1. nodejs

**1.异步IO的优势**

（1）面向网络是node选择异步的重要原因。如同在浏览器中单线程运行的js，如果网页需要临时获取1个网络资源，假如使用同步，将出现卡顿，导致用户体验非常差。同理，node使用单线程，同时使用异步IO，充分利用cpu时间分片，加快响应速度。

（2）因多线程存在锁、状态同步等问题，所以顺序执行的单线程更符合人的思维方式。为弥补无法利用多核cpu能力的缺点，node提供类似前端的webworkers的子进程，以高效利用cpu。

**2.node中的异步IO**

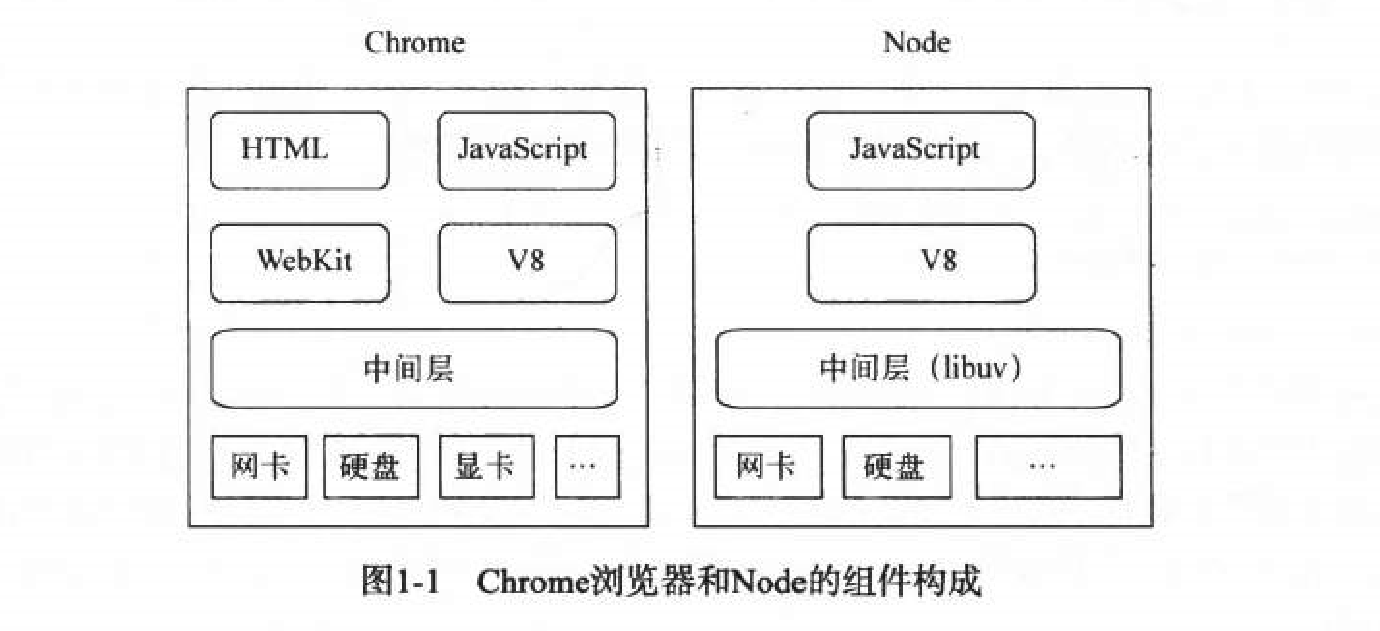
（1）node异步IO设计事件循环、观察者、请求对象等概念。

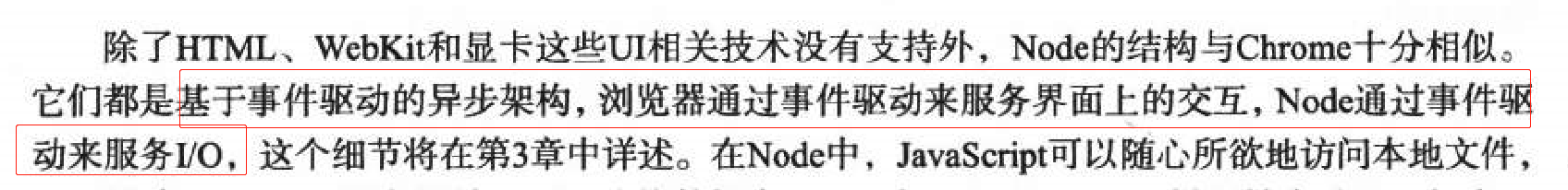
（2）进程启动时，node便会创建1个类似while(true)的循环。没执行1个循环就被称为tick。

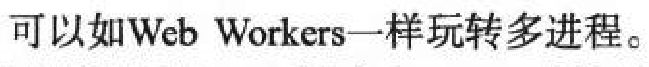
对于普通的异步函数，由编程者自己调用。而对于异步IO的回调函数，则不由开发者调用。从js发起调用到内核执行完IO操作，存在1种中间产物，叫做请求对象。

高性能服务器要点：事件驱动、非阻塞式。

V8是js引擎、webkit是布局引擎。

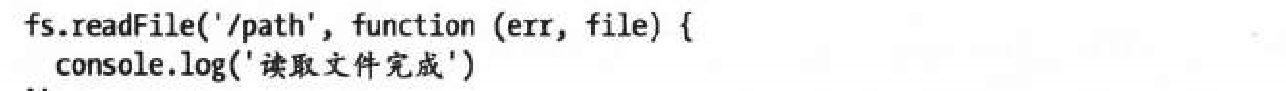




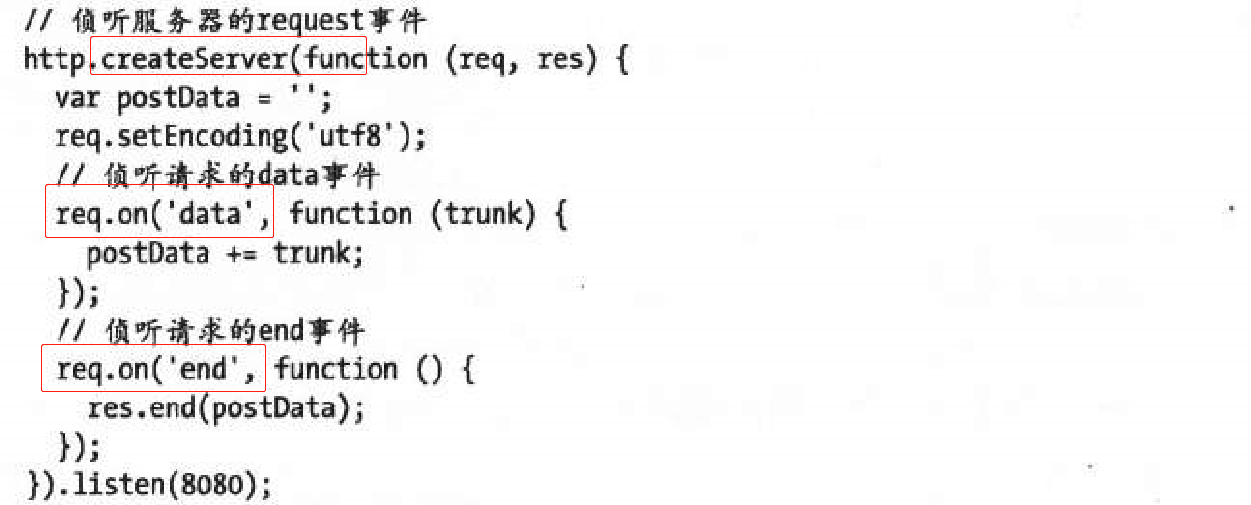


nodejs特点

异步I/O。如图ajax，并不知道明确收到响应的明确时间。node在底层构建了很多异步IO的api，从而可以在语言层面，进行并行IO操作。



事件驱动。如ajax，当成功的时候会调用绑定的success回调。在node中，如创建1个web服务器，可以为服务器绑定request事件，并未请求对象request，绑定data和end事件。



事件编程方式具有轻量级、松耦合、只关注事务点等特点，但在多个异步场景下，事件之间相互独立，如何协作是1个问题。

回调函数。在js中，函数作为第一公民来对待。无论是异步还是事件，回调无处不在。

单线程。js与其他线程不共享任何状态。好处在于不用像多线程编程在意状态同步问题，缺点为：1）无法利用多核cpu；2）大量计算会占用cpu导致无法继续调用异步IO；3）错误会导致整个应用退出。不过后来有了child\_process方案（如同h5种的webworkers）。

node应用场景

IO密集型

* 1. 模块机制
     1. 概要

（1）node借鉴CommonJs的Modules规范实现自身的模块系统。

（2）CommonJs规范中，由require方法引入模块；由module对象的exports对象导出模块的方法或变量。

* + 1. node模块系统的实现

**1.模块分类**

（1）node模块分为两类：node提供的核心模块和用户编写的文件模块。

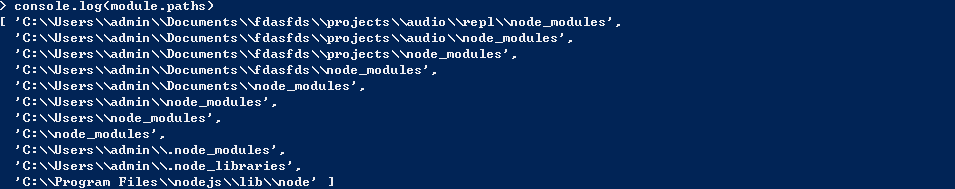
（2）文件模块是在运行时动态加载，会经历路径分析、文件定位、编译执行等过程。

（3）无论是核心模块还是文件模块，相同模块的加载都会优先从缓存中加载。

**2.模块标识符**

（1）在node中主要有：1）核心模块，如http、fs等；2）相对路径的文件模块；3）绝对路径的文件模块；4）自定义模块。

（2）自定义模块可能是1个文件或者是1个包，查找最为耗时，会按照模块查找顺序查找。如在1个项目中，可以使用module.paths变量查看模块查找顺序：



即沿项目路径逐级向上查找node\_modules，及用户node\_modules。

（3）标识符可以不带文件扩展名，默认按.js、.node、.json顺序补齐并查找。因此，如果扩展名是.node或.json，最好加上。

（4）如果最好查找得到的是1个目录，则会当做1个包处理。获取目录下的package.json中的main指定的文件，或目录下的index文件。

**3.模块编译**

（1）模块编译时，每个文件模块都是1个对象，定义如下：



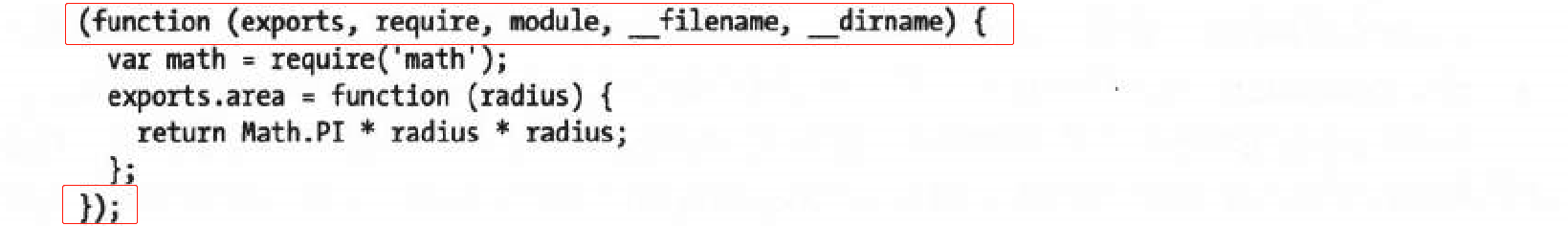
（2）对于.node文件，会当做c/c++编写的扩展文件编译；对于.json文件，会使用fs模块“同步”读取文件后，用JSON.parse解析；对于.js或其他文件，使用fs模块“同步”读取文件后编译执行。编译成功后，会以模块的绝对路径为索引，编译后的结果为值，保存到Module.\_cache对象上。

（3）不同的文件扩展名，会保存到require()的extension的不同属性上。



如json文件，会如下执行

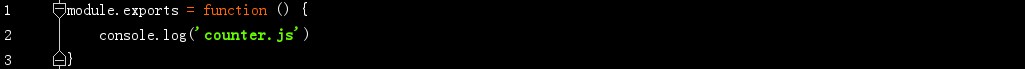
（4）编译过程中，node会对获取的js文件内容进行头尾包装，如下：



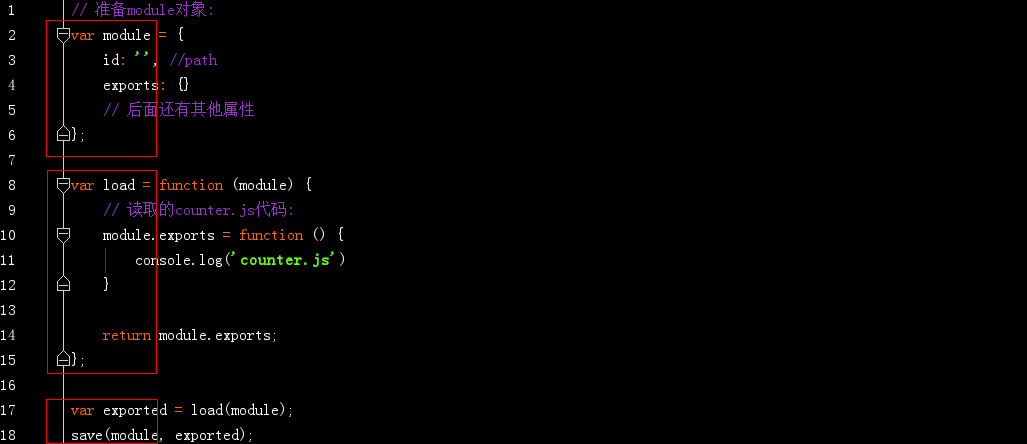
于是，每个模块作用域都进行了隔离。然后调用vm原生模块的runThisContext方法执行，返回1个function对象。最后在执行这个模块函数的时候，传入exports等对象。

（5）总结：首先准备Module对象，注意，每个模块1个Module对象。然后对js文件头尾包装为1个函数，将这个module对象exports、module等，以及其他变量传入这个函数执行，最后将执行结果保存到缓存中。

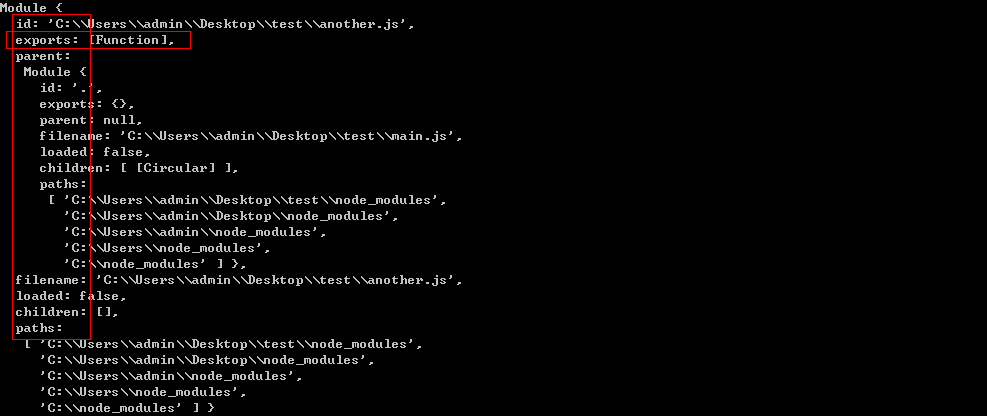
以下面的counter.js模块文件为例。



在main.js文件require这个文件时，首先会如下读取并写入到另外1个文件中并执行（简化）。



写入文件会：（1）准备module对象；（2）用1个函数表达式包裹counter.js并执行；（3）保存至缓存。此时的module（可以在模块内部打印）如下：



结论：1）只有执行require函数时，才会执行上面的操作。2）module看似1个全局变量，但实际只是这个模块的1个变量。不同模块的module不一样。

（6）对于c/c++模块和json，并不需要包装，但仍需要准备Module，在执行后，将结果保存到Module的exports内，并保存到缓存中。

**4.核心模块**

（1）核心模块分为c/c++编写部分和js编写部分，其中，c/c++编写部分在node下的src目录下，js编写部分在node下的lib目录下。在编译c/c++部分之前，会先将js部分编译为c/c++形式。

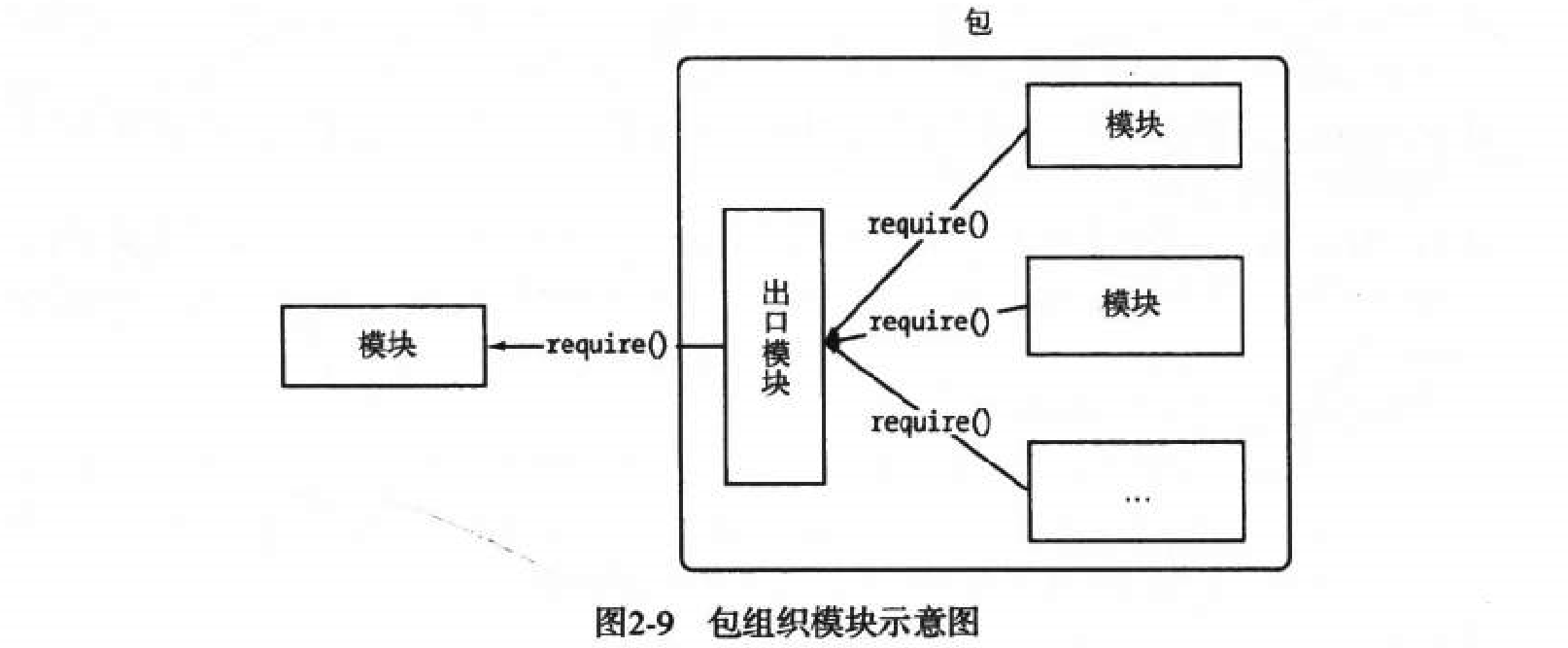
（2）c/c++主内完成核心代码，js主外实现封装以提高编程速度。

（3）更底层的是node的内建模块，由纯c编写，供核心模块调用（使用process.bing()方法），但尽量不要直接调用。

* + 1. 包管理

**1.概要**

（1）包组织示意如下：



（2）符合CommonJS规范的包应该是如下这种结构：1）package.json文件应该存在于包顶级目录下。2）二进制文件应该包含在bin目录下。3）JavaScript代码应该包含在lib目录下。4）文档应该在doc目录下。5）单元测试应该在test目录下。

（3）package.json中比较重要的字段：scripts：脚本说明对象，用于被包管理器安装、编译、测试、卸载等；dependencies：依赖项列表；respositories：托管源码的位置列表等。

* + - 1. npm

**1.模块管理**



更新自己。

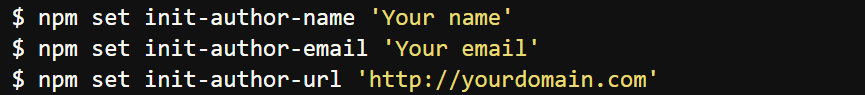




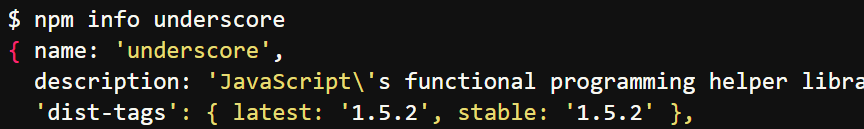
初始化package.json，跳过提问阶段-y：



设置环境变量：



查看模块的具体信息：



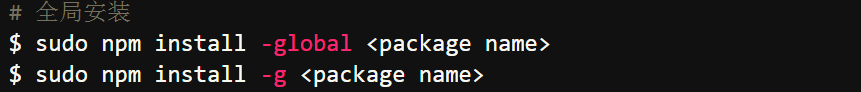
搜索npm仓库：



以树型结构列出当前项目安装的所有模块，以及它们依赖的模块：

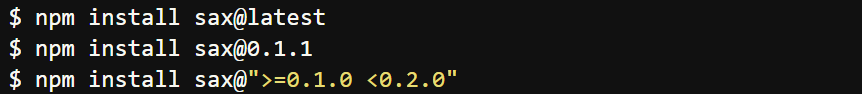








安装指定版本

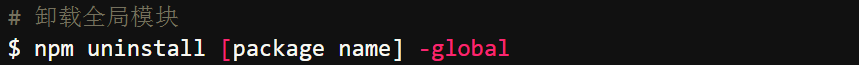




npminstall默认会安装dependencies字段和devDependencies字段中的所有模块，如果使用--production参数，可以只安装dependencies字段的模块。









**2.执行脚本**

package.json文件有一个scripts字段，可以用于指定脚本命令，供npm直接调用。







如下，执行npmrunlint，就会执行jshint\*\*.js。



**3.淘宝镜像**

（1）临时使用



（2）持久使用



* + - 1. yarn















* 1. 异步IO

**1.阻塞式和非阻塞式**

（1）所谓IO主要指与libuv支持的系统磁盘和网络的交互。

（2）阻塞是指Node.js进程执行js代码必须等到非js操作完成。

（3）事件循环无法在发生阻塞操作时继续运行js。执行CPU密集型操作，而不是等待非js操作（如IO），而产生的性能较差的js通常不称为阻塞。使用libuv的Node.js标准库中的同步方法是最常用的阻塞操作。

（4）Node.js标准库中的所有IO方法都提供阻塞和非阻塞版本。

**2.事件循环（区别于主线程）**

（1）当node进程启动后，node便会创建1个类似while(true)的循环，检查是否有事件待处理，如果有，就取出对应的事件回调函数执行。每执行1次循环，称为1次tick。



（2）事件循环是1个典型的生产者/观察者模式。异步IO等是事件的生产者，为node提供不同类型的事件。根据事件分类，每类事件都有自己的观察者。事件传递到观察者哪里，而事件循环则从观察者哪里取出事件并处理。就如客户向服务员下单，厨房询问服务员有没有接下来要做的菜。

（3）每次事件循环都包含了6个阶段：1）timers阶段：检查有无过期的timer（setTimeout、setInterval），有则把它的回调压入timer的任务队列中等待执行；2）I/O callbacks阶段：执行一些系统调用错误，比如网络通信的错误回调；3）idle、prepare阶段：仅node内部使用；4）poll阶段：执行I/O事件,适当的条件下node将阻塞在这里；4）check阶段：执行setImmediate()的回调；5）close callbacks阶段：执行socket的close事件回调（统一关闭）。

通常主要涉及timers、poll、check这3个阶段。

timers阶段虽然是事件循环的第一个阶段，但里面其回调却是在poll阶段执行。

poll阶段主要有2个功能：处理poll队列的事件，以及执行已超时的timer回调函数。如果poll队列，事件循环将同步执行poll队列的事件，直到poll队列执行完（理论上，如果执行中有新的IO操作，IO操作结束后又会附在后面，但是，这里的操作是同步操作，而IO操作不会那么快，所以还是会等到下一次tick执行）或达到系统上限。之后检查是否预设setImmediate，有则进入check阶段，否则一直停留在本阶段。

如果进入check阶段执行完后，就会进入下一阶段，然后进入下次循环。如果停留在poll阶段，则会检查timer队列是否为空，不为空则进入下一次tick。

**3.请求对象**

（1）对于异步IO，在执行过程中，会生成1个中间产物即请求对象。事件回调函数将绑定到这个请求对象上（作为属性）。待将来IO操作执行完毕，也会将执行结果保存到这个请求对象上，并放入IO观察者队列（poll队列）中，等待事件循环处理。

（2）事件循环处理IO事件，就是将请求对象的回调函数和执行结果执行实现。

**4.执行顺序**

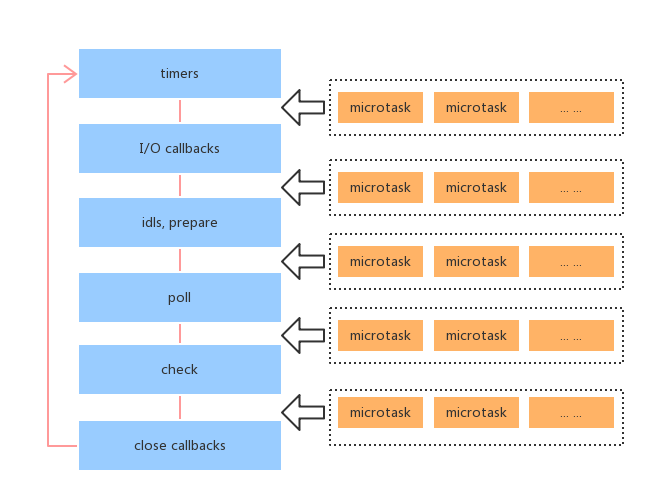
（1）node环境下，执行顺序为：



原因如下：

1）process.nextTick为idle阶段的观察者，timers阶段此时只是压入timer队列，IO队列因为IO操作还没有那么快，因此先执行process.nextTick。

2）任务队列分为两类：microtasks和macrotasks。macrotasks就是上面的setTimeout、setInterval、setImmediate、I/O、UI渲染等；microtasks是promise、process.nextTick、Object.observe等。在Node.js中，microtask会在事件循环的各个阶段之间执行，即任何阶段执行完毕，就会去执行microtask队列的任务。



因此，在idle阶段执行完后，会立即执行microtask的promise。

3）还有1个注意事项，process.nextTick的回调函数保存在1个数组中，而setImmediate保存在1个链表中。执行时，process.nextTick的回调会一次性执行完，而setImmediate每次事项循环只会取出其中1项执行。

* 1. 异步编程
     1. 概要

**1.函数式编程**

（1）函数作为一等公民，可以当做变量、参数、返回值等。

（2）高阶函数是指将函数作为参数或返回值的函数。



如数组sort方法，通过接受不同的回调函数，实现不同的排序。



（3）偏函数是指固定函数的1个参数（类似）。

**2.异步编程注意事项**

（1）异常处理。如下，常规try-catch无法捕获异常。



node约定，将异步异常作为回调函数的第一个参数返回。如果为空值，则表明没有异常。



因此，在编写异步方法时，应：1）必须执行调用者传入的回调；2）正确传回异常供调用者判断。



（2）回调函数嵌套过深。

* + 1. 异步编程解决方案

**1.事件发布/订阅模式**

（1）使用node提供的events模块。该模块比前端DOM事件简单，因为不存在冒泡、默认事件等。模块提供了addListener/on、removeListener、once、removeAllListener、emit等接口。

（2）如果给1个事件绑定了超过10个侦听器，则会得到1个警告，以防止内存泄漏。

（3）对于执行过程中的错误，如果对error绑定了侦听，则交给侦听处理，否则作为异常抛出。

**2.Promise/Defferred模式**

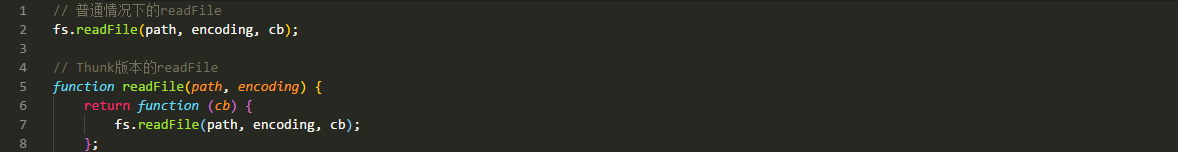
Promise/Defferred模式包括两部分：Promise和Defferred。

**3.generator**

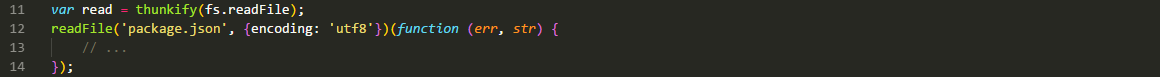
（1）协程：如当协程a执行过程中暂停执行，将执行权交给协程b，b执行完后再将执行权交回a，a恢复执行。

（2）generator函数是协程在ES6的实现。当遇到yield时就会暂停执行。但存在的问题是：两个yield之间有何关联？何时执行下一个next（）？

（3）js中的Thunk函数，即具备以下两个条件的函数：1）有且只有一个参数是callback的函数；2）callback的第一个参数是error。任何函数，只要参数有回调函数，就能写成Thunk函数的形式。

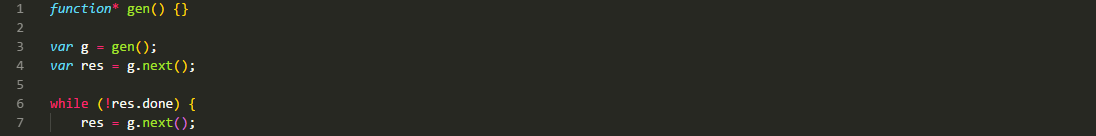


手动改写太麻烦，可以使用thunkify自动转换。

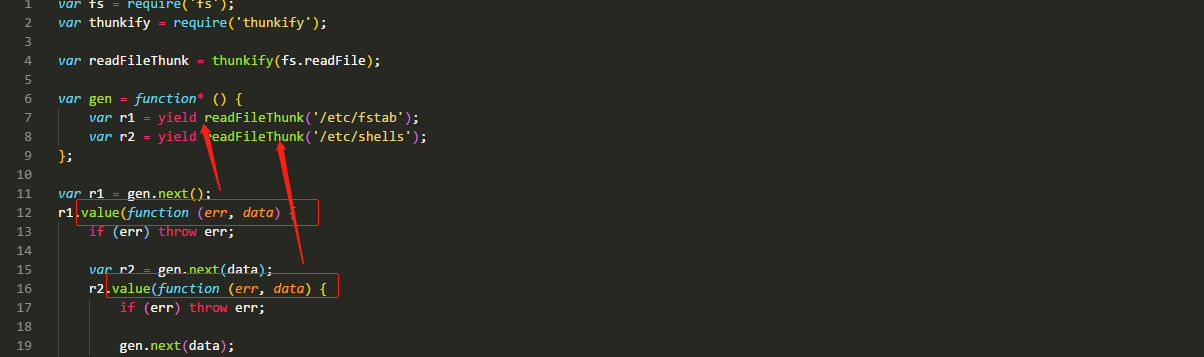


（4）Thunk函数可用于generator函数的自动流程管理，前提是yield命令后面的必须是Thunk函数。

下面的方式虽然实现了generator函数的自动执行，但不适合前一步执行完，才能执行后一步的异步操作。



如下，yield命令用于将程序的执行权移出Generator函数，那么就需要一种方法，将执行权再交还给Generator函数。使用Thunk函数可以在其回调函数里，将执行权交还给Generator函数。



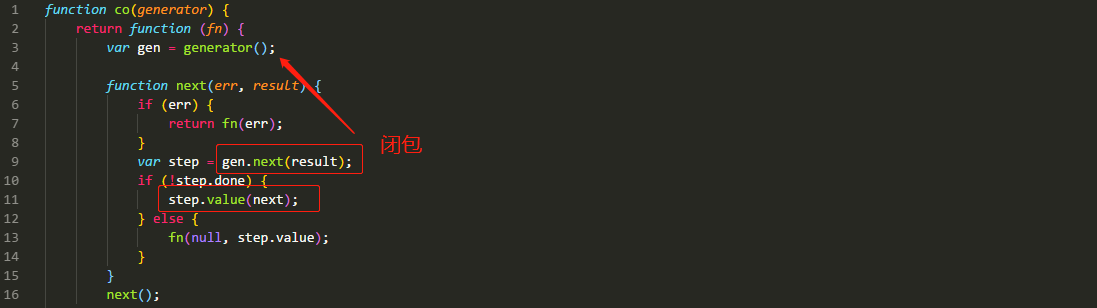
（5）上面还是需要手动逐个调用next方法。可以使用递归方式自动调用。



（6）可以使用co模块自动执行Generator函数。前提是将异步操作包装成Thunk函数或promise对象或promise对象数组。



co的精简版（只支持thunk函数）如下：



**4.async函数**

* 1. 进程管理

**1.process对象**

（1）process对象就是处理与进程相关信息的全局对象，不需要require引用，且是EventEmitter的实例。



（2）可以通过process.argv获取命令行指令参数；可以绑定进程事件：



**2.web服务器架构变迁历史**

（1）同步：每次只服务一个请求，当前请求处理完毕后处理下一个请求。

（2）复制进程：如apache的进程模式，每个请求起1个进程进行服务。因启动进程代价较高，因此会预启动（prefork）一定数量进程备用，以及复用不用的进程。

（3）多线程：使用线程服务请求。

（4）事件驱动：node和Nginx均采用这种方式，但没有充分利用cpu的多线程。

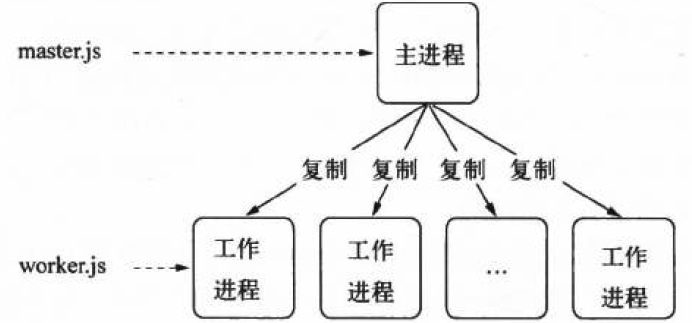
**3.多进程架构**

（1）为处理node单进程单线程对多核使用不足的问题，node提供child\_process模块，可用于启用多个进程，每个进程使用1个cpu。

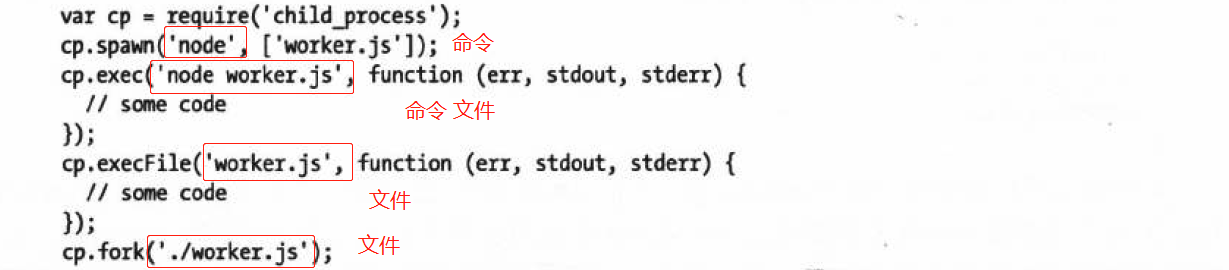
如，主进程程序为master.js，代码如下：



这就是典型的Master-Worker模式（主从模式）。



（2）child\_process模块提供spawn、exec、execFile、fork四种方法创建子进程，均返回子进程对象。它们的区别如下：



在V8引擎下，使用fork复制1个独立的进程，需要30ms启动和10MB内存。

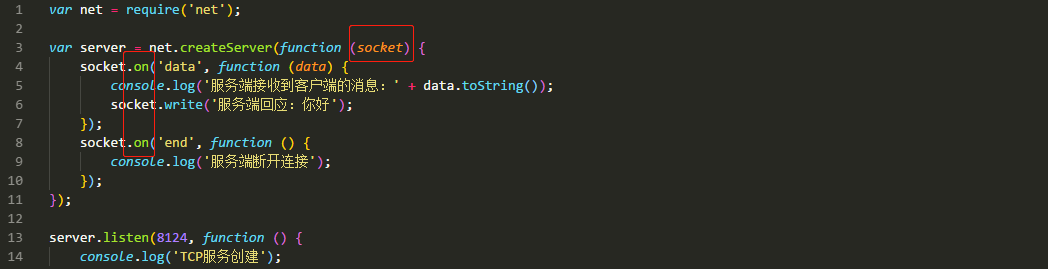
（3）如同浏览器环境使用onmessage和postMessage进行通信，node环境下使用on和send实现进程间通信。

* 1. 网络编程

**1.概要**

（1）node提供net、dgram、http、https四个模块，分别用于处理udp、tcp、http、https服务。

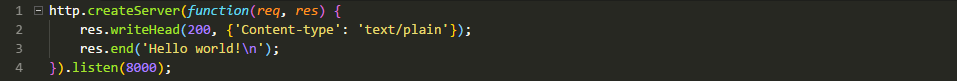
（2）如下，创建tcp服务：



net.createServer是EventEmitter的实例，可以监听data、end、error等事件。

（3）创建udp服务和tcp服务类似。

（4）创建http服务端：



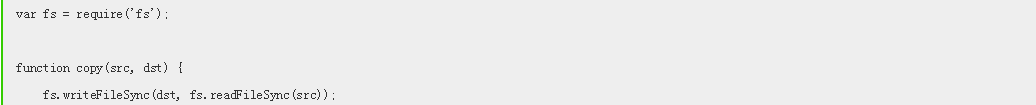
（5）http模块提供http.request接口，可用于创建http客户端。

* 1. 文件操作

**1.文件复制**

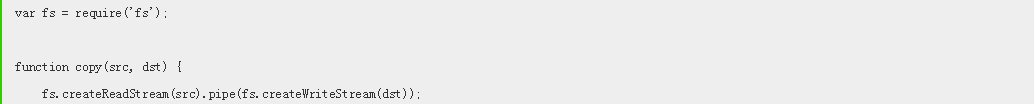
（1）小文件拷贝

可以使用fs.readFileSync从源路径读取文件内容，并使用fs.writeFileSync将文件内容写入目标路径。



（2）大文件拷贝

一次性把所有文件内容都读取到内存中后再一次性写入磁盘的方式不适合拷贝大文件，内存会爆仓。可以使用fs.createReadStream创建了一个源文件的只读数据流，并使用fs.createWriteStream创建了一个目标文件的只写数据流，并且用pipe方法把两个数据流连接了起来。



**2.API**

（1）Stream

（2）FileSystem

（3）Path

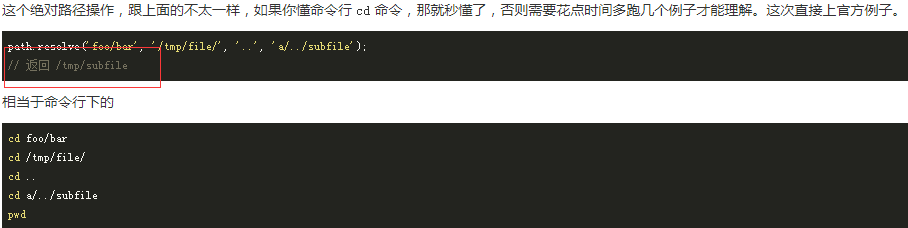


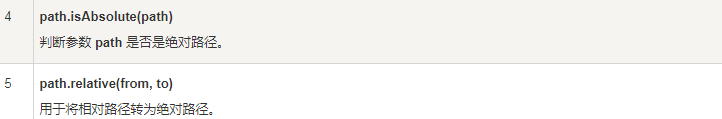










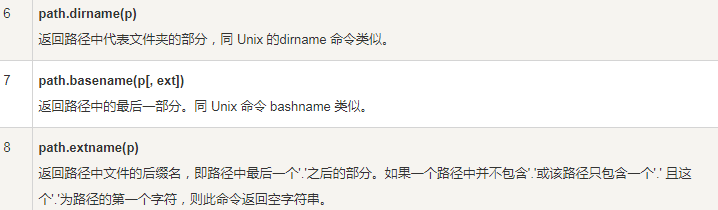






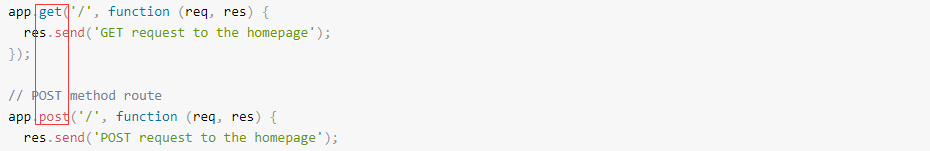






1. express
   1. 路由

**1.路由方法**



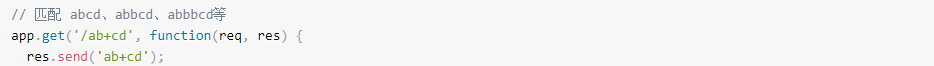
app.all()没有任何HTTP方法与其对应，它的作用是对于一个路径上的所有请求加载中间件。



**2.路由路径**

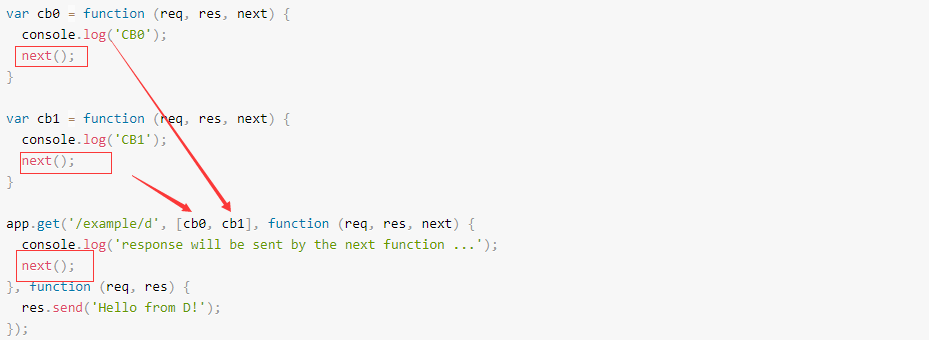
（1）查询字符串不是路由路径的一部分。

（2）可以是字符串、字符串模式或者正则表达式。



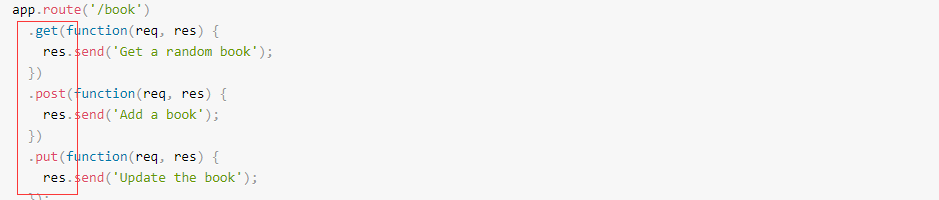
**3.路由句柄**

行为类似中间件，可以定义多个路由句柄，可以是数组，但前面的路由句柄中需要调用next方法，否则无法传到下一个路由句柄。



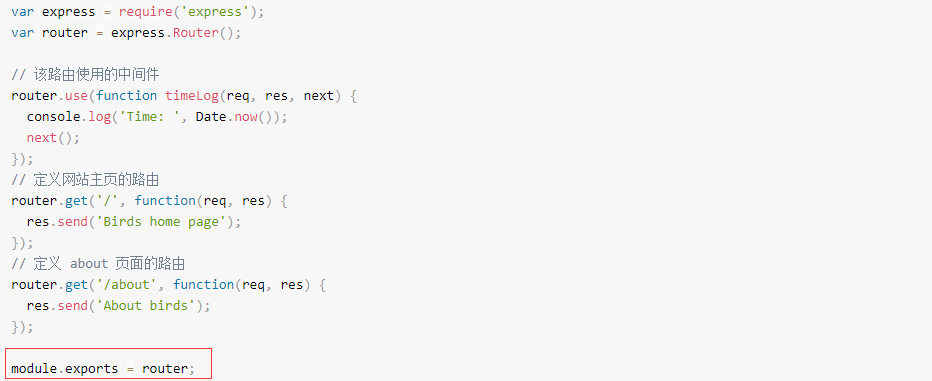
**4.app.route**

创建路由路径的链式路由句柄，可用于创建模块化路由，减少代码冗余和拼写错误。



**5.express.Router**

Router实例是一个完整的中间件和路由系统，也可用于创建模块化、可挂载的路由句柄。





**6.响应方法**

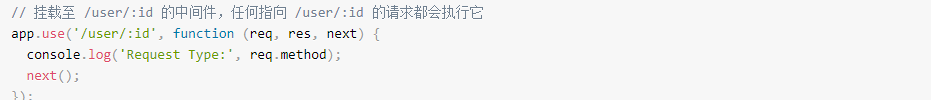
响应对象（res）的方法向客户端返回响应，终结请求。如果在路由句柄中一个也不调用，来自客户端的请求会一直挂起。



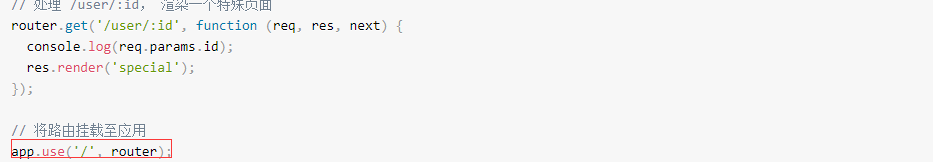
* 1. 中间件

**1.应用级中间件**

使用app.use()和app.METHOD()，其中，METHOD是需要处理的HTTP请求的方法，例如GET,PUT,POST等。

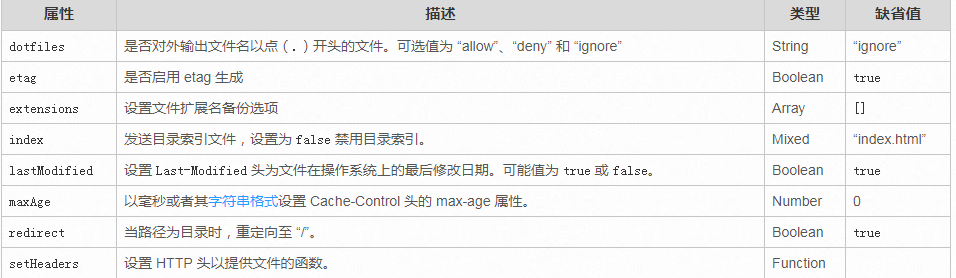


**2.路由级中间件**



**3.内置中间件**

唯一内置的中间件express.static(root,[options])。负责在Express应用中提托管静态资源。参数root指提供静态资源的根目录。options可以为：





staticPath是1个目录，express.static指定静态资源跟目录。上面表示，如获取static目录下的静态文件，如http://localhost:3000/static/images/kitten.jpg，将从express.static('./public')中获取。

**4.第三方中间件**



* 1. 模板引擎

（1）在应用中如下设置模板文件。



（2）安装相应的模板引擎。



（3）返回模板文件。



* 1. 集成数据库

（1）安装



（2）配置



1. 二进制处理

Buffer（缓冲区）是系统两端处理速度平衡（从长时间尺度上看）时使用的。它的引入是为了减小短期内突发I/O的影响，起到流量整形的作用。比如生产者——消费者问题，他们产生和消耗资源的速度大体接近，加一个buffer可以抵消掉资源刚产生/消耗时的突然变化。

Cache（缓存）则是系统两端处理速度不匹配时的一种折衷策略。因为CPU和memory之间的速度差异越来越大，所以人们充分利用数据的局部性（locality）特征，通过使用存储系统分级（memoryhierarchy）的策略来减小这种差异带来的影响。

arraybuffer就是一片内存空间，不能直接引用里面的数据，可以通过typedarray类型引用，用户只能通过typedarray使用这片内存，不能直接通过arraybuffer使用这片内存。