Bootstrap and Bindings

2024, OSSCA, Node.js - Code and Learn

Daeyeon Jeong

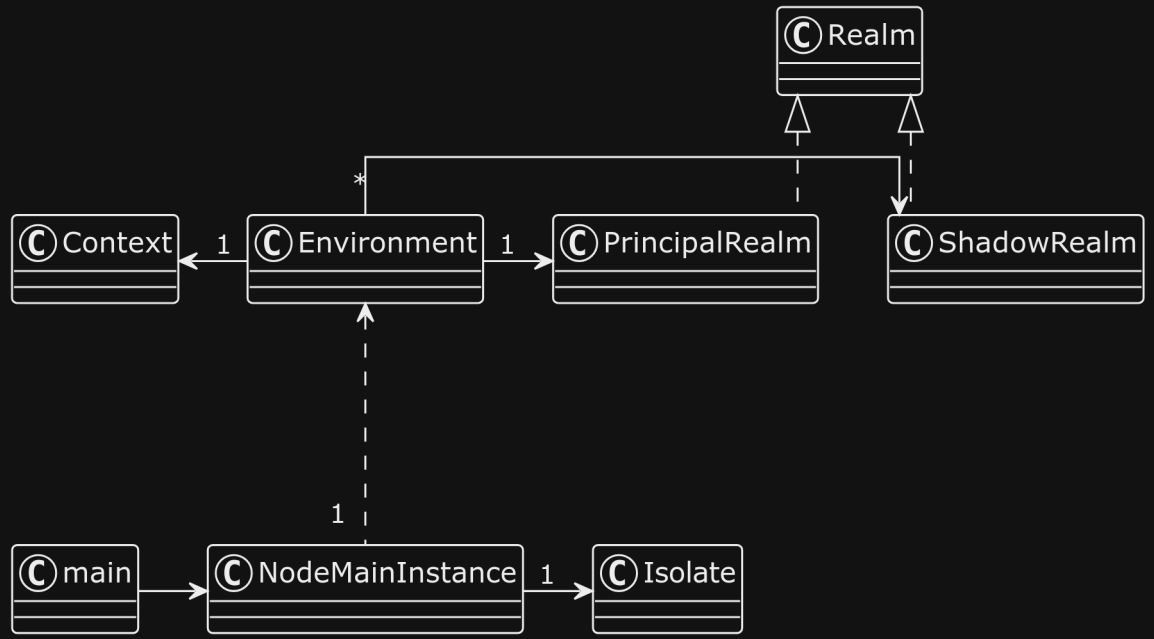
Index

- Brief Node.js Bootstrap
- Binding C++ and JavaScript

Reminder

```cpp
// src/node_main_instance.cc
ExitCode NodeMainInstance::Run() {
  // Sets the isolate and creates a handle scope for V8 handle.
  Isolate::Scope isolate_scope(isolate_);
  HandleScope handle_scope(isolate_);

  ExitCode exit_code = ExitCode::kNoFailure;

  // 1. CreateMainEnvironment initializes an environment for main thread.
  //    - CreateEnvironment (Setting up context and state for execution)
  DeleteFnPtr<Environment, FreeEnvironment> env = CreateMainEnvironment(&exit_code);
  CHECK_NOT_NULL(env);

  // Sets the context created in CreateMainEnvironment for this scope.
  Context::Scope context_scope(env->context());

  // 2. Run starts the execution of the environment, which involves:
  //    - LoadEnvironment (Loading modules and setting up runtime)
  //    - SpinEventLoopInternal (Running the event loop)
  Run(&exit_code, env.get());
  return exit_code;
}
```

NodeMainInstance



Bootstrap

CreateMainEnvironment

- CreateEnvironment
  - env->RunBootstrapping()
    - Realm::RunBootstrapping
      - `internal/bootstrap/realm`
        - `internalBinding(...)` : private internal C++ binding loader
    - Realm::BootstrapRealm (src/node_realm.cc)
      - `internal/bootstrap/node`
      - `internal/bootstrap/switches/is_main_thread`
        - `internal/modules/cjs/loader`
        - `internal/modules/run_main`
        - `internal/process/pre_execution` (loader, global objects...)

Bootstrap

Run(&exit_code, env.get());

- LoadEnvironment
  - StartExecution
    - `lib/internal/main/run_main_module`
    - `internal/main/eval_string`
    - `internal/main/repl`
    - ...
- SpinEventLoopInternal
  - uv_run(env->event_loop(), UV_RUN_DEFAULT);

Creating and Setting JS Objects in V8

- `v8::Object`

```
// let info = {};
// let servername_str = '...';
// info['servername'] = servername_str;

// 1. Creates a new JavaScript object in the current isolate.
// 2. Converts the servername C-string to a V8 String.
// 3. Set the 'servername' property of the 'info' object

Local<Object> info = Object::New(env->isolate());
Local<String> servername_str = OneByteString(env->isolate(), servername, strlen(servername));
info->Set(env->context(), env->servername_string(), servername_str);
```

Creating and Setting JS Functions in V8

- `v8::Function`

```
// let thrower = function() { // ProtoThrower
//    throw new Error('Accessing Object.prototype.__proto__ has been disallowed ...');
// }

void ProtoThrower(const FunctionCallbackInfo<Value>& info) {
  THROW_ERR_PROTO_ACCESS(info.GetIsolate());
}

Local<Value> thrower;
if (!Function::New(context, ProtoThrower).ToLocal(&thrower)) {
  return Nothing<bool>();
}
```

Example: process.argv

- Creating a native module follows a coding pattern, bridging native and JS.

```
# 0. How does `process.argv` output the following value ?
$ node -e "console.log(process.argv)";
[ '/home/daeyeon/.volta/tools/image/node/20.16.0/bin/node' ]
```

```
// lib/internal/process/pre_execution.js

// Patch the process object with legacy properties and normalizations.
function patchProcessObject(expandArgv1) {

  // 1. Check the binding name, 'process_methods', loaded by `internalBinding`. (clue 1)
  const binding = internalBinding('process_methods');

  // 2. Check the function property name, 'patchProcessObject', used via the binding. (clue 2)
  binding.patchProcessObject(process);

  ...
}
```

Example: process.argv

```
// src/node_process_object.cc

// 3. C++ macros define various bindings. Search for `NODE_BINDING_CONTEXT_AWARE_INTERNAL` to find them.
// With clue 1, we can find 'process_methods' and it's set up by the `process::CreatePerContextProperties`.
NODE_BINDING_CONTEXT_AWARE_INTERNAL(process_methods, node::process::CreatePerContextProperties)
NODE_BINDING_PER_ISOLATE_INIT(process_methods, node::process::CreatePerIsolateProperties)
```

```
// src/node_process_object.cc
static void CreatePerIsolateProperties(IsolateData* isolate_data,
                                       Local<ObjectTemplate> target) {
  Isolate* isolate = isolate_data->isolate();
  ...

  // 4. The function sets `patchProcessObject` to `PatchProcessObject`.
  SetMethod(isolate, target, "patchProcessObject", PatchProcessObject);
}
```

Example: process.argv

```
// src/node_process_object.cc

// 5. With clue 2, we can find a native function named `PatchProcessObject` that
// starts with an uppercase letter, which matches the name, `patchProcessObject`,
// in JavaScript. (This pattern is commonly used for binding functions.)

void PatchProcessObject(const FunctionCallbackInfo<Value>& args) {
  ...
  CHECK(args[0]->IsObject());

  // 6. Here, we can see code that sets the 'argv' property on the `process` object, which
  // is passed as arguments[0] from JavaScript.
  Local<Object> process = args[0].As<Object>();

  // process.argv
  process->Set(context,FIXED_ONE_BYTE_STRING(isolate, "argv"),
               ToV8Value(context, env->argv()).ToLocalChecked()).Check();
}
```

V8 Object & Function creation with a template

- Simple objects or functions use class methods for setup, as previously mentioned.
- More complex cases use Templates like `v8::ObjectTemplate` or `v8::FunctionTemplate`.
- Template Advantages :
  - Structure Definition : Clearly defines the object’s structure, allowing for consistent setup.
  - Inheritance Support : Allows prototype-based inheritance from the template.

V8 Object creation with a template

```
// Note: `Symbol` is used conceptually for explanation, not an exact match for the internal field count.
// const kInternalFieldCount = Symbol('internalFieldCount');
// const kMessagePortInternalFieldCount = 1;
//
// function HandleWrap() { ... }
//
// function MessagePort() {                                                  // a
//   this[kInternalFieldCount] = kMessagePortInternalFieldCount;             // c
// }
//
// MessagePort.prototype = Object.create(HandleWrap.prototype);             // b
// MessagePort.prototype.constructor = MessagePort;                         // b
//
// MessagePort.prototype.postMessage = function(message) { ... };           // d


// src/node_messaging.cc
Local<FunctionTemplate> m = NewFunctionTemplate(isolate, MessagePort::New);     // a
m->SetClassName(isolate_data->message_port_constructor_string());
m->Inherit(HandleWrap::GetConstructorTemplate(isolate_data));                   // b
m->InstanceTemplate()->SetInternalFieldCount(MessagePort::kInternalFieldCount); // c
SetProtoMethod(isolate, m, "postMessage", MessagePort::PostMessage);            // d
```

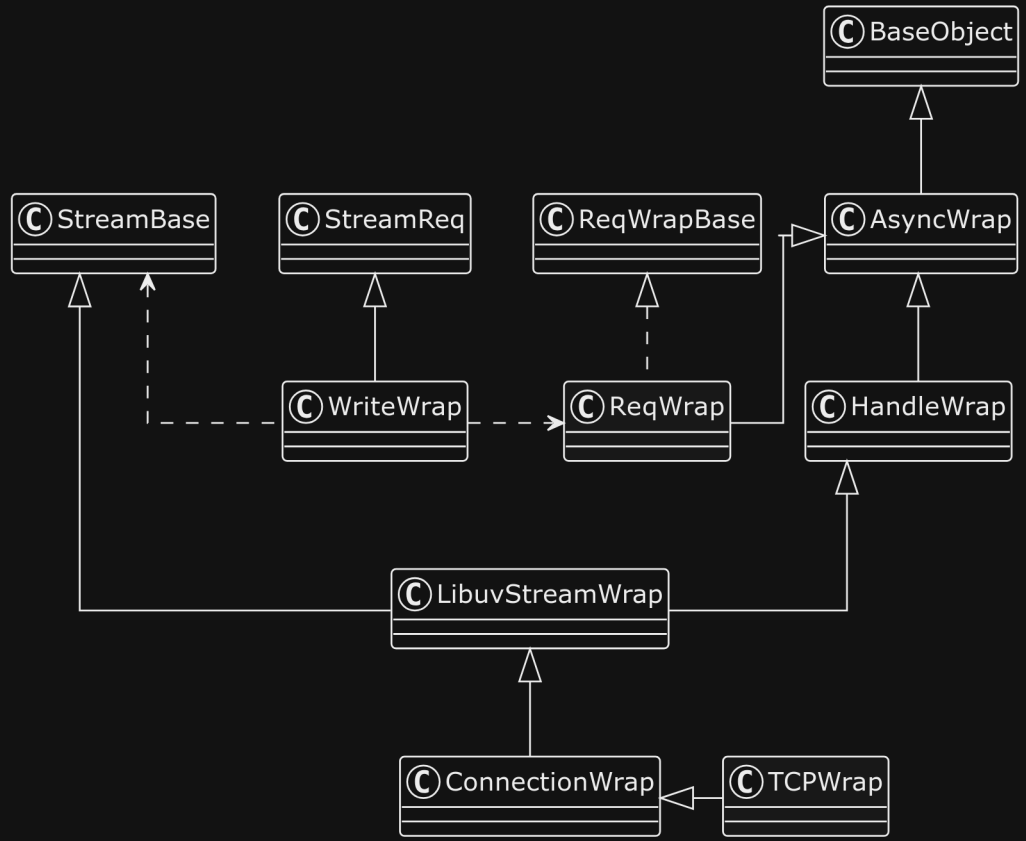
V8 Object creation with a template

Set Usage : Object Method vs. ObjectTemplate

```
// Set Method: Configures each object individually, which can be
// less efficient for multiple objects.
Local<Object> obj1 = Object::New(isolate);
obj1->Set(context, String::NewFromUtf8(isolate, "key"), value);
Local<Object> obj2 = Object::New(isolate);
obj2->Set(context, String::NewFromUtf8(isolate, "key"), value);
```

```
// ObjectTemplate: Creates multiple objects with the same structure more efficiently
// by defining the template once.
Local<ObjectTemplate> tmpl = ObjectTemplate::New(isolate);
tmpl->Set(isolate, "key", value);
Local<Object> obj1 = tmpl->NewInstance(context).ToLocalChecked();
Local<Object> obj2 = tmpl->NewInstance(context).ToLocalChecked();
```

C++ and JS Object Wrapping



C++ and JS Object Wrapping

```
#define UV_REQ_FIELDS uv_req_type type; ...

struct uv_req_s {               // ReqWrap
  UV_REQ_FIELDS ...
};

struct uv_write_s {             // WriteWrap , uv_write_t is a subclass of uv_req_t.
  UV_REQ_FIELDS ...
  uv_stream_t* handle;
};

#define UV_HANDLE_FIELDS \
  uv_loop_t* loop;       \
  int fd; ...

struct uv_stream_s {            // LibuvStreamWrap
  UV_HANDLE_FIELDS ...
};

struct uv_tcp_s {               // TCPWrap
  UV_HANDLE_FIELDS
  UV_STREAM_FIELDS ...
};
```

C++ and JS Object Wrapping

- BaseObject : Holds or detaches its lifetime from the JavaScript object.
- xxxWrap
  - AsyncWrap : Tracks asynchronous operations.
  - ReqWrap : Manages asynchronous requests based on `uv_req_t`.
  - StreamBase : Provides common functionality for streams using `uv_stream_t`.
  - WriteWrap : Manages asynchronous writes using `uv_write_t`.
  - TCPWrap : Handles TCP operations based on `uv_tcp_t`.
  - ...

Example: TCPWrap

```
// lib/net.js
const {
  TCP, // 1. Check 'tcp_wrap' loaded by `internalBinding`. (clue 1)
} = internalBinding('tcp_wrap')

function createHandle(fd, is_server) {
  return new TCP(...); // 2. Check the 'TCP' keyword, a constructor function. (clue 2)
}

this._handle = createHandle(fd, false);
```

```
// src/tcp_wrap
void TCPWrap::Initialize(Local<Object> target,...) {
  Local<FunctionTemplate> t = NewFunctionTemplate(isolate, New);
  t->Inherit(LibuvStreamWrap::GetConstructorTemplate(env));
  // 4. With clue 2, we can find that 'TCP' is set as a constructor in the Function Template.
  SetConstructorFunction(context, target, "TCP", t);
  ...
}
// 3. With clue 1, we see that 'tcp_wrap' is set up in TCPWrap::Initialize.
NODE_BINDING_CONTEXT_AWARE_INTERNAL(tcp_wrap, node::TCPWrap::Initialize)
```

Example: TCPWrap

```
// lib/net.js
this._handle = createHandle(fd, false);

function closeSocketHandle(self, isException, isCleanupPending = false) {
    // 5. 'close' isn't set as a prototype method in TCPWrap::Initialize,
    // but it's inherited by `t->Inherit(LibuvStreamWrap::GetConstructorTemplate(env));`.
    self._handle.close(() => {
      self.emit('close', isException);
      self._handle = null;
    });
  }
}
```

```
// src/handle_wrap.cc
Local<FunctionTemplate> HandleWrap::GetConstructorTemplate(...) {
  ...
  // 6. 'close' is a common part of uv_handle, so it's handled here, HandleWrap.
  SetProtoMethod(isolate, tmpl, "close", HandleWrap::Close);
  SetProtoMethod(isolate, tmpl, "ref", HandleWrap::Ref);
  SetProtoMethod(isolate, tmpl, "unref", HandleWrap::Unref);
}
```

Wrap C++ Object in JS with `internalField`

- Make a link between a C++ object and a JavaScript object via `SetInternalFields`.

```cpp
// src/base_object.cc
BaseObject::BaseObject(Realm* realm, Local<Object> object)
    : persistent_handle_(realm->isolate(), object), realm_(realm) {
  CHECK_EQ(false, object.IsEmpty());
  CHECK_GE(object->InternalFieldCount(), BaseObject::kInternalFieldCount);

  // 1. Here, we associate this C++ object with the given JS object by storing
  // 'this' pointer in the JS object's InternalField.
  SetInternalFields(realm->isolate_data(), object, static_cast<void*>(this));

  realm->AddCleanupHook(DeleteMe, static_cast<void*>(this));
  realm->modify_base_object_count(1);
}

void BaseObject::SetInternalFields(IsolateData* isolate_data,
                                   v8::Local<v8::Object> object,
                                   void* slot) {
  // 2. Store the pointer in the internal field at position BaseObject::kSlot.
  object->SetAlignedPointerInInternalField(BaseObject::kSlot, slot);
}
```

Unwrap C++ Object in JS with `internalField`

```cpp
// src/tcp_wrap.cc
// err = this._handle.open(fd); // lib/net.js
void TCPWrap::Open(const FunctionCallbackInfo<Value>& args) {
  TCPWrap* wrap;

  // 3. ASSIGN_OR_RETURN_UNWRAP extracts the native C++ object (TCPWrap) from the
  // JavaScript object (args.This()).
  ASSIGN_OR_RETURN_UNWRAP(
      &wrap, args.This(), args.GetReturnValue().Set(UV_EBADF));

  if (!args[0]->IntegerValue(args.GetIsolate()->GetCurrentContext()).To(&val))
    return;
  int fd = static_cast<int>(val); ...
}
```

```cpp
#define ASSIGN_OR_RETURN_UNWRAP(ptr, obj, ...)                                 \
    *ptr = static_cast<typename std::remove_reference<decltype(*ptr)>::type>(  \
        BaseObject::FromJSObject(obj));                                        \

BaseObject* BaseObject::FromJSObject(v8::Local<v8::Value> value) {
  v8::Local<v8::Object> obj = value.As<v8::Object>();
  // 4. Retrieve the pointer from the internal field at position BaseObject::kSlot.
  return static_cast<BaseObject*>(obj->GetAlignedPointerFromInternalField(BaseObject::kSlot));
}
```

Wrap up

- **Bootstrap**
  - High-Level overview of the bootstrap process and relevant JavaScript files.
  - Omits details like libuv and V8 setup (platform/snapshot/inspector), and more.
- **Object and Function Creation**
  - Simple setups use each class method, while complex cases use Templates.
- **Binding C++ class instances and JavaScript Objects**
  - JS objects, C++ objects, and uv handles are hierarchically wrapped.

Thank you

https://github.com/daeyeon/code-and-learn

daeyeon.dev@gmail.com