Node.js 全栈基础

1. Node.js 光速入门

1.1 Node.js 概述

1. Node.js 是什么

   Node.js 不是一门编程语言，它是一个执行 JavaScript 代码的工具。工具是指可以安装在计算机操作系统之上的软件。

2. 为什么浏览器和 Node.js 都可以运行 JavaScript

   因为浏览器和 Node.js 都内置了 JavaScript V8 Engine。

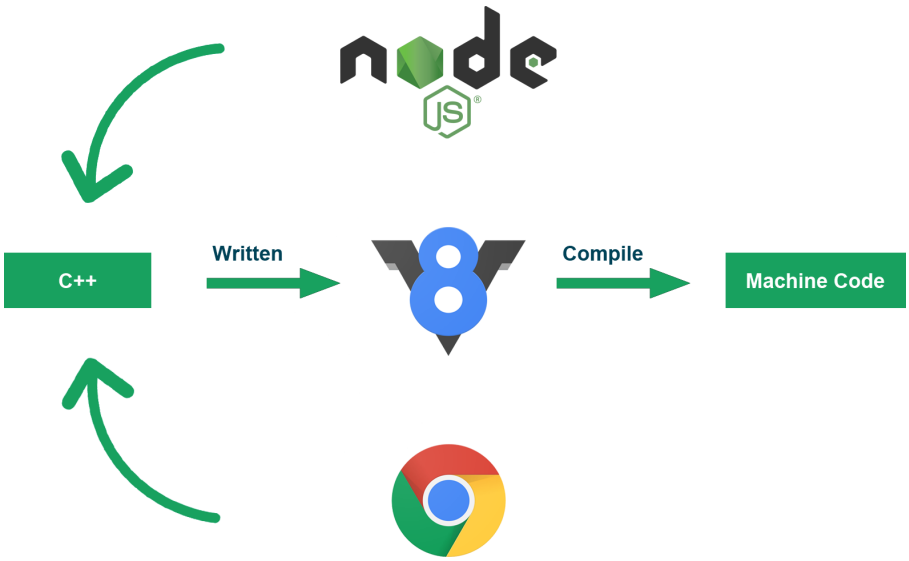
   它可以将 JavaScript 代码编译为计算机能够识别的机器码。

3. 浏览器中运行的 JavaScript 和 Node.js 中运行的 JavaScript 有区别吗

   在内置了 JavaScript V8 Engine 以后实际上只能执行 ECMAScript，就是语言中的语法部分。

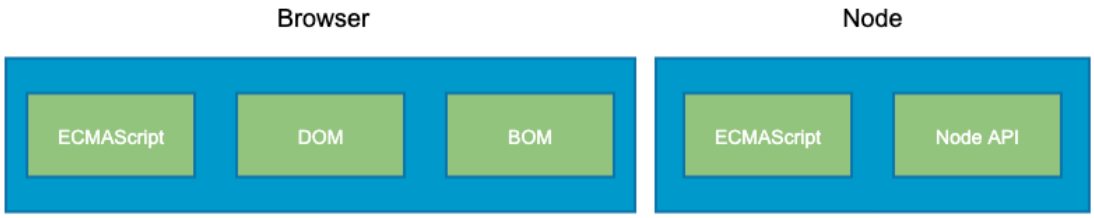
   浏览器为了能够让 JavaScript 操作浏览器窗口以及 HTML 文档，所以在 JavaScript V8 Engine 中添加了控制它们的 API, 就是 DOM 和 BOM. 所以 JavaScript 在浏览器中运行时是可以控制浏览器窗口对象和DOM文档对象的。

   和浏览器不同，在 Node.js 中是没有 DOM 和 DOM 的，所以在 Node.js 中不能执行和它们相关的代码，比如 `window.alert()` 或者 `document.getElementById()`. DOM 和 DOM 是浏览器环境中特有的。在 Node.js 中，作者向其中添加了很多系统级别的 API，比如对操作系统中的文件和文件夹进行操作。获取操作系统信息，比如系统内存总量是多少，系统临时目录在哪，对系统的进程进行操作等等。



   JavaScript 运行在浏览器中控制的是浏览器窗口和 DOM 文档。

   JavaScript 运行在 Node.js 中控制的操作系统级别的内容。



4. 为什么浏览器中的 JavaScript 不能控制系统级别的 API？

   浏览器是运行在用户的操作系统中的，如果能控控制系统级别的 API 就会存在安全问题。

   Node.js 是运行在远程的服务器中的，访问的是服务器系统 API，不存在这方面的安全问题。

5. Node.js 能够做什么

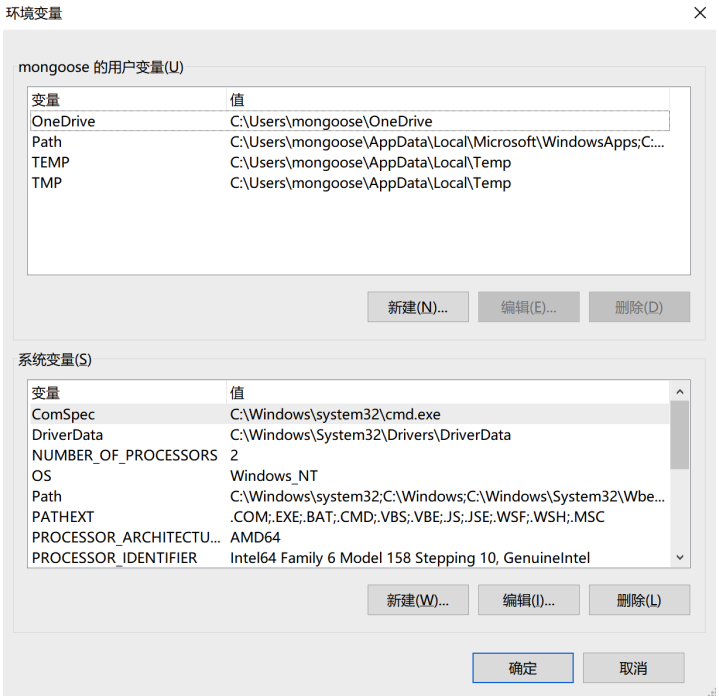
   我们通常使用它来构建服务器端应用和创建前端工程化工具。

   JavaScript 运行在浏览器中我们就叫它客户端 JavaScript。

   JavaScript 运行在 Node.js 中我们就叫它服务器端 JavaScript。

1.2 系统环境变量

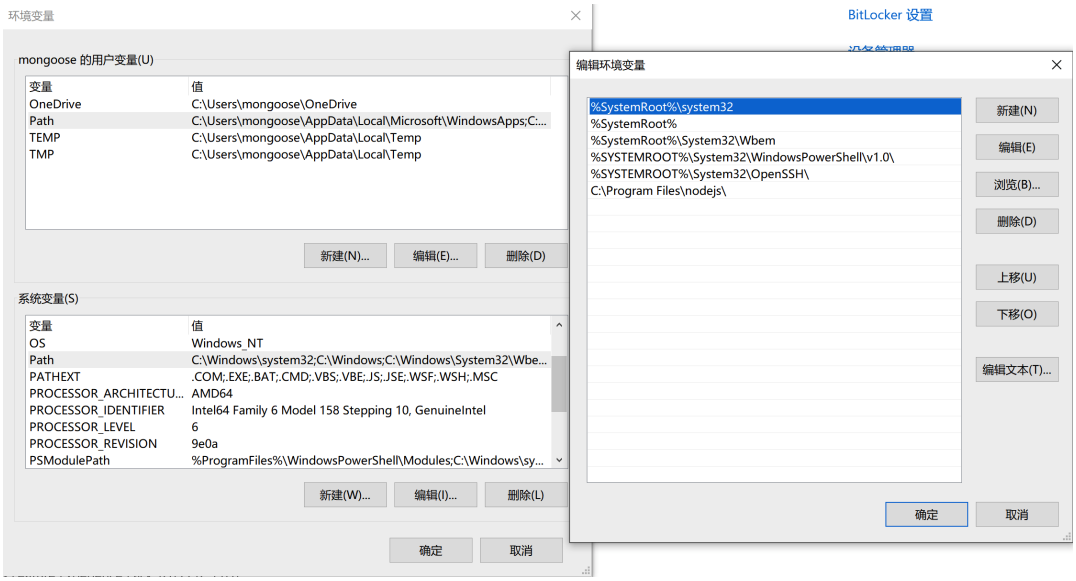
系统环境变量是指在操作系统级别上定义的变量，变量中存储了程序运行时所需要的参数。



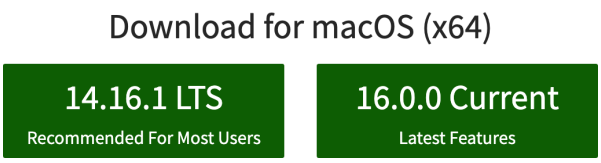
比如在使用 webpack 构建前端应用时就使用到了系统环境变量，因为 webpack 需要根据系统环境变量判断当前为开发环境还是生产环境，根据环境决定如何构建应用。

在开发环境的操作系统中定义 NODE_ENV 变量，值为 development，在生产环境的操作系统中定义 NODE_ENV 变量，值为 production。webpack 在运行时通过 process.env.NODE_ENV 获取变量的值，从而得出当前代码的运行环境是什么。

环境变量 PATH：系统环境变量 PATH 中存储的都是应用程序路径。当要求系统运行某一个应用程序又没有告诉它程序的完整路径时，此时操作系统会先在当前文件夹中查找应用程序，如果查找不到就会去系统环境变量 PATH 中指定的路径中查找。

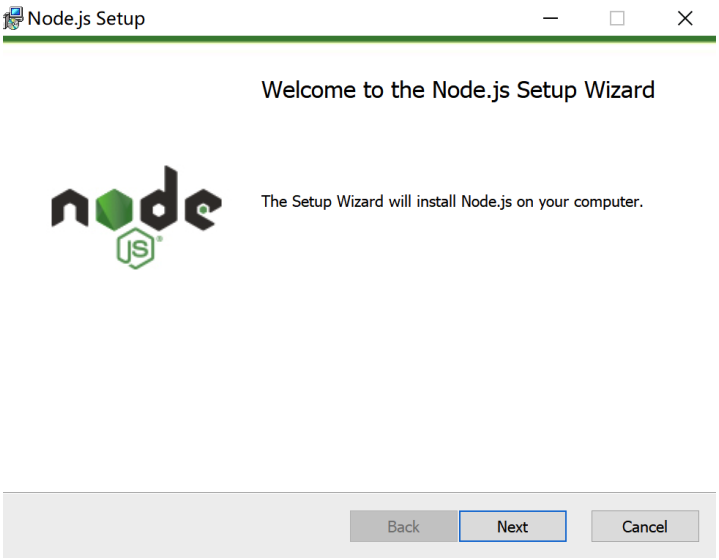


1.3 安装 Node.js

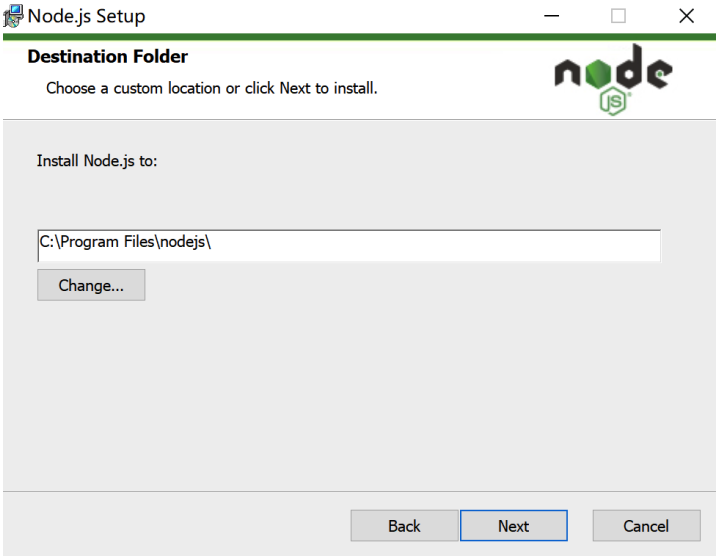


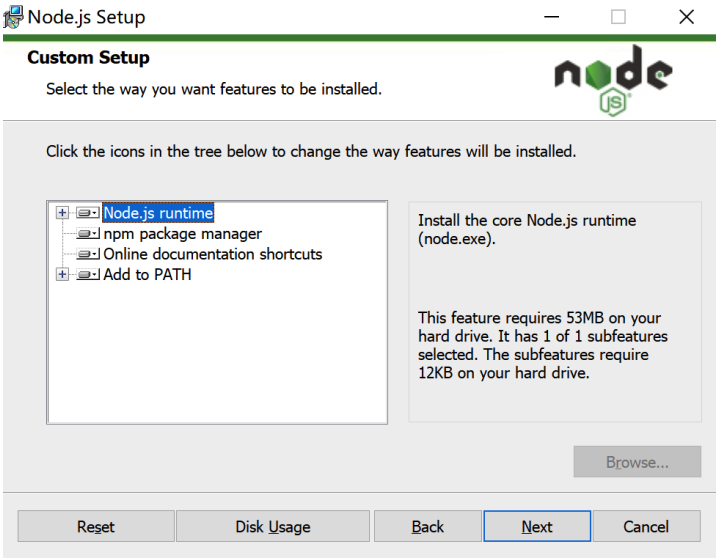
LTS：长期支持版 (稳定版) 可以运行在生产环境中。

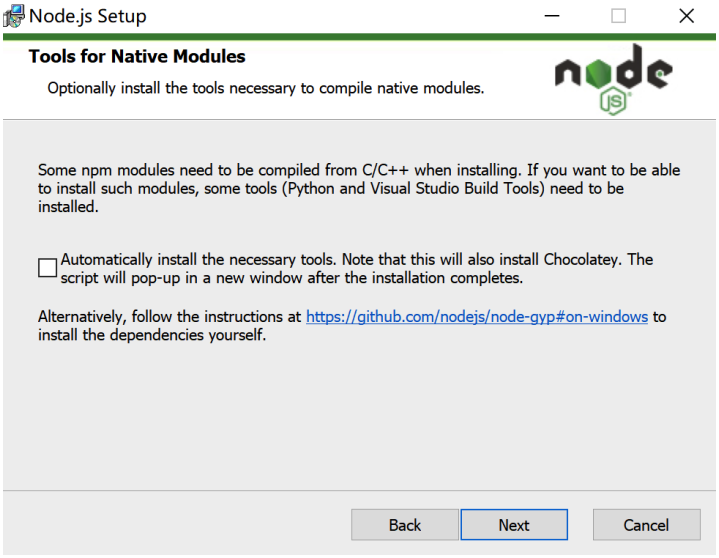
Current：最新版 (预览版) 不建议运行在生产环境中，因为可能有 BUG。

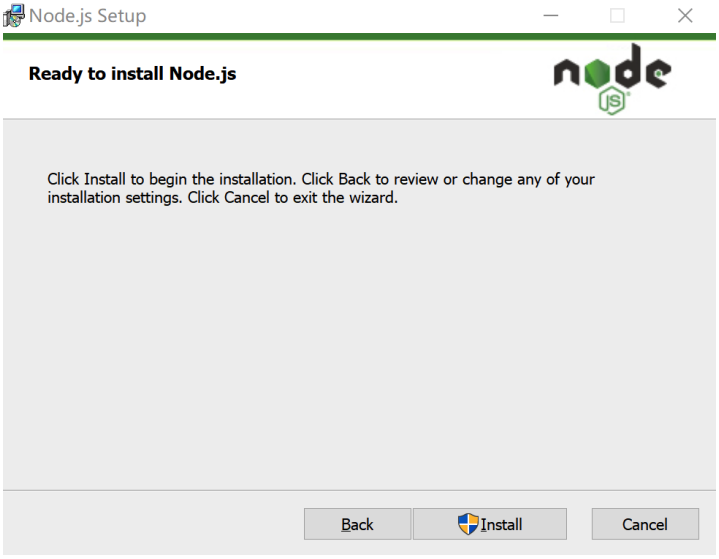


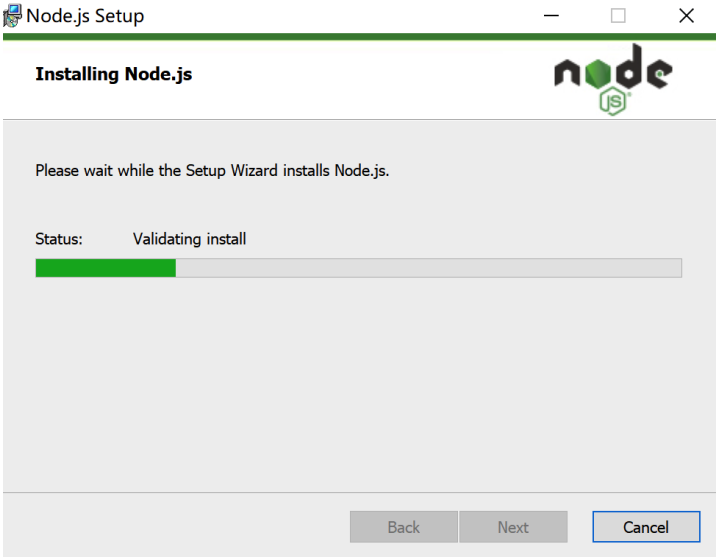

Node.js Setup
End-User License Agreement
Please read the following license agreement carefully
Node.js is licensed for use as follows:
Copyright Node.js contributors. All rights reserved.
Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so,
I accept the terms in the License Agreement
Print
Back
Next
Cancel

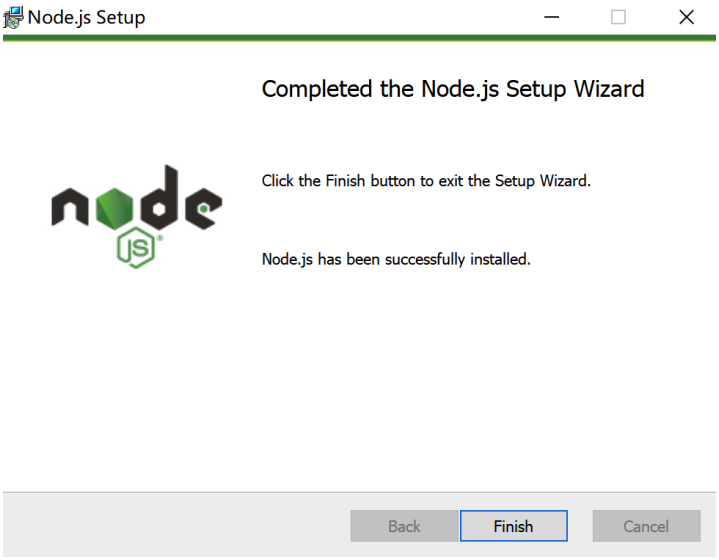

Node.js Setup
Destination Folder
Choose a custom location or click Next to install.
Install Node.js to:
C:\Program Files\nodejs\
Change...
Back
Next
Cancel


Node.js Setup
Custom Setup
Select the way you want features to be installed.
Click the icons in the tree below to change the way features will be installed.
Node.js runtime
npm package manager
Online documentation shortcuts
Add to PATH
Install the core Node.js runtime (node.exe).
This feature requires 53MB on your hard drive. It has 1 of 1 subfeatures selected. The subfeatures require 12KB on your hard drive.
Browse...
Reset
Disk Usage
Back
Next
Cancel


Node.js Setup
Tools for Native Modules
Optionally install the tools necessary to compile native modules.
Some npm modules need to be compiled from C/C++ when installing. If you want to be able to install such modules, some tools (Python and Visual Studio Build Tools) need to be installed.
Automatically install the necessary tools. Note that this will also install Chocolatey. The script will pop-up in a new window after the installation completes.
Alternatively, follow the instructions at https://github.com/nodejs/node-gyp#on-windows to install the dependencies yourself.
Back
Next
Cancel


Node.js Setup
Ready to install Node.js
Click Install to begin the installation. Click Back to review or change any of your installation settings. Click Cancel to exit the wizard.
Back
Install
Cancel


Node.js Setup
Installing Node.js
Please wait while the Setup Wizard installs Node.js.
Status:
Validating install
Back
Next
Cancel



查看 Node 版本：`node -v`

查看 Npm 版本：`npm -v`

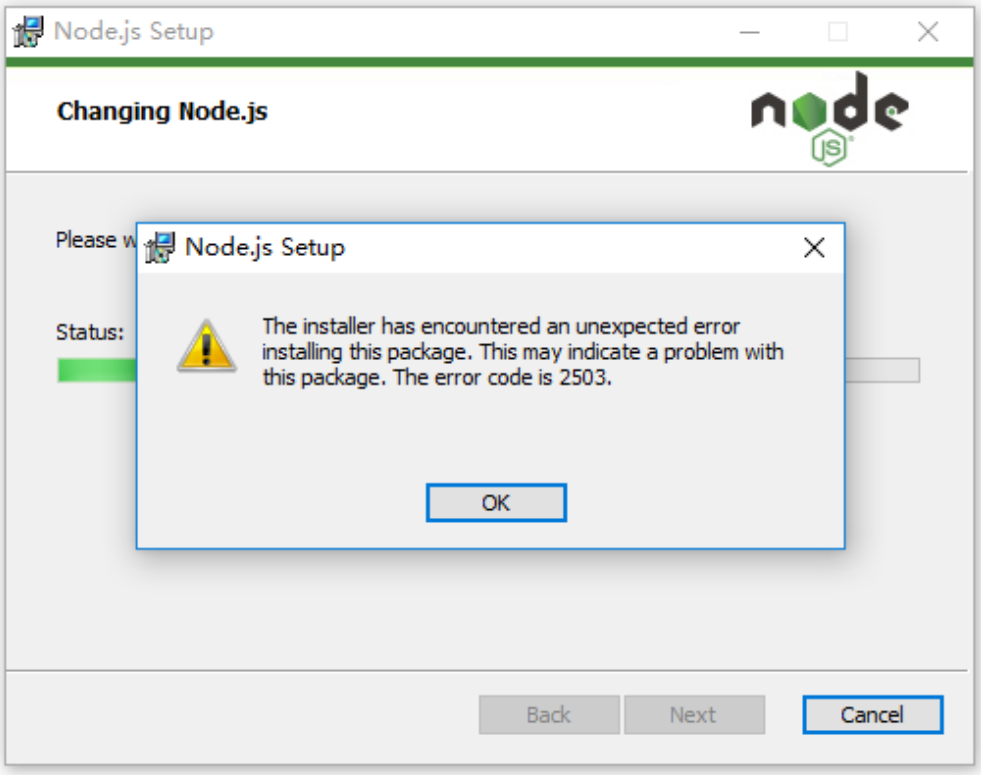
1.4 解决安装异常

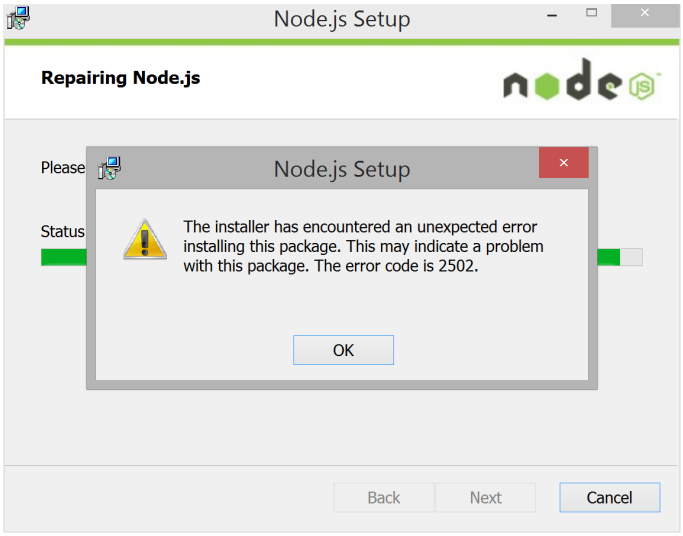
1. 解决在运行 node 命令时提示 "不是内部或外部命令, 也不是可运行的程序或批处理文件"。

```
C:\Program Files\cmder_mini
λ node -v
'node' 不是内部或外部命令，也不是可运行的程序
或批处理文件。
```

   将 Node 应用程序目录添加到系统环境变量中, 然后重新启动命令行工具再次执行 node 命令.

2. 解决在安装 Node 的过程中出现代码为 2502 和 2503 的错误。





1. 通过管理员权限打开命令行工具
2. 切换到 node 安装包所在的目录
3. 通过 `msiexec /package node-v10.15.0-x64.msi` 运行 Node 应用程序安装包

1.5 Node.js 初体验

```
function sayHello (name) {
  console.log('Hello' + name)
}
sayHello('Node')
```

在命令行工具中通过 `node JavaScript 文件` 的方式执行代码。

1.6 全局对象

```
console.log(window) // window is not defined
```

在 Node.js 环境中是没有 window 的，所以 window 对象自然是未定义的。

在 Node.js 环境中全局对象为 global，在 global 对象中会存在一些和 window 对象中名字相同且作用相同的方。

```
global.console.log
global.setInterval
global.clearInterval
global.setTimeout
global.clearTimeout
global.setImmediate
```

在 Node.js 环境中声明的变量不会被添加到全局对象中，变量声明后只能在当前文件中使用。

```
var message = "hello"
console.log(global.message) // undefined
```

2. 模块系统

2.1 模块概述

在 Node.js 环境中，默认就支持模块系统，该模块系统遵循 CommonJS 规范。

一个 JavaScript 文件就是一个模块，在模块文件中定义的变量和函数默认只能在模块文件内部使用，如果需要在其他文件中使用，必须显式声明将其进行导出。

2.2 模块成员导出

在每一个模块文件中，都会存在一个 module 对象，即模块对象。在模块对象中保存了和当前模块相关信息。

在模块对象中有一个属性 exports，它的值是一个对象，模块内部需要被导出的成员都应该存储在到这个对象中。

```
Module {
  exports: {}
}
```

```
// logger.js
const url = "http://mylogger.io/log";

function log (message) {
  console.log(message)
}
module.exports.endPoint = url
module.exports.log = log
```

2.3 模块成员导入

在其他文件中通过 require 方法引入模块，require 方法的返回值就是对应模块的 module.exports 对象。

在导入模块时，模块文件后缀 .js 可以省略，文件路径不可省略。

require 方法属于同步导入模块，模块导入后可以立即使用。

```
// app.js
const logger = require("./logger")
console.log(logger) // { endPoint: 'http://mylogger.io/log', log: [Function:
log] }
console.log(logger.endPoint) // http://mylogger.io/log
logger.log('Hello Module')   // Hello Node
```

通过 require 方法引入模块时会执行该模块中的代码。

```
// logger.js
console.log("running...")

// app.js
require("./logger") // running...
```

在导入其他模块时，建议使用 const 关键字声明常量，防止模块被重置。

```
var logger = require("./logger")
logger = 1;
logger.log("Hello") // logger.log is not a function

const logger = require("./logger")
logger = 1;  // Assignment to constant variable.
logger.log("Hello")
```

有时在一个模块中只会导出一个成员，为方便其他模块使用，可以采用以下导入方式。

```
// logger.js
module.exports = function (message) {
  console.log(message)
}

// app.js
const logger = require("./logger")
logger("Hello")
```

2.4 Module Wrapper Function

Node.js 是如何实现模块的，为什么在模块文件内部定义的变量在模块文件外部访问不到?

每一个模块文件中都会有 module 对象和 require 方法，它们是从哪来的?

在模块文件执行之前，模块文件中的代码会被包裹在模块包装函数当中，这样每个模块文件中的代码就都拥有了自己的作用域，所以在模块外部就不能访问模块内部的成员了。


```
(function(exports, require, module, __filename, __dirname) {
  // entire module code lives here
});
```

从这个模块包装函数中可以看到，module 和 require 实际上模块内部成员, 不是全局对象 global 下面的属性。

__filename：当前模块文件名称。

__dirname：当前文件所在路径。

exports：引用地址指向了 module.exports 对象，可以理解为是 module.exports 对象的简写形式。

```
exports.endPoint = url;
exports.log = log
```

在导入模块时最终导入的是 module.exports 对象，所以在使用 exports 对象添加导出成员时不能修改引用地址。

```
exports = log //这是错误的写法.
```

2.5 Node.js 内置模块

在 Node.js 安装完成后，会内置一些非常有用的模块。

Path：模块内提供了一些和路径操作相关的方法。

File system：文件操作系统，提供了和操作文件相关的方法。

在引入内置模块时, 使用的是模块的名字，前面不需要加任何路径。

2.5.1 Path 模块

```
const path = require("path")
console.log(path.parse(__filename))

{
  root: '/',
  dir: '/Users/administrators/Desktop/node_test',
  base: 'app.js',
  ext: '.js',
  name: 'app'
}
```

2.5.2 File system 模块

```
const fs = require("fs")

const files = fs.readdirSync("./")
console.log(files)  [ 'app.js', 'logger.js' ]

fs.readdir("./", function (error, files) {
  console.log(error) // null | Error {}
  console.log(files) // [ 'app.js', 'logger.js' ] | undefined
})
```

3.NPM

3.1 Node.js 软件包

每一个基于 Node.js 平台开发的应用程序都是 Node.js 软件包。

所有 Node.js 软件包都被托管在 www.npmjs.com 中。

3.2 什么是 NPM

Node Package Manager，Node.js 环境中的软件包管理器。随 Node.js 一起被安装。

它可以将 Node 软件包添加到我们的应用程序中并对其进行管理，比如下载，删除，更新，查看版本等等。

它没有用户界面，需要在命令行工具中通过命令的方式使用，对应的命令就是 npm。

NPM 和 Node 是两个独立的应用程序，只是被捆绑安装了，可以通过版本号证明。

3.3 package.json

Node.js 规定在每一个软件包中都必须包含一个叫做 package.json 的文件。

它是应用程序的描述文件，包含和应用程序相关的信息，比如应用名称，应用版本，应用作者等等。

通过 package.json 文件可以方便管理应用和发布应用。

创建 package.json 文件: `npm init`

快速创建 package.json 文件: `npm init --yes`

```
{
  "name": "project-name",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

3.4 下载 Node.js 软件包

在应用程序的根目录执行命令：`npm install <pkg>` 或者 `npm i <pkg>`

`npm install lodash`

软件包下载完成后会发生三件事：

1. 软件包会被存储在 node_modules 文件夹中，如果在应用中不存在此文件夹，npm 会自动创建。
2. 软件包会被记录在 `package.json` 文件中. 包含软件包的名字以及版本号。
3. npm 会在应用中创建 `package-lock.json` 文件, 用于记录软件包及软件包的依赖包的下载地址及版本。

3.5 使用 Node.js 软件包

在引入第三方软件包时，在 require 方法中不需要加入路径信息，只需要使用软件包的名字即可，require 方法会自动去 node_modules 文件夹中进行查找。

```
const _ = require("lodash")

const array = ["a", "b", "c", "d"]
// chunk 对数组中的元素进行分组
// 参数一表示要进行操作的数组
// 参数二表示每一组中包含的元素个数
console.log(_.chunk(array, 2)) // [ [ 'a', 'b' ], [ 'c', 'd' ] ]
```

3.6 软件包依赖问题说明

1. 比如在我的应用中要依赖 mongoose 软件包，于是我下载了它，但是在 node_modules 文件夹中除了包含 mongoose 以外还多出了很多其他软件包，为什么会多出这么多软件包呢?

   实际上它们又是 mongoose 依赖的软件包。

2. 为什么 mongoose 依赖的软件包不放在 mongoose 文件夹中呢?

   在早期的 npm 版本中, 某个软件包依赖的其他软件包都会被放置在该软件包内部的 node_modules 文件夹中，但是这样做存在两个问题，第一个问题是很多软件包都会有相同的依赖，导致开发者在一个项目中会下载很多重复的软件包，比如 A 依赖 X，B 依赖 X，C 依赖 X，在这种情况下 X 就会被重复下载三次。第二个问题是文件夹嵌套层次太深，导致文件夹在 windows 系统中不能被直接删除。比如 A 依赖 B, B 依赖 C, C 依赖 D ... , 就会发生文件夹依次嵌套的情况。

3. 所有的软件包都放置在 node_modules 文件夹中不会导致软件包的版本冲突吗?

   在目前的 npm 版本中，所有的软件包都会被直接放置在应用根目录的 node_modules 文件夹中，这样虽然解决了文件夹嵌套层次过深和重复下载软件包的问题，但如果只这样做肯定会导致软件包版本冲突的问题，如何解决呢?

   比如 A 依赖 X 的 1 版本，B 依赖 X 的 2 版本，如果你先下载的是 A，那么 A 依赖的 X 会被放置在根目录的 node_modules 文件夹中, 当下载 B 时，由于在根目录中已经存在 X 并且版本不一致，那么 B 依赖的 X 就会被放置在 B 软件包中的 node_module 文件夹中，通过此方式解决软件包版本冲突的问题。

4. node_modules 文件夹中的软件包都需要提交到 git 仓库中吗?

   在 node_modules 文件夹中有很多软件包，随着应用程序的增长，软件包也会越来越多，甚至会达到几百兆。

   当我们将应用提交到版本库时，我们不想提交它，因为它们不是我们应用中的源代码，而且由于碎文件比较多，其他人在检出代码时需要等待的时间会很久。当其他人拿到应用程序时没有依赖软件包应用程序是运行不起来的，如何解决呢?

实际上应用程序依赖了哪些软件包在 package.json 文件中都会有记录，其他人可以通过 `npm install` 命令重新下载它们。为了保持下载版本一直，npm 还会根据 package-lock.json 文件中的记录的地址进行下载。

将应用程序提交到版本库之前，将 node_modules 文件夹添加到 .gitignore 文件中。

```
git init
git status
echo "node_modules/" > .gitignore
git status
git add .
git commit -m "our first commit"
```

3.7 语义版本控制

1. 版本号规范

   `Major Version 主要版本`：添加新功能 (破坏现有 API) -> 6.0.0

   `Minor version 次要版本`：添加新功能 (不会破坏现有 API, 在现有 API 的基础上进行添加) -> 5.13.0

   `Patch version 补丁版本`：用于修复 bug -> 5.12.6

2. 版本号更新规范

   ^5.12.5: 主要版本不变，更新次要版本和补丁版本

   ~5.12.5: 主要版本和次要版本不变，更新补丁版本

   5.12.5: 使用确切版本，即主要版本，次要版本，补丁版本固定

3.8 查看软件包实际版本

当过了一段时间以后，其他人从版本库中下载了你的应用程序，并通过 `npm install` 命令恢复了应用程序的依赖软件包，但是此时应用程序的依赖软件包版本可能会发生变化，而应用程序的 package.json 文件中记录的只是大致版本，如何查看依赖软件包的具体版本呢?

方式一：在 node_modules 文件夹中找到对应的依赖软件包，找到它的 package.json 文件，可以在这个文件中的 version 字段中找到它的具体版本。

方式二：通过 `npm list` 命令查看所有依赖软件包的具体版本， --depth 选项指定查看依赖包的层级。

3.9 查看软件包元数据

```
npm view mongoose
npm view mongoose versions
npm view mongoose dist-tags dependencies
```

3.10 下载特定版本的软件包

```
npm i <pkg>@<version>
npm i mongoose@2.4.2 lodash@4.7.0
```

```
cat package.json
npm list --depth 0
```

3.11 删除软件包

```
npm uninstall <pkg>
npm uninstall mongoose
npm un mongoose
```

3.12 更新软件包

通过 `npm outdated` 命令可以查看哪些软件包已经过期，对应的新版本是什么。

通过 `npm update` 更新过期的软件包，更新操作遵循语义版本控制规则。

3.13 项目依赖 VS 开发依赖

项目依赖：无论在开发环境还是线上环境只要程序在运行的过程中需要使用的软件包就是项目依赖。比如 lodash，mongoose。

开发依赖：在应用开发阶段使用，在生产环境中不需要使用的软件包，比如 TypeScript 中的类型声明文件。

在 `package.json` 文件中, 项目依赖和开发依赖要分别记录，项目依赖被记录在 `dependencies` 对象中，开发依赖被记录在 `devDependencies` 中，使开发者可以在不同的环境中下载不同的依赖软件包。

在下载开发依赖时，要在命令的后面加上 `--save-dev` 选项或者 `-D` 选项。 `npm i eslint -D`

在开发坏境中下载所有依赖软件包: `npm install`

在生产环境中只下载项目依赖软件包: `npm install --prod`

3.14 本地安装与全局安装

1. 本地安装与全局安装

   本地安装：将软件包下载到应用根目录下的 node_modules 文件夹中，软件包只能在当前应用中使用。

   全局安装：将软件包下载到操作系统的指定目录中，可以在任何应用中使用。

   通过 `-g` 选项将软件包安装到全局： `npm install <pkg> -g`

   查看全局软件包安装位置： `npm root -g`

   删除全局中的软件包: `npm un npm-check-updates -g`

   查看全局中安装了哪些软件包: `npm list -g --depth 0`

   查看全局中有哪些过期软件包: `npm outdated -g`

2. nodemon

   问题：在 node 环境中每次修改 JavaScript 文件后都需要重新执行该文件才能看到效果。

   通过 nodemon 可以解决此烦恼，它是命令工具软件包，可以监控文件变化，自动重新执行文件。

   `npm install nodemon@2.0.7 -g`

   `nodemon app.js`

3. npm-check-updates 强制更新

   npm-check-updates 可以查看应用中有哪些软件包过期了，可以强制更新 `package.json` 文件中软件包版本

   1. 将 `npm-check-updates` 安装到全局： `npm install npm-check-updates -g`
4. 查看过期软件包： `npm-check-updates`

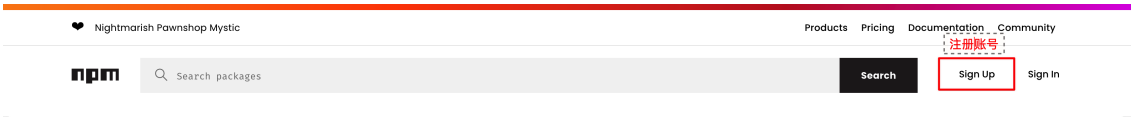
3. 更新 package.json：`ncu -u`

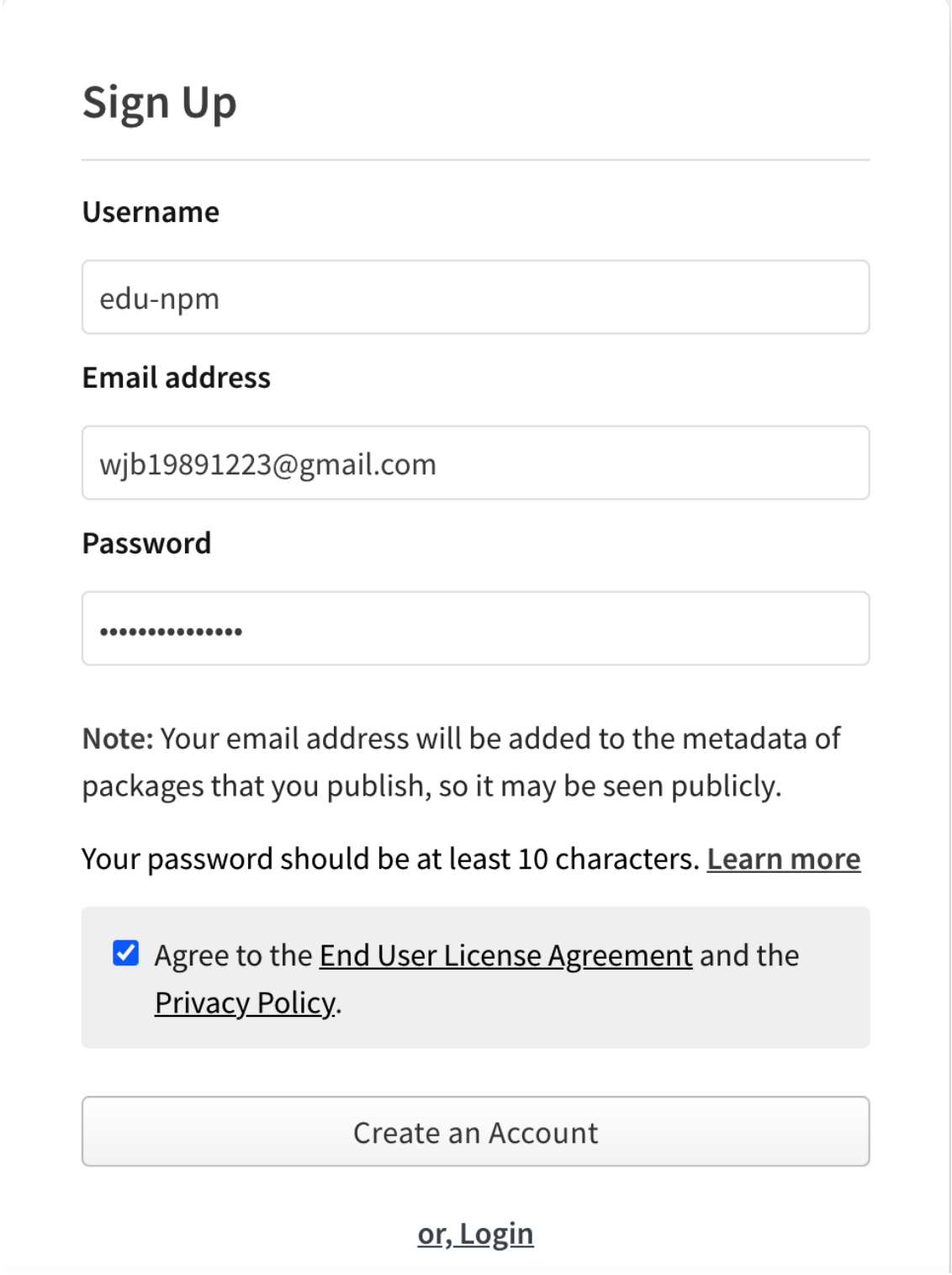
5. 安装软件包：`npm i`

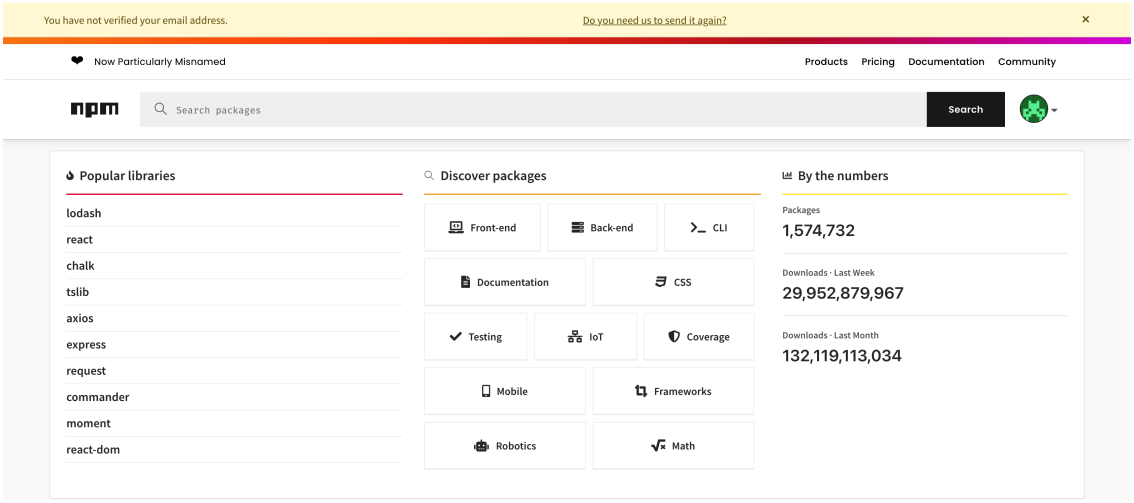
6. 检测：`npm outdated` 或 `npm-check-updates`

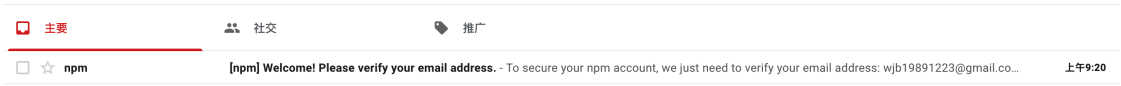
3.15 发布软件包

1. 注册 npm 账号









Welcome to npm, **edu-npm**! To complete your npm sign up, we just need to verify your email address: wjb19891223@gmail.com.

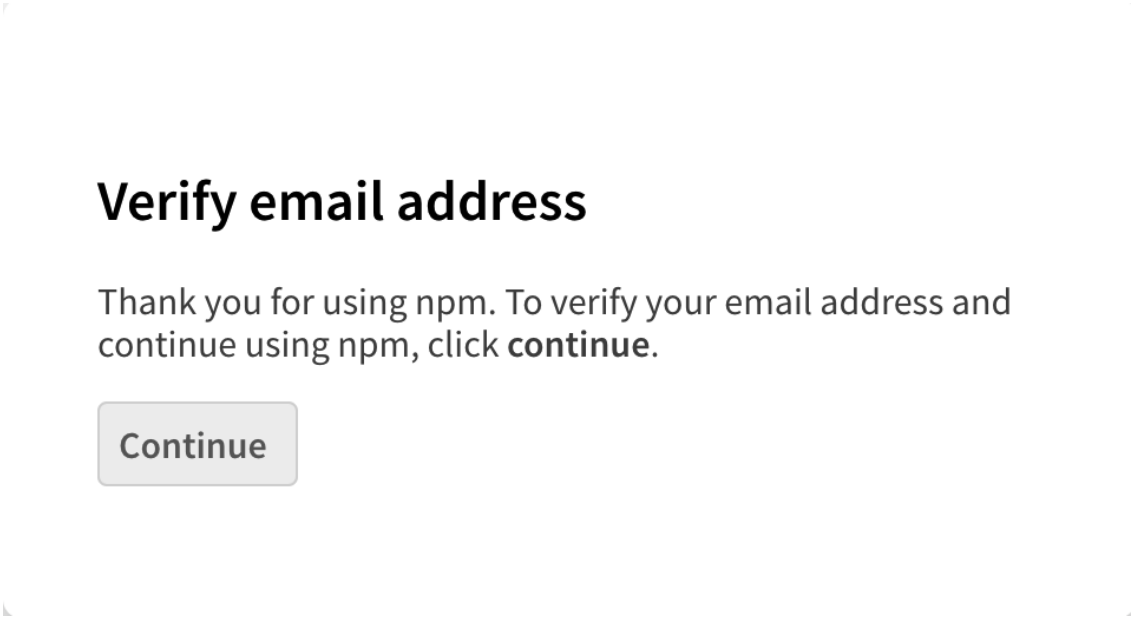
Verify email address

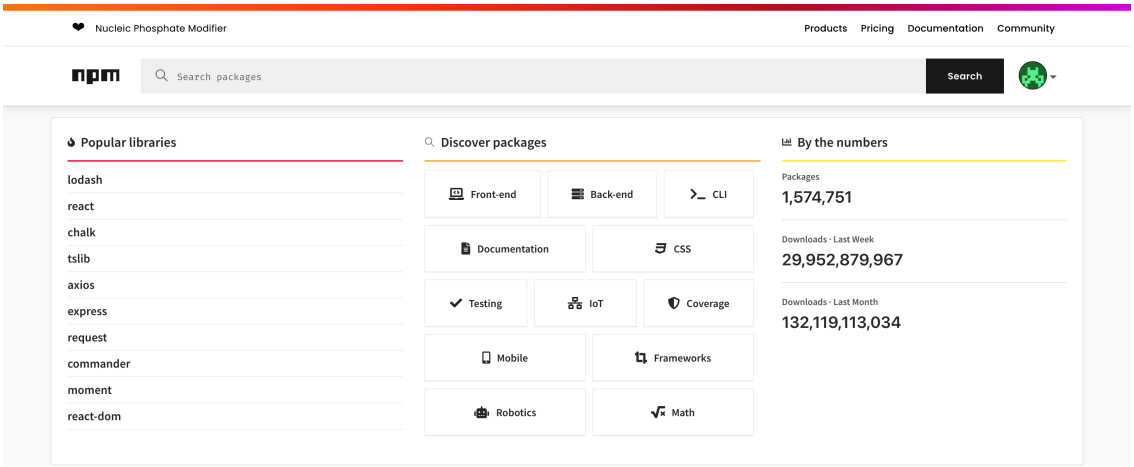
Once verified, you can start using all of npm's features to explore and publish packages.

Button not working? Paste the following link into your browser: https://www.npmjs.com/verify/eac413c3-b775-4894-8b06-8479aea4047b

You're receiving this email because you recently created an npm account. If this wasn't you, please ignore this email.

Preferences · Terms · Privacy · Sign in to npm





2. 创建软件包

```
mkdir lagou-node-test && cd "$_"
npm init --yes
```

3. 创建模块 index.js

```
module.exports = function (a, b) {
  return a + b
}
```

4. 登录 npm (npm 镜像地址必须为 npmjs.com)

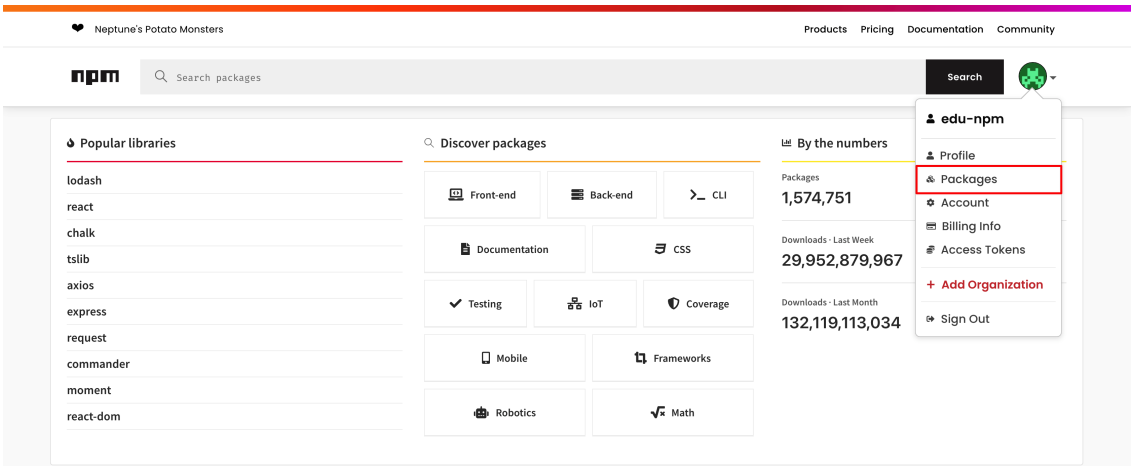
```
npm login
```

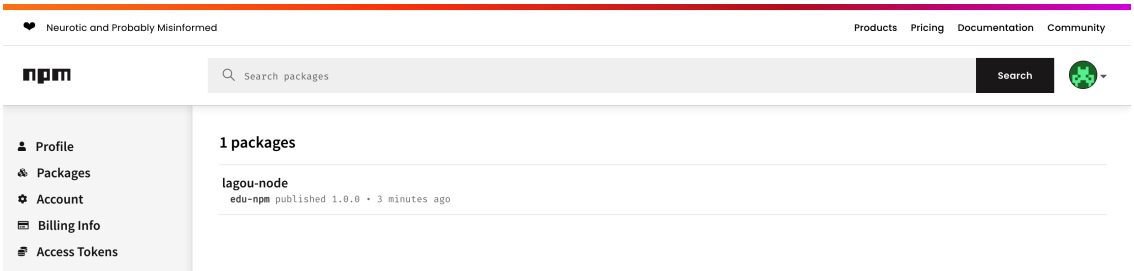
```
Administrators@mongoose lagou-node % npm login
Username: edu-npm
Password:
Email: (this IS public) wjb19891223@gmail.com
Logged in as edu-npm on https://registry.npmjs.org/.
Administrators@mongoose lagou-node %
```

5. 发布软件包

```
npm publish
```

```
Administrators@mongoose lagou-node % npm publish
npm notice
npm notice 📦  lagou-node@1.0.0
npm notice === Tarball Contents ===
npm notice 56B  index.js
npm notice 224B package.json
npm notice === Tarball Details ===
npm notice name:          lagou-node
npm notice version:       1.0.0
npm notice package size:  324 B
npm notice unpacked size: 280 B
npm notice shasum:        990c3153b937a089ffcc6cbbf59c93f680cc4ad7
npm notice integrity:     sha512-OGqEO43pnRaTY[...]yhpMIL0OzTz9A==
npm notice total files:   2
npm notice
+ lagou-node@1.0.0
Administrators@mongoose lagou-node %
```





6. 测试: 在其他应用中使用该软件包

```
npm install lagou-node-test
```

创建 index.js 模块

```
const lagouNodeTest = require("lagou-node-test")
console.log(lagouNodeTest.add(1, 2)) // 3
```

3.16 更新版本号

在软件包的源代码发生更改后, 是不能直接发布的, 应该新更新软件包的版本号然后再进行发布.

更新主要版本号：`npm version major`

更新次要版本号：`npm version minor`

更新补丁版本号：`npm version patch`

3.17 撤销已发布的软件包

1. 只有在发布软件包的24小时内才允许撤销
2. 软件包撤销后 24 小时以后才能重新发布
3. 重新发布时需要修改包名称和版本号

`npm unpublish <pkg> --force`

3.18 更改 npm 镜像地址

由于 npmjs.com 是国外的网站，大多数时候下载软件包的速度会比较慢，如何解决呢?

可以通过配置的方式更改 npm 工具的下载地址。

1. 获取 npm 配置

   `npm config list -l --json`

   -l 列表所有默认配置选项

   --json 以 json 格式显示配置选项

2. 设置 npm 配置

   获取 npm 下载地址：`npm config get registry`

   获取 npm 用户配置文件: `npm config get userconfig`

3. 更改 npm 镜像地址

   ```
   npm config set registry https://registry.npm.taobao.org
   npm config set registry https://registry.npmjs.org/
   cat .npmrc
   ```

3.19 npx 命令

npx 是 npm 软件包提供的命令，它是 Node.js 平台下软件包执行器。主要用途有两个，第一个是临时安装软件包执行后删除它，第二个是执行本地安装的提供命令的软件包。

1. 临时安装软件包执行后删除软件包

   有些提供命令的软件包使用的频率并不高，比如 create-react-app 脚手架工具，我能不能临时下载使用，然后再删掉它。

   ```
   npx create-react-app react-test
   ```

2. 执行本地安装的软件包

   现在有两个项目都依赖了某个命令工具软件包，但是项目 A 依赖的是它的 1 版本，项目 B 依赖的是它的 2 版本，我在全局到底应该安装什么版本呢？

   该软件包可以在本地进行安装，在 A 项目中安装它的 1 版本, 在 B 项目中安装它的 2 版本，在应用中可以通过 npx 调用 node_modules 文件夹中安装的命令工具。

将所有软件包安装到应用本地是现在最推荐的做法，一是可以防止软件包的版本冲突问题，二是其他开发者在恢复应用依赖时可以恢复全部依赖，因为软件包安装到本地后会被 package.json 文件记录，其他开发者在运行项目时不会因为缺少依赖而报错。

3.20 配置入口文件的作用

应用程序入口文件就是应用程序执行的起点，就是启动应用程序时执行的文件。

场景一：其他开发者拿到你的软件包以后，通过该文件可以知道应用的入口文件是谁，通过入口文件启动应用。

场景二：通过 `node 应用文件夹` 命令启动应用。node 命令会执行 package.json 文件中 main 选项指定的入口文件，如果没有指定入口文件，则执行 index.js。

3.21 模块查找规则

1. 在指定了查找路径的情况下

```
require("./server")
```

    1. 查找 server.js
    2. 查找 server.json
    3. 查找 server 文件夹, 查看入口文件 (package.json -> main)
    4. 查找 server 文件夹 中的 index.js 文件
2. 在没有指令查找路径的情况下

```
require('server')
```

```
paths: [
  '/Users/administrators/Desktop/Node/code/node_modules',
  '/Users/administrators/Desktop/Node/node_modules',
  '/Users/administrators/Desktop/node_modules',
  '/Users/administrators/node_modules',
  '/Users/node_modules',
  '/node_modules'
]
```

4. 异步编程

4.1 CPU 与存储器

目标: 了解程序运行过程中 CPU 和存储器起到了什么作用或者说扮演了什么角色.

1. CPU

   中央处理器，计算机核心部件，负责运算和指令调用。

   开发者编写的 JavaScript 代码在被编译为机器码以后就是通过 CPU 执行的。

2. 存储器

   内存：用于临时存储数据，断电后数据丢失。由于数据读写速度快，计算机中的应用都是在内存中运行的。

   磁盘：用于持久存储数据，断电后数据不丢失。内部有磁头依靠马达转动在盘片上读写数据, 速度比内存慢。

计算机应用程序在没有运行时是存储在磁盘中的，当我们启动应用程序后，应用程序会被加载到内存中运行，应用程序中的指令会被中央处理器CPU来执行。

4.2 什么是 I/O

I 就是 Input 表示输入，O 就是 Output 表示输出，I/O 操作就是输入输出操作。什么样的操作属于 I/O 操作呢？

比如数据库的读写操作就是 I/O 操作，因为数据库文件是存储在磁盘中的，而我们编写的程序是运行在内存中的，将内存中的数据写入数据库对于内存来说就是输出，查询数据库中的数据就是将磁盘中的数据读取到内存中，对于内存来说就是输入。

4.3 I/O 模型

从数据库中查询数据(将磁盘中的文件内容读取到内存中)，由于磁盘的读写速度比较慢，查询内容越多花费时间越多。无论 I/O 操作需要花费多少时间，在 I/O 操作执行完成后，CPU 都是需要获取到操作结果的，那么问题就来了，CPU 在发出 I/O 操作指令后是否要等待 I/O 操作执行完成呢？这就涉及到 I/O 操作模型了，I/O 操作的模型有两种。

第一种是 CPU 等待 I/O 操作执行完成获取到操作结果后再去执行其他指令，这是同步 I/O 操作 (阻塞 I/O)。

第二种是 CPU 不等待 I/O 操作执行完成，CPU 在发出 I/O 指令后，内存和磁盘开始工作，CPU 继续执行其他指令。当 I/O 操作完成后再通知 CPU I/O 操作的结果是什么。这是异步 I/O 操作 (非阻塞 I/O) 。

同步 I/O 在代码中的表现就是代码暂停执行等待 I/O 操作，I/O 操作执行完成后再执行后续代码。

异步 I/O 在代码中的表现就是代码不暂停执行，I/O 操作后面的代码可以继续执行，当 I/O 操作执行完成后通过回调函数的方式通知 CPU，说 I/O 操作已经完成了，基于 I/O 操作结果的其他操作可以执行了(通知 CPU 调用回调函数)。

同步 I/O 和 异步 I/O 区别就是是否等待 I/O 结果。

Node 采用的就是异步非阻塞 I/O 模型。

```
const fs = require("fs")

fs.readFile("./x.txt", "utf-8", function (error, data) { console.log(data) })
console.log("Hello")
```

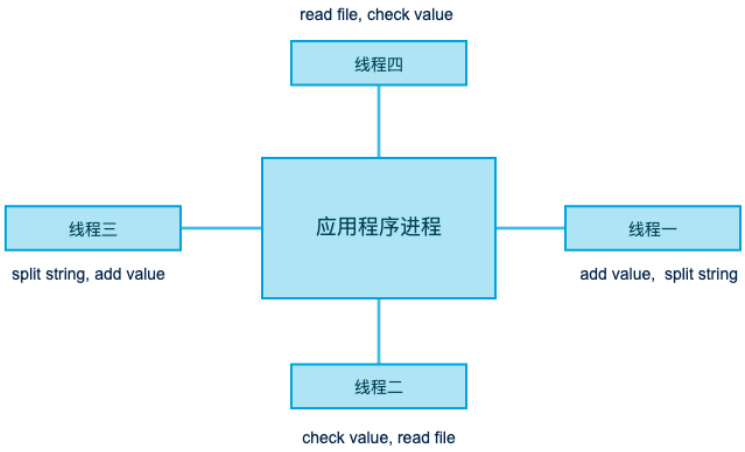
```
const fs = require("fs")

const data = fs.readFileSync("./y.txt", { encoding: "utf-8" })
console.log(data)
```

4.4 进程与线程

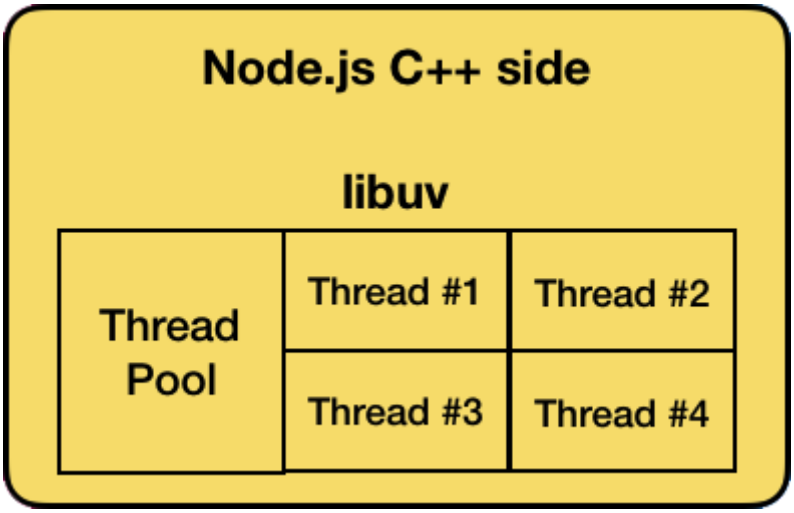
每当我们运行应用程序时，操作系统都会创建该应用程序的实例对象，该实例对象就是应用程序的进程，操作系统会按照进程为单位为应用程序分配资源，比如内存，这样程序才能够在计算机的操作系统中运行起来。

线程被包裹在进程之中，是进程中的实际运作单位，一条线程指的就是进程中的一个单一顺序的控制流。也就是说，应用程序要做的事情都存储在线程之中。可以这样认为，一条线程就是一个待办列表，供 CPU 执行。

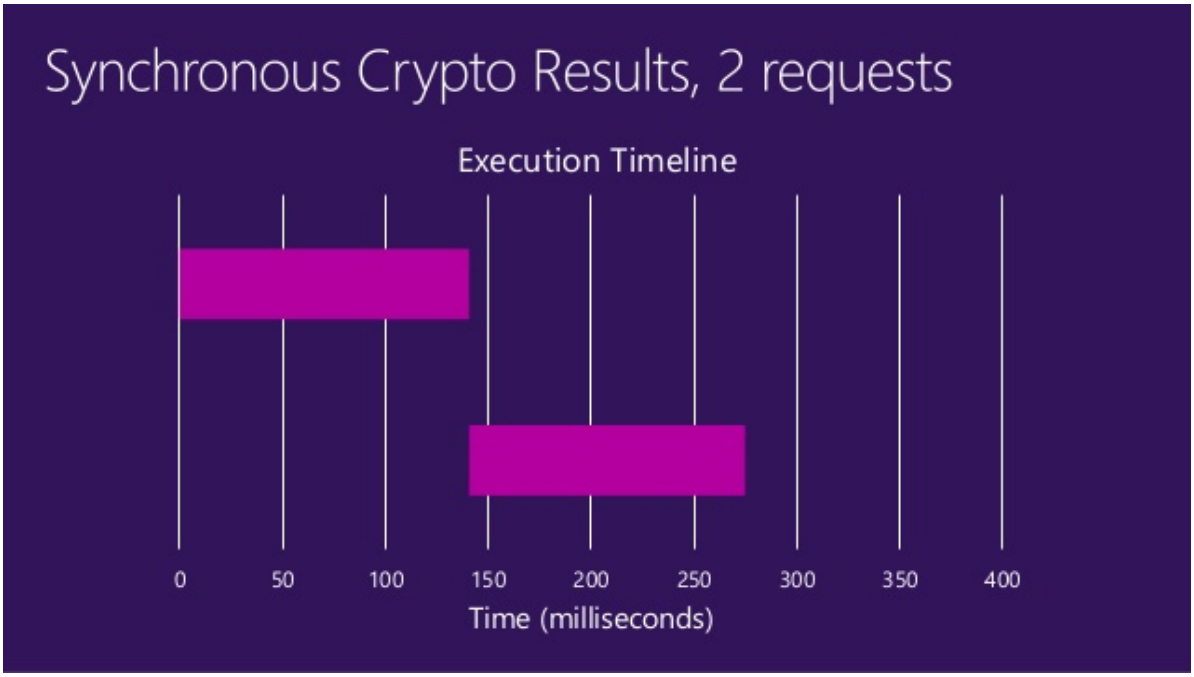


4.5 JS 单线程 OR 多线程？

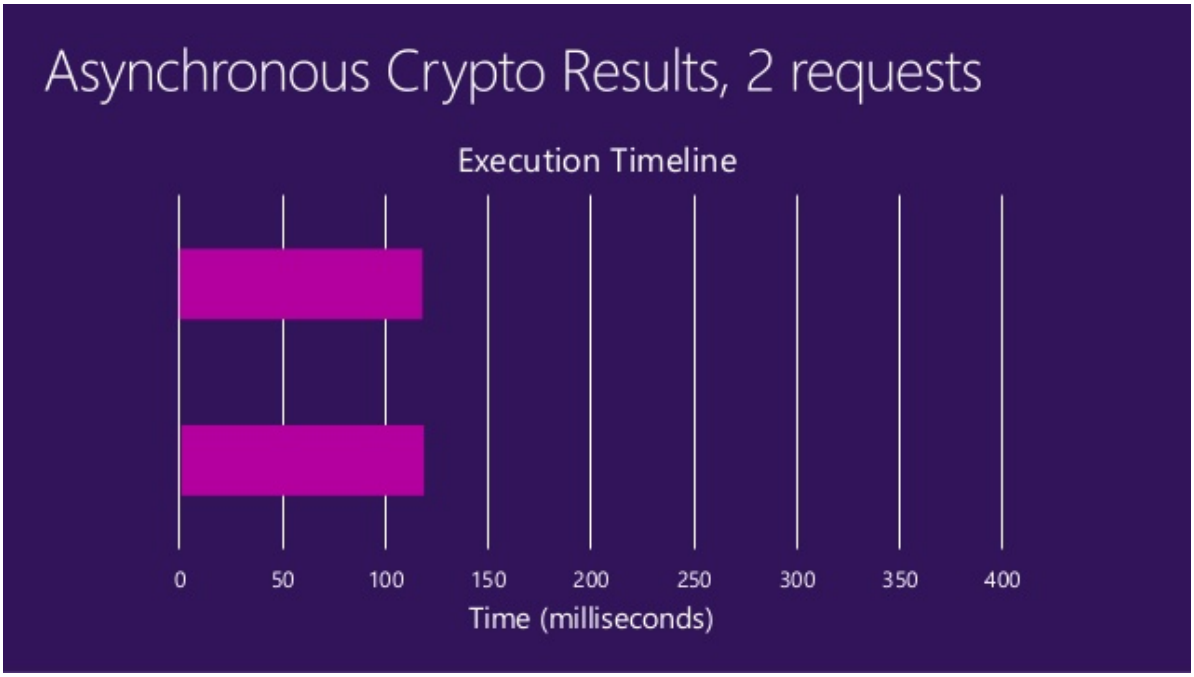
在 Node.js 代码运行环境中，它为 JavaScript 代码的执行提供了一个主线程，通常我们所说的单线程指的就是这个主线程，主线程用来执行所有的同步代码。但是 Node.js 代码运行环境本身是由 C++ 开发的，在 Node.js 内部它依赖了一个叫做 libuv 的 c++ 库，在这个库中它维护了一个线程池，默认情况下在这个线程池中存储了 4 个线程，JavaScript 中的异步代码就是在这些线程中执行的，所以说 JavaScript 代码的运行依靠了不止一个线程，所以 JavaScript 本质上还是多线程的。




```
const crypto = require('crypto')
const NUM_REQUESTS = 2;
for (let i = 0; i < NUM_REQUESTS; i++) {
    crypto.pbkdf2Sync('srcret', 'salt', 10000, 512, 'sha512')
}
```




```
const crypto = require('crypto')
const NUM_REQUESTS = 2;
for (let i = 0; i < NUM_REQUESTS; i++) {
    crypto.pbkdf2('srcret', 'salt', 10000, 512, 'sha512')
}
```



4.6 基于回调函数的异步编程

1. 什么是回调函数

   回调函数是指通过函数参数的方式将一个函数传递到另一个函数中，参数函数就是回调函数。

```
function A() {
  console.log("A is running")
}

function B(callback) {
  console.log("B Start")
  callback() // A is running
  console.log("B End")
}

B(A)
```

   我们经常将回调函数写成 callback，实际上它是 call then back 的简写，含义是调用后返回，就是在主函数中调用参数函数，参数函数调用完成后返回主函数继续执行主函数中的代码。

   为什么在 B 函数中不直接调用 A 函数而要通过参数的方式传递进去？

   通常在编写应用程序时，B 函数都是语言内部或者其他开发者定义好的，我们看不到内部代码或者说不能直接在他内部代码中插入我们的代码，而我们又想介入程序的执行，此时就可以通过回调函数的方式将我们的逻辑传递给 B 函数，B 函数在内部再来调用这个回调函数。

2. 回调函数传递参数

   在主函数中调用回调函数时，可以为回调函数传递参数。

```
function A(arg) {
  console.log("A is running")
  console.log(arg)
}

function B(callback) {
  console.log("B Start")
  callback("我是B函数传递给A函数的参数") // A is running
  console.log("B End")
}

B(A)
```

3. 回调函数在异步编程中的应用

   在异步编程中，异步 API 执行的结果就是通过回调函数传递参数的方式传递到上层代码中的。

```
const fs = require("fs")

fs.readFile("./index.html", "utf-8", function (error, data) {
  if (error) console.log("发生了错误")
  console.log(data)
})
```

4. 回调地狱

   回调地狱是回调函数多层嵌套导致代码难以维护的问题。

   基于回调函数的异步编程一不小心就会产生回调地狱的问题。

```
const fs = require("fs")

fs.readFile("./x.txt", "utf-8", function (error, x) {
  fs.readFile("./y.txt", "utf-8", function (error, y) {
    fs.readFile("./z.txt", "utf-8", function (error, z) {
      console.log(x)
      console.log(y)
      console.log(z)
    })
  })
})
```

```
const x = fs.readFile('./x.txt', 'utf-8')
const y = fs.readFile('./y.txt', 'utf-8')
const z = fs.readFile('./z.txt', 'utf-8')
console.log(x)
console.log(y)
console.log(z)
```

4.7 基于 Promise 的异步编程

1. Promise 概述

   Promise 是 JavaScript 中异步编程解决方案，可以解决回调函数方案中的回调地狱问题。

   可以将 Promise 理解为容器，用于包裹异步 API 的容器，当容器中的异步 API 执行完成后，Promise 允许我们在容器的外面获取异步 API 的执行结果，从而避免回调函数嵌套。

Promise 翻译为承若，表示它承若帮我们做一些事情，既然它承若了它就要去做，做就会有一个过程，就会有一个结果，结果要么是成功要么是失败。

所以在 Promise 中有三种状态, 分别为等待(pending)，成功(fulfilled)，失败(rejected)。

默认状态为等待，等待可以变为成功，等待可以变为失败。

状态一旦更改不可改变，成功不能变回等待，失败不能变回等待，成功不能变成失败，失败不能变成成功。

2. Promise 基础语法

```
const fs = require("fs")

const promise = new Promise(function (resolve, reject) {
  fs.readFile("./x.txt", "utf-8", function (error, data) {
    if (error) {
      // 将状态从等待变为失败
      reject(error)
    } else {
      // 将状态从等待变为成功
      resolve(data)
    }
  })
})

promise
  .then(function (data) {
    console.log(data)
  })
  .catch(function (error) {
    console.log(error)
  })
```

3. Promise 链式调用

```
const fs = require("fs")

function readFile(path) {
  return new Promise(function (resolve, reject) {
    fs.readFile(path, "utf-8", function (error, data) {
      if (error) return reject(error)
      resolve(data)
    })
  })
}

readFile("./x.txt")
  .then(function (x) {
    console.log(x)
    return readFile("./y.txt")
  })
  .then(function (y) {
    console.log(y)
    return readFile("./z.txt")
  })
  .then(function (z) {
    console.log(z)
```

```
  })
  .catch(function (error) {
    console.log(error)
  })
  .finally(function () {
    console.log("finally")
  })
```

4. Promise.all 并发异步操作

```
const fs = require("fs")

Promise.all([
  readFile("./x.txt"),
  readFile("./y.txt"),
  readFile("./z.txt")
]).then(function (data) {
  console.log(data)
})
```

4.8 基于异步函数的异步编程

Promise 虽然解决了回调地狱的问题，但是代码看起来仍然不简洁。

使用异步函数简化代码提高异步编程体验。

1. 异步函数概述

```
const fs = require("fs")

function readFile(path) {
  return new Promise(function (resolve, reject) {
    fs.readFile(path, "utf-8", function (error, data) {
      if (error) return reject(error)
      resolve(data)
    })
  })
}

async function getFileContent() {
  let x = await readFile("./x.txt")
  let y = await readFile("./y.txt")
  let z = await readFile("./z.txt")
  return [x, y, z]
}

getFileContent().then(console.log)
```

async 声明异步函数的关键字，异步函数的返回值会被自动填充到 Promise 对象中。

await 关键字后面只能放置返回 Promise 对象的 API。

await 关键字可以暂停函数执行，等待 Promise 执行完后返回执行结果。

await 关键字只能出现在异步函数中。

2. util.promisify

在 Node.js 平台下，所有异步方法使用的都是基于回调函数的异步编程。为了使用异步函数提高异步编程体验，可以使用 util 模块下面的 promisify 方法将基于回调函数的异步 API 转换成返回 Promise 的API。

```
const fs = require("fs")
const util = require("util")
const readFile = util.promisify(fs.readFile)

async function getFileContent() {
  let x = await readFile("./x.txt", "utf-8")
  let y = await readFile("./y.txt", "utf-8")
  let z = await readFile("./z.txt", "utf-8")
  return [x, y, z]
}

getFileContent().then(console.log)
```

4.9. Event Loop 机制概述

1. 为什么要学习事件循环机制?

   学习事件循环可以让开发者明白 JavaScript 的运行机制是怎么样的。

2. 事件循环机制做的是什么事情?

   事件循环机制用于管理异步 API 的回调函数什么时候回到主线程中执行。

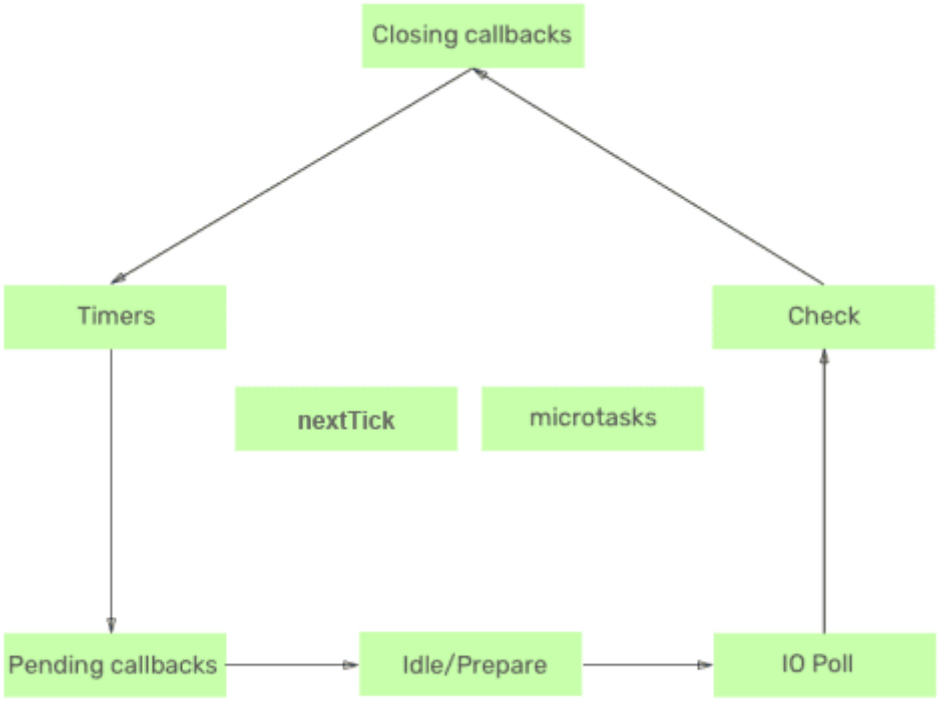
   Node.js 采用的是异步 I/O 模型。同步 API 在主线程中执行，异步 API 在底层的 C++ 维护的线程中执行，异步 API 的回调函数在主线程中执行。在 JavaScript 应用运行时，众多异步 API 的回调函数什么时候能回到主线程中调用呢？这就是事件循环机制做的事情，管理异步 API 的回调函数什么时候回到主线程中执行。

3. 为什么这种机制叫做事件循环?

   因为 Node.js 是事件驱动的。事件驱动就是当什么时候做什么事情，做的事情就定义在回调函数中，可以将异步 API 的回调函数理解为事件处理函数，所以管理异步API回调函数什么时候回到主线程中调用的机制叫做事件循环机制。

4.10. Event Loop 的六个阶段

事件循环是一个循环体，在循环体中有六个阶段，在每个阶段中，都有一个事件队列，不同的事件队列存储了不同类型的异步API 的回调函数。



1. Timers：用于存储定时器的回调函数(setInterval, setTimeout)。
2. Pending callbacks：执行与操作系统相关的回调函数，比如启动服务器端应用时监听端口操作的回调函数就在这里调用。
3. Idle, prepare：系统内部使用。
4. IO Poll：存储 I/O 操作的回调函数队列，比如文件读写操作的回调函数。

   如果事件队列中有回调函数，执行它们直到清空队列。

   否则事件循环将在此阶段停留一段时间以等待新的回调函数进入，这个等待取决于以下两个条件：

   1. setImmediate 队列(check 阶段)中存在要执行的回调函数.
   2. timers 队列中存在要执行的回调函数. 在这种情况下, 事件循环将移至 check 阶段, 然后移至 Closing callbacks 阶段, 并最终从 timers 阶段进入下一次循环。
5. Check：存储 setImmediate API 的回调函数。
6. Closing callbacks：执行与关闭事件相关的回调，例如关闭数据库连接的回调函数等。

循环体会不断运行以检测是否存在没有调用的回调函数，事件循环机制会按照先进先出的方式执行他们直到队列为空。

4.11. 宏任务与微任务

4.11.1 宏任务与微任务

宏任务：setInterval, setTimeout, setImmediate, I/O

微任务：Promise.then Promise.catch Promise.finally, process.nextTick

4.11.2 微任务与宏任务的区别

1. 微任务的回调函数被放置在微任务队列中，宏任务的回调函数被放置在宏任务队列中。
2. 微任务优先级高于宏任务。

   当微任务事件队列中存在可以执行的回调函数时，事件循环在执行完当前阶段的回调函数后会暂停进入事件循环的下一个阶段，事件循环会立即进入微任务的事件队列中开始执行回调函数，当微任务队列中的回调函数执行完成后，事件循环再进入到下一个阶段开始执行回调函数。

nextTick 的优先级高于 microTask，在执行任务时，只有 nextTick 中的所有回调函数执行完成后才会开始执行 microTask。

不同阶段的宏任务的回调函数被放置在了不同的宏任务队列中，宏任务与宏任务之间没有优先级的概念，他们的执行顺序是按照事件循环的阶段顺序进行的。

4.12 Event Loop 代码解析

在 Node 应用程序启动后，并不会立即进入事件循环，而是先执行输入代码，从上到下开始执行，同步 API 立即执行，异步 API 交给 C++ 维护的线程执行，异步 API 的回调函数被注册到对应的事件队列中。当所有输入代码执行完成后，开始进入事件循环。

```
console.log("start")

setTimeout(() => {
  console.log("setTimeout 1")
}, 0)

setTimeout(() => {
  console.log("setTimeout 2")
}, 0)

console.log("end")
// start end 1 2
```

```
setTimeout(() => console.log("1"), 0)
setImmediate(() => console.log("2"))

function sleep(delay) {
  var start = new Date().getTime()
  while (new Date().getTime() - start < delay) {
    continue
  }
}
sleep(1000)
// 1 2
```

```
setTimeout(() => console.log("1"), 0)
setImmediate(() => console.log("2"))
//  2 1 或 1 2
```

```
const fs = require("fs")

fs.readFile("./index.html", () => {
  setTimeout(() => console.log("1"), 0)
  setImmediate(() => console.log("2"))
})
// 2 1
```

```
setTimeout(() => console.log("1"), 50)
process.nextTick(() => console.log("2"))
setImmediate(() => console.log("3"))
process.nextTick(() => console.log("4"))
// 2 4 3 1
```

```
setTimeout(() => console.log(1))
setImmediate(() => console.log(2))
process.nextTick(() => console.log(3))
Promise.resolve().then(() => console.log(4))
;(() => console.log(5))()
// 5 3 4 1 2
```

```
process.nextTick(() => console.log(1))
Promise.resolve().then(() => console.log(2))
process.nextTick(() => console.log(3))
Promise.resolve().then(() => console.log(4))
// 1 3 2 4
```

```
setTimeout(() => console.log("1"), 50)
process.nextTick(() => console.log("2"))
setImmediate(() => console.log("3"))
process.nextTick(() =>
  setTimeout(() => {
    console.log("4")
  }, 1000)
)
// 2 3 1 4
```

4.13. process.nextTick 与 setImmediate()

1. process.nextTick()

   此方法的回调函数优先级最高，会在事件循环之前被调用。

   如果你希望异步任务尽可能早地执行，那就使用 process.nextTick。

   ```
   const fs = require("fs")

   function readFile(fileName, callback) {
     if (typeof fileName !== "string") {
       return callback(new TypeError("filename 必须是字符串类型"))
     }
     fs.readFile(filename, function (err, data) {
       if (err) return callback(err)
       return callback(null, data)
     })
   }
   ```

   此段代码的问题在于 readFile 方法根据传入的参数类型，callback 可能会在主线程中直接被调用，callback 也可能在事件循环的 IO 轮询阶段被调用，这可能会导致不可预测的问题发生。如何使 readFile 方法变成完全异步的呢?

```
const fs = require("fs")

function readFile(fileName, callback) {
  if (typeof fileName !== "string") {
    return process.nextTick(callback, new TypeError("filename 必须是字符串类型"))
  }
  fs.readFile(fileName, (err, data) => {
    if (err) return callback(err)
    return callback(null, data)
  })
}
```

经过以上更改以后，无论 fileName 参数是否是字符串类型，callback 都不会在主线程中直接被调用。

2. setImmediate()

   setImmediate 表示立即执行，它是宏任务，回调函数会被会放置在事件循环的 check 阶段。

   在应用中如果有大量的计算型任务，它是不适合放在主线程中执行的，因为计算任务会阻塞主线程，主线程一旦被阻塞，其他任务就需要等待，所以这种类型的任务最好交给由 C++ 维护的线程去执行。

   可以通过 setImmediate 方法将任务放入事件循环中的 check 阶段，因为代码在这个阶段执行不会阻塞主线程，也不会阻塞事件循环。

```
function sleep(delay) {
  var start = new Date().getTime()
  while (new Date().getTime() - start < delay) {
    continue
  }
  console.log("ok")
}
```

```
console.log("start")
sleep(2000)
console.log("end")
```

```
console.log("start")
setImmediate(sleep, 2000)
console.log("end")
```

结论：Node 适合 I/O 密集型任务，不适合 CPU 密集型任务，因为主线程一旦阻塞，程序就卡主了。

5. 网站概述

5.1 网站的组成

从开发者的角度来看，web 应用主要由三部分组成：用户界面，业务逻辑，数据。

1. 用户界面 (视图层)：用于将数据展示给用户的地方，采用 HTML，CSS，JavaScript 编写。
2. 业务逻辑 (控制层)：实现业务需求和控制业务流程的地方，可以采用 Java, PHP, Python, JavaScript 编写。
3. 数据 (模型层)：应用的核心部分, 应用业务逻辑的实现，用户界面的展示都是基于数据的， web 应用中的数据通常是存储在数据库中的，数据库可以采用 MySql, Mongodb 等。

5.2 什么是 web 服务器

服务器是指能够向外部(局域网或者万维网)提供服务的机器(计算机)就是服务器。

在硬件层面，web 服务器就是能够向外部提供网站访问服务的计算机。

在这台计算机中存储了网站运行所必须的代码文件和资源文件。

在软件层面，web 服务器控制着用户如何访问网站中的资源文件，控制着用户如何与网站进行交互。

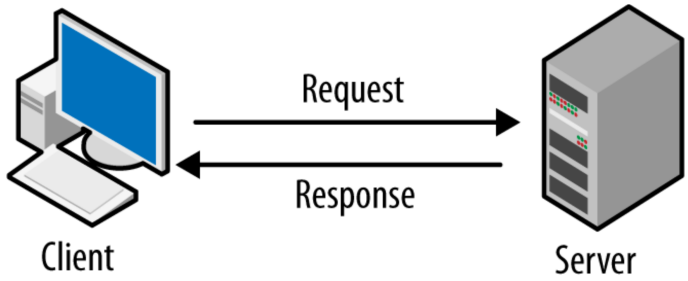
5.3 客户端

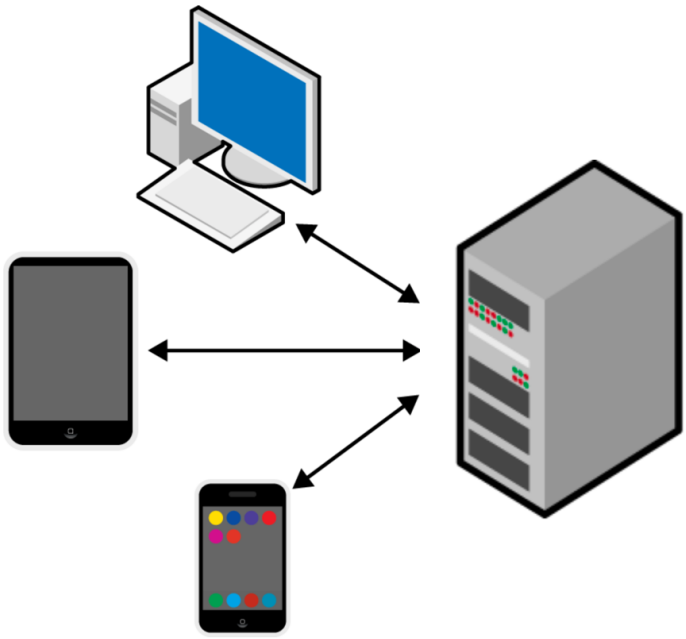
web 应用中的客户端是指用户界面的载体，实际上就是浏览器。

用户可以通过浏览器这个客户端访问网站应用的界面，通过用户界面与网站应用进行交互。

5.4 网站的运行

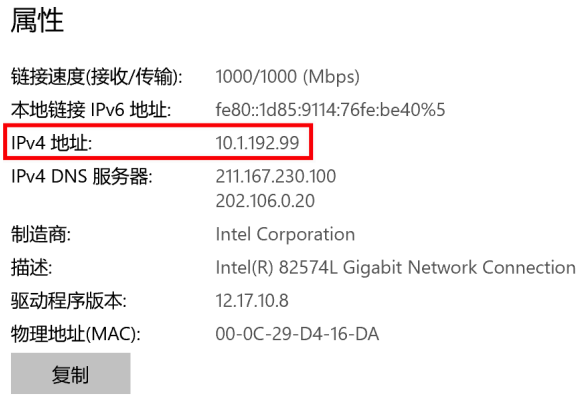
web 应用是基于请求和响应模型的。





5.5 IP和域名

Internet Protocol address：互联网协议地址，标识网络中设备的地址，具有唯一性。例如: 45.113.192.101

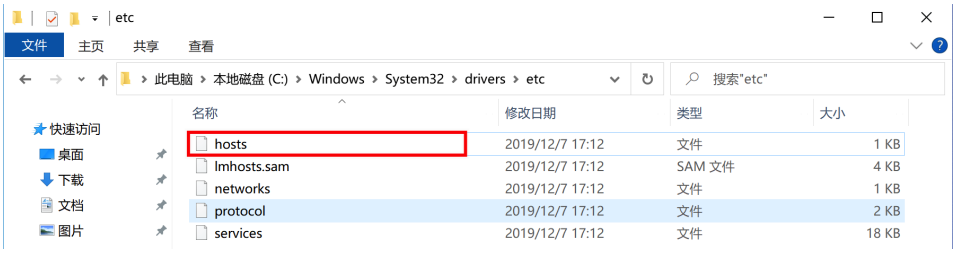


域名 (Domain Name)：是由一串用点分隔的字符组成的互联网上某一台计算机或计算机组的名称，用于在数据传输时标识计算机的电子方位 (摘自维基百科)。

ping www.baidu.com

5.6 DNS 服务器

Domain Name Server：域名服务器，互联网域名解析系统，它可以将"人类可识别"的标识符映射为系统内部通常为数字形式的标识码。(摘自维基百科)




```
14   # For example:
15   #
16   #      102.54.94.97     rhino.acme.com          # source server
17   #       38.25.63.10     x.acme.com              # x client host
18
19   # localhost name resolution is handled within DNS itself.
20   #   127.0.0.1       localhost
21   #   ::1             localhost
```

5.7 端口

是设备与外界通讯交流的出口，此处特指计算机中的虚拟端口。0 ~ 65535

比如在一座大厦当中有很多房间，每间房间都提供着不同的服务，我们可以通过房间号找到提供不同服务的房间。

服务器就是这座大厦，在服务器中可以提供很多服务，比如 web 访问服务，邮件的收发服务，文件的上传下载服务，用户在找到服务器以后如何去找具体的服务呢？答案就是端口号，端口号就是大厦中的房间号，在服务器中通过端口号区分不同的服务。

也就是说，服务器中的各种应用，要想向外界提供服务，必须要占用一个端口号。

通常 web 应用占用 80 端口，在浏览器中访问应用时 80 可以省略，因为默认就访问 80。

5.8 URL

URL：统一资源定位符，表示我们要访问的资源在哪以及要访问的资源是什么。

protocol://hostname[:port]/path

http://www.example.com/index.html

5.9 台前和后台, 前端与后端

前台和后台都是指用户界面。前台是为客户准备的，每个人都可以访问的用户界面。后台是为网站管理员准备的，只有登录以后才能访问的用户界面，用于管理网站应用中的数据。

前端是指开发客户端应用的程序员。

后端是指开发服务器端应用程序的程序员。

5.10 开发环境说明

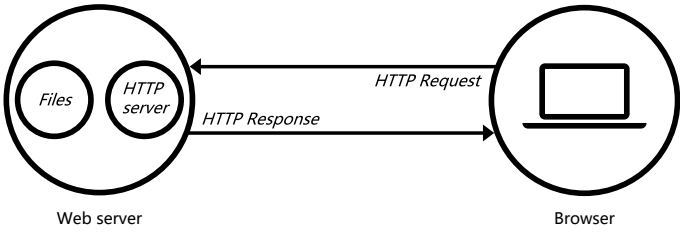
在开发环境中，开发者机器既充当了客户端的角色又充当了服务器的角色。

本机IP: 127.0.0.1

本机域名: localhost

5.11 创建 web server

1. 创建软件层面的 web 服务器，用于控制资源要如何被访问。




```
const http = require("http")

// 创建 web server
const server = http.createServer(function (req, res) {
  // req 请求对象, 包含请求信息
  // res 响应对应, 用于对请求进行响应
  if (req.url === "/") {
    res.write("Hello Node.js")
    res.end()
  }
  if (req.url === "/api/course") {
    res.write(JSON.stringify([1, 2, 3]))
    res.end()
  }
})

server.listen(3000)
console.log("web server is running...")
```

这种方法无法构建复杂的大型应用程序，因为在大型应用程序中会有各种各种的请求需要处理，我们不想将所有的 if 条件判断都写在 createServer 方法的回调函数中。

扫码联系老师

技能评估、福利资料、课程优惠

Made with ♥ by LagouFed