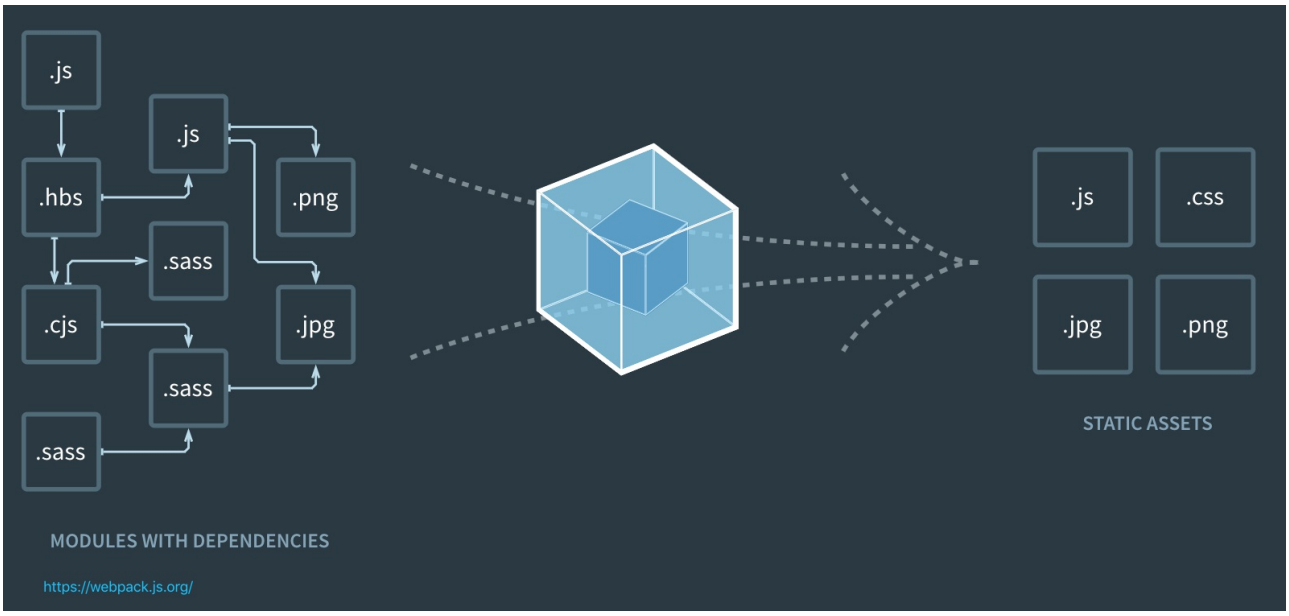
源码挖掘： Webpack 中用到 Node 的 10 个核心基础能力



Webpack v4.23.1 (https://webpack.js.org/)，可以把 JS 文件中依赖的各种资源，分门别类的摘立出来，根据一定的配置规则，该编译编译，该合并合并，该压缩压缩，最终生成干干净净的静态资源文件，那么如此强大的 Webpack，又是站在 Node 的生态和能力之上，一定用到了 Node 的诸多能力，结合小册子，这里面我们可以看看它有用到哪些基础但核心的知识呢?

首先我们安装一个命令到本地： `npm i countapi -g`，然后在 webpack 的目录下执行：`countapi ./`：

```
➜  countapi ./
Node API Used Times
┌──────────────────────┬─────────┐
| module               | count   |
├──────────────────────┼─────────┤
| path                 | 134     |
```

fs	121
util	16
os	8
vm	8
child_process	7
crypto	5
assert	5
events	5
stream	3
url	2
zlib	2
http	2
https	2
http	2
readline	1
dns	1

```
| inspector      | 1     |
├────────────────┼───────┤
| cluster        | 1     |
├────────────────┼───────┤
| net            | 0     |
├────────────────┼───────┤
| v8             | 0     |
├────────────────┼───────┤
| net            | 0     |
├────────────────┼───────┤
| http2          | 0     |
├────────────────┼───────┤
| repl           | 0     |
├────────────────┼───────┤
| perf_hooks     | 0     |
├────────────────┼───────┤
| dgram          | 0     |
├────────────────┼───────┤
| string_decoder | 0     |
├────────────────┼───────┤
| querystring    | 0     |
├────────────────┼───────┤
| worker_threads | 0     |
├────────────────┼───────┤
| async_hooks    | 0     |
```

这里打印了 webpack 仓库源码中，用到的 Node 的一些核心 API 的引用次数，排名靠前的是需要我们关注的，但像断言 （assert）我们当下并不关注，在本册中，我们会结合 webpack 的源码，会选择性的学习 Node 里面较为基础和常用的 API 知识点。

1. Javascript 语言基础

作为 Javascript 社区和 Node 社区的成果，毫无疑问，对于 Node 所支持的 Javascript 语法特性的使用，是搭建 Webpack 最核心的语言基础，那么我们需要关注的第一个核心知识就是：在 Node 里面对于 Javascript(ECMA) 的支持程度以及它主要语法特性的使用，在 Webpack 里面，Promise/Class/Arrow Function/Set/Symbol 等等这些现代的 JS 语法特性也是大规模的使用。

Webpack 有着庞大的开发者阵营和用户阵营，意味着他们无论是参与 Node 社区的建设，还是基于 Node 做工具栈技术栈拓展，Javascript 都是逃不掉的必备技能，Webpack 主仓库 6 万行的 JS 代码也说明了这一点。

```
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
JavaScript                    2303           6539           6767          60728
Markdown                       103           2613              0          17994
JSON                            72              0              0           3504
YAML                             6             95             26            883
TypeScript                      16             35           1264            819
CSS                              8             37              0            190
HTML                            11              0              1            118
CoffeeScript                     6              4              3             58
LESS                             4              2              0             15
Pug                              4              1              0              8
JSX                              2              0              0              2
-------------------------------------------------------------------------------
SUM:                          2535           9326           8061          84319
-------------------------------------------------------------------------------
```

2. Node 的模块/CommonJS 规范

我们掌握了 Javascript 的主要语法知识，还不能在 Node 里面施展拳脚，像 Webpack 里面有 2000 多个 JS 文件，这样的数量级即便再缩小 10 倍，若没有一个机制来管理彼此错综复杂的关系，这个项目依然是梦魇一般的存在。

```
const Compiler = require('./Compiler')
const MultiCompiler = require('./MultiCompiler')
const NodeEnvironmentPlugin =
require('./node/NodeEnvironmentPlugin')
const WebpackOptionsApply =
require('./WebpackOptionsApply')
const WebpackOptionsDefaulter =
require('./WebpackOptionsDefaulter')
const validateSchema =
require('./validateSchema')
const WebpackOptionsValidationError =
require('./WebpackOptionsValidationError')
const webpackOptionsSchema =
require('../schemas/WebpackOptions.json')
const RemovedPluginError =
require('./RemovedPluginError')
```

幸运的是 Node 面世那一年是 2009 年，Javascript 社区中也发展出了 CommonJS 规范，而使用了该规范的 Node 在社区大受欢迎，自此服务端 JS 的模块规范与浏览器端的模块规范走向了不同的道路。

简而言之，在 Node 里面写 JS 代码，CommonJS 的模块包关系如何组织是我们需要掌握的第二个核心知识。

3. Node 的生态能力 - NPM

除了在项目中彼此依赖的模块，Webpack 的主仓库包括它的实际配置场景中，还依赖许许多多的三方 Plugins 和 Loaders，这些三方的模块也会有他们自己的依赖，所有这些依赖会形成一颗深度嵌套的依赖树，源码都分布在 NPM Registry 上面，需要把他们都下载到本地才能使用：

```
"dependencies": {
  "@webassemblyjs/ast": "1.7.8",
  "@webassemblyjs/helper-module-context":
"1.7.8",
  "@webassemblyjs/wasm-edit": "1.7.8",
  "@webassemblyjs/wasm-parser": "1.7.8",
  "acorn": "^5.6.2",
  "acorn-dynamic-import": "^3.0.0",
  "ajv": "^6.1.0",
  "ajv-keywords": "^3.1.0",
  "chrome-trace-event": "^1.0.0",
  "memory-fs": "~0.4.1",
  "mkdirp": "~0.5.0",
  "neo-async": "^2.5.0",
  "uglifyjs-webpack-plugin": "^1.2.4",
  "watchpack": "^1.5.0",
  "webpack-sources": "^1.3.0"
  ...
}
```

这时候就需要用到 Node 包管理工具 - [NPM (https://www.npmjs.com/)](https://www.npmjs.com/) 下载指定模块到当前项目中放到 node_modules 目录下配置使用，那么 NPM 对于 Node 到底是怎么样的一个存在呢，这又是我们学习 Node 必须熟练掌握的技能。

4. Node 的工具集 - path/url/util/zlib

我们扒开 Webpack 的源码库，如果把它的依赖也通过 npm install 后，在这些 node_modules 里面，随便点开一些代码，就能发现满屏的 util 和 path 的使用，比如:

```
/webpack-master/lib/Compiler.js:
336  if (targetFile.match(/\/|\\/)) {
337:   const dir = path.dirname(targetFile)

/webpack-master/lib/ContextModuleFactory.js:
191  files.filter(p => p.indexOf(".") !== 0),
192  (segment, callback) => {
193:   const subResource = path.join(directory,
segment)

/webpack-master/lib/ContextReplacementPlugin.js:
84   if (resourceRegExp.test(result.resource)) {
85     if (newContentResource !== undefined) {
86:      result.resource =
path.resolve(result.resource, newContentResource)
87     }
```

```
201  function getFullPath(id, normalize) {
202    if (normalize !== false) id =
normalizeId(id)
203    var p = url.parse(id, false, true)
...
220  function resolveUrl(baseId, id) {
221    id = normalizeId(id)
222:   return url.resolve(baseId, id)
```

```
/Users/black/Downloads/webpack-
master/lib/Chunk.js:
755  Object.defineProperty(Chunk.prototype,
"forEachModule", {
756    configurable: false,
757:   value: util.deprecate(
758      /**
759       * @deprecated
```

这些常用的工具套件就像 Node 的贴心小助手，从琐碎杂烦的任务中解脱出来，无论是模块的代码结构，还是功能实现的便携程度，都有很大的提升，那么这些工具件方法也是我们需要掌握的，也是使用频次比较高的 API。

5. Node 的文件操作能力 - fs

无论多么松散耦合的 JS 代码，Webpack 都能把它里面不同类型的文件抽离出来，最终经过一系列处理放到某个目标目录下，在这个过程中，就用到了 Node 非常重要的一个能力，就是文件操作能力，比如把文件写入到某个位置，或者复制移动到某个位置，我们可以到 Webpack 的仓库，找到 /lib/webpack 第 11 行 (https://github.com/webpack/webpack/blob/4d3fe000b5f17

```
const WebpackOptionsDefaulter =
require("./WebpackOptionsDefaulter")
```

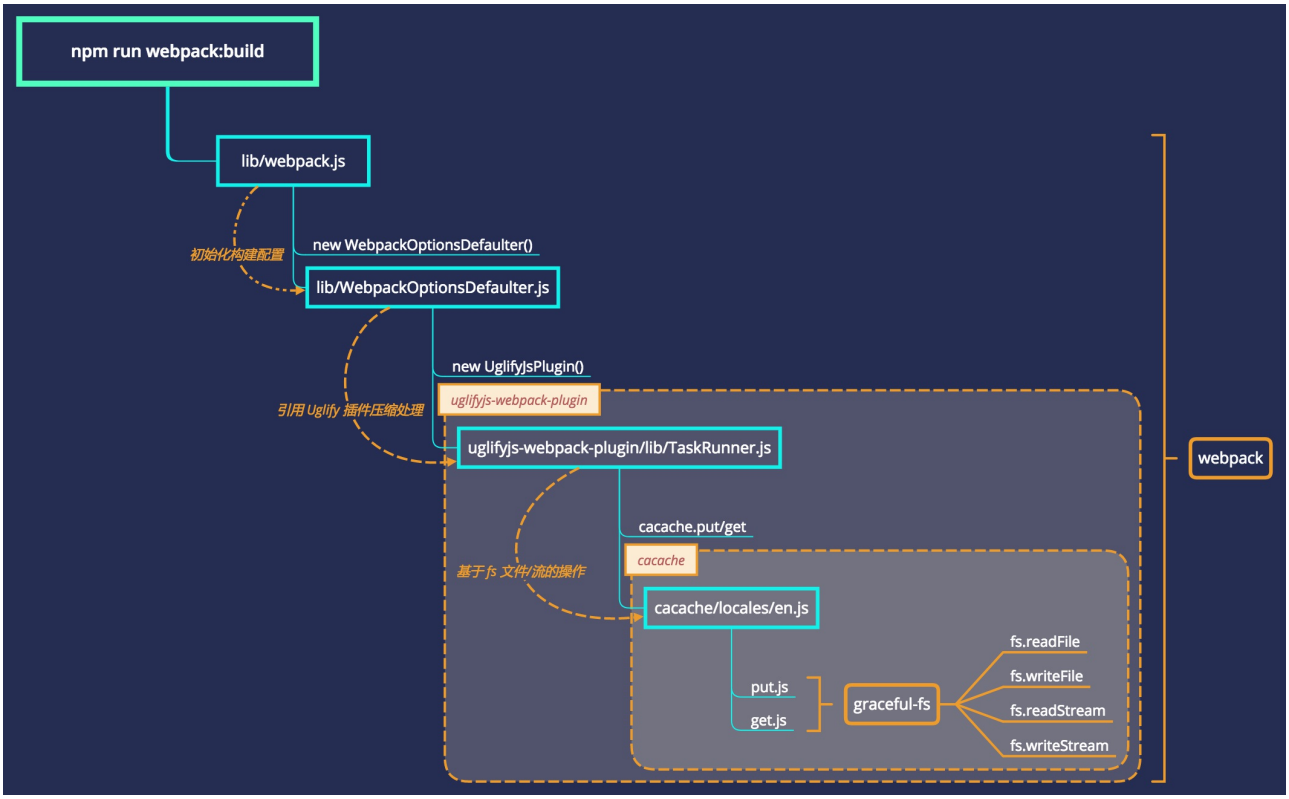
这样一层层找下去，就能发现：

- Webpack 配置文件的读取，就用到了 /lib/WebpackOptionsDefaulter (https://github.com/webpack/webpack/blob/4d3fe000b
- 在 WebpackOptionsDefaulter.js 里面，编译压缩的代码这里依赖了三方模块 uglifyjs-webpack-plugin

(https://github.com/webpack-contrib/uglifyjs-webpack-plugin/blob/master/src/TaskRunner.js#L73)

- 在 uglify 插件中的 src/TaskRunner.js 里面，用到了 cacache/put.js (https://github.com/zkat/cacache/blob/latest/put.js), cacache/get.js (https://github.com/zkat/cacache/blob/latest/get.js)
- 在 cacache/put.js (https://github.com/zkat/cacache/blob/latest/put.js) 里面又用到了 lib/content/write.js (https://github.com/zkat/cacache/blob/latest/lib/conte 以及 graceful-fs (https://github.com/isaacs/node-graceful-fs/blob/master/graceful-fs.js#L65)
- 在 write.js 及 graceful-fs 里面，则是 fs 能力的各种封装了

我们来用一张依赖关系图来展示 Webpack 文件操作能力吧：



可以发现，Node 的 fs 能力作为底层，最终在 webpack 里面，作为插件来支持文件的生成，跨目录的复制转移等等这些骚操作。

文件的操作能力，也就是本地文件资源的交互与操作能力，在 Node 里面是我们需要掌握的又一个核心能力，包括 Path 路径这些周边 API 的配套使用。

6. Node 的缓冲与流

日常使用 Webpack，我们动辄会有几分钟的构建等待时间，这里面有大量的文件读写操作，需要有比较好的机制来保证数据操作效率，基于上面第 5 点的 Webpack 文件操作能力，在源码中其实就能看到大量的 stream 读写的封装了，比如第 5 点的关系图，里面的 cacache 其实还用到了移动复制文件的模块 move-concurrently，而 move-concurrently 则依赖于 copy-concurrently/copy.js 实现拷贝文件的 Promise API:

```
fs.createReadStream(from)
  .once('error', onError)
  .pipe(writeStreamAtomic(to, writeOpts))
  .once('error', onError)
  .once('close', function () {
    if (errored) return
    if (opts.mode != null) {
      resolve(chmod(to, opts.mode))
    } else {
      resolve()
    }
  })
```

在这个 API 里面，就是基于 fs.createReadStream 来对文件内容以流的方式来输送数据，既然提到 stream， Buffer 也就是缓冲也需要了解，那么到底 Node 里面，Stream 和 Buffer 是怎样一个存在，如何使用，这也是一个比较核心的能力等我们掌握。

7. Node 的事件机制 – EventEmitter

基于上面第 6 点，我们处理流的过程中，有 once('error|close') 这样的错误或者流结束这样的事件监听，来挂载对应的回调函数，为什么流的中转过程可以捕获到这样的触发事件呢，我们可以去查看 Node 仓库的源码 – lib/internal/streams/legacy.js (https://github.com/nodejs/node/blob/master/lib/internal/s

```
const EE = require('events')
const util = require('util')

function Stream () {
  EE.call(this)
}
util.inherits(Stream, EE)

Stream.prototype.pipe = function(dest, options) {
  // 伪代码
  source.on('data', ondata)
  source.on('end', onend)
  source.on('close', onclose)
  source.on('end', cleanup)
  source.on('close', cleanup)
  source.on('error', onerror)

  dest.on('drain', ondrain)
  dest.on('error', onerror)
  dest.on('close', cleanup)
  dest.emit('pipe', source)
  return dest
}

module.exports = Stream
```

可以看到最底层的这个 Stream，是继承 events 也就是 EventEmitter，那么自然所有继承了 Stream 的模块，都拥有了 EventEmitter 的特性，可以订阅事件发布事件，为特定事件注册回调函数，甚至我们也可以基于 EventEmitter 来封装我们自己的模块，那么 EventEmitter 这个非常核心的模块到底如何使用呢，我们也会在小册子中进行探讨。

8. Node 的 HTTP 处理 – 请求与响应

以上的基础能力，大多跟跟客户端/本地/系统资源处理有关，一个前端工程师，差不多理解和掌握这几个模块就够用了，但是要继续夯实整个 Node 的基础，为前端开发提供更多可能性，我们就不可避免的会摸到服务端这个领域，也就是 HTTP 的另外一头，服务器上的请求与响应。

而且我们用 webpack 在本地起 dev 环境开发的时候，经常需要用到 Server 来代理静态资源，包括做接口转发，Webpack 插件用到的是 Express，我们基本是插件配置好，就开箱即用，如果想要定制改造，那么不可避免的还是要理解 Node 里面的 HTTP 整套流程。

所以关于 Node 里面 HTTP 的部分，无论是作为服务端响应请求，还是作为请求的发起方，向第三方请求 API，甚至是爬取网页，都逃不开 HTTP 的网络通信能力，那么这个技能的掌握对于我们扩充服务端领域的知识很有意义。

9. Node 的事件循环 – Event Loop

Webpack 的整个打包构建流里面，有着很多在执行的任务，这些任务有的是异步的有的是同步的，比如 webpack-master/lib/Compilation.js 里面等待构建结束时候检查是否有回调队列，如果没有就会定义一个新的 callback 动作去执行：

```
waitForBuildingFinished(module, callback) {
  let callbackList =
this._buildingModules.get(module)
  if (callbackList) {
    callbackList.push(() => callback())
  } else {
    process.nextTick(callback)
  }
}
```

除了 Process.nextTick，还有 setImmediate/Promise 等等这些任务， 到底这些任务到底是按照怎样的规则或者顺序来依次执行呢，答案就是 Event Loop - 事件循环，这也 Node 所谓高性能高并发卖点背后的任务调度机制，任务在哪个时机执行，哪些任务会优先执行，当给我们要深入学习 Node 的时候，事件循环整套流程和特点，就是我们需要掌握的，它会让我们更了解代码的底层运行状况，也是面试中最常问到的一个问题。

```
const EventEmitter = require('events')
class EE extends EventEmitter {}
const yy = new EE()

yy.on('event', () => console.log('粗大事啦'))
setTimeout(() => console.log('0 毫秒后到期的定时器回
调'), 0)
setTimeout(() => console.log('100 毫秒后到期的定时器
回调'), 100)
setImmediate(() => console.log('immediate 立即回
调'))
process.nextTick(() =>
console.log('process.nextTick 的回调'))

Promise.resolve().then(() => {
  yy.emit('event')
  process.nextTick(() =>
console.log('process.nextTick 的回调'))
  console.log('promise 第一次回调')
})
.then(() => console.log('promise 第二次回调'))
```

10. Node 的进程集群 - Cluster

基于上面的 HTTP，我们只要摸到服务端，就需要对所谓单线程事件驱动的高并发 Node IO 模型，能从自己嘴中讲出个所以然来，也要了解它的优劣势，那么如果再放大下场景，我们来对 Node 服务进行横向扩展，让它可以支持更多的并发，更好的利用 CPU，应该怎么做呢？那么 cluster 就是 Node 给出的答案，在小侧子里面，我们也会来探讨下 cluster 在 Node 中的使用，以及有哪些更成熟的工具，比如 PM2 的配套使用。

总结

我们拿 Webpack 只是个举个例子而已，其他前端构建框架也都类似，总而言之，现在前端的整个工程工具体系都是站在了 Node 的生态之上，而抛开生态就是 Node 的单兵作战能力，我们不会为了学 Node 而学 Node，而是通过学习 Node 让自己能摸到更多的知识领域，能带来更多视角和边界的觉醒，同时能给自己带来更多竞争力的知识积累，这就可以作为我们早期学习 Node 的动力，那接下来我们就针对 Webpack 刨出来的这些技能知识点开始学习和练习吧。