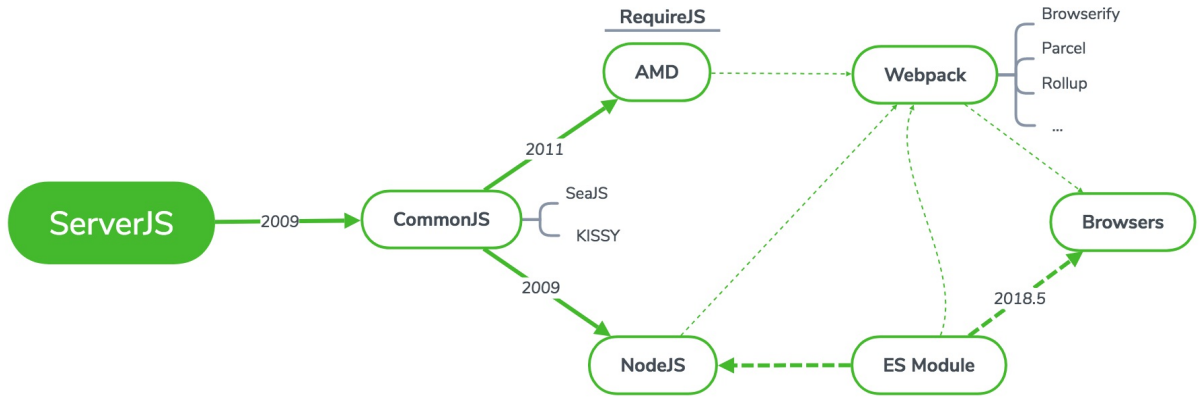
[视频时长统计] Node 的模块机制（CommonJS）与包管理

本节目标：【实现一个视频时长统计工具】，你包我包他的包，依赖加载怎么搞，模块关系的组织与加载是 Node 中 JS 动态语言处理的一大利器。



https://www.cyj.me/programming/2018/05/22/about-module-i/

Node 世界，一切（独立 JS 文件）皆模块，模块之间互相隔离互不影响，通过引用来互相调用。

一个模块本质是一个模块对象，通过 module.exports（exports 只是 module.exports 的一个引用）对外暴露接口，比如创建一个 step.js：

```
const pad = '.'

exports.step = (t) => `|${pad.repeat(t)}》`
// console.log(module)
```

再创建一个 race.js：

```
const { step } = require('./step')
const steps = step(10)

module.exports = { steps }
console.log(steps)
// console.log(module)
```

打印的结果：丨.....》，step.js 暴露一个拼接字符串的能力，race 把这个接口拎过来直接用，我们把代码中注释去掉，通过日志打印下 module：

```
# 模块对象 step
Module {
  id: '/Users/c/d/node-10.x/demo/step.js',
  exports: { step: [Function] },
  parent:
   Module {
     id: '.', exports: {}, parent: null, loaded:
false,
     filename: '/Users/c/d/node-
10.x/demo/race.js',
     children: [ [Circular] ],
     paths: [ '../node_modules' ] },
  filename: '/Users/c/d/node-10.x/demo/step.js',
  loaded: false, children: [],
  # 查找模块的方式，就是一级级往上查，直到根目录
  paths: [ '/Users/c/d/node-
10.x/demo/node_modules',
     '/Users/c/d/node-10.x/node_modules',
     '/Users/c/d/node_modules',
     '/Users/c/node_modules',
     '/Users/node_modules',
     '/node_modules' ] }
```

这是第一个模块 step.js 的模块内部信息，下面是第二个，两个模块结构都是一样的，大家可以比对下有哪些不同?

```
# 模块对象 race
Module {
  id: '.', parent: null, loaded: false,
  exports: { steps: '|..........》' },
  filename: '/Users/c/d/node-10.x/demo/race.js',
  children:
   [ Module {
       id: '/Users/c/d/node-10.x/demo/step.js',
       exports: [Object], parent: [Circular],
       loaded: true, children: [], paths: [Array]
       filename: '/Users/c/d/node-
10.x/demo/step.js' ],
  paths: [ '.../node_modules' ] }
```

打印出来的是 module 对象，有 id/loaded/paths/filename 这些基本信息外，被引用的模块会有一个 parent，里面是引用它的父亲模块的信息，反过来，引用了别的接口的模块，它里面则会有 children，里面包含了被它引用的模块信息，一个模块既可以被引用也可以引用别人。

我们来看下 step.js 的 module.exports，它也是一个对象，包含一个 step 的函数，race.js 里面的 module.exports 也是一个对象，里面包含的是 steps 这个字符串，那么 step 函数和 steps 都是这两个模块对外暴露的接口，暴露的方式，都是通过 module.exports 或 exports，它俩作用等价，而引用很简单，require 就够了。

那这个 module 和 require 又是怎么来的，我们就得结合前面 Node 源码分析启动流程来接着讲了，同时要先回忆下 CommonJS 规范。

CommonJS 是模块管理的规范

CommonJS (https://en.wikipedia.org/wiki/CommonJS) 的前身是 ServerJS (https://wiki.mozilla.org/ServerJS)。Node 在采用 CommonJS 规范来管理模块关系后，如日中天，拿下了服务端 JavaScript 市场的几乎全部江山，所谓青出于蓝而更胜于蓝，Node 在 CommonJS 的基础上继续衍化，加上 CommonJS 脱离群众太久，最终大家愿意买单的竟然是 Node Modules。再看下浏览器端，既有对 CommonJS 实现的前端模块加载框架 SeaJS/KISSY 等，也有基于 CommonJS 继续演进的 RequireJS，AMD 规范也应运而生，无论是 CommonJS 还是 AMD，基于他们所实现的模块加载库的背后也都有各自的构建工具，花开各家几年后，Webpack 横空出世，以 Webpack 为代表的构建工具，把不同模块理念下的模块代码全部收拢进来，彻底一统江湖。

关于模块的历史大家可以看 前端模块的历史沿革 (https://www.cyj.me/programming/2018/05/22/about-module-i/)，我们今天只关注 Node 里面的 CommonJS。

我们首先问自己这样一个问题，Node 里面的模块规范还是 CommonJS 么？问题先放这儿，我们先去寻找文章开头提到的：module 和 require 又是怎么来的这个问题的答案。

Node 中的模块加载机制

我们已经知道 CommonJS 是一套模块规范，约定了模块如何定义、加载与执行等等，那在 Node 里面是如何实现的呢？带着这样的问题，我们回到 Node 源码中找寻答案。

首先，我们把 源码解读：Node 程序架构和启动流程 (https://juejin.im/book/5bc1bf3e5188255c3272e315/sectior 这一节，我们分析 Node 程序架构和启动流程所学习到的知识，也

就是我们的纸篓子先拎过来，这是结论部分，我们先从这个结论部分直接跳到 CommonJS 这里来学习：

```
纸篓子 = [
  '1. Node 源码有一坨依赖，大部分是 C/C++ 底层',
  '2. Node 启动入口是 node_main.cc 的 main 函数',
  '3. 入口函数找到 node.cc 的 3 个 Start，依次调用',
  '4. node.cc 的第一个 Start 初始化了 v8，调用第二个
Start',
  '5. 第二个 Start 让 v8 准备了引擎实例，调用第三个
Start',
  '6. 第三个 Start：',
  '   6.1 首先准备了 v8 的上下文 Context',
  '   6.2 其次准备了 Node 的启动环境，对各种需要的变量做
整理',
  '   6.3 再把 Node 原生模块和我们的 JS 代码都加载进来
运行',
  '   6.4 最后把主持人 libuv 请上场，执行 JS 里的各种任
务',
  '7. libuv 没活干了，就一层层来退出进程、收拾场地，退出
程序',
]
```

以上就是 Node 的简要启动过程，从 6.3 这里，Node 正式进入了 JS 的语言世界，那 6.3 里面应该有我们希望看到的答案，它到底做了哪些事呢?

我们再声明一个纸箱子：

```
let 纸箱子 = [
  '6.3.1 Node 底层环境均已 Ready，准备装载 JS 模块'
]
```

加载内部模块的 Loader

首先回到 6.3 的 LoadEnvironment，在 src/node.cc 2115 行 (https://github.com/nodejs/node/blob/v10.x/src/node.cc#L2 精简如下：

```
void LoadEnvironment(Environment* env) {
  // 1. 首先载入 loader.js 和 node.js 拿到 JS 文件内容（字符串），通过 GetBootstrapper 解析
  // 注意这两个 JS 是会被 node_js2c 编译成字符串数组，存储到 node_javascript.cc 里面，这里只是源码而已
  <String> loaders_name = FIXED_STRING(env->isolate(), "internal/bootstrap/loaders.js");
  <Function> loaders_bootstrapper = GetBootstrapper(env, LoadersBootstrapperSource(env), loaders_name);
  Local<String> node_name = FIXED_STRING(env->isolate(), "internal/bootstrap/node.js");
  <Function> node_bootstrapper = GetBootstrapper(env, NodeBootstrapperSource(env), node_name);
  // 2. 创建各种 bindings，后面会丢到 JS 函数中用
  // ...
  // 3. 拼装 loaders 的函数参数数组，分别是 process 和后面的 binding function
  // 注意这里的几个参数跟下文的 loaders.js 是有对应关系的
  Local<Value> loaders_bootstrapper_args[] = {
      env->process_object(),
      get_binding_fn,
      get_linked_binding_fn,
      get_internal_binding_fn
  };
```

```
  // 4. 通过 ExecuteBootstrapper 来陆续启动内部模块的
loader 和 node.js
  // 其中启动的时候，会传入环境参数、loader 函数体，以及
上面拼好的参数数组
  ExecuteBootstrapper(env,
loaders_bootstrapper.ToLocalChecked(),

arraysize(loaders_bootstrapper_args),

loaders_bootstrapper_args,
                              &bootstrapped_loaders)

  // 5. 拼装 node.js 的函数参数数组，分别是 process 和
后面的 bootstrapper
  Local<Value> node_bootstrapper_args[] = {
    env->process_object(),
    bootstrapper,
    bootstrapped_loaders
  };
  // 6. 启动 node.js
  ExecuteBootstrapper(env,
node_bootstrapper.ToLocalChecked(),

arraysize(node_bootstrapper_args),

node_bootstrapper_args,
                              &bootstrapped_node)
}
```

在注释 1 的位置，JS 源码经过 GetBootstrapper 后，会定义成一个可以执行的 C++ 函数，也就是 loaders_bootstrapper，它是 Local 类型的 Function，在 v8 引擎里面，可以通过 call 直接执行它对应的 JS 函数，可以理解为 v8 里面调用 C++ 函数，来运行一

段 JS 代码，另外在执行这个 JS 代码的时候，可以对 JS 里面的函数传入 C++ 构造的一些对象或者函数，这样就达到让被执行的 JS 函数，它里面也能调用到 C++ 层面的函数的目的。

也就是到了注释 4，通过执行 JS 代码来启动模块的 loader，我们看下 ExecuteBootstrapper 的代码，在 [src/node.cc 2094 行 (https://github.com/nodejs/node/blob/v10.x/src/node.cc#L2](https://github.com/nodejs/node/blob/v10.x/src/node.cc#L2) 精简如下：

```
static bool ExecuteBootstrapper(Environment* env,
Local<Function> bootstrapper, int argc,
Local<Value> argv[], Local<Value>* out) {
  bootstrapper->Call(
      env->context(), Null(env->isolate()), argc,
argv).ToLocal(out);
}
```

核心就是 bootstrapper->Call()，来执行 bootstrapper 函数，实际上就是执行 internal/bootstrap/loaders.js 的 JS 函数表达式 (function(){ })，同时对它传入 C++ 生成的 process 对象和 bindings 函数，也就是 loaders_bootstrapper_args 里面的：

```
{
  env->process_object(),  # 对应 process
  get_binding_fn,         # 对应 GetBinding
  get_linked_binding_fn,  # 对应 GetLinkedBinding
  get_internal_binding_fn # 对应
GetInternalBinding
}
```

在 GetBinding GetLinkedBinding 和 GetInternalBinding 里面，又是各自通过 get_builtin_module get_internal_module 和 get_linked_module 来找到对应的

模块以及进行一些初始化工作，这些代码都在 src/node.cc (https://github.com/nodejs/node/blob/v10.x/src/node.cc#L1 里面，可以发现它们也都是通过 FindModule 函数来遍历查找的，关于模块注册和查找我们不再往上面继续深究，继续回来到 loaders_bootstrapper_args 的几个参数，我们此时执行 internal/bootstrap/loaders.js，对它传入这 4 个参数，看下简版的 loaders.js 代码：

```
(function bootstrapInternalLoaders(process,
 getBinding, getLinkedBinding,
getInternalBinding) {
 function NativeModule(id) {}

 return loaderExports;
});
```

发现它所接收的参数刚好是 4 个，跟 loaders_bootstrapper_args 里的参数一一对应，同时这个函数里面，有一个 NativeModule 的函数对应，望名生义，应该就是原生模块了，整个 Loaders 函数执行后，还会返回一个 loaderExports 对象，这个对 internal/bootstrap/loaders.js 是有用的。

大白话翻译下，node.cc 里面从 C++ 层面把 loader.js 的源码拎过来解析执行，同时对它传入几个 C++ 对象，这样就可以从 loaders.js 里面以 getBinding 的形式获取原生模块了，费了这么大力气，终于可以来更新下纸箱子了：

```
纸箱子 = [
  '6.3.1 Node 底层环境均已 Ready，准备装载 JS 模块',
  '6.3.2 node.cc 加载 loaders.js，对 JS 函数传入
process、binding 等 C++ 接口',
]
```

internal/bootstrap/loaders.js 的文档非常详实，我简单翻译下：

首先它是 Node 启动的前置条件：

- loaders 的作用是创建内部模块，以及用来 binding 的 loaders，来给 built-in 模块使用
- 我们自己写的代码，包括 node_modules 下的三方模块，都由 lib/internal/modules/cjs/loader.js
  和 lib/internal/modules/esm/* (ES Modules) 接管处理
- loaders.js 本身最终会被编译，编译后被 node.cc 所调用，等到它生效后，才会去继续调用 bootstrap/node.js
  也就是说，要等到 loaders 启动之后 Nodejs 才算是真正启动

其次，它把 C++ binding 能力挂载到了 process 对象上：

- process.binding(): 是 C++ binding loader, 从用户这可以直接访问
- process._linkedBinding(): 目的是让 C++ 作为扩展被项目嵌入进来引用，本质是 C++ binding
- internalBinding(): 私有内部（internal） C++ binding loader, 用户无权访问，只给 NativeModule.require() 使用

再次，它提供了内部原生模块的 loader 能力：

- NativeModule: 一个迷你的模块系统，用来加载 Node 的核心 JS 模块
    - 这些模块在 lib/**/*.js deps/**/*.js 里面
    - 这些核心模块会被 node_javascript.cc 编译成 node 二进制文件，这样没有 I/O 开销，加载更快
    - 这个类还允许核心模块访问 lib/internal/* deps/internal/* 里的模块和 internalBinding()，也允许核心模块通过 require 加载它，即便它不是一个 CommonJS 的模块

- process.moduleLoadList 则是按照加载顺序，记录了 bindings 和已经 load 的模块

最后，binding 和 loader 的能力，都被放到了 loaderExports 里面，作为函数执行的返回值，以 CommonJS 的方式暴露出去，可以这样理解：

```
module.exports = { internalBinding, NativeModule
}
```

再来更新下纸箱子：

```
纸箱子 = [
  '6.3.1 Node 底层环境均已 Ready，准备装载 JS 模块',
  '6.3.2 node.cc 加载 loaders.js，对 JS 函数传入
process、binding 等 C++ 接口',
    '6.3.2.1 loaders.js 封装了原生模块的加载，同时把加
载能力和 internalBinding 也暴露出去',
]
```

我们再稍微的看下 internal/bootstrap/loaders.js 的源码，它里面一共分为三部分：

首先是往 process 上挂 binding:

```
process.binding = function binding(module) {
  mod = bindingObj[module] = getBinding(module);
};
process._linkedBinding = function
_linkedBinding(module) {
  mod = bindingObj[module] =
getLinkedBinding(module);
}
```

然后就是声明 NativeModule，实现代码编译等操作：

```
function NativeModule(id) {
  this.filename = `${id}.js`;
  this.id = id;
  this.exports = {};
  this.script = null;
}

// require 时代码拎过来组装编译，再把 exports 丢出去，缓存代码都略去不表
NativeModule.require = function (id) {
  const nativeModule = new NativeModule(id);
  nativeModule.compile();
  return nativeModule.exports;
};

// contextify 这个模块的作用就是执行 JS 代码
const { ContextifyScript } =
process.binding('contextify');

NativeModule.prototype.compile = function () {
  // 拿到传入模块的源码，包裹成 CommonJS 的样子
  let source = NativeModule.getSource(id);
  source = NativeModule.wrap(source);

  // ContextifyScript 类上面主要有 RunInContext、RunInThisContext 两个方法
  const script = new ContextifyScript(
    source, this.filename, 0, 0,
    cache, false, undefined
  );
  const fn = script.runInThisContext(-1, true,
```

```
false);
  const requireFn =
this.id.startsWith('internal/deps/') ?
    NativeModule.requireForDeps :
    NativeModule.require;
  fn(this.exports, requireFn, this, process);
};

// internal 这些内部模块不会暴露给用户使用，代码略去不表
NativeModule.requireForDeps = function (id) {
  return
NativeModule.require(`internal/deps/${id}`);
};
NativeModule.wrapper = ['(function (exports,
require, module, process) {', '\n});'];
NativeModule.wrap = (script) =>
(NativeModule.wrapper[0] + script +
NativeModule.wrapper[1])
NativeModule._source = getBinding('natives');
NativeModule.getSource = function (id) {
  return NativeModule._source[id];
};
```

最后，来把 loaderExports 暴露出去：

```
let internalBinding = function
internalBinding(module) {
  let mod = bindingObj[module];
  if (typeof mod !== 'object') {
    mod = bindingObj[module] =
getInternalBinding(module);
    moduleLoadList.push(`Internal Binding
${module}`);
  }
  return mod;
};

const loaderExports = { internalBinding,
NativeModule }

return loaderExports
```

NativeModule 的工作产出，我们来举个简单例子，比如加载 internal/steam.js，源码大概是：

```
const { Buffer } = require('buffer');
const Stream = module.exports =
require('internal/streams/legacy');
Stream.Readable = require('_stream_readable');
Stream.Writable = require('_stream_writable');
Stream.Duplex = require('_stream_duplex');
Stream.Transform = require('_stream_transform');
Stream.PassThrough =
require('_stream_passthrough');
```

那么通过 internal/bootstrap/loaders.js 的 loaderExports 中 NativeModule 加载之后，实际是这样的代码在 v8 里面运行：

```
(function (exports, require, module, process) {
  const { Buffer } = require('buffer');
  const Stream = module.exports =
require('internal/streams/legacy');
  Stream.Readable = require('_stream_readable');
  Stream.Writable = require('_stream_writable');
  Stream.Duplex = require('_stream_duplex');
  Stream.Transform =
require('_stream_transform');
  Stream.PassThrough =
require('_stream_passthrough');
})
```

运行时，里面的 require，在 NativeModule 里面是有区分的，对于 internal 走 requireForDeps，其他模块就是 require：

```
const requireFn =
this.id.startsWith('internal/deps/') ?
    NativeModule.requireForDeps :
    NativeModule.require;
fn(this.exports, requireFn, this, process)
```

总而言之，作为 native 模块的 loader，internal/bootstrap/loaders.js 依然可以看做是准备工作，主要负责原生模块的加载，那我们在项目中写的 JS 是怎么加载进来的呢？比如 server.js 是怎么被加载进去的呢？

真正要让 JS 代码运行起来，还需要 internal/bootstrap/node.js 的赞助：

```
(function bootstrapNodeJSCore(process,
  { _setupProcessObject, _setupNextTick,
_setupPromises, ... },
  { internalBinding, NativeModule }) {
  startup();
});
```

先忽略它第二个通过解构拿到的一坨参数组成的参数对象，我们看第三个参数 `{ internalBinding, NativeModule }` 其实就是我们之前 `internal/bootstrap/loaders.js` 的 loaderExports 所传下来的参数，也就是 binding 能力和 NativeModule 的加载能力，有了这两个能力，我们看下它 startup() 所做的主要事情：

```
function startup() {
  // 通过 NativeModule 拿到 cjs/loader 这个用来加载外部（用户）的 JS loader
  const CJSModule = NativeModule.require('internal/modules/cjs/loader');
  preloadModules();
  // 调用 CJSModule 的 runMain 方法，让代码运行起来
  CJSModule.runMain();
}

function preloadModules() {
  const {
    _preloadModules
  } = NativeModule.require('internal/modules/cjs/loader');
  _preloadModules(process._preload_modules);
}

startup();
```

那么接下来的事情，自然就发生在了 internal/modules/cjs/loader 里面了，其实我们可以这样来测试下调用栈，在本地写一个 test.js，里面放上：

```
require('./notexist.js')
```

我们调用一个不存在的 JS 模块， node test.js 跑一下，会报错如下：

```
# 代码终止在了 583 行，抛出错误
internal/modules/cjs/loader.js:583
  throw err;
  ^

Error: Cannot find module './notexist.js'
  at Function.Module._resolveFilename
(internal/modules/cjs/loader.js:581:15)
  at Function.Module._load
(internal/modules/cjs/loader.js:507:25)
  at Module.require
(internal/modules/cjs/loader.js:637:17)
  at require
(internal/modules/cjs/helpers.js:20:18)
  at Object.<anonymous>
(/Users/black/Downloads/node-10.x/bind.js:1:75)
  at Module._compile
(internal/modules/cjs/loader.js:689:30)
  at Object.Module._extensions..js
(internal/modules/cjs/loader.js:700:10)
  at Module.load
(internal/modules/cjs/loader.js:599:32)
  at tryModuleLoad
(internal/modules/cjs/loader.js:538:12)
  at Function.Module._load
(internal/modules/cjs/loader.js:530:3)
```

调用栈由下向上，依次调用，比如 require('./notexist.js') 就是调用到了internal/modules/cjs/loader _load 方法，我们一一对应整理下来就是：

```
|- loader.js 530 行 _load
 |- loader.js 538 行 tryModuleLoad
  |- loader.js 599 行 load
   |- loader.js 700 行 _extensions
    |- loader.js 275 行 _compile
     |- bind.js 1 行 匿名函数
      |- helpers.js 20 行 require
       |- loader.js 637 行 require
        |- loader.js 507 行 _load
         |- loader.js 581 行 _resolveFilename
```

具体代码的行数大家不用计较，因为 Node 版本不同，跟我们读的源码不一定能对上，但是函数名基本是可以对上的，来把 internal/modules/cjs/loader 代码精简一下，删减到了 50 行，其实跟 NativeModule 差不多，我们找到 Module.runMain 从上向下看：

```
function Module(id, parent) {
  this.id = id;
  this.exports = {};
}

Module.wrap = function(script) {
  return Module.wrapper[0] + script +
Module.wrapper[1];
};
Module.wrapper = [
  '(function (exports, require, module,
__filename, __dirname) { ',
  '\n});'
];

Module.runMain = function() {
```

```
  Module._load(process.argv[1], null, true);
};

Module._load = function(request, parent, isMain)
{
  var module = new Module(filename, parent);

  tryModuleLoad(module, filename);
};

function tryModuleLoad(module, filename) {
  module.load(filename);
}

Module.prototype.load = function(filename) {
  Module._extensions['.js'](this, filename);
};

Module._extensions['.js'] = function(module,
filename) {
  var content = fs.readFileSync(filename,
'utf8');
  module._compile(stripBOM(content), filename);
};

Module.prototype._compile = function(content,
filename) {
  var wrapper = Module.wrap(content);
  var compiledWrapper =
vm.runInThisContext(wrapper, {
    filename: filename,
    lineOffset: 0,
    displayErrors: true
```

```
  });
  var require = makeRequireFunction(this);

  compiledWrapper.call(this.exports,
this.exports, require, this, filename, dirname);
};

Module.prototype.require = function(id) {
  return Module._load(id, this, /* isMain */
false);
};

Module._resolveFilename = function(request,
parent, isMain, options) {};
```

那么我们平时手写的 JS 代码，包括 node_modules 下的代码，就会被这个 CJS Loader 给接管了，拿到代码后的第一件事就是对它包裹一个函数表达式，传入一些变量，我们可以在 node 命令行模式下，输入 require('module').wrap.toString() 和 require('module').wrapper 来查看到包裹的方法：

```
➜  node-10.x_2 node
> require('module').wrap.toString()
'function(script) {\n  return Module.wrapper[0] + script + Module.wrapp
er[1];\n}'
> require('module').wrapper
[ '(function (exports, require, module, __filename, __dirname) { ',
  '\n});' ]
>
```

我们可以拿一段 webpack 源代码举例：

```
// ChunkRenderError.js
const WebpackError = require('./WebpackError')

class ChunkRenderError extends WebpackError {
  constructor(chunk, file, error) {
    super()

    this.name = 'ChunkRenderError'
    this.error = error
    this.message = error.message
    this.details = error.stack
    this.file = file
    this.chunk = chunk
    Error.captureStackTrace(this,
this.constructor)
  }
}

module.exports = ChunkRenderError
```

在 require 的时候，经过 CJS Loader 的编译，就编程了这样子：

```
(function (exports, require, module, __filename,
__dirname) {
  const WebpackError = require('./WebpackError')

  class ChunkRenderError extends WebpackError {
    constructor(chunk, file, error) {
      super()

      this.name = 'ChunkRenderError'
      this.error = error
      this.message = error.message
      this.details = error.stack
      this.file = file
      this.chunk = chunk
      Error.captureStackTrace(this,
this.constructor)
    }
  }

  module.exports = ChunkRenderError
})
```

于是我们很直观的得到两个结论：

- internal/bootstrap/loaders.js 和 internal/modules/cjs/loader 都是 Loader，但作用不同，前者是加载 Native 模块，后者加载我们项目中的 JS 模块，且后者依赖前者
- 前者是非 CommonJS 的 Loader，后者是 CommonJS 的 Loader

扒了这一圈，我们就可以来回答文章开头的问题了：Node 里面的模块规范还是 CommonJS 么？

CommonJS 与 Node Modules

上面提到，虽然基于 CommonJS 来实现模块管理，但 Node 的 modules 体系演化至今，已经自成一套，跟 CommonJS 虽有大量血缘关系，但也确实有不同之处，最明显的，Node 里面 `require('./index')` 的依赖查找有后缀名的优先级，分别是 .js > .json > .node，同时一个模块的入口文件路径，是在 package.json 的 main 里面定义，以及 Node 里面依赖的模块统一在 node_modules 下面管理，这也是 Node 所独有的，还有其他比较大的差异之处，比如：

1. CommonJS 为 require 定义了 main 和 paths 两个静态属性，而 Node 不支持 require.paths， 且暴露了额外的 cache 属性和 resolve() 方法，可以 node 命令行打印 `require`。
2. CommonJS 的 module 对象有 id 和 uri，而 Node 里面增加了 children/exports/filename/loaded/parent 属性，以及 require() 方法，它的接口通过 exports 和 module.exports 对外暴露，而在 CommonJS 里面，暴露模块 API 的唯一办法就是对 exports 对象增加方法或者属性，module.exports 在 CommonJS 里面不存在。

总而言之，就像 Node 社区所说，CommonJS is dead，Node 里的 modules 体系已经不再是严格意义的 CommonJS，只是大家对这个叫法习惯了，现在依然用 CommonJS 来代指 Node 里面的模块规范，而事实上，Node 社区的开发者已经抛弃 CommonJS 而去，只不过里面的大量血液仍源于 CommonJS。

编程练习 - 实现视频数量与时长统计小工具

最后，我们来实现一个小工具，可以检查当前目录里面，所有的 mp4 文件的总时长，我自己平时有下载一些抖音视频，存了一大堆，有时候想要统计下每一类视频，平均是多少时长之类的数据，是个小玩具，简单实现如下：

```
// 这里会有 1.mp4 2.mp4 等几十上百个视频
// 可以用 promise 并发计算时长，最后汇总叠加
// 叠加总时长如果不超过 1 个小时，比如 55 分钟，那就打印
55 分钟
// 如果超过 1 个小时，比如 65 分钟，打印 1 小时 5 分钟
const fs = require('fs')
const path = require('path')
const moment = require('moment')
const util = require('util')
const open = util.promisify(fs.open)
const read = util.promisify(fs.read)

function getTime (buffer) {
  const start =
buffer.indexOf(Buffer.from('mvhd')) + 17
  const timeScale = buffer.readUInt32BE(start)
  const duration = buffer.readUInt32BE(start + 4)
  const movieLength = Math.floor(duration /
timeScale)
  return movieLength
}

function getLocaleTime (seconds) {
  return moment
    .duration(seconds, 'seconds')
    .toJSON()
    .replace(/[PTHMS]/g, str => {
      switch (str) {
        case 'H': return '小时'
        case 'M': return '分钟'
        case 'S': return '秒'
        default: return ''
      }
```

```
    })
}

;(async function () {
  const dir = path.resolve(__dirname + '/video')
  const files = fs.readdirSync(dir).map(file =>
path.resolve(dir, file))
  const videos = await Promise.all(
    files.map(async file => {
      const fd = await open(file, 'r')
      const buff = Buffer.alloc(100)
      const { buffer } = await read(fd, buff, 0,
100, 0)
      const time = getTime(buffer)
      return { file, time }
    })
  )
  const res = {
    '视频总数': videos.length,
    '视频总时长': getLocaleTime(
      videos.reduce((prev, e) => {
        return prev + e.time
      }, 0)
    )
  }

  console.log(res)
  return res
})()
```