# Node模块实现

2.2

Node在实现它们的过程中经历了什么，这个过程要了解。

在node引入模块需要经过三个步骤：

1. 路径分析
2. 文件定位
3. 编译执行

在node中，模块分两类，一是node自带的核心模块，二是用户编写的文件模块。

* 核心模块在node源码编译时，编译进了二进制执行文件。在node进程启动时，部分核心模块就被加载进了内存中，所以核心模块省去了文件定位和编译执行两个过程，并且路径分析优先判断，加载速度最快。
* 文件模块在运行时加载，需要完整的路径分析、文件定位、编译执行三个过程，所以加载比核心模块慢。

2.2.1 优先从缓存中加载。

浏览器对引用过的文件会进行缓存，node类似，对引用过的模块会进行缓存，并且缓存的格式是编译和执行过的对象。

引入模块时，优先从缓存中加载，优先级比核心模块高。

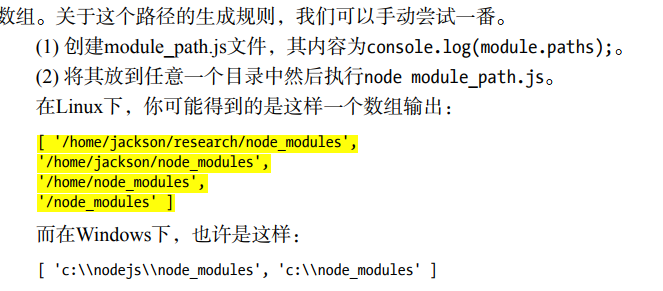
2.2.2 路径分析和文件定位

1.路径分析

* 核心模块优先级仅次于缓存，自定义模块的名字不得与核心模块重复，否则不会加载自定义模块。
* 文件模块以 .、..、/为开头，require（）方法会先将其转换为真实路径，然后再文件定位、编译执行。

由于指定了真实路径，引入模块比较快，但比核心模块慢。

* 自定义模块，一种特殊的文件或者包形式。这类模块查找最费时。



沿路径向上逐级递归，直到根目录下的node\_modules。

文件的路径越深，模块的查找越耗时。

1. 文件定位

文件定位的过程中有些细节需要注意，其中包括文件扩展名的分析、文件目录和包的处理。

* 扩展名分析require()过程中会出现文件没有扩展名的情况，此时就按照js，json，node依次尝试补齐扩展名。

尝试过程用fs模块同步判断文件是否存在，所以这里可能出现堵塞进程的耗性能情况。这里建议不是js扩展名的都补齐。

* + 目录分析和包

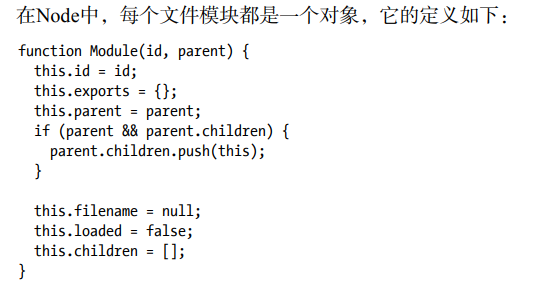
通过分析文件扩展名之后，没有找到文件，则可能找到一个目录。

首先node在当前目录下查找package.json，找到main属性指定的文件名进行定位。

若果main属性指定的文件名错误或者没有package文件，则默认使用index文件名一次查找index.js .json .node文件。

如果目录分析过程没有找得到任何文件，则进行模块路径数组的下一个数组里查找。全都没有找到则抛出错误。

2.2.3 模块编译

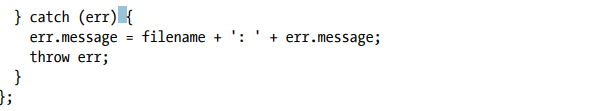
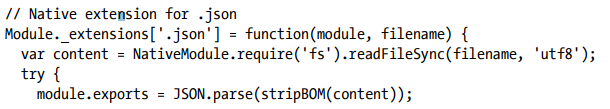


编译和执行是引入文件模块的最后一个阶段，定位到具体文件之后node会新建一个模块对象，然后根据路径载入并编译。不同扩展名载入方法也不同。

* .js文件。通过js模块同步读取文件后编译执行
* .node文件。通过C/C++编写的扩展文件，通过dlopen（）方法加载再编译。
* .json文件。通过fs模块同步读取文件，再用JSON。Parse（）解析返回结果。
* .其余扩展名文件。都当做js文件载入

每一个编译成功的模块都会以文件名为key缓存在Module。\_cache对象上。

如果扩展名为json，则调用



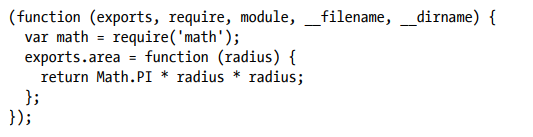
其中Module.extension会复制给require.extension，可以通过log打印出extension来查看，得到如下结果

console.log(require.extensions);

{ '.js': [Function], '.json': [Function], '.node': [Function] }

如果想对自定义的扩展名进行特殊的加载，可以通过require.extensions['.ext']的方式实现。早起的coffeescript就是通过添加require.extensions['.coffee']来实现。但从v0.10.6版本开始，官方不鼓励通过这种方式来加载，希望先将其他语言转成js再引入，以减少繁琐的编译加载过程。

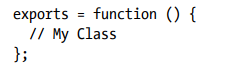
1. js模块的编译



每个文件都会经过这样的封装，然后自带5个变量。

在执行之后，exports属性上的任何属性或者方法都会返回给调用方。Exports会被外界访问到，但是其余的变量则不可直接被调用。

此外还有module.exports，理想情况只要传值给exports就行。但是exports方法返回的都只能是json对象，如果想返回一个function时，一般首先会想到如下方法：



但是会得到一个失败的结果，因为exports是一个参数，这种方法就直接改变了形参的引用，所以会将函数赋值给module.exports。

1. C/C++模块编译

Node调用process.dlopen方法进行加载和执行，不需要编译。

1. JSON文件编译

Node通过fs模块读取，JSON.parse过后，直接赋值给exports。

一般一些配置文件会使用json后缀，用require读取json模块node会对其进行缓存。

2.3核心模块

首先将所有的内置js代码转换成C++里的数组，以字符串形式保存起来，加载进内存。

再通过与文件模块一样的函数封装，导出exports对象，不过核心模块是从内存中获取源码，编译完保存的位置也不同。

核心模块源文件通过process.binding('natives')取出，编译成功后保存于NativeModule.\_cache对象，而文件模块在保存在Module.\_cache对象。



2.3.2 C/C++核心模块编译过程

核心模块中有些由C/C++全部编写，有些则C/C++完成核心部分，其他部分由js实现包装活向外导出。后者为常见方式。Node的buffer、crypto、evals、fs、os等模块都是部分通过C/C++编写。

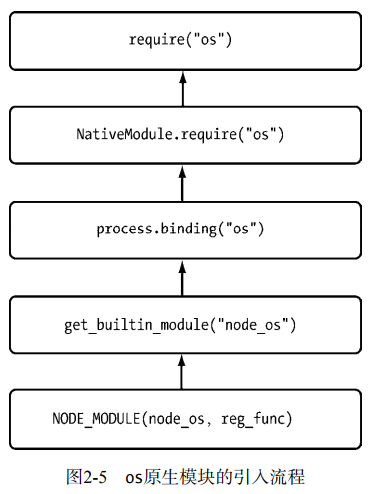
1. 内建模块组织形式

node\_extensions.h文件这些内建模块统一放进了一个node\_module\_list的数组中。Node提供了get\_builtin\_module()方法从node\_module\_list数组中取出这些模块。

1. 内部模块导出

通常，文件模块先调用核心模块，核心模块再调用内建模块方式来调用内建模块。

引入流程如下



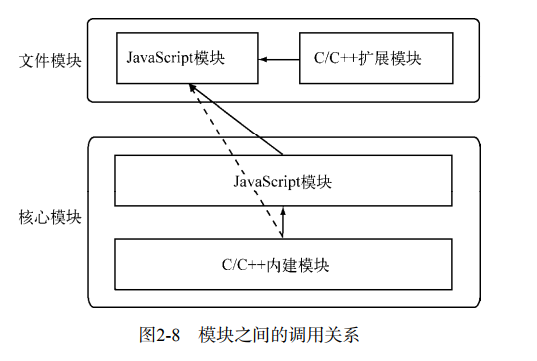
2.3.4 编写核心模块 见书25页（pdf43张）

2.4 C/C++扩展模块

Js位运算符效率很低，因为js都是double类型数据，需要先将数据转换为int类型才能进行为运算，这时候就可以使用C/C++来进行计算。书27页（pdf45张）

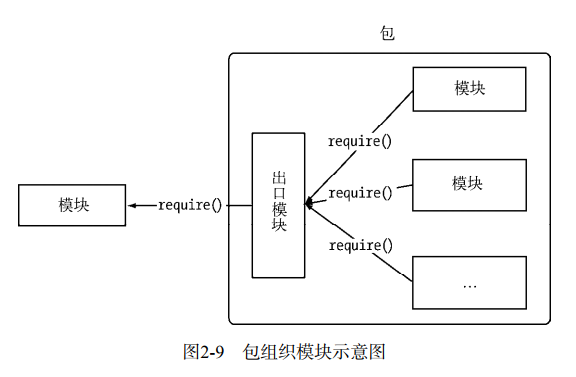
2.5 模块调用栈

各模块之间的关系如图

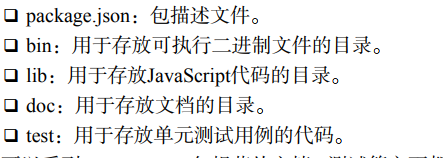


如果不是对C/C++模块非常熟悉的话，不推荐使用process.binding()直接调用。

2.6 npm和包



2.6.1 包模块结构

包是存档文件，即一个目录直接打包为.zip或.rar.gz文件，安装后解压还原为目录。一个包一般包含以下文件。

2.6.2 包描述文件和npm

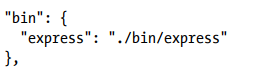
Package.json一般需要包含以下字段：

* Name：包名，可以有.、\_、-符号但是不允许有空格。
* Description：包介绍。
* Version：版本号。
* Maintainers：包维护者。
* Contributors：贡献者。
* Bugs：可以反馈bug的地址。
* Licenses：包许可证。
* Repositories：托管源码的地址。
* Dependencies：依赖包，npm会根据这个属性自动加载依赖。
* Homepage：当前包的网页地址。
* Os：支持的操作系统。
* Cpu：支持的cpu。
* Engine：支持的js引擎。
* Builtin：是否存在底层系统的标准组件。
* Directories：包目录说明。
* Implements：实现的common.js哪些规范。
* Scripts：脚本说明。主要被包管理器用来安装、测试、编译、卸载。

2.6.3 npm日常使用

全局模式安装

全局模式并不是将一个包安装为全局包，并不意味着可以在任何地方调用这个模块，它会根据package.json中的bin描述字段配置



所以的全局安装的包都存在

path.resolve(process.execPath, '..', '..', 'lib', 'node\_modules');

目录下

从本地安装

对于没有发布或者因为网络原因无法下载的npm包，可以将包下载到本地，通过本地安装。本地安装可以通过一个包含package.json的存档文件，也可以是一个url，或者一个包含package.json的文件目录。

npm install <tarball file>

npm install <tarball url>

npm install <folder>

从非官网安装

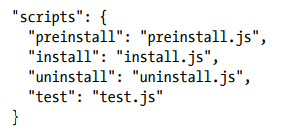
可以通过镜像源进行安装：

npm install express --registry=http://registry.url

或者更换下载源：

npm config set registry http://registry.url

Npm钩子命令

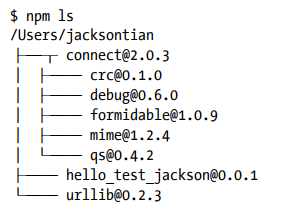


npm install <package>时，preinstall指向的脚本会先加载，然后加载install指向的脚本。在npm uninstall <package>时，uninstall指向的脚本也会执行。

发布包 书39页（pdf57张）

分析包

npm ls命令可以输出模块路径下的所以包以及相关依赖。



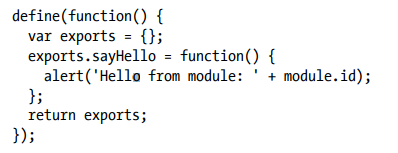
2.6.4 局域npm

企业内部搭建npm 书42页（pdf60张）

2.7前后端公用模块

2.7.2 AMD规范

define(id?, dependencies?, factory);

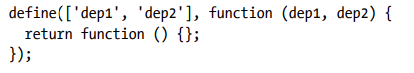


其中id和规范是可选的，不同commonJs的是，AMD需要自己包装内容，以返回内容形式exports。

2.7.3 CMD规范

和AMD规范不同之处在于定义模块和依赖引入部分。

AMD需要在声明模块的时候指定所以依赖



与AMD相比，CMD更接近node的commonJs，CMD规范可以动态引入依赖，与node相同。

