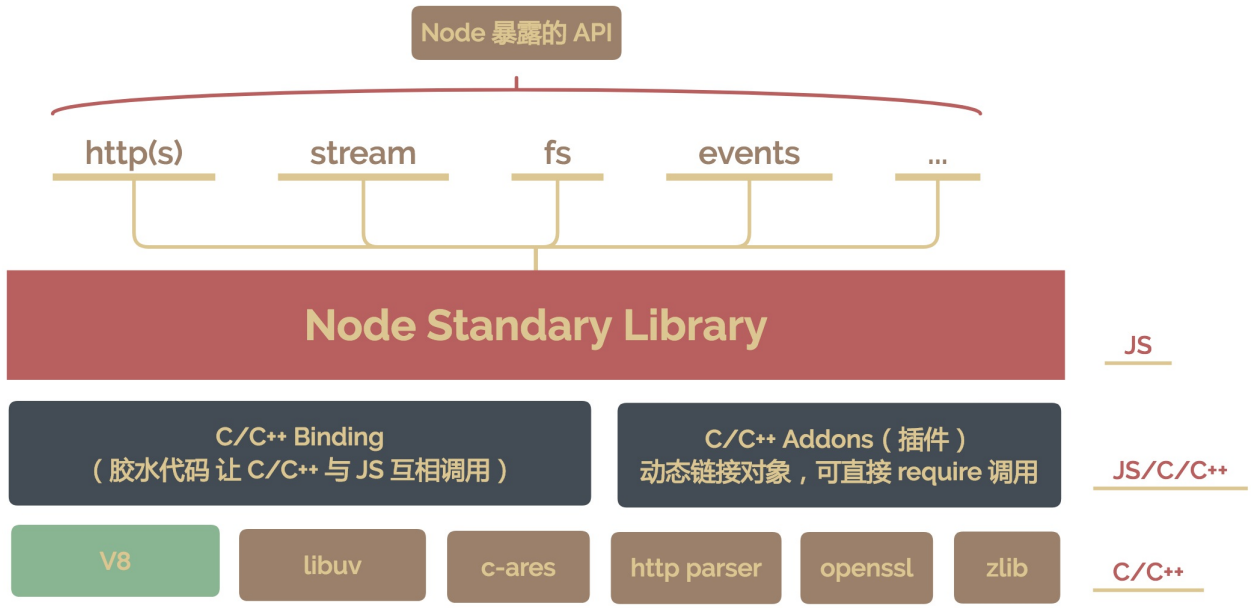
源码解读：Node 的程序架构及启动流程

> 本节有一定理解难度，建议新手同学在完成前面章节后，再来消化本节。



通常，网上搜 Node 的架构或者源码，经常搜到这样一张图，大体把 Node 分为了 3 层：

- 第一层是对外暴露的 API，比如 fs/buffer/net 等，直接 require 进来用
- 第二层可以看做是桥接层，一头连 JS，一头连 C++，让这两种不同语言直接借助 layer 互相调用，比如 Node 项目中针对底层模块所封装的各种 bindings，或者我们可以直接从外部来引入 C++ 模块作为插件使用，通过 JS 直接调用第三方 C++ 模块
- 最后一层，就是 Node 整个底层所依赖的一坨 C/C++ 库，包括提供 JS 解释与运行的 v8 引擎，提供 crypto 加密算法的 openssl 等等。

那么这三层是如何分工协作的，他们的关系是什么，内部调用机制如何，我们先埋下一个伏笔在这里。

一切美好的事情，总是从源码开始

我们首先在命令行里输入 node（输入 .exit 则是退出）进入命令行模式，然后输入 global 回车，可以看到类似下面的一坨内容：

```
Object [global] {
  ...
  global: [Circular],
  process:
   process {
     execArgv: [],
     argv: [
'/Users/black/.nvm/versions/node/v10.11.0/bin/nod
e' ],
     env: { ..., _:
'/Users/black/.nvm/versions/node/v10.11.0/bin/nod
e' },
     moduleLoadList:
      [ 'Binding contextify',
        'NativeModule buffer',
        'Binding fs',
        'Binding v8', ... ],
     binding: [Function: binding] }},
  Buffer: { [Function: Buffer] },
  setImmediate: { [Function: setImmediate] },
```

可以看到一个 global 的对象上挂载了一堆的属性，比如 setTimeout process 都是可以直接访问的， 如果大家对浏览器熟悉，会知道在浏览器里面会有一个顶层全局变量 window，在 window 里面有 setTimeout alert 等各种属性或者方法。

可以这样简单理解，Node 里面有一个顶层全局对象 global，我们所写的所有 JS 代码，都是活跃在这个 global 下面，就像我们网页上的 JS 变量/函数都活跃在 window 下面一样，那么 window 或者 global 可以想象它是存在于某一个 context 或者说沙箱（说盒子也行吧）里面，这个沙箱呢是在 v8 的引擎实例里面运行的，也就是说，浏览器里的代码也好，我们所写的 Nodejs 代码也好，都运行在这个 Chrome v8 里面，在 v8 实例的 context 里面，我们具备访问 window/global 的能力。

在 global 下面，有一个很重要的对象 process，我们继续命令行输入： process.moduleLoadList，可以看到所有按照打印的顺序所加载进来的模块列表：

```
[ 'Binding contextify',
'Internal Binding worker',
'NativeModule events',
'NativeModule internal/async_hooks',
'Binding uv',
'NativeModule util'
...
```

这些模块有很多，如果再仔细辨认一下，会发现主要有这样几种：

- Binding 的一类
- Internal Binding 的一类
- NativeModule 的一类
- NativeModule internal 的一类

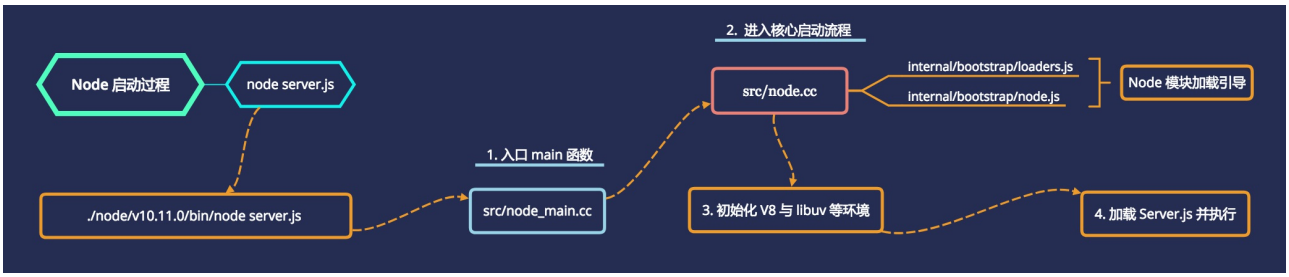
我们继续命令行输入：module，会打印出来：

```
Module {
  id: '<repl>',
  exports: {},
  parent: undefined,
  filename: null,
  loaded: false,
  children: [],
  paths:
   [ '/Users/black/Downloads/node-
10.x/repl/node_modules',
     '/Users/black/Downloads/node_modules', ... ]
}
```

发现 module 也是一个对象，还有模块 id 啊，文件名称 filename 等等这些属性，继续命令行输入 require：

```
{ [Function: require]
  resolve: { [Function: resolve] paths:
[Function: paths] },
  main: undefined,
  extensions: { '.js': [Function], '.json':
[Function], '.node': [Function] },
  cache: {} }
```

发现 require 是一个函数而已，通过命令行，我们可以看到许多 Node 运行中提供的对象、方法和属性，比如 process module require，那它们都是怎么来的呢，我们还是得回到源头。



源码里面藏着一切答案，我们就从源码开始吧，本册源码基于 v10.x (https://github.com/nodejs/node/tree/v10.x)，下载地址 https://github.com/nodejs/node/archive/v10.x.zip (https://github.com/nodejs/node/archive/v10.x.zip)，不同版本的源码差异有大有小，但整体加载流程大概一致，首次阅读源码，建议以本册下载的版本为准。

首先在本地，创建一个 server.js，写入如下代码：

```
const path = require('path')

console.log(path.resolve(__dirname,
'./server.js'))
```

这段代码做的事情，是导入 path 模块，通过 path.resolve 拼接当前 server.js 的完整路径，用 console.log 打印出来，在我的电脑上打印结果是：/Users/black/Downloads/node-10.x/server.js。

node server.js 的时候发生了什么?

在对 node 不熟悉的时候，我们发挥想象力，凭空猜测一下：node 应该是一个可执行程序，可能跟 windows 上的 exe 差不多，只不过是以命令的形式在终端上调用了一下，然后告诉它来把 server.js 代码运行一下，这个代码通过 require 加载了 path，运行后的结果它通过 console.log 再告诉我们电脑的命令行（终端），打印出来，整个运行过程的细节我们可能是不清楚的，特别是 node 是如何启动的，如何把 server.js 加载进来，以及如何提供运行环境来执行这个 JS 文件的，今天我们只关注比较粗的大流程，细节和深度方面都不涉及，大家不用担心难度，整本小册也会尽量多配插图，帮助大家消化理解。

开始之前，我们先定义一个便签纸篓子，来存放我们阶段性得出的结论，以便于我们脑海中形成记忆：

```
let 纸篓子 = []
```

Node 的源码目录和 C++ 代码占比

```
-------------------------------------------------------------------------------
Language                              files          blank        comment           code
-------------------------------------------------------------------------------
C++                                    1879         169825         139881        1344851
JavaScript                            13590         162983         225324        1165656
Perl                                    379          20151          17465         639110
Assembly                                305          40620           7776         490674
C                                      1277          57788          53170         361706
C/C++ Header                           2031          81289         184455         343021
```

Node 的整个底层代码，大量使用 C/C++，JS 和 C/C++ 各 100 多万行，我们再看下 Node 的源码主要目录结构（暴力删减版）：

```
.$ /Users/black/Downloads/node-10.x/
├── deps                    # Node 各种依赖
|   ├── acorn                 * Javascript 解析库
|   ├── cares                 * 异步 DNS 解析库
|   ├── gtest                 * C/C++ 单元测试框架
|   ├── http_parser           * C 语言的 http 解析库
|   ├── icu-small             * 跨平台 unicode 编码集
|   ├── nghttp2               * HTTP/2 协议库
|   ├── node-inspect          * Node 调试工具
|   ├── npm                   * Node 包管理工具
|   ├── openssl               * 通信/算法加密库
|   ├── uv(libuv)             * C 语言封装的异步 I/O 库
|   ├── v8                    * 提供 JS 的运行环境的 vm
|   └── zlib                  * 数据压缩解压的类库
├── lib                     # 原生 JS 模块库
|   ├── fs.js
|   ├── http.js
|   ├── buffer.js
|   ├── events.js
|   ├── internal
```

```
|   |   ├── bootstrap
|   |   |   ├── cache.js
|   |   |   ├── loaders.js
|   |   |   └── node.js
|   |   ├── http.js
|   |   ├── modules
|   |   |   ├── cjs
|   |   |   └── esm
|   |   ├── process
├── src                     # Node 底层源码
|   ├── node_main.cc         * Node 启动的入口
|   ├── node.cc              * Node 的启动主逻辑
└── tools                   # 编译所需要的工具
```

```
纸篓子 = [
  '1. Node 源码有一坨依赖，大部分是 C/C++ 底层'
]
```

Node 初步启动 - 调用入口函数 main

我们大学上 C/C++ 语言课，可能对这坨代码印象比较深刻：

```
int main(void) {

}
```

没错，main 函数就是 C 语言世界里的程序入口了，我们到 Node 目录下，找到 node-master/src/node_main.cc (https://github.com/nodejs/node/blob/v10.x/src/node_main 第 124 行，核心就干了一点事，根据操作系统干了些处理额外参数的活儿，就跑去调用 Start 函数了。

```
94 int main(int argc, char* argv[]) {
25  #ifdef _WIN32
..
72:   return node::Start(argc, argv);
73  }
74  #else
..
124:   return node::Start(argc, argv);
125  }
126  #endif
```

```
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',
]
```

进入多层 Start 函数跑通主逻辑

到 node-master/src/node.cc (https://github.com/nodejs/node/blob/v10.x/src/node.cc#L3 里面能找到好几个 Start 函数以及函数调用，我们从 2891 行 (https://github.com/nodejs/node/blob/v10.x/src/node.cc#L2 2987 行 (https://github.com/nodejs/node/blob/v10.x/src/node.cc#L2 3034 行 (https://github.com/nodejs/node/blob/v10.x/src/node.cc#L3 能拎出来这 3 个 Start 函数定义，他们存在依次调用关系。

```
inline int Start(Isolate* isolate,..
inline int Start(uv_loop_t* event_loop...
int Start(int argc, char** argv) {
```

```
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',
]
```

先来看第一个 Start 函数，它里面的入参 (int argc, char** argv)，跟我们在上面 main 函数里面，调用 Start 的入参保持一致，然后开始各种忙活，比如 Init v8 参数处理和 v8 初始化啊，最后清理战场，帮助 Node 退出，我们关注 3034 行 (https://github.com/nodejs/node/blob/v10.x/src/node.cc#L3 这里：

```
int Start(int argc, char** argv) {
  // 注册内置模块/参数预处理等工作
  Init(&args, &exec_args);
  // v8 初始化
  V8::Initialize();

  Start(uv_default_loop(), args, exec_args);
}
```

```
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 做了初始化工作，调用第二个 Start',
]
```

这里的 Start(uv_default_loop(), args, exec_args) 调用了第二个 Start 函数，且对这个 Start 传了 3 个参数，第一个参数是一个函数，直接执行掉了，它来初始化了 Node 的事件循环，也

就是 Event Loop，后两个参数略去不表，我们继续前往第二个 Start，也就是 2987 行 (https://github.com/nodejs/node/blob/v10.x/src/node.cc#L2

```
inline int Start(uv_loop_t* event_loop,
                 <std::string>& args,
                 <std::string>& exec_args) {
  // 生成一个独立 v8 引擎实例，所有 JS 代码都将丢到它里面执行
  const isolate = NewIsolate(allocator.get());

  // 配置 v8 的引擎实例和里面的工作区，准备干活
  Locker locker(isolate);
  Isolate::Scope isolate_scope(isolate);
  HandleScope handle_scope(isolate);
  ...

  // 第三个 Start，传进去引擎实例，准备编译我们的 JS 代码了
  Start(isolate, isolate_data.get(), args, exec_args);
}
```

```
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',
]
```

如上面代码中的注释，前面都是准备工作，我们继续前往 2891 行 (https://github.com/nodejs/node/blob/v10.x/src/node.cc#L2
这个 Start 有点贪心，做的事情太多，我们先精简下：

```
inline int Start(isolate, isolate_data,
                 args, exec_args) {

  // 先在一个引擎实例中准备 v8 上下文
  Context::Scope context_scope(context);

  // 拼凑起来一个 node 启动环境，然后把它收拾舒服
  // 比如 libuv 事件循环，process 全局变量之类
  Environment env(isolate_data,
      context,
v8_platform.GetTracingAgentWriter())
  env.Start(args, exec_args, v8_is_profiling);

  // 把原生模块和我们的 JS 代码加载进来
  LoadEnvironment(&env);

  // libuv 上场，不断轮询有没有活儿干
  uv_run(env.event_loop(), UV_RUN_DEFAULT);
  more = uv_loop_alive(env.event_loop());
  if (more) continue;
  more = uv_loop_alive(env.event_loop());

  // 没活儿了就抛 exit 事件，拼命退出进程，各种清理工作
  EmitExit(&env)
  env.RunCleanup();
  RunAtExit(&env);
}
```

跟着上面的注释，我们获取到了一个信息，那就是这个 Start 把所有的活儿都干了，既有场地准备，又有各种环境条件准备，又把演员们（模块和 JS） 拉过来，表演一个又一个节目，节目演完了就收拾场地跑人，来把纸篓子扩充下：

```
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',
]
```

对于 6.3 的 LoadEnvironment(&env)，它是台柱子，代码在 [2120 行](https://github.com/nodejs/node/blob/v10.x/src/node.cc#L2

```
void LoadEnvironment(Environment* env) {
  // 先把 loaders.js 和 node.js 的代码拎过来
  // 配置各种对象、变量，如全局对象 global，内置模块的
bind 等等
  loaders_name = ...
("internal/bootstrap/loaders.js");
  loaders_bootstrapper = ...
(LoadersBootstrapperSource(env), loaders_name);
  node_name = ...("internal/bootstrap/node.js");
  node_bootstrapper = ...
(NodeBootstrapperSource(env), node_name);

  // 把 global 全局对象挂载到 context 上
  Local<Object> global = env->context()-
>Global();
  global->Set(env->isolate(), "global", global);

  // 各种配置函数的参数后，执行 Bootstrap 的
loaders.js 和 Node.js
  ExecuteBootstrapper(env, loaders_bootstrapper,
    loaders_bootstrapper_args,
&bootstrapped_loaders))
  ExecuteBootstrapper(env, node_bootstrapper,
    node_bootstrapper_args, &bootstrapped_node))
}
```

这两个 JS 是靠 [2099 行 ExecuteBootstrapper (https://github.com/nodejs/node/blob/v10.x/src/node.cc#L2](https://github.com/nodejs/node/blob/v10.x/src/node.cc#L2) 执行的，在它里面，很简单，就是通过 bootstrapper->Call() 来调用执行。

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

调用执行后，Node 里面的模块系统就 Ready 了，有了模块系统，我们各种 require 就能跑起来了。

至此，Node 启动过程一日游结束，再来看下纸篓子：

```
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',
]
```

简单总结一下，Node 的运行是按照一定的顺序，来分别把 v8 启动，libuv 初始化，再把 v8 实例创建，Context 准备好，最后把模块代码导进来，最后把 libuv 跑起来，按照一定策略执行模块代码里的任务，直到任务跑完。

这里面还有 1 个遗留问题：

- 到底 loaders.js 和 node.js 是怎么让我们的模块系统生效的?

第一个问题，它的背后涉及到 require/exports 如何生效，Node 里面的模块和我们 npm install 的模块是如何加载进来工作的，是非常核心的基础知识，我们在 [视频时长统计] Node 的模块

机制（CommonJS）与包管理
(https://juejin.im/editor/book/5bc1bf3e5188255c3272e315/
有过探讨。

思考

最后给大家留一个小作业，比如 Node server.js 的这个 server.js 作为被执行文件的路径参数，到底是如何一层层传下来的，以及从 main 到 3 个 Start，到 bootstrapper->Call()，中间有引擎实例啊，上下文啊各种参数，它们一路传下来，参数也经过不断加工，在每个环节又各是什么意思，大家可以自己尝试思考下，结合源码在本地画一画答案。