JavaScript and C++ in Node.js core

How do they talk to each other?

Joyee Cheung @ Blink NodeConf EU 2023

Agenda

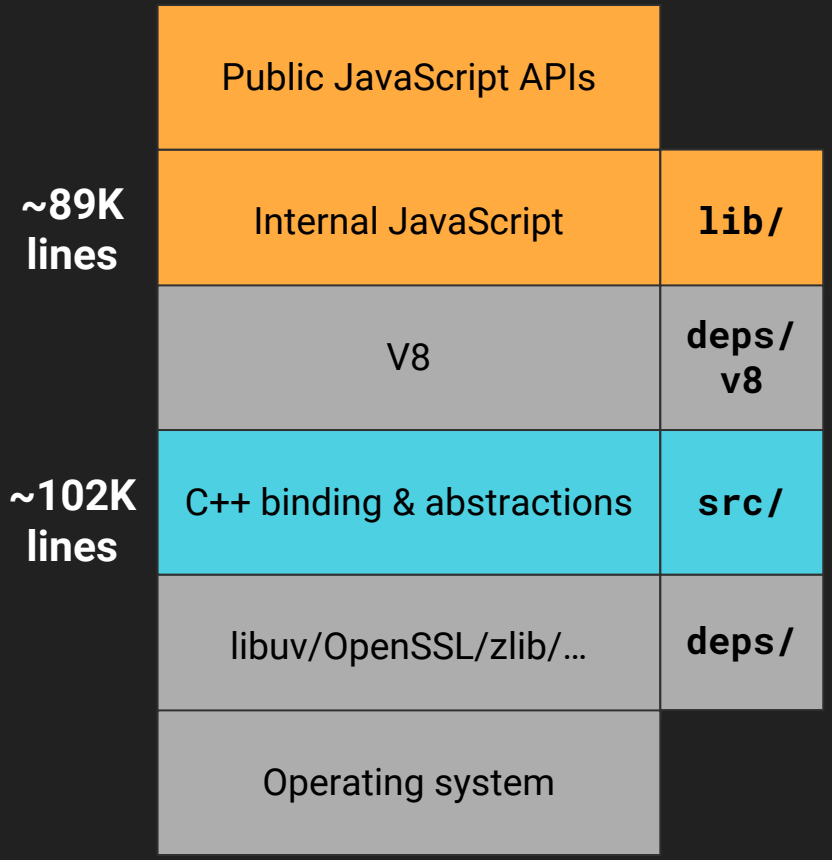
- **Overview of layers in Node.js**
- **Code invocations between C++ and JavaScript**
  - JavaScript -> C++
  - V8 Fast API calls
  - C++ -> JavaScript
- **Cross-layer memory management**
  - JavaScript -> C++ references
  - C++ -> JavaScript references
  - Trace-based GC & unified heap

About me..

- Work on Igalia & Bloomberg collaboration
- Node.js TSC member & V8 committer
- Recent focus
  - Memory infrastructure
  - Startup snapshot
  - Code caching

- Twitter/GitHub: @joyeecheung

Overview of layers in Node.js



e.g. **`fs.open()`**

Validate & normalize arguments. Some APIs are mostly implemented in JS e.g. streams

JavaScript virtual machine

Convert JS values <-> C++ values
Take care of permission, tracing and hooks, invoke libraries..

Dependencies

Warning: internal property names, especially underscore-prefixed ones, are for demonstration purposes only, don’t count on them in the user land

…and some names used in the examples are invented pseudo-names

JavaScript -> C++ invocation

JavaScript land - `fs.openSync()`

```
// lib/fs.js
const binding = internalBinding('fs');
function openSync(path, flags, mode) {
  path = toNamespacedPath(
    getValidatedPath(path));
  flags = stringToFlags(flags);
  mode = parseFileMode(mode, 'mode',
                       0o666);
  return binding.open(path, flags, mode);
}
```

- Internal version of **`process.binding()`** (doc-deprecated)
- Used to organize functionalities C++ expose to JS

C++ land - `binding.open()`

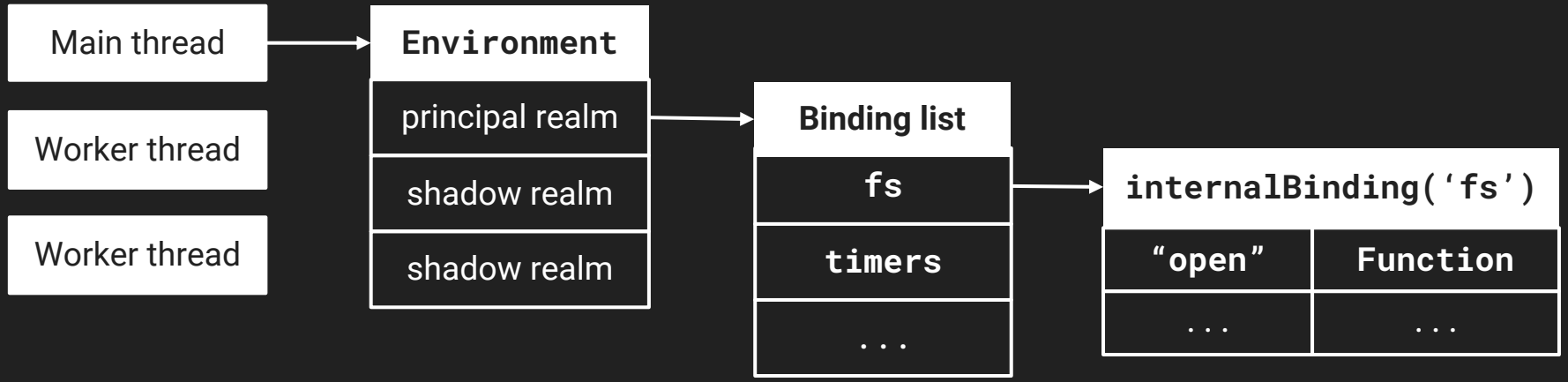
```
static void Open(const FunctionCallbackInfo<Value>& args) {
  Environment* env = Environment::GetCurrent(args);

  // Convert JS arguments into native ones
  BufferValue path(env->isolate(), args[0]);
  const int flags = args[1].As<Int32>()->Value();
  const int mode = args[2].As<Int32>()->Value();

  // A bunch of abstractions but essentially...
  int result = uv_fs_open(nullptr, req, *path, flags, mode,
                          nullptr /* synchronous */);
  if (!is_uv_error(result)) {
    Local<Integer> ret = v8::Integer::New(isolate, result);
    args.GetReturnValue().Set(ret); return;
  }
}
```

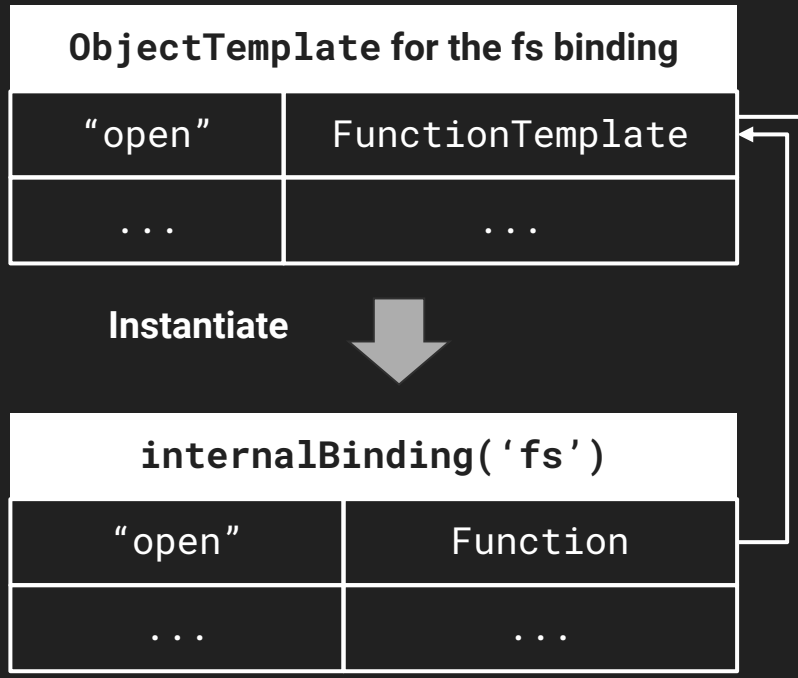
How the bindings are organized - binding list

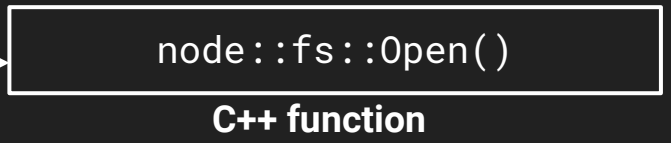
- Each Environment contains one V8 VM and one or more realms



Node.js process

Binding list





- When **`binding.open()`** is called, V8 looks up the C++ function following the pointers
- Then V8 sets up some internal states (e.g. for arguments) before calling the C++ function

JavaScript -> C++ invocation

JavaScript land - `fs.openSync()`

```
// lib/fs.js
const binding = internalBinding('fs');
function openSync(path, flags, mode) {
  path = toNamespacedPath(
    getValidatedPath(path));
  flags = stringToFlags(flags);
  mode = parseFileMode(mode, 'mode',
                       0o666);
  return binding.open(path, flags, mode);
}
module.exports = { openSync, ... };
```

* Simplified version of **`fs.openSync()`** - success path only

C++ land - `binding.open()`

```
static void Open(const FunctionCallbackInfo<Value>& args) {
  Environment* env = Environment::GetCurrent(args);

  // Convert JS arguments into native ones
  BufferValue path(env->isolate(), args[0]);
  const int flags = args[1].As<Int32>()->Value();
  const int mode = args[2].As<Int32>()->Value();

  // A bunch of abstractions but essentially...
  int result = uv_fs_open(nullptr, req, *path, flags, mode,
                          nullptr /* synchronous */);
  if (!is_uv_error(result)) {
    Local<Integer> ret = v8::Integer::New(isolate, result);
    args.GetReturnValue().Set(ret); return;
  }
}
```

JavaScript -> C++ invocation

Yes, today Node.js still uses raw V8 stuff internally instead of using anything like Node-API.

JavaScript land - `fs.openSync()`

```
// lib/fs.js
const binding = internalBinding('fs');
function openSync(path, flags, mode) {
  path = toNamespacedPath(
    getValidatedPath(path));
  flags = stringToFlags(flags);
  mode = parseFileMode(mode, 'mode',
                       0o666);
  return binding.open(path, flags, mode);
}
module.exports = { openSync, ... };
```

* Simplified version of **`fs.openSync()`** - success path only

C++ land - `binding.open()`

```
static void Open(const FunctionCallbackInfo<Value>& args) {
  Environment* env = Environment::GetCurrent(args);

  // Convert JS arguments into native ones
  BufferValue path(env->isolate(), args[0]);
  const int flags = args[1].As<Int32>()->Value();
  const int mode = args[2].As<Int32>()->Value();

  // A bunch of abstractions but essentially...
  int result = uv_fs_open(nullptr, req, *path, flags, mode,
                          nullptr /* synchronous */);
  if (!is_uv_error(result)) {
    Local<Integer> ret = v8::Integer::New(isolate, result);
    args.GetReturnValue().Set(ret); return;
  }
}
```

JavaScript -> C++ invocation

JavaScript land - `fs.openSync()`

```
// lib/fs.js
const binding = internalBinding('fs');
function openSync(path, flags, mode) {
  path = toNamespacedPath(
    getValidatedPath(path));
  flags = stringToFlags(flags);
  mode = parseFileMode(mode, 'mode',
                       0o666);
  return binding.open(path, flags, mode);
}
module.exports = { openSync, ... };
```

* Simplified version of **`fs.openSync()`** - success path only

C++ land - `binding.open()`

```
static void Open(const FunctionCallbackInfo<Value>& args) {
  Environment* env = Environment::GetCurrent(args);

  // Convert JS arguments into native ones
  BufferValue path(env->isolate(), args[0]);
  const int flags = args[1].As<Int32>()->Value();
  const int mode = args[2].As<Int32>()->Value();

  // A bunch of abstractions but essentially...
  int result = uv_fs_open(nullptr, req, *path, flags, mode,
                          nullptr /* synchronous */);
  if (!is_uv_error(result)) {
    Local<Integer> ret = v8::Integer::New(isolate, result);
    args.GetReturnValue().Set(ret); return;
  }
}
```

JavaScript -> C++ invocation

JavaScript land - `fs.openSync()`

```
// lib/fs.js
const binding = internalBinding('fs');
function openSync(path, flags, mode) {
  path = toNamespacedPath(
    getValidatedPath(path));
  flags = stringToFlags(flags);
  mode = parseFileMode(mode, 'mode',
                       0o666);
  return binding.open(path, flags, mode);
}
module.exports = { openSync, ... };
```

* Simplified version of **`fs.openSync()`** - success path only

C++ land - `binding.open()`

```
static void Open(const FunctionCallbackInfo<Value>& args) {
  Environment* env = Environment::GetCurrent(args);

  // Convert JS arguments into native ones
  BufferValue path(env->isolate(), args[0]);
  const int flags = args[1].As<Int32>()->Value();
  const int mode = args[2].As<Int32>()->Value();

  // A bunch of abstractions but essentially...
  int result = uv_fs_open(nullptr, req, *path, flags, mode,
                          nullptr /* synchronous */);
  if (!is_uv_error(result)) {
    Local<Integer> ret = v8::Integer::New(isolate, result);
    args.GetReturnValue().Set(ret); return;
  }
}
```

JavaScript -> C++ invocation

JavaScript land - `fs.openSync()`

```
// lib/fs.js
const binding = internalBinding('fs');
function openSync(path, flags, mode) {
  path = toNamespacedPath(
    getValidatedPath(path));
  flags = stringToFlags(flags);
  mode = parseFileMode(mode, 'mode',
                       0o666);
  return binding.open(path, flags, mode);
}
module.exports = { openSync, ... };
```

* Simplified version of **`fs.openSync()`** - success path only

C++ land - `binding.open()`

```
static void Open(const FunctionCallbackInfo<Value>& args) {
  Environment* env = Environment::GetCurrent(args);

  // Convert JS arguments into native ones
  BufferValue path(env->isolate(), args[0]);
  const int flags = args[1].As<Int32>()->Value();
  const int mode = args[2].As<Int32>()->Value();

  // A bunch of abstractions but essentially...
  int result = uv_fs_open(nullptr, req, *path, flags, mode,
                          nullptr /* synchronous */);
  if (!is_uv_error(result)) {
    Local<Integer> ret = v8::Integer::New(isolate, result);
    args.GetReturnValue().Set(ret); return;
  }
}
```

That works…but can we make it faster?

Yes..with V8 fast API calls

```
void GuessHandleType(const FunctionCallbackInfo<Value>& args) {
  Isolate* isolate = args.GetIsolate();
  Local<Context> context = isolate->GetCurrentContext();
  int fd = args[0]->Int32Value(context).ToChecked();
  uv_handle_type t = uv_guess_handle(fd);
  uint32_t result = GetUVHandleTypeCode(t);
  args.GetReturnValue().Set(Uint32::New(isolate, result));
}
```

* An equivalent version of the internal `guessHandleType()` binding

V8 Fast API calls

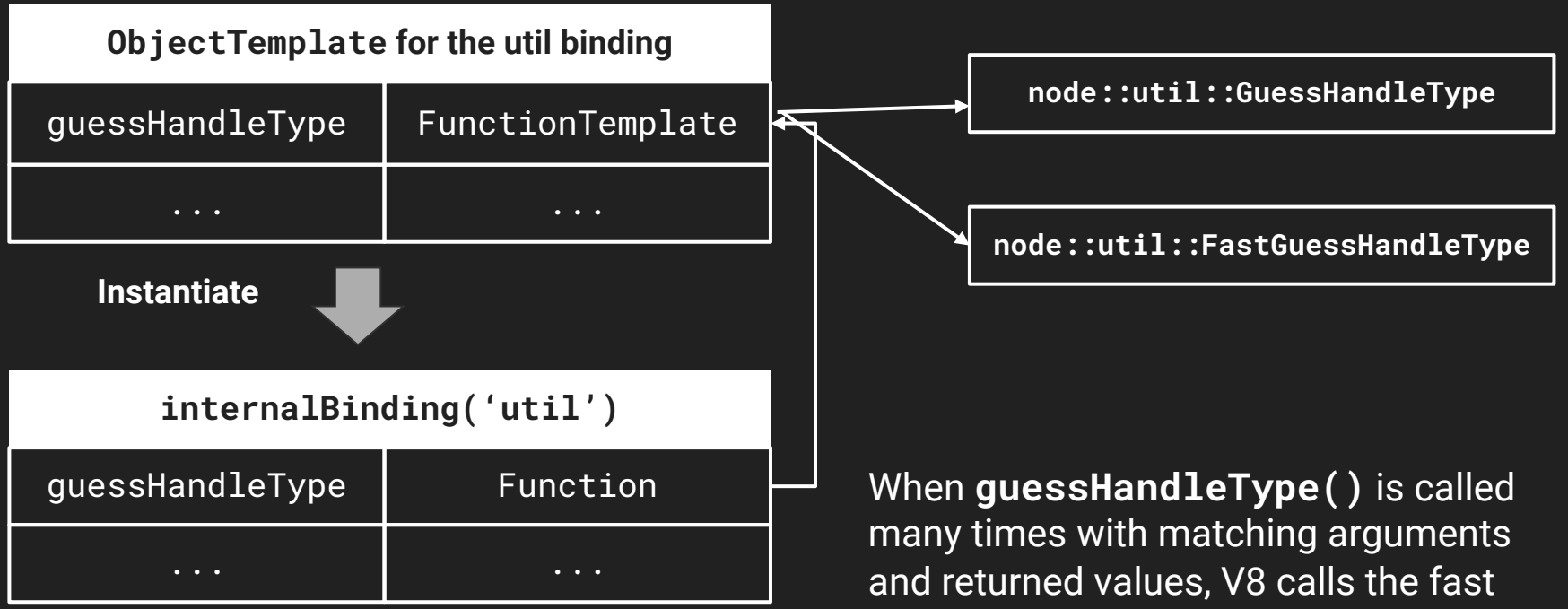
```
void GuessHandleType(const FunctionCallbackInfo<Value>& args) {
  Isolate* isolate = args.GetIsolate();
  Local<Context> context = isolate->GetCurrentContext();
  int fd = args[0]->Int32Value(context).ToChecked();
  uv_handle_type t = uv_guess_handle(fd);
  uint32_t result = GetUVHandleTypeCode(t);
  args.GetReturnValue().Set(Uint32::New(isolate, result));
}

uint32_t FastGuessHandleType(Local<Value> receiver, const uint32_t fd) {
  uv_handle_type t = uv_guess_handle(fd);
  return GetUVHandleTypeCode(t);
}
```

* An equivalent version of the internal `guessHandleType()` binding

- Add a new version whose argument/return value conversions are inlined by v8

Binding list



When **guessHandleType()** is called many times with matching arguments and returned values, V8 calls the fast version for subsequent calls with matching signatures.

V8 Fast API calls

```
uint32_t FastGuessHandleType(Local<Value> receiver, const uint32_t fd) {
  uv_handle_type t = uv_guess_handle(fd);
  return GetUVHandleTypeCode(t);
}
```

- No allocations, no calling back to JS, no exceptions
- In return, smaller overhead due to not having to set things up for handling them
- ~10% faster for most bindings in Node.js - and they add up
- More bindings in Node.js are being converted to fast API calls if applicable

C++ -> JavaScript invocation

* Simplified version of **`fs.open()`**

JavaScript land - `fs.openSync()`

```
// lib/fs.js
function open(path, flags, mode, cb) {
  // ...validate and normalize the args
  const req = new FSReqCallback();
  req.oncomplete = cb;
  binding.open(path, flags, mode, req);
}

// user land
fs.open('./f.txt', O_RDONLY, 0o666,
        (err, fd) => { /* … */ )});
```

C++ land - `binding.open()`

```
static void Open(const FunctionCallbackInfo<Value>& args) {
  // arg handling similar to that of fs.openSync()..
  uv_fs_open(loop, req, *path, flags, mode, [](uv_fs_t* req) {
    FSReqCallback* native_req = Unwrap(req);
    Local<Object> js_req = native_req->object();
    // req.oncomplete
    Local<Function> callback = GetOnComplete(req);
    // Convert results into callback arguments
    Local<Value> args[] = {
       is_uv_error(req->result)? v8::Null(isolate) : Err(req);
      v8::Integer::New(isolate, req->result)
    };
    callback->Call(context, js_req, 2, args);
  });
}
```

C++ -> JavaScript invocation

* Simplified version of **`fs.open()`**

JavaScript land - `fs.openSync()`

```
// lib/fs.js
function open(path, flags, mode, cb) {
  // ...validate and normalize the args
  const req = new FSReqCallback();
  req.oncomplete = cb;
  binding.open(path, flags, mode, req);
}

// user land
fs.open('./f.txt', O_RDONLY, 0o666,
        (err, fd) => { /* … */ }));
```

C++ land - `binding.open()`

```
static void Open(const FunctionCallbackInfo<Value>& args) {
  // arg handling similar to that of fs.openSync()..
  uv_fs_open(loop, req, *path, flags, mode, [](uv_fs_t* req) {
    FSReqCallback* native_req = Unwrap(req);
    Local<Object> js_req = native_req->object();
    // req.oncomplete
    Local<Function> callback = GetOnComplete(req);
    // Convert results into callback arguments
    Local<Value> args[] = {
       is_uv_error(req->result)? v8::Null(isolate) : Err(req);
      v8::Integer::New(isolate, req->result)
    };
    callback->Call(context, js_req, 2, args);
  });
}
```

C++ -> JavaScript invocation

* Simplified version of **`fs.open()`**

JavaScript land - `fs.openSync()`

```
// lib/fs.js
function open(path, flags, mode, cb) {
  // ...validate and normalize the args
  const req = new FSReqCallback();
  req.oncomplete = cb;
  binding.open(path, flags, mode, req);
}

// user land
fs.open('./f.txt', O_RDONLY, 0o666,
        (err, fd) => { /* … */ }));
```

C++ land - `binding.open()`

```
static void Open(const FunctionCallbackInfo<Value>& args) {
  // arg handling similar to that of fs.openSync()..
  uv_fs_open(loop, req, *path, flags, mode, [](uv_fs_t* req) {
    FSReqCallback* native_r
    Local<Object> js_req =
    // req.oncomplete
    Local<Function> callback = GetOnComplete(req);
    // Convert results into callback arguments
    Local<Value> args[] = {
       is_uv_error(req->result)? v8::Null(isolate) : Err(req);
      v8::Integer::New(isolate, req->result)
    };
    callback->Call(context, js_req, 2, args);
  });
}
```

Post the asynchronous request to the event loop with a callback

C++ -> JavaScript invocation

* Simplified version of **`fs.open()`**

JavaScript land - `fs.openSync()`

```
// lib/fs.js
function open(path, flags, mode, cb) {
  // ...validate and normalize the args
  const req = new FSReqCallback();
  req.oncomplete = cb;
  binding.open(path, flags, mode, req);
}

// user land
fs.open('./f.txt', O_RDONLY, 0o666,
        (err, fd) => { /* … */ )});
```

C++ land - `binding.open()`

```
static void Open(const FunctionCallbackInfo<Value>& args) {
  // arg handling similar to that of fs.openSync()..
  uv_fs_open(loop, req, *path, flags, mode, [](uv_fs_t* req) {
    FSReqCallback* native_req = Unwrap(req);
    Local<Object> js_req = native_req->object();
    // req.oncomplete
    Local<Function> callback = GetOnComplete(req);
    // Convert results into callback arguments
    Local<Value> args[] = {
       is_uv_error(req->result)? v8::Null(isolate) : Err(req);
      v8::Integer::New(isolate, req->result)
    };
    callback->Call(context, js_req, 2, args);
  });
}
```

C++ -> JavaScript invocation

* Simplified version of **`fs.open()`**

JavaScript land - `fs.openSync()`

```
// lib/fs.js
function open(path, flags, mode, cb) {
  // ...validate and normalize the args
  const req = new FSReqCallback();
  req.oncomplete = cb;
  binding.open(path, flags, mode, req);
}

// user land
fs.open('./f.txt', O_RDONLY, 0o666,
        (err, fd) => { /* … */ )});
```

C++ land - `binding.open()`

```
static void Open(const FunctionCallbackInfo<Value>& args) {
  // arg handling similar to that of fs.openSync()..
  uv_fs_open(loop, req, *path, flags, mode, [](uv_fs_t* req) {
    FSReqCallback* native_req = Unwrap(req);
    Local<Object> js_req = native_req->object();
    // req.oncomplete
    Local<Function> callback = GetOnComplete(req);
    // Convert results into callback arguments
    Local<Value> args[] = {
       is_uv_error(req->result)? v8::Null(isolate) : Err(req),
      v8::Integer::New(isolate, req->result)
    };
    callback->Call(context, js_req, 2, args);
  });
}
```

That seems *simple* enough…
except that this is a bit too simplified

C++ -> JavaScript invocation

* Simplified version of **`fs.open()`**

JavaScript land - `fs.openSync()`

```
// lib/fs.js
function open(path, flags, mode, cb) {
  // ...validate and normalize the args
  const req = new FSReqCallback();
  req.oncomplete = cb;
  binding.open(path, flags, mode, req);
}

// user land
fs.open('./f.txt', O_RDONLY, 0o666,
        (err, fd) => { /* … */ )});
```

C++ land - `binding.open()`

```
static void Open(const FunctionCallbackInfo<Value>& args) {
  // arg handling similar to that of fs.openSync()..
  uv_fs_open(loop, req, *path, flags, mode, [](uv_fs_t* req) {
    FSReqCallback* native_req = Unwrap(req);
    Local<Object> js_req = native_req->object();
    // req.oncomplete
    Local<Function> callback = GetOnComplete(req);
    // Convert results into callback arguments
    Local<Value> args[] = {
       is_uv_error(req->result)? v8::Null(isolate) : Err(req);
      v8::Integer::New(isolate, req->result)
    };
    callback->Call(context, js_req, 2, args);
  });
}
```

C++ -> JavaScript invocation

* **Less** simplified version of `fs.open()`

JavaScript land - `fs.openSync()`

```
// lib/fs.js
function open(path, flags, mode, cb) {
  // ...validate and normalize the args
  const req = new FSReqCallback();
  req.oncomplete = cb;
  binding.open(path, flags, mode, req);
}

// user land
fs.open('./f.txt', O_RDONLY, 0o666,
        (err, fd) => { /* … */ )});
```

C++ land - `binding.open()`

```
static void Open(const FunctionCallbackInfo<Value>& args) {
  // arg handling similar to that of fs.openSync()..
  uv_fs_open(loop, req, *path, flags, mode, [](uv_fs_t* req) {
    FSReqCallback* native_req = Unwrap(req);
    Local<Object> js_req = native_req->object();
    // req.oncomplete
    Local<Function> callback = GetOnComplete(req);
    // Convert results into callback arguments
    Local<Value> args[] = {
       is_uv_error(req->result)? v8::Null(isolate) : Err(req);
      v8::Integer::New(isolate, req->result)
    };
    MakeCallback(... callback, 2, args);
  });
}
```

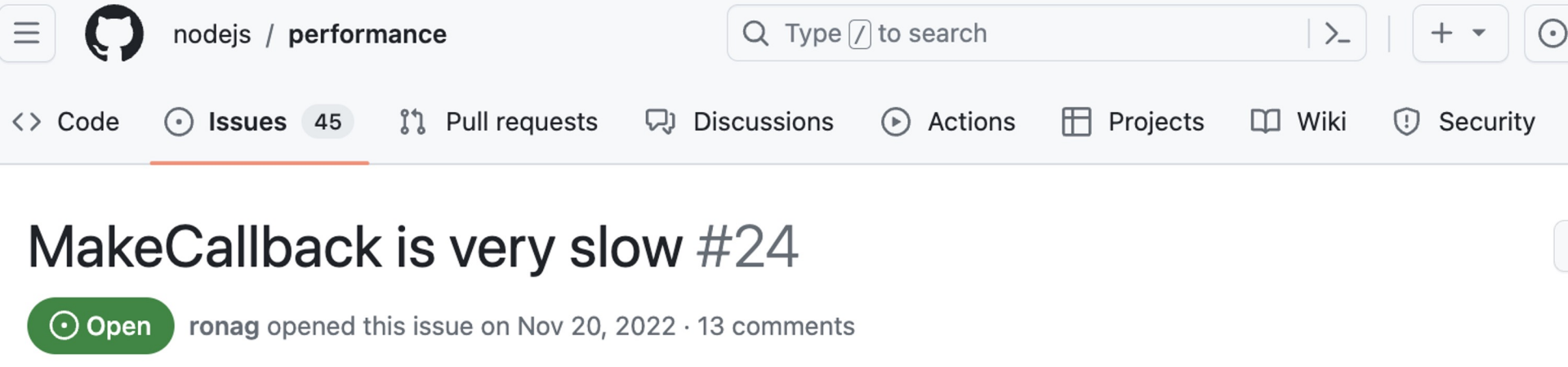
What does MakeCallback does?

Besides just invoking the JS function…

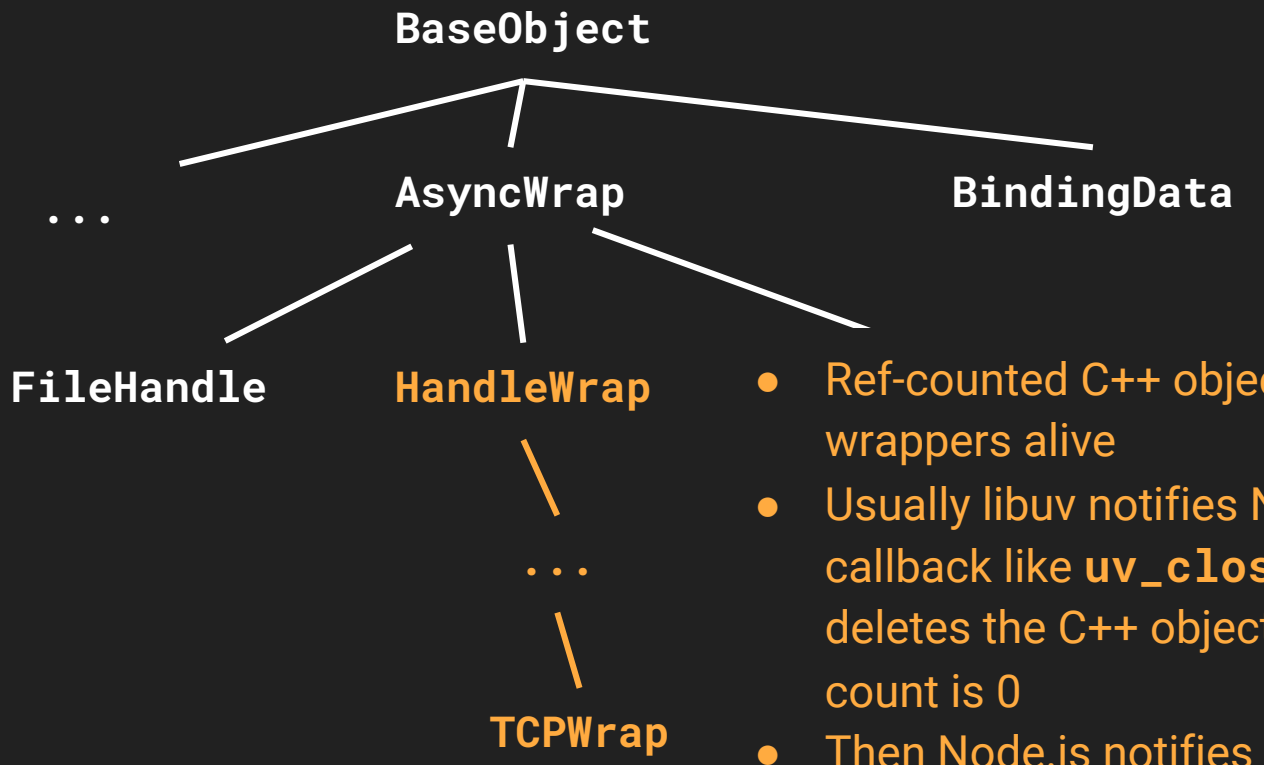
- Run microtasks (from promises), if any (common)
- Run callbacks queued by `process.nextTick()`, if any (common)
- Emit AsyncHooks, if any (not as often)

`node::MakeCallback()`

- Theoretically, there's no overhead if there are no microtasks, ticks, hooks
- In practice, many of them are in the way in real-world applications
- **`MakeCallback()`** tends to be slower than "simply calling back to JS in V8" - despite optimizations like callback coalescing
- ...changing it breaks ordering (some assumed by users), how the overhead can be reduced without hurting compatibility is still under investigation

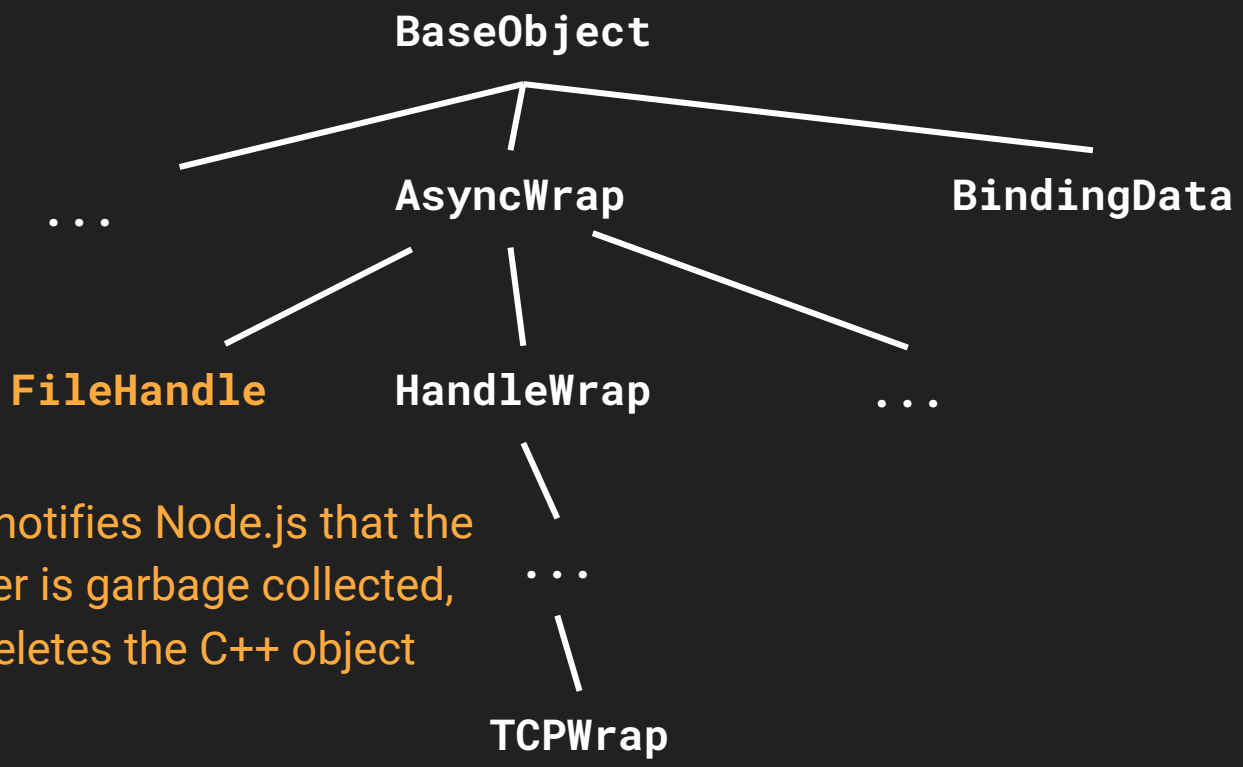


Classes of wrappers for resource management



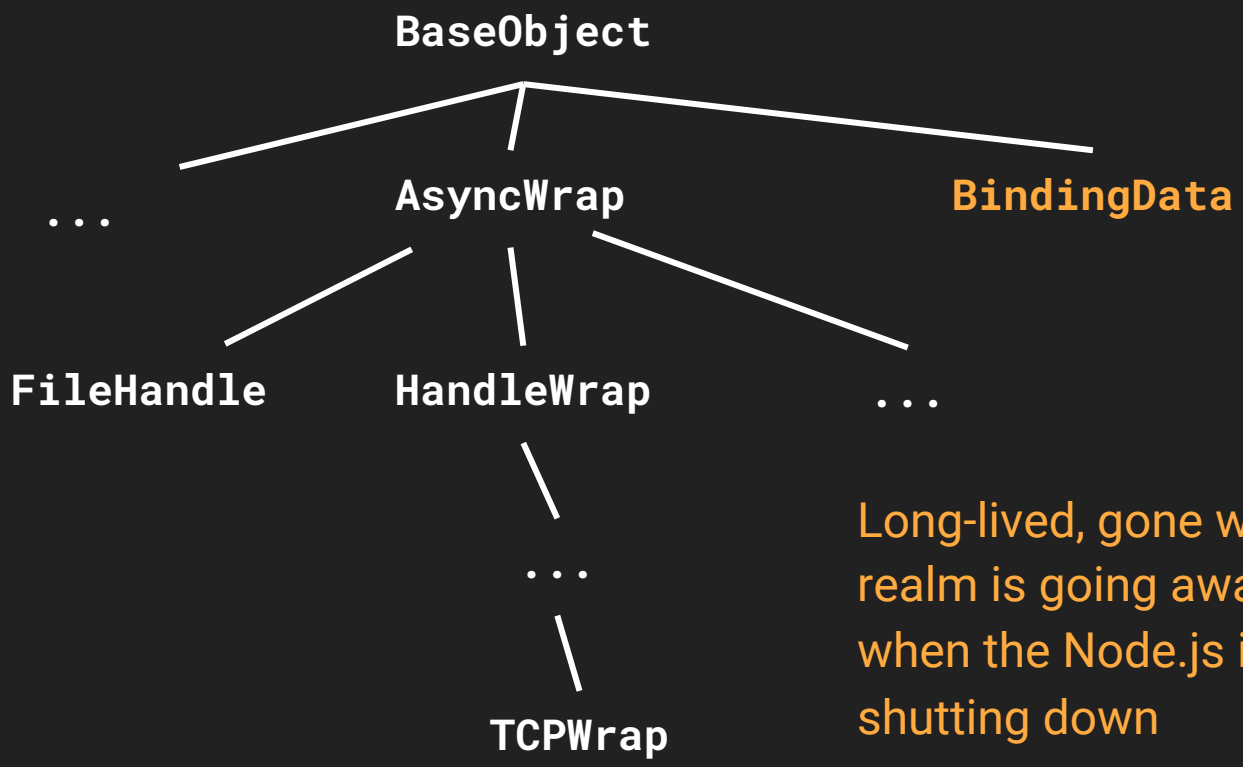
- Ref-counted C++ objects hold JS wrappers alive
- Usually libuv notifies Node.js in a callback like `uv_close()`, Node.js deletes the C++ object when ref count is 0
- Then Node.js notifies V8 to release the JS wrapper

Classes of wrappers for resource management



When V8 notifies Node.js that the JS wrapper is garbage collected, Node.js deletes the C++ object

Classes of wrappers for resource management



Long-lived, gone when the realm is going away e.g. when the Node.js instance is shutting down

JavaScript -> C++ memory reference

* Simplified version of **`net.Socket`**
  - TCP path only

JavaScript land

```
Socket.prototype.connect = function(...args) {
  this[kHandle] = new TCP(SOCKET);
  initSocketHandle(this);
}

Socket.prototype.destroy = function(err, cb) {
  closeSocketHandle(this, ...);
}

function closeSocketHandle(self, ...) {
  self[kHandle].close(() => {
    self.emit('close', ...);
    // ...
  });
}
```

C++ land

```
void TCPWrap::New(const FunctionCallbackInfo<Value>& args) {
  new TCPWrap(env, args.This(), provider);
}

void HandleWrap::Close(Local<Value> close_callback) {
  uv_close(handle_, OnClose);
  object()->Set(context, onclose_symbol, close_callback);
}

void HandleWrap::OnClose(uv_handle_t* handle) {
  BaseObjectPtr<HandleWrap> wrap { handle->data };
  wrap->Detach();
  wrap->MakeCallback(onclose_symbol, 0, nullptr);
}
```

JavaScript -> C++ memory reference

* Simplified version of **`net.Socket`**
  - TCP path only

JavaScript land

```
Socket.prototype.connect = function(...args) {
  this[kHandle] = new TCP(SOCKET);
  initSocketHandle(this);
}

Socket.prototype.destroy = function(err, cb) {
  closeSocketHandle(this, ...);
}

function closeSocketHandle(self, ...) {
  self[kHandle].close(() => {
    self.emit('close', ...);
    // ...
  });
}
```

C++ land

```
void TCPWrap::New(const FunctionCallbackInfo<Value>& args) {
  new TCPWrap(env, args.This(), provider);
}

void HandleWrap::Close(Local<Value> close_callback) {
  uv_close(handle_, OnClose);
  object()->Set(context, onclose_symbol, close_callback);
}

void HandleWrap::OnClose(uv_handle_t* handle) {
  BaseObjectPtr<HandleWrap> wrap { handle->data };
  wrap->Detach();
  wrap->MakeCallback(onclose_symbol, 0, nullptr);
}
```

JavaScript -> C++ memory reference

* Simplified version of **`net.Socket`**
  - TCP path only

JavaScript land

```
Socket.prototype.connect = function(...args) {
  this[kHandle] = new TCP(SOCKET);
  initSocketHandle(this);
}

Socket.prototype.destroy = function(err, cb) {
  closeSocketHandle(this, ...);
}

function closeSocketHandle(self, ...) {
  self[kHandle].close(() => {
    self.emit('close', ...);
    // ...
  });
}
```

C++ land

```
void TCPWrap::New(const FunctionCallbackInfo<Value>& args) {
  new TCPWrap(env, args.This(), provider);
}

void HandleWrap::Close(Local<Value> close_callback) {
  uv_close(handle_, OnClose);
  object()->Set(context, onclose_symbol, close_callback);
}

void HandleWrap::OnClose(uv_handle_t* handle) {
  BaseObjectPtr<HandleWrap> wrap { handle->data };
  wrap->Detach();
  wrap->MakeCallback(onclose_symbol, 0, nullptr);
}
```

Holds a strong reference to the JS-land TCP object

JavaScript -> C++ memory reference

* Simplified version of **`net.Socket`**
  - TCP path only

JavaScript land

```
Socket.prototype.connect = function(...args) {
  this[kHandle] = new TCP(SOCKET);
  initSocketHandle(this);
}

Socket.prototype.destroy = function(err, cb) {
  closeSocketHandle(this, ...);
}

function closeSocketHandle(self, ...) {
  self[kHandle].close(() => {
    self.emit('close', ...);
    // ...
  });
}
```

C++ land

```
void TCPWrap::New(const FunctionCallbackInfo<Value>& args) {
  new TCPWrap(env, args.This(), provider);
}
```

TCPWrap inherits from HandleWrap

```
void HandleWrap::Close(Local<Value> close_callback) {
  uv_close(handle_, OnClose);
  object()->Set(context, onclose_symbol, close_callback);
}

void HandleWrap::OnClose(uv_handle_t* handle) {
  BaseObjectPtr<HandleWrap> wrap { handle->data };
  wrap->Detach();
  wrap->MakeCallback(onclose_symbol, 0, nullptr);
}
```

JavaScript -> C++ memory reference

* Simplified version of **`net.Socket`**
  - TCP path only

JavaScript land

```
Socket.prototype.connect = function(...args) {
  this[kHandle] = new TCP(SOCKET);
  initSocketHandle(this);
}

Socket.prototype.destroy = function(err, cb) {
  closeSocketHandle(this, ...);
}

function closeSocketHandle(self, ...) {
  self[kHandle].close(() => {
    self.emit('close', ...);
    // ...
  });
}
```

C++ land

```
void TCPWrap::New(const FunctionCallbackInfo<Value>& args) {
  new TCPWrap(env, args.This(), provider);
}

void HandleWrap::Close(Local<Value> close_callback) {
  uv_close(handle_, OnClose);
  object()->Set(context, onclose_symbol, close_callback);
}

void HandleWrap::OnClose(uv_handle_t* handle) {
  BaseObjectPtr<HandleWrap> wrap { handle->data };
  wrap->Detach();
  wrap->MakeCallback(onclose_symbol, 0, nullptr);
}
```

`tcp._handle[onclose]` assigned to the callback from JS

JavaScript -> C++ memory reference

* Simplified version of **`net.Socket`**
  - TCP path only

JavaScript land

```
Socket.prototype.connect = function(...args) {
  this[kHandle] = new TCP(SOCKET);
  initSocketHandle(this);
}

Socket.prototype.destroy = function(err, cb) {
  closeSocketHandle(this, ...);
}

function closeSocketHandle(self, ...) {
  self[kHandle].close(() => {
    self.emit('close', ...);
    // ...
  });
}
```

C++ land

```
void TCPWrap::New(const FunctionCallbackInfo<Value>& args) {
  new TCPWrap(env, args.This(), provider);
}

void HandleWrap::Close(Local<Value> close_callback) {
  uv_close(handle_, OnClose);
  object()->Set(context, onclose_symbol, close_callback);
}

void HandleWrap::OnClose(uv_handle_t* handle) {
  BaseObjectPtr<HandleWrap> wrap { handle->data };
  wrap->Detach();
  wrap->MakeCallback(onclose_symbol, 0, nullptr);
}
```

Async callback

JavaScript -> C++ memory reference

* Simplified version of **`net.Socket`**
  - TCP path only

JavaScript land

```
Socket.prototype.connect = function(...args) {
  this[kHandle] = new TCP(SOCKET);
  initSocketHandle(this);
}

Socket.prototype.destroy = function(err, cb) {
  closeSocketHandle(this, ...);
}

function closeSocketHandle(self, ...) {
  self[kHandle].close(() => {
    self.emit('close', ...);
    // ...
  });
}
```

C++ land

```
void TCPWrap::New(const FunctionCallbackInfo<Value>& args) {
  new TCPWrap(env, args.This(), provider);
}

void HandleWrap::Close(Local<Value> close_callback) {
  uv_close(handle_, OnClose);
  object()->Set(context, onclose_symbol, close_callback);
}

void HandleWrap::OnClose(uv_handle_t* handle) {
  BaseObjectPtr<HandleWrap> wrap { handle->data };
  wrap->Detach();
  wrap->MakeCallback(onclose_symbol, 0, nullptr);
}
```

Destroy TCPWrap when ref count in C++ is 0

JavaScript -> C++ memory reference

* Simplified version of **`net.Socket`**
  - TCP path only

JavaScript land

```
Socket.prototype.connect = function(...args) {
  this[kHandle] = new TCP(SOCKET);
  initSocketHandle(this);
}

Socket.prototype.destroy = function(err, cb) {
  closeSocketHandle(this, ...);
}

function closeSocketHandle(self, ...) {
  self[kHandle].close(() => {
    self.emit('close', ...);
    // ...
  });
}
```

C++ land

```
void TCPWrap::New(const FunctionCallbackInfo<Value>& args) {
  new TCPWrap(env, args.This(), provider);
}

void HandleWrap::Close(Local<Value> close_callback) {
  uv_close(handle_, OnClose);
  object()->Set(context, onclose_symbol, close_callback);
}

void HandleWrap::OnClose(uv_handle_t* handle) {
  BaseObjectPtr<HandleWrap> wrap { handle->data };
  wrap->Detach();
  wrap->MakeCallback(onclose_symbol, 0, nullptr);
}
```

..and with it, C++ reference to the JS land TCP wrapper

JavaScript -> C++ memory reference

* Simplified version of **net.Socket**
  - TCP path only

JavaScript land

```
Socket.prototype.connect = function(...args) {
  this[kHandle] = new TCP(SOCKET);
  initSocketHandle(this);
}

Socket.prototype.destroy = function(err, cb) {
  closeSocketHandle(this, ...);
}

function closeSocketHandle(self, ...) {
  self[kHandle].close(() => {
    self.emit('close', ...);
    // ...
  });
}
```

C++ land

```
void TCPWrap::New(const FunctionCallbackInfo<Value>& args) {
  new TCPWrap(env, args.This(), provider);
}

void HandleWrap::Close(Local<Value> close_callback) {
  uv_close(handle_, OnClose);
  object()->Set(context, onclose_symbol, close_callback);
}

void HandleWrap::OnClose(uv_handle_t* handle) {
  BaseObjectPtr<HandleWrap> wrap { handle->data };
  wrap->Detach();
  wrap->MakeCallback(onclose_symbol, 0, nullptr)
}
```

tcp._handle[onclose]

JavaScript -> C++ memory reference

* Simplified version of **`net.Socket`**
  - TCP path only

JavaScript land

```
Socket.prototype.connect = function(...args) {
  this[kHandle] = new TCP(SOCKET);
  initSocketHandle(this);
}

Socket.prototype.destroy = function(err, cb) {
  closeSocketHandle(this, ...);
}

function closeSocketHandle(self, ...) {
  self[kHandle].close(() => {
    self.emit('close', ...);
    // ...
  });
}
```

C++ land

```
void TCPWrap::New(const FunctionCallbackInfo<Value>& args) {
  new TCPWrap(env, args.This(), provider);
}

void HandleWrap::Close(Local<Value> close_callback) {
  uv_close(handle_, OnClose);
  object()->Set(context, onclose_symbol, close_callback);
}

void HandleWrap::OnClose(uv_handle_t* handle) {
  BaseObjectPtr<HandleWrap> wrap { handle->data };
  wrap->Detach();
  wrap->MakeCallback(onclose_symbol, 0, nullptr);
}
```

After close, everything will be gone when Socket and the TCP handle is unreachable from JS

Deciphering heap snapshots

Summary ▼ socket ⊗ All objects ▼

Constructor ▲	Distance	Shallow Size		Reta
▼ Socket	9	488	0 %	
▼ Socket @77145	9	488	0 %	
▶ <symbol kBuffer> :: system / Oddball @67	4	48	0 %	
▶ <symbol kBufferCb> :: system / Oddball @67	4	48	0 %	
▶ <symbol kBufferGen> :: system / Oddball @67	4	48	0 %	
▶ <symbol kCapture> :: system / Oddball @71	5	48	0 %	
▼ <symbol kHandle> :: TCP @6285	10	56	0 %	
▶ 3 :: *onStreamRead()* @75257	8	64	0 %	
▶ <symbol owner_symbol> :: Socket @77145	9	488	0 %	
▼ javascript_to_native :: Node / TCPWrap<Socket> @50	11	424	0 %	
native_to_javascript :: TCP @6285	10	56	0 %	

Retainers

Object	Distance	Shallow Size		Reta
▶ javascript_to_native in TCP @6285	10	56	0 %	
▼ [1] in Node / CleanupQueue @48	–	320	0 %	
▼ cleanup_queue in Node / PrincipalRealm @46	–	752	0 %	
principal_realm in Node / Environment @8	–	2 328	0 %	

Deciphering heap snapshots

Summary ▼ socket ⊗ All objects ▼

Constructor ▲	Distance	Shallow Size		Reta
▼ Socket	9	488	0 %	
▼ Socket @77145	9	488	0 %	
▶ <symbol kBuffer> :: system / Oddball @67	4	48	0 %	
▶ <symbol kBufferCb> :: system / Oddball @67	4	48	0 %	
▶ <symbol kBufferGen> :: system / Oddball @67	4	48	0 %	
▶ <symbol kCapture> :: system / Oddball @71	5	48	0 %	
▼ <symbol kHandle> :: TCP @6285	10	56	0 %	
▶ 3 :: *onStreamRead()* @75257	8	64	0 %	
▶ <symbol owner_symbol> :: Socket @77145	9	488	0 %	
▼ javascript_to_native :: Node / TCPWrap<Socket> @50	11	424	0 %	
native_to_javascript :: TCP @6285	10	56	0 %	

Native nodes are prefixed with “Node /”

Retainers

Object	Distance	Shallow Size		Reta
▶ javascript_to_native in TCP @6285	10	56	0 %	
▼ [1] in Node / CleanupQueue @48	–	320	0 %	
▼ cleanup_queue in Node / PrincipalRealm @46	–	752	0 %	
principal_realm in Node / Environment @8	–	2 328	0 %	

Deciphering heap snapshots

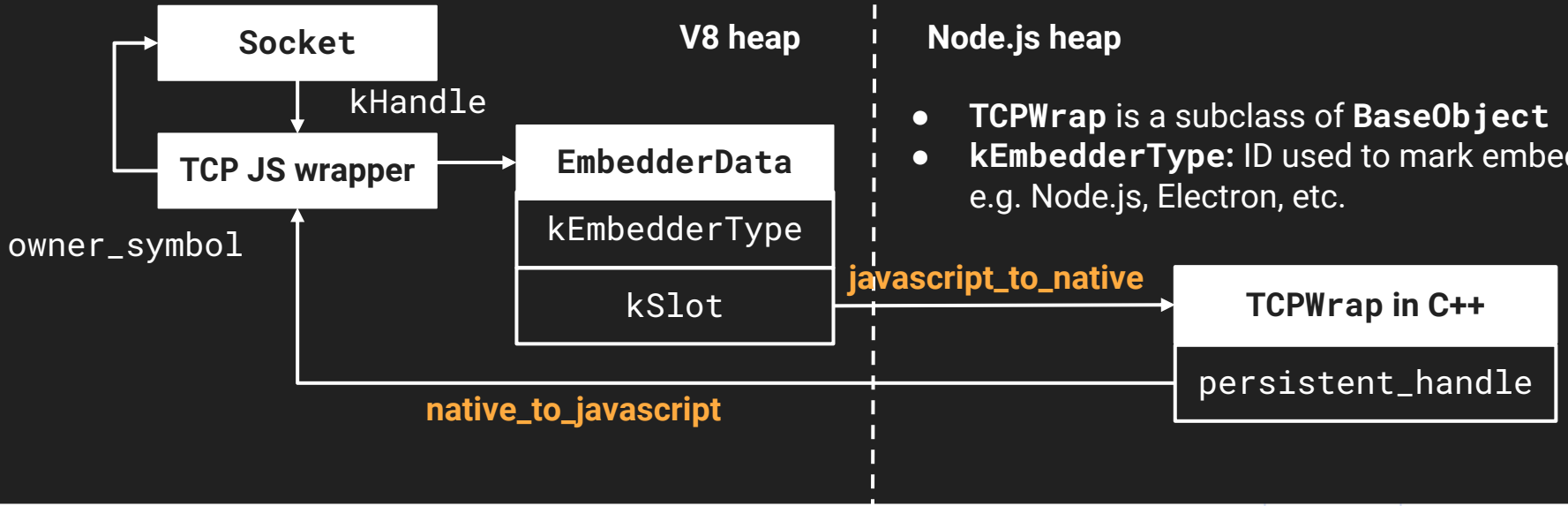
Summary ▼ socket ⊗ All objects ▼

Constructor ▲	Distance	Shallow Size		Reta
▼ Socket	9	488	0 %	
▼ Socket @77145	9	488	0 %	
▶ <symbol kBuffer> :: system / Oddball @67	4	48	0 %	
▶ <symbol kBufferCb> :: system / Oddball @67	4	48	0 %	
▶ <symbol kBufferGen> :: system / Oddball @67	4	48	0 %	
▶ <symbol kCapture> :: system / Oddball @71	5	48	0 %	
▼ <symbol kHandle> :: TCP @6285	10	56	0 %	
▶ 3 :: *onStreamRead()* @75257	8	64	0 %	
▶ <symbol owner_symbol> :: Socket @77145	9	488	0 %	
▼ javascript_to_native :: Node / TCPWrap<Socket> @50	11	424	0 %	
native_to_javascript :: TCP @6285	10	56	0 %	

Retainers

Object	Distance	Shallow Size		Reta
▶ javascript_to_native in TCP @6285	10	56	0 %	
▼ [1] in Node / CleanupQueue @48	–	320	0 %	
▼ cleanup_queue in Node / PrincipalRealm @46	–	752	0 %	
principal_realm in Node / Environment @8	–	2 328	0 %	

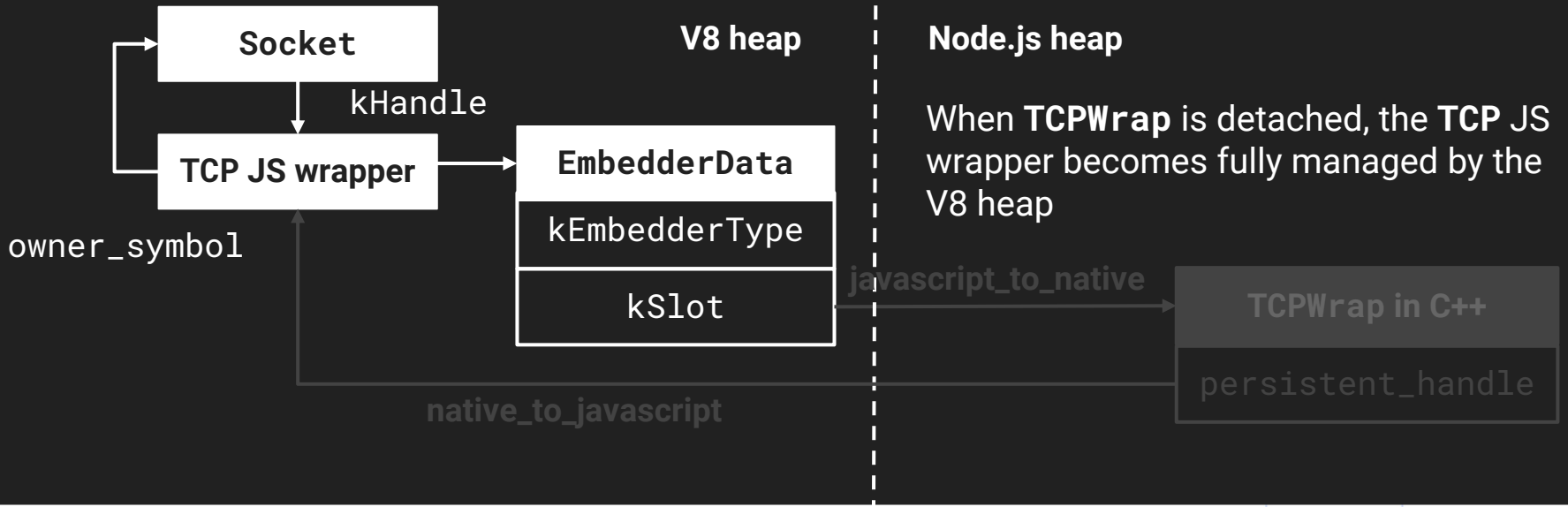
Deciphering heap snapshots



- **`TCPWrap`** is a subclass of **`BaseObject`**
- **`kEmbedderType`:** ID used to mark embedder e.g. Node.js, Electron, etc.

▼ <symbol kHandle> :: TCP @6285	10	56	0 %
▶ 3 :: *onStreamRead()* @75257	8	64	0 %
▶ <symbol owner_symbol> :: Socket @77145	9	488	0 %
▼ javascript_to_native :: Node / TCPWrap<Socket> @50	11	424	0 %
native_to_javascript :: TCP @6285	10	56	0 %

Deciphering heap snapshots



▼ <symbol kHandle> :: TCP @6285	10	56	0 %
▶ 3 :: *onStreamRead()* @75257	8	64	0 %
▶ <symbol owner_symbol> :: Socket @77145	9	488	0 %
▼ javascript_to_native :: Node / TCPWrap<Socket> @50	11	424	0 %
native_to_javascript :: TCP @6285	10	56	0 %

C++ -> JavaScript memory reference

- Some objects reference the JS wrapper using a weak handle so the C++ object does not hold the JS one alive, it’s the other way around
- e.g. **`FileHandle`** (handle wrapped by **`fs.promises.open()`**)

```
FileHandle::FileHandle(BindingData* binding_data, Local<Object> obj, int fd) {
  MakeWeak();
  // ...
}
void BaseObject::MakeWeak() {
  persistent_handle_.SetWeak(this, [](const WeakCallbackInfo<BaseObject>& data) {
      BaseObject* obj = data.GetParameter();
      obj->persistent_handle_.Reset();
      delete obj;
  }, WeakCallbackType::kParameter);
}
```

C++ -> JavaScript memory reference

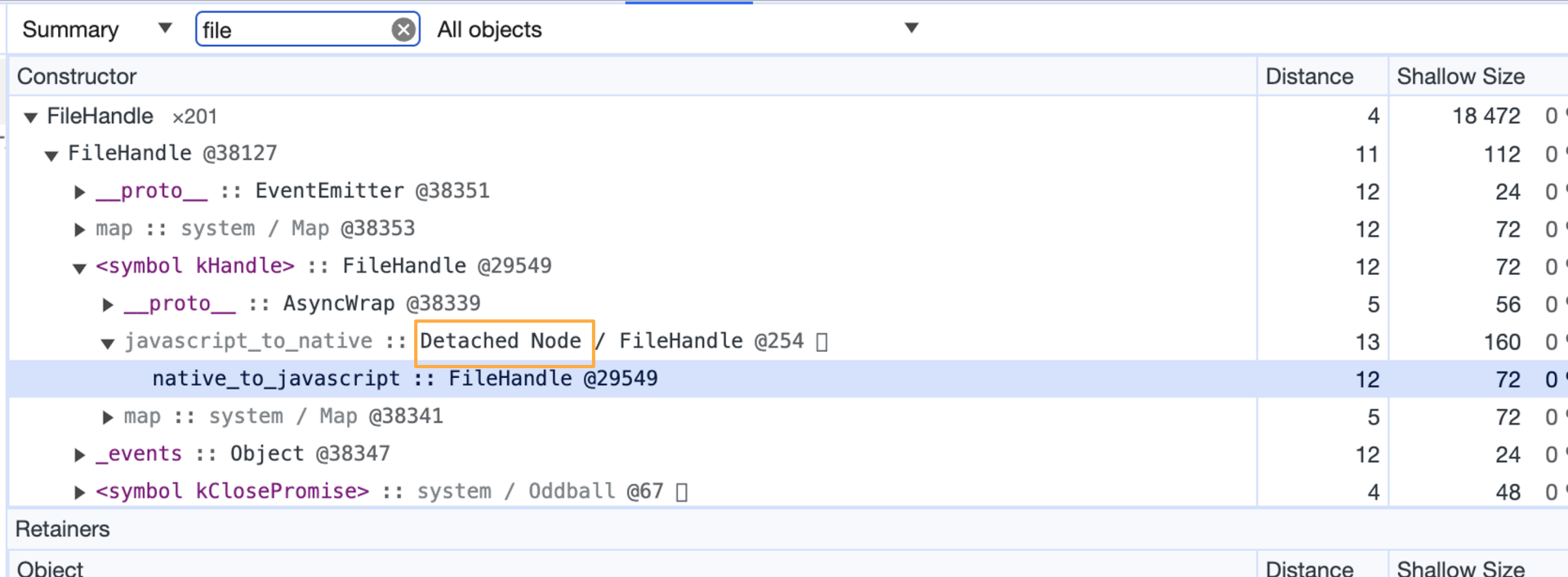
- Some objects reference the JS wrapper using a weak handle so the C++ object does not hold the JS one alive, it's the other way around
- e.g. **`FileHandle`** (handle wrapped by **`fs.promises.open()`**)

```
FileHandle::FileHandle(BindingData* binding_data, Local<Object> obj, int fd) {
  MakeWeak();
  // ...
}
void BaseObject::MakeWeak() {
  persistent_handle_.SetWeak(this, [](const WeakCallbackInfo<BaseObject>& data) {
      BaseObject* obj = data.GetParameter();
      obj->persistent_handle_.Reset();
      delete obj;
  }, WeakCallbackType::kParameter);
}
```

When V8 notifies Node.js that the JS wrapper is garbage collected, Node.js delete the C++ object

C++ -> JavaScript memory reference

The C++ objects being held alive only by JS objects and/or with no other references from C++ are shown as **Detached** in the heap snapshot



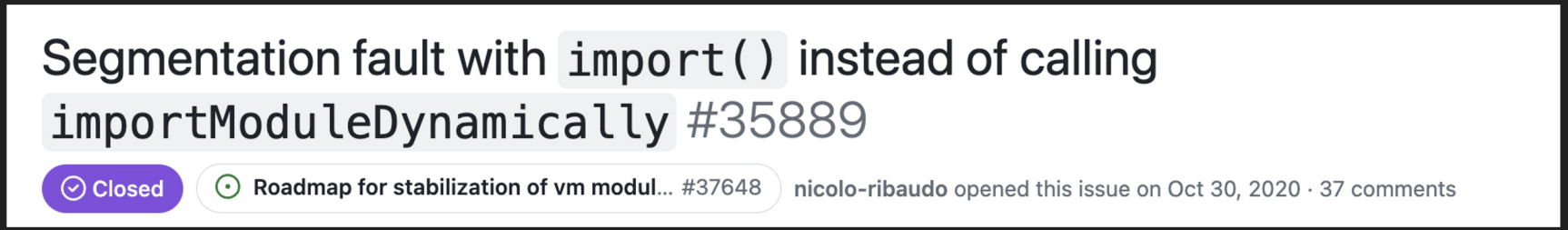
C++ -> JavaScript memory reference

- Many BaseObjects have additional references to JS values
- Callback-based memory management is also a hack (though it's been working)

```
/**
* Install a finalization callback on this object.
* NOTE: There is no guarantee as to *when* or even *if* the callback is
* invoked. The invocation is performed solely on a best effort basis.
* As always, GC-based finalization should *not* be relied upon for any
* critical form of resource management!
*
* The callback is supposed to reset the handle. No further V8 API may be
* called in this callback. In case additional work involving V8 needs to be
* done, a second callback can be scheduled using
* WeakCallbackInfo<void>::SetSecondPassCallback.
*/
template <typename P>
V8_INLINE void SetWeak(P* parameter,
                       typename WeakCallbackInfo<P>::Callback callback,
                       WeakCallbackType type);
```

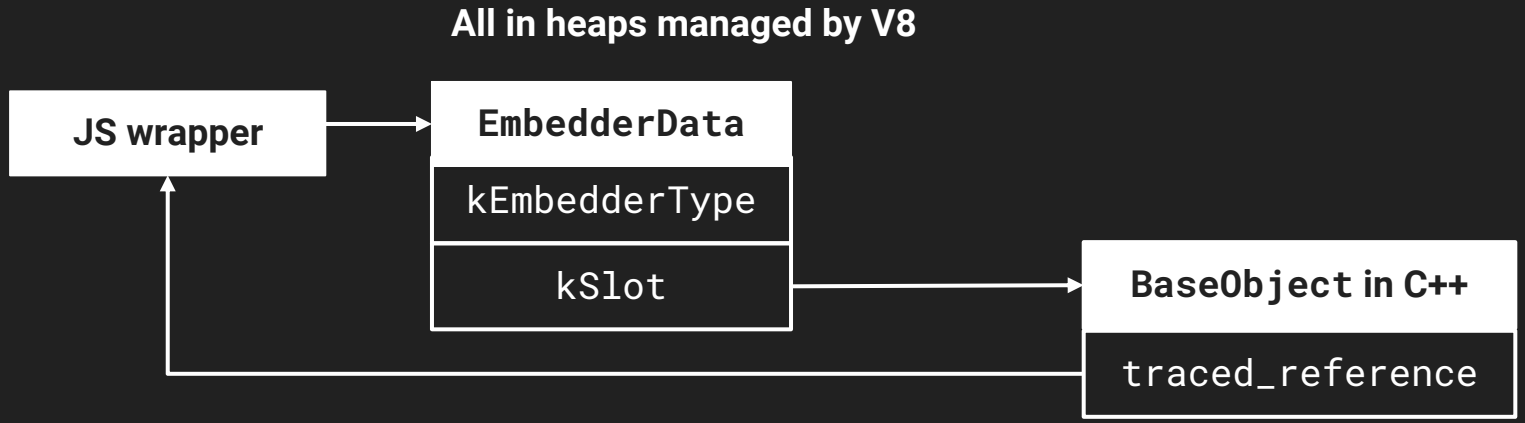
C++ -> JavaScript memory reference

- When the callbacks are not configured properly, bugs happen
- If you are one of the Jest users getting stuck at 16 because of some memory and/or performance issue from Node.js, please try 21 (fixes will be backported to older LTS soon)



WIP: trace-based GC & unified heap

- Oilpan: trace-based GC library in V8 carved out from Blink
- Allows implementing an interface to create C++ -> JS references fully known to V8’s garbage collector - and it’s not a hack
- Bootstrapped in Node.js now, internal object migration in progress



Thank you