.

For startup snapshot integration, this interface needs two method overrides:

1. One that allows Node.js to dispose certain native states/references to JS objects in order for it to be snapshottable, before the serialization begins.
2. One that knows how to serialize information about the snapshottable native states into `v8::StartupData`, while the serialization happens

For heap snapshot integration, `node::NameProvider::GetHumanReadableName()` should be enough to replace what Node.js internally does with the `node::MemoryRetainer::MemoryInfoName()` override. `node::MemoryRetainer::IsRootNode()` and `node::MemoryRetainer::GetDetachedness()` (whether it's detached from JS) can probably be simply inferred by cppgc. That leaves us finding replacements for `node::MemoryRetainer::MemoryInfo()` and `MemoryRetainer::SelfSize()`

1. `node::MemoryRetainer::MemoryInfo()` is used to track size of members as children nodes in the heap snapshot.
    - For example, `crypto::Hash` tracks the the OpenSSL `EVP_MD_CTX` it holds as a node called `mdctx` whose size is `kSizeOf_EVP_MD_CTX`, and the computed hash as a node called `md` whose size is returned by `EVP_MD_size()`, as it depends on the hashing algorithm
    - The interface should embdders to report this information either by creating some synthetic cppgc-specific data structure that the heap snapshot generator can convert into part of the heap snapshot graph, or just pass a `v8::HeapProfiler::EmbedderGraph*` into an overridable method for the class to generate nodes and edges in it. The latter probably takes less effort.
2. `node::MemoryRetainer::SelfSize()` is used to report size of additional heap-allocated memory held by the object when we don't want/need to split children nodes, or if that size depends on the state of the retainer.
    - This just needs a method override to return a `size_t`.

Finished: v8::TracedReference<v8::Data> support

Implemented in https://chromium-review.googlesource.com/c/v8/v8/+/5403888

Some classes, like ContextifyScript, hold v8::Global<v8::Data> alive, and these have been particularly tricky to manage to avoid leaks. It could greatly simplify the management if v8::TracedHandle accepts v8::Data classes, and we can use cppgc to manage these references. A preliminary implementation shows that it's possible though it requires quite a bit of

breakages in the v8 API (it should also be possible to create alternative APIs instead of breaking existing signatures). In particular this also needs support for v8::Data in the EmbedderGraph API, which can also be handy for us to track them in the heap snapshot.

Support for Messaging transfer

TBD: we only need to consider this when we start migrating embedder objects that can actually be transferred.

Migrating different types of objects

AliasedBuffer

This is not in the BaseObject hierarchy but it's what many of the BaseObjects references. We can use the internal fields of the TypedArrays to integrate them into cppgc.

BindingData classes

The BindingData classes are effectively all held strongly through the internal binding list in a Node.js context, so they are an easier case of BaseObject.

BaseObjectPtr/BaseObjectWeakPtr

This is, in general, used to maintain BaseObjects managed in the category 2 mentioned in the Background section. We could migrate these to cppgc::Member/cppgc::Persistent and cppgc::WeakMember/cppgc::WeakPersistent.

TBD(joyee): how the replacement should work

BaseObject with no custom destructor

An example can be seen at https://github.com/nodejs/node/pull/51017

BasObject with custom destructor

We need to be careful with the destructor. Since we will not have a cleanup queue again, it should be prepared that an Environment may not be there or the context may not be available when the destructor is run (incidentally this is also required by the ShadowRealm integration) and skip whatever operation that needs a live Environment or active context, like this

```
Environment* env = Environment::GetCurrent(owning_isolate_);
if (env == nullptr) {
  return;
}
// Do cleanup with env e.g. execute some JavaScript with the context
```

This means that cppgc-allocated objects need to store a pointer to the owning Isolate, instead of the owning Environment (what BaseObjects currently do).

AsyncWrap/HandleWrap with active Close

Many of the AsyncWrap/HandleWrap are managed as category 2 mentioned in the Background section. The plan roughly is:

1. Some of the internal ones may be migrated to category 1 if they are always closed right before being released from JS land and there's no way to reach back to them otherwise.
2. In case the active close must come from userland, we can keep the current implementation.
3. We can also devise new APIs that reclaim the native resources upon GC e.g. similar to the FileHandle API, which may also help users address leaks.

TBD(joyee): classify the existing AsyncWrap/HandleWraps

Thread Safety

There are a few constraints we need to consider:

1. CppHeap can only be accessed from one thread at a time
2. Cppgc persistent must be created and destructed in the same thread
3. To retain persistent reference to a cppgc object created from a different thread, use CrossThreadPersistent

To simplify the management, we should only *allocate* cppgc objects from the thread owning the isolate - in general in Node.js we have a 1:1 correspondence between an isolate and its owning thread. The only exception is that with worker threads, it's possible to access a non-owning isolate from the parent/child thread. We need to make sure that they do not *allocate* cppgc objects on the wrong thread (because allocation requires access to the CppHeap allocation handle) - this doesn't seem to happen anywhere across the codebase anyway, but it would be good to have some assertion in place for this.

This means that TransferData and Message should not be allocated with cppgc. They are usually not BaseObject or need to create global handles to JS objects there's not much motivation to migrate them anyway, we can simply leave them as-is.

Test plan

We will need tests for the embedders' use case and the addons' use case: added in #45704. For internal objects being migrated to cppgc, the Node.js core test suites should be enough to catch obvious bugs.

Rollout considerations

ABI compatibility

To allow addons to make use of cppgc, we need to publish the cppgc headers, this may result in more potential ABI breakages to take care of when we upgrade V8.

At the moment, however, the cppgc headers are already quite stable, and do not change more than other V8 headers. To be on the safe side, we could still mark the ABI compatibility of any API in these headers as experimental at the beginning, and make it stable later when we have a better idea about how often breakages actually are. See discussions in v8:14062.

Flags

We can begin the migration with the simpler classes and roll them out onto current releases. If we notice any regression, revert them quickly or consider introducing runtime/configure-time flags to toggle between cppgc/BaseObject management (this would add some complexity to the implementation, so we should defer until there are actually regressions that warrant the complexity).

Appendix: list of native objects and weather they are currently weak persistents

- quic::BindingData: weak
- Http2State: weak
- http_parser::BindingData: weak
- X509Certificate: weak
- SignBase (Sign/Verify): weak
- NativeKeyObject: weak

- KeyObjectHandle: weak
- Hmac: weak
- Hash: weak
- ECDH: weak
- DiffieHellman: weak
- SecureContext: weak
- CipherBase: weak
- WasmStreamingObject: weak
- WASI: weak
- GCProfiler: weak
- NodeCategorySet: weak
- SocketAddressBase: weak
- SocketAddressBlockListWrap: weak
- DeserializerContext: weak
- SerializerContext: weak
- JSTransferable: weak
- ConverterObject: weak
- ConnectionsList: weak
- MicrotaskQueueWrap: weak
- ContextifyScript: weak
- ContextifyContext: weak
- Blob: weak
- HistogramBase: weak

- AsyncWrap: ?
  - ChannelWrap: weak
  - QueryWrap: not weak
  - HandleWrap:
    - FSEventWrap: not weak
    - IntervalHistogram: weak
    - MessagePort: not weak
    - StatWatcher: not weak
    - TraceSigintWatchdog: not weak
    - ProcessWrap: not weak
    - SignalWrap: not weak
    - LibuvStreamWrap: not weak
    - UDPWrap: not weak
    - Endpoint::UDP::Impl: not weak
  - HeapSnapshotStream: weak
  - JSBindingsConnection: not weak
  - JSStream: weak
  - JSUDPWrap: weak
  - Blob::Reader: weak

  - DirHandle: weak
  - FileHandle: weak
  - Parser: not weak (can be?)
  - Http2Stream: weak
  - Http2Session: weak
  - Http2Ping: not weak (can be?)
  - Http2Settings: not weak (can be?)
  - WorkerHeapSnapshotTaker: not weak (can be?)
  - Worker: weak
  - CompressionStream: weak
  - ReqWrap: weak
  - StreamPipe: weak
  - TLSWrap: weak
  - CrytpoJob: weak if sync, ...not weak if async?
  - Endpoint: weak
  - LogStream: weak
  - quic::Session: weak
  - quic::Stream: weak
- SnapshotableObject:
  - timers::BindingData: weak
  - v8::BindingData: weak
  - util::WeakReference: weak
  - url::BindingData: weak
  - process::BindingData: weak
  - fs::BindingData: weak
  - BlobBindingData: weak
  - encoding_binding: weak
- CompiledFnEntry: is not actually necessary
- ModuleWrap: could be made weak with https://github.com/nodejs/node/pull/48510
- TLSContext: weak? from quic::Session