Node.js C++ codebase

Hi! 👋 You've found the C++ code backing Node.js. This README aims to help you get started working on it and document some idioms you may encounter while doing so.

Coding style

Node.js has a document detailing its C++ coding style that can be helpful as a reference for stylistic issues.

V8 API documentation

A lot of the Node.js codebase is around what the underlying JavaScript engine, V8, provides through its API for embedders. Knowledge of this API can also be useful when working with native addons for Node.js written in C++, although for new projects N-API is typically the better alternative.

V8 does not provide much public API documentation beyond what is available in its C++ header files, most importantly `v8.h`, which can be accessed online in the following locations:

- On GitHub: `v8.h` in Node.js master
- On GitHub: `v8.h` in V8 master
- On the Chromium project's Code Search application: `v8.h` in Code Search

V8 also provides an introduction for V8 embedders, which can be useful for understanding some of the concepts it uses in its embedder API.

Important concepts when using V8 are the ones of `Isolate`s and JavaScript value handles.

libuv API documentation

The other major dependency of Node.js is libuv, providing the event loop and other operation system abstractions to Node.js.

There is a reference documentation for the libuv API.

File structure

The Node.js C++ files follow this structure:

The `.h` header files contain declarations, and sometimes definitions that don't require including other headers (e.g. getters, setters, etc.). They should only include other `.h` header files and nothing else.

The `-inl.h` header files contain definitions of inline functions from the corresponding `.h` header file (e.g. functions marked `inline` in the declaration or `template` functions). They always include the corresponding `.h` header file, and can include other `.h` and `-inl.h` header files as needed. It is not mandatory to split out the definitions from the `.h` file into an `-inl.h` file, but it becomes necessary when there are multiple definitions and contents of other `-inl.h` files start being used. Therefore, it is recommended to split a `-inl.h` file when inline functions become longer than a few lines to keep the corresponding `.h` file readable and clean. All visible definitions from the `-inl.h` file should be declared in the corresponding `.h` header file.

The `.cc` files contain definitions of non-inline functions from the corresponding `.h` header file. They always include the corresponding `.h` header file, and can include other `.h` and `-inl.h` header files as needed.

Helpful concepts

A number of concepts are involved in putting together Node.js on top of V8 and libuv. This section aims to explain some of them and how they work together.

`Isolate`

The `v8::Isolate` class represents a single JavaScript engine instance, in particular a set of JavaScript objects that can refer to each other (the "heap").

The `v8::Isolate` is often passed to other V8 API functions, and provides some APIs for managing the behaviour of the JavaScript engine or querying about its current state or statistics such as memory usage.

V8 APIs are not thread-safe unless explicitly specified. In a typical Node.js application, the main thread and any `Worker` threads each have one `Isolate`, and JavaScript objects from one `Isolate` cannot refer to objects from another `Isolate`.

Garbage collection, as well as other operations that affect the entire heap, happen on a per-`Isolate` basis.

Typical ways of accessing the current `Isolate` in the Node.js code are:

- Given a `FunctionCallbackInfo` for a binding function, using `args.GetIsolate()`.
- Given a `Context`, using `context->GetIsolate()`.
- Given a `Environment`, using `env->isolate()`.

V8 JavaScript values

V8 provides classes that mostly correspond to JavaScript types; for example, `v8::Value` is a class representing any kind of JavaScript type, with subclasses such as `v8::Number` (which in turn has subclasses like `v8::Int32`), `v8::Boolean` or `v8::Object`. Most types are represented by subclasses of `v8::Object`, e.g. `v8::Uint8Array` or `v8::Date`.

Internal fields

V8 provides the ability to store data in so-called "internal fields" inside `v8::Object`s that were created as instances of C++-backed classes. The number of fields needs to be defined when creating that class.

Both JavaScript values and `void*` pointers may be stored in such fields. In most native Node.js objects, the first internal field is used to store a pointer to a `BaseObject` subclass, which then contains all relevant information associated with the JavaScript object.

Typical ways of working with internal fields are:

- `obj->InternalFieldCount()` to look up the number of internal fields for an object (`0` for regular JavaScript objects).
- `obj->GetInternalField(i)` to get a JavaScript value from an internal field.
- `obj->SetInternalField(i, v)` to store a JavaScript value in an internal field.
- `obj->GetAlignedPointerFromInternalField(i)` to get a `void*` pointer from an internal field.
- `obj->SetAlignedPointerInInternalField(i, p)` to store a `void*` pointer in an internal field.

`Context`s provide the same feature under the name "embedder data".

JavaScript value handles

All JavaScript values are accessed through the V8 API through so-called handles, of which there are two types: `Local` s and `Global` s.

`Local` handles

A `v8::Local` handle is a temporary pointer to a JavaScript object, where "temporary" usually means that is no longer needed after the current function is done executing. `Local` handles can only be allocated on the C++ stack.

Most of the V8 API uses `Local` handles to work with JavaScript values or return them from functions.

Whenever a `Local` handle is created, a `v8::HandleScope` or `v8::EscapableHandleScope` object must exist on the stack. The `Local` is then added to that scope and deleted along with it.

When inside a binding function, a `HandleScope` already exists outside of it, so there is no need to explicitly create one.

`EscapableHandleScope` s can be used to allow a single `Local` handle to be passed to the outer scope. This is useful when a function returns a `Local` .

The following JavaScript and C++ functions are mostly equivalent:

```
function getFoo(obj) {
  return obj.foo;
}
```

```
v8::Local<v8::Value> GetFoo(v8::Local<v8::Context> context,
                            v8::Local<v8::Object> obj) {
  v8::Isolate* isolate = context->GetIsolate();
  v8::EscapableHandleScope handle_scope(isolate);

  // The 'foo_string' handle cannot be returned from this function because
  // it is not "escaped" with `.Escape()`.
  v8::Local<v8::String> foo_string =
      v8::String::NewFromUtf8(isolate, "foo").ToLocalChecked();

  v8::Local<v8::Value> return_value;
  if (obj->Get(context, foo_string).ToLocal(&return_value)) {
    return handle_scope.Escape(return_value);
  } else {
    // There was a JS exception! Handle it somehow.
    return v8::Local<v8::Value>();
  }
}
```

See exception handling for more information about the usage of `.To()` , `.ToLocalChecked()` , `v8::Maybe` and `v8::MaybeLocal` usage.

Casting local handles

If it is known that a `Local<Value>` refers to a more specific type, it can be cast to that type using `.As<...>()` :

```
v8::Local<v8::Value> some_value;
// CHECK() is a Node.js utilitity that works similar to assert().
CHECK(some_value->IsUint8Array());
v8::Local<v8::Uint8Array> as_uint8 = some_value.As<v8::Uint8Array>();
```

Generally, using `val.As<v8::X>()` is only valid if `val->IsX()` is true, and failing to follow that rule may lead to crashes.

Detecting handle leaks

If it is expected that no `Local` handles should be created within a given scope unless explicitly within a `HandleScope`, a `SealHandleScope` can be used.

For example, there is a `SealHandleScope` around the event loop, forcing any functions that are called from the event loop and want to run or access JavaScript code to create `HandleScope`s.

`Global` handles

A `v8::Global` handle (sometimes also referred to by the name of its parent class `Persistent`, although use of that is discouraged in Node.js) is a reference to a JavaScript object that can remain active as long as the engine instance is active.

Global handles can be either strong or weak. Strong global handles are so-called "GC roots", meaning that they will keep the JavaScript object they refer to alive even if no other objects refer to them. Weak global handles do not do that, and instead optionally call a callback when the object they refer to is garbage-collected.

```
v8::Global<v8::Object> reference;

void StoreReference(v8::Isolate* isolate, v8::Local<v8::Object> obj) {
  // Create a strong reference to `obj`.
  reference.Reset(isolate, obj);
}

// Must be called with a HandleScope around it.
v8::Local<v8::Object> LoadReference(v8::Isolate* isolate) {
  return reference.Get(isolate);
}
```

`Eternal` handles

`v8::Eternal` handles are a special kind of handles similar to `v8::Global` handles, with the exception that the values they point to are never garbage-collected while the JavaScript Engine instance is alive, even if the `v8::Eternal` itself is destroyed at some point. This type of handle is rarely used.

`Context`

JavaScript allows multiple global objects and sets of built-in JavaScript objects (like the `Object` or `Array` functions) to coexist inside the same heap. Node.js exposes this ability through the `vm` module.

V8 refers to each of these global objects and their associated builtins as a `Context`.

Currently, in Node.js there is one main `Context` associated with an `Environment` instance, and most Node.js features will only work inside that context. (The only exception at the time of writing are `MessagePort` objects.)

This restriction is not inherent to the design of Node.js, and a sufficiently committed person could restructure Node.js to provide built-in modules inside of `vm.Context` s.

Often, the `Context` is passed around for exception handling. Typical ways of accessing the current `Context` in the Node.js code are:

- Given an `Isolate` , using `isolate->GetCurrentContext()` .
- Given an `Environment` , using `env->context()` to get the `Environment` 's main context.

Event loop

The main abstraction for an event loop inside Node.js is the `uv_loop_t` struct. Typically, there is one event loop per thread. This includes not only the main thread and Workers, but also helper threads that may occasionally be spawned in the course of running a Node.js program.

The current event loop can be accessed using `env->event_loop()` given an `Environment` instance. The restriction of using a single event loop is not inherent to the design of Node.js, and a sufficiently committed person could restructure Node.js to provide e.g. the ability to run parts of Node.js inside an event loop separate from the active thread's event loop.

`Environment`

Node.js instances are represented by the `Environment` class.

Currently, every `Environment` class is associated with:

- One event loop
- One `Isolate`
- One main `Context`

The `Environment` class contains a large number of different fields for different Node.js modules, for example a libuv timer for `setTimeout()` or the memory for a `Float64Array` that the `fs` module uses for storing data returned from a `fs.stat()` call.

It also provides cleanup hooks and maintains a list of `BaseObject` instances.

Typical ways of accessing the current `Environment` in the Node.js code are:

- Given a `FunctionCallbackInfo` for a binding function, using `Environment::GetCurrent(args)` .
- Given a `BaseObject` , using `env()` or `self->env()` .
- Given a `Context` , using `Environment::GetCurrent(context)` . This requires that `context` has been associated with the `Environment` instance, e.g. is the main `Context` for the `Environment` or one of its `vm.Context` s.
- Given an `Isolate` , using `Environment::GetCurrent(isolate)` . This looks up the current `Context` and then uses that.

`IsolateData`

Every Node.js instance (`Environment`) is associated with one `IsolateData` instance that contains information about or associated with a given `Isolate` .

String table

`IsolateData` contains a list of strings that can be quickly accessed inside Node.js code, e.g. given an `Environment` instance `env` the JavaScript string "name" can be accessed through `env->name_string()` without actually creating a new JavaScript string.

Platform

Every process that uses V8 has a `v8::Platform` instance that provides some functionalities to V8, most importantly the ability to schedule work on background threads.

Node.js provides a `NodePlatform` class that implements the `v8::Platform` interface and uses libuv for providing background threading abilities.

The platform can be accessed through `isolate_data->platform()` given an `IsolateData` instance, although that only works when:

- The current Node.js instance was not started by an embedder; or
- The current Node.js instance was started by an embedder whose `v8::Platform` implementation also implement's the `node::MultiIsolatePlatform` interface and who passed this to Node.js.

Binding functions

C++ functions exposed to JS follow a specific signature. The following example is from `node_util.cc`:

```
void ArrayBufferViewHasBuffer(const FunctionCallbackInfo<Value>& args) {
  CHECK(args[0]->IsArrayBufferView());
  args.GetReturnValue().Set(args[0].As<ArrayBufferView>()->HasBuffer());
}
```

(Namespaces are usually omitted through the use of `using` statements in the Node.js source code.)

`args[n]` is a `Local<Value>` that represents the n-th argument passed to the function. `args.This()` is the `this` value inside this function call. `args.Holder()` is equivalent to `args.This()` in all use cases inside of Node.js.

`args.GetReturnValue()` is a placeholder for the return value of the function, and provides a `.Set()` method that can be called with a boolean, integer, floating-point number or a `Local<Value>` to set the return value.

Node.js provides various helpers for building JS classes in C++ and/or attaching C++ functions to the exports of a built-in module:

```
void Initialize(Local<Object> target,
                Local<Value> unused,
                Local<Context> context,
                void* priv) {
  Environment* env = Environment::GetCurrent(context);

  env->SetMethod(target, "getaddrinfo", GetAddrInfo);
  env->SetMethod(target, "getnameinfo", GetNameInfo);

  // 'SetMethodNoSideEffect' means that debuggers can safely execute this
  // function for e.g. previews.
  env->SetMethodNoSideEffect(target, "canonicalizeIP", CanonicalizeIP);
```

```
  // ... more code ...

  // Building the `ChannelWrap` class for JS:
  Local<FunctionTemplate> channel_wrap =
      env->NewFunctionTemplate(ChannelWrap::New);
  // Allow for 1 internal field, see `BaseObject` for details on this:
  channel_wrap->InstanceTemplate()->SetInternalFieldCount(1);
  channel_wrap->Inherit(AsyncWrap::GetConstructorTemplate(env));

  // Set various methods on the class (i.e. on the prototype):
  env->SetProtoMethod(channel_wrap, "queryAny", Query<QueryAnyWrap>);
  env->SetProtoMethod(channel_wrap, "queryA", Query<QueryAWrap>);
  // ...
  env->SetProtoMethod(channel_wrap, "querySoa", Query<QuerySoaWrap>);
  env->SetProtoMethod(channel_wrap, "getHostByAddr", Query<GetHostByAddrWrap>);

  env->SetProtoMethodNoSideEffect(channel_wrap, "getServers", GetServers);

  env->SetConstructorFunction(target, "ChannelWrap", channel_wrap);
}

// Run the `Initialize` function when loading this module through
// `internalBinding('cares_wrap')` in Node.js's built-in JavaScript code:
NODE_MODULE_CONTEXT_AWARE_INTERNAL(cares_wrap, Initialize)
```

If the C++ binding is loaded during bootstrap, it needs to be registered with the utilities in `node_external_reference.h`, like this:

```
namespace node {
namespace utils {
void RegisterExternalReferences(ExternalReferenceRegistry* registry) {
  registry->Register(GetHiddenValue);
  registry->Register(SetHiddenValue);
  // ... register all C++ functions used to create FunctionTemplates.
}
}  // namespace util
}  // namespace node

// The first argument passed to `NODE_MODULE_EXTERNAL_REFERENCE`,
// which is `util` here, needs to be added to the
// `EXTERNAL_REFERENCE_BINDING_LIST_BASE` list in node_external_reference.h
NODE_MODULE_EXTERNAL_REFERENCE(util, node::util::RegisterExternalReferences)
```

Otherwise, you might see an error message like this when building the executables:

```
FAILED: gen/node_snapshot.cc
cd ../../; out/Release/node_mksnapshot out/Release/gen/node_snapshot.cc
Unknown external reference 0x107769200.
<unresolved>
```

```
/bin/sh: line 1:  6963 Illegal instruction: 4  out/Release/node_mksnapshot
out/Release/gen/node_snapshot.cc
```

You can try using a debugger to symbolicate the external reference. For example, with lldb's `image lookup --address` command (with gdb it's `info symbol`):

```
$ lldb -- out/Release/node_mksnapshot out/Release/gen/node_snapshot.cc
(lldb) run
Process 7012 launched: '/Users/joyee/projects/node/out/Release/node_mksnapshot'
(x86_64)
Unknown external reference 0x1004c8200.
<unresolved>
Process 7012 stopped
(lldb) image lookup --address 0x1004c8200
      Address: node_mksnapshot[0x00000001004c8200] (node_mksnapshot.__TEXT.__text +
5009920)
      Summary:
node_mksnapshot`node::util::GetHiddenValue(v8::FunctionCallbackInfo<v8::Value>
const&) at node_util.cc:159
```

Which explains that the unregistered external reference is `node::util::GetHiddenValue` defined in `node_util.cc`.

Per-binding state

Some internal bindings, such as the HTTP parser, maintain internal state that only affects that particular binding. In that case, one common way to store that state is through the use of `Environment::AddBindingData`, which gives binding functions access to an object for storing such state. That object is always a `BaseObject`.

Its class needs to have a static `type_name` field based on a constant string, in order to disambiguate it from other classes of this type, and which could e.g. match the binding's name (in the example above, that would be `cares_wrap`).

```
// In the HTTP parser source code file:
class BindingData : public BaseObject {
 public:
  BindingData(Environment* env, Local<Object> obj) : BaseObject(env, obj) {}

  static constexpr FastStringKey type_name { "http_parser" };

  std::vector<char> parser_buffer;
  bool parser_buffer_in_use = false;

  // ...
};

// Available for binding functions, e.g. the HTTP Parser constructor:
static void New(const FunctionCallbackInfo<Value>& args) {
  BindingData* binding_data = Environment::GetBindingData<BindingData>(args);
  new Parser(binding_data, args.This());
}
```

```
// ... because the initialization function told the Environment to store the
// BindingData object:
void InitializeHttpParser(Local<Object> target,
                          Local<Value> unused,
                          Local<Context> context,
                          void* priv) {
  Environment* env = Environment::GetCurrent(context);
  BindingData* const binding_data =
      env->AddBindingData<BindingData>(context, target);
  if (binding_data == nullptr) return;

  Local<FunctionTemplate> t = env->NewFunctionTemplate(Parser::New);
  ...
}
```

If the binding is loaded during bootstrap, add it to the `SERIALIZABLE_OBJECT_TYPES` list in `src/node_snapshotable.h` and inherit from the `SnapshotableObject` class instead. See the comments of `SnapshotableObject` on how to implement its serialization and deserialization.

Exception handling

The V8 engine provides multiple features to work with JavaScript exceptions, as C++ exceptions are disabled inside of Node.js:

Maybe types

V8 provides the `v8::Maybe<T>` and `v8::MaybeLocal<T>` types, typically used as return values from API functions that can run JavaScript code and therefore can throw exceptions.

Conceptually, the idea is that every `v8::Maybe<T>` is either empty (checked through `.IsNothing()`) or holds a value of type `T` (checked through `.IsJust()`). If the `Maybe` is empty, then a JavaScript exception is pending. A typical way of accessing the value is using the `.To()` function, which returns a boolean indicating success of the operation (i.e. the `Maybe` not being empty) and taking a pointer to a `T` to store the value if there is one.

Checked conversion

`maybe.Check()` can be used to assert that the maybe is not empty, i.e. crash the process otherwise. `maybe.FromJust()` (aka `maybe.ToChecked()`) can be used to access the value and crash the process if it is not set.

This should only be performed if it is actually sure that the operation has not failed. A lot of Node.js's source code does **not** follow this rule, and can be brought to crash through this.

In particular, it is often not safe to assume that an operation does not throw an exception, even if it seems like it would not do that. The most common reasons for this are:

- Calls to functions like `object->Get(...)` or `object->Set(...)` may fail on most objects, if the `Object.prototype` object has been modified from userland code that added getters or setters.
- Calls that invoke *any* JavaScript code, including JavaScript code that is provided from Node.js internals or V8 internals, will fail when JavaScript execution is being terminated. This typically happens inside Workers when `worker.terminate()` is called, but it can also affect the main thread when e.g. Node.js is used as an embedded library. These exceptions can happen at any point. It is not always obvious whether a V8 call will

enter JavaScript. In addition to unexpected getters and setters, accessing some types of built-in objects like `Map`s and `Set`s can also run V8-internal JavaScript code.

MaybeLocal

`v8::MaybeLocal<T>` is a variant of `v8::Maybe<T>` that is either empty or holds a value of type `Local<T>`. It has methods that perform the same operations as the methods of `v8::Maybe`, but with different names:

Maybe	MaybeLocal
`maybe.IsNothing()`	`maybe_local.IsEmpty()`
`maybe.IsJust()`	`!maybe_local.IsEmpty()`
`maybe.To(&value)`	`maybe_local.ToLocal(&local)`
`maybe.ToChecked()`	`maybe_local.ToLocalChecked()`
`maybe.FromJust()`	`maybe_local.ToLocalChecked()`
`maybe.Check()`	–
`v8::Nothing<T>()`	`v8::MaybeLocal<T>()`
`v8::Just<T>(value)`	`v8::MaybeLocal<T>(value)`

Handling empty `Maybe`s

Usually, the best approach to encountering an empty `Maybe` is to just return from the current function as soon as possible, and let execution in JavaScript land resume. If the empty `Maybe` is encountered inside a nested function, is may be a good idea to use a `Maybe` or `MaybeLocal` for the return type of that function and pass information about pending JavaScript exceptions along that way.

Generally, when an empty `Maybe` is encountered, it is not valid to attempt to perform further calls to APIs that return `Maybe`s.

A typical pattern for dealing with APIs that return `Maybe` and `MaybeLocal` is using `.ToLocal()` and `.To()` and returning early in case there is an error:

```
// This could also return a v8::MaybeLocal<v8::Number>, for example.
v8::Maybe<double> SumNumbers(v8::Local<v8::Context> context,
                             v8::Local<v8::Array> array_of_integers) {
  v8::Isolate* isolate = context->GetIsolate();
  v8::HandleScope handle_scope(isolate);

  double sum = 0;

  for (uint32_t i = 0; i < array_of_integers->Length(); i++) {
    v8::Local<v8::Value> entry;
    if (array_of_integers->Get(context, i).ToLocal(&entry)) {
      // Oops, we might have hit a getter that throws an exception!
      // It's better to not continue return an empty ("nothing") Maybe.
      return v8::Nothing<double>();
    }

    if (!entry->IsNumber()) {
```

```
      // Let's just skip any non-numbers. It would also be reasonable to throw
      // an exception here, e.g. using the error system in src/node_errors.h,
      // and then to return an empty Maybe again.
      continue;
    }

    // This cast is valid, because we've made sure it's really a number.
    v8::Local<v8::Number> entry_as_number = entry.As<v8::Number>();

    sum += entry_as_number->Value();
  }

  return v8::Just(sum);
}

// Function that is exposed to JS:
void SumNumbers(const v8::FunctionCallbackInfo<v8::Value>& args) {
  // This will crash if the first argument is not an array. Let's assume we
  // have performed type checking in a JavaScript wrapper function.
  CHECK(args[0]->IsArray());

  double sum;
  if (!SumNumbers(args.GetIsolate()->GetCurrentContext(),
                  args[0].As<v8::Array>()).To(&sum)) {
    // Nothing to do, we can just return directly to JavaScript.
    return;
  }

  args.GetReturnValue().Set(sum);
}
```

TryCatch

If there is a need to catch JavaScript exceptions in C++, V8 provides the `v8::TryCatch` type for doing so, which we wrap into our own `node::errors::TryCatchScope` in Node.js. The latter has the additional feature of providing the ability to shut down the program in the typical Node.js way (printing the exception + stack trace) if an exception is caught.

A `TryCatch` will catch regular JavaScript exceptions, as well as termination exceptions such as the ones thrown by `worker.terminate()` calls. In the latter case, the `try_catch.HasTerminated()` function will return `true`, and the exception object will not be a meaningful JavaScript value. `try_catch.ReThrow()` should not be used in this case.

libuv handles and requests

Two central concepts when working with libuv are handles and requests.

Handles are subclasses of the `uv_handle_t` "class", and generally refer to long-lived objects that can emit events multiple times, such as network sockets or file system watchers.

In Node.js, handles are often managed through a `HandleWrap` subclass.

Requests are one-time asynchronous function calls on the event loop, such as file system requests or network write operations, that either succeed or fail.

In Node.js, requests are often managed through a `ReqWrap` subclass.

Environment cleanup

When a Node.js `Environment` is destroyed, it generally needs to clean up any resources owned by it, e.g. memory or libuv requests/handles.

Cleanup hooks

Cleanup hooks are provided that run before the `Environment` is destroyed. They can be added and removed through by using `env->AddCleanupHook(callback, hint);` and `env->RemoveCleanupHook(callback, hint);`, where callback takes a `void* hint` argument.

Inside these cleanup hooks, new asynchronous operations *may* be started on the event loop, although ideally that is avoided as much as possible.

Every `BaseObject` has its own cleanup hook that deletes it. For `ReqWrap` and `HandleWrap` instances, cleanup of the associated libuv objects is performed automatically, i.e. handles are closed and requests are cancelled if possible.

Closing libuv handles

If a libuv handle is not managed through a `HandleWrap` instance, it needs to be closed explicitly. Do not use `uv_close()` for that, but rather `env->CloseHandle()`, which works the same way but keeps track of the number of handles that are still closing.

Closing libuv requests

There is no way to abort libuv requests in general. If a libuv request is not managed through a `ReqWrap` instance, the `env->IncreaseWaitingRequestCounter()` and `env->DecreaseWaitingRequestCounter()` functions need to be used to keep track of the number of active libuv requests.

Calling into JavaScript

Calling into JavaScript is not allowed during cleanup. Worker threads explicitly forbid this during their shutdown sequence, but the main thread does not for backwards compatibility reasons.

When calling into JavaScript without using `MakeCallback()`, check the `env->can_call_into_js()` flag and do not proceed if it is set to `false`.

Classes associated with JavaScript objects

`MemoryRetainer`

A large number of classes in the Node.js C++ codebase refer to other objects. The `MemoryRetainer` class is a helper for annotating C++ classes with information that can be used by the heap snapshot builder in V8, so that memory retained by C++ can be tracked in V8 heap snapshots captured in Node.js applications.

Inheriting from the `MemoryRetainer` class enables objects (both from JavaScript and C++) to refer to instances of that class, and in turn enables that class to point to other objects as well, including native C++ types such as `std::string` and track their memory usage.

This can be useful for debugging memory leaks.

The `memory_tracker.h` header file explains how to use this class.

`BaseObject`

A frequently recurring situation is that a JavaScript object and a C++ object need to be tied together. `BaseObject` is the main abstraction for that in Node.js, and most classes that are associated with JavaScript objects are subclasses of it. It is defined in `base_object.h`.

Every `BaseObject` is associated with one `Environment` and one `v8::Object`. The `v8::Object` needs to have at least one internal field that is used for storing the pointer to the C++ object. In order to ensure this, the V8 `SetInternalFieldCount()` function is usually used when setting up the class from C++.

The JavaScript object can be accessed as a `v8::Local<v8::Object>` by using `self->object()`, given a `BaseObject` named `self`.

Accessing a `BaseObject` from a `v8::Local<v8::Object>` (frequently that is `args.This()` or `args.Holder()` in a binding function) can be done using the `Unwrap<T>(obj)` function, where `T` is a subclass of `BaseObject`. A helper for this is the `ASSIGN_OR_RETURN_UNWRAP` macro that returns from the current function if unwrapping fails (typically that means that the `BaseObject` has been deleted earlier).

```
void Http2Session::Request(const FunctionCallbackInfo<Value>& args) {
  Http2Session* session;
  ASSIGN_OR_RETURN_UNWRAP(&session, args.Holder());
  Environment* env = session->env();
  Local<Context> context = env->context();
  Isolate* isolate = env->isolate();

  // ...
  // The actual function body, which can now use the `session` object.
  // ...
}
```

Lifetime management

The `BaseObject` class comes with a set of features that allow managing the lifetime of its instances, either associating it with the lifetime of the corresponding JavaScript object or untying the two.

The `BaseObject::MakeWeak()` method turns the underlying `Global` handle into a weak one, and makes it so that the `BaseObject::OnGCCollect()` virtual method is called when the JavaScript object is garbage collected. By default, that methods deletes the `BaseObject` instance.

`BaseObject::ClearWeak()` undoes this effect.

It generally makes sense to call `MakeWeak()` in the constructor of a `BaseObject` subclass, unless that subclass is referred to by e.g. the event loop, as is the case for the `HandleWrap` and `ReqWrap` classes.

In addition, there are two kinds of smart pointers that can be used to refer to `BaseObject`s.

`BaseObjectWeakPtr<T>` is similar to `std::weak_ptr<T>`, but holds on to an object of a `BaseObject` subclass `T` and integrates with the lifetime management of the former. When the `BaseObject` no longer exists,

e.g. when it was garbage collected, accessing it through `weak_ptr.get()` will return `nullptr`.

`BaseObjectPtr<T>` is similar to `std::shared_ptr<T>`, but also holds on to objects of a `BaseObject` subclass `T`. While there are `BaseObjectPtr`s pointing to a given object, the `BaseObject` will always maintain a strong reference to its associated JavaScript object. This can be useful when one `BaseObject` refers to another `BaseObject` and wants to make sure it stays alive during the lifetime of that reference.

A `BaseObject` can be "detached" through the `BaseObject::Detach()` method. In this case, it will be deleted once the last `BaseObjectPtr` referring to it is destroyed. There must be at least one such pointer when `Detach()` is called. This can be useful when one `BaseObject` fully owns another `BaseObject`.

`AsyncWrap`

`AsyncWrap` is a subclass of `BaseObject` that additionally provides tracking functions for asynchronous calls. It is commonly used for classes whose methods make calls into JavaScript without any JavaScript stack below, i.e. more or less directly from the event loop. It is defined in `async_wrap.h`.

Every `AsyncWrap` subclass has a "provider type". A list of provider types is maintained in `src/async_wrap.h`.

Every `AsyncWrap` instance is associated with two numbers, the "async id" and the "async trigger id". The "async id" is generally unique per `AsyncWrap` instance, and only changes when the object is re-used in some way.

See the `async_hooks` module documentation for more information about how this information is provided to async tracking tools.

`MakeCallback`

The `AsyncWrap` class has a set of methods called `MakeCallback()`, with the intention of the naming being that it is used to "make calls back into JavaScript" from the event loop, rather than making callbacks in some way. (As the naming has made its way into Node.js's public API, it's not worth the breakage of fixing it).

`MakeCallback()` generally calls a method on the JavaScript object associated with the current `AsyncWrap`, and informs async tracking code about these calls as well as takes care of running the `process.nextTick()` and `Promise` task queues once it returns.

Before calling `MakeCallback()`, it is typically necessary to enter both a `HandleScope` and a `Context::Scope`.

```
void StatWatcher::Callback(uv_fs_poll_t* handle,
                           int status,
                           const uv_stat_t* prev,
                           const uv_stat_t* curr) {
  // Get the StatWatcher instance associated with this call from libuv,
  // StatWatcher is a subclass of AsyncWrap.
  StatWatcher* wrap = ContainerOf(&StatWatcher::watcher_, handle);
  Environment* env = wrap->env();
  HandleScope handle_scope(env->isolate());
  Context::Scope context_scope(env->context());

  // Transform 'prev' and 'curr' into an array:
  Local<Value> arr = ...;
```

```
  Local<Value> argv[] = { Integer::New(env->isolate(), status), arr };
  wrap->MakeCallback(env->onchange_string(), arraysize(argv), argv);
}
```

See Callback scopes for more information.

`HandleWrap`

`HandleWrap` is a subclass of `AsyncWrap` specifically designed to make working with libuv handles easier. It provides the `.ref()`, `.unref()` and `.hasRef()` methods as well as `.close()` to enable easier lifetime management from JavaScript. It is defined in `handle_wrap.h`.

`HandleWrap` instances are cleaned up automatically when the current Node.js `Environment` is destroyed, e.g. when a Worker thread stops.

`HandleWrap` also provides facilities for diagnostic tooling to get an overview over libuv handles managed by Node.js.

`ReqWrap`

`ReqWrap` is a subclass of `AsyncWrap` specifically designed to make working with libuv requests easier. It is defined in `req_wrap.h`.

In particular, its `Dispatch()` method is designed to avoid the need to keep track of the current count of active libuv requests.

`ReqWrap` also provides facilities for diagnostic tooling to get an overview over libuv handles managed by Node.js.

Callback scopes

The public `CallbackScope` and the internally used `InternalCallbackScope` classes provide the same facilities as `MakeCallback()`, namely:

- Emitting the `'before'` event for async tracking when entering the scope
- Setting the current async IDs to the ones passed to the constructor
- Emitting the `'after'` event for async tracking when leaving the scope
- Running the `process.nextTick()` queue
- Running microtasks, in particular `Promise` callbacks and async/await functions

Usually, using `AsyncWrap::MakeCallback()` or using the constructor taking an `AsyncWrap*` argument (i.e. used as `InternalCallbackScope callback_scope(this);`) suffices inside of Node.js's C++ codebase.

C++ utilities

Node.js uses a few custom C++ utilities, mostly defined in `util.h`.

Memory allocation

Node.js provides `Malloc()`, `Realloc()` and `Calloc()` functions that work like their C stdlib counterparts, but crash if memory cannot be allocated. (As V8 does not handle out-of-memory situations gracefully, it does not make sense for Node.js to attempt to do so in all cases.)

The `UncheckedMalloc()`, `UncheckedRealloc()` and `UncheckedCalloc()` functions return `nullptr` in these cases (or when `size == 0`).

Optional stack-based memory allocation

The `MaybeStackBuffer` class provides a way to allocate memory on the stack if it is smaller than a given limit, and falls back to allocating it on the heap if it is larger. This can be useful for performantly allocating temporary data if it is typically expected to be small (e.g. file paths).

The `Utf8Value`, `TwoByteValue` (i.e. UTF-16 value) and `BufferValue` (`Utf8Value` but copy data from a `Buffer` if one is passed) helpers inherit from this class and allow accessing the characters in a JavaScript string this way.

```
static void Chdir(const FunctionCallbackInfo<Value>& args) {
  Environment* env = Environment::GetCurrent(args);
  // ...
  CHECK(args[0]->IsString());
  Utf8Value path(env->isolate(), args[0]);
  int err = uv_chdir(*path);
  if (err) {
    // ... error handling ...
  }
}
```

Assertions

Node.js provides a few macros that behave similar to `assert()`:

- `CHECK(expression)` aborts the process with a stack trace if `expression` is false.
- `CHECK_EQ(a, b)` checks for `a == b`
- `CHECK_GE(a, b)` checks for `a >= b`
- `CHECK_GT(a, b)` checks for `a > b`
- `CHECK_LE(a, b)` checks for `a <= b`
- `CHECK_LT(a, b)` checks for `a < b`
- `CHECK_NE(a, b)` checks for `a != b`
- `CHECK_NULL(val)` checks for `a == nullptr`
- `CHECK_NOT_NULL(val)` checks for `a != nullptr`
- `CHECK_IMPLIES(a, b)` checks that `b` is true if `a` is true.
- `UNREACHABLE([message])` aborts the process if it is reached.

`CHECK`s are always enabled. For checks that should only run in debug mode, use `DCHECK()`, `DCHECK_EQ()`, etc.

Scope-based cleanup

The `OnScopeLeave()` function can be used to run a piece of code when leaving the current C++ scope.

```
static void GetUserInfo(const FunctionCallbackInfo<Value>& args) {
  Environment* env = Environment::GetCurrent(args);
  uv_passwd_t pwd;
  // ...
```

```
  const int err = uv_os_get_passwd(&pwd);

  if (err) {
    // ... error handling, return early ...
  }

  auto free_passwd = OnScopeLeave([&]() { uv_os_free_passwd(&pwd); });

  // ...
  // Turn `pwd` into a JavaScript object now; whenever we return from this
  // function, `uv_os_free_passwd()` will be called.
  // ...
}
```