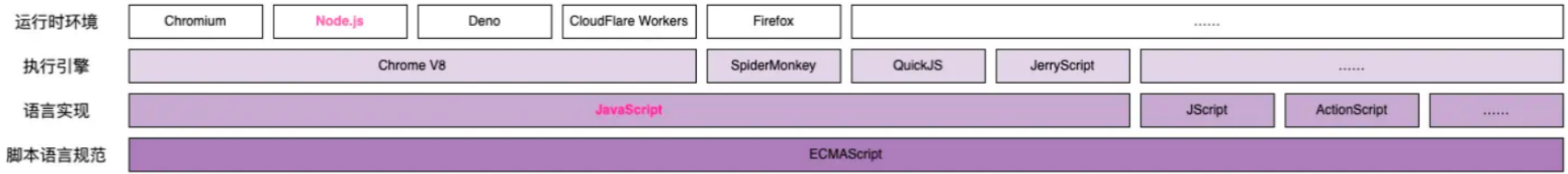
Node.js 基础

Node.js不是一门编程语言，它是一个开源的、跨平台的 JavaScript 运行时环境。

Node.js与JavaScript的关系



这里我们从下往上梳理。

最下面一层是脚本语言规范（Spec），由于我们讲的是 Node.js，所以这里最下层只写 ECMAScript。

再往上一层就是对于该规范的实现了，如 JavaScript、JScript 以及 ActionScript 等都属于对 ECMAScript 的实现。

然后就是执行引擎，JavaScript 常见的引擎有 V8、SpiderMonkey、QuickJS 等（其相关 Logo 如下图所示）。

最上面就是运行时环境了，比如基于 V8 封装的运行时环境有 Chromium、Node.js、Deno、CloudFlare Workers 等等。而我们所说的 Node.js 就是在运行时环境这一层。

可以看到，JavaScript 在第二层，Node.js 则在第四层。

Node.js发展历史

2009年

- **Node.js 诞生**：Ryan Dahl 在一个会议上首次介绍了 Node.js，标志着这个项目正式启动。
- **第一版 npm 创建**：npm（Node Package Manager）作为 Node.js 的包管理器，最初是为了帮助开发者管理 Node.js 项目中的依赖。

2010年

- **Express 诞生**：Express 是一个灵活的 Node.js Web 应用框架，提供了一套强大的功能来构建各种 Web 应用。
- **Socket.io 诞生**：Socket.io 是一个实时通信库，使得在 Web 应用中实现实时功能变得更加容易。

2011年

- **npm 发布 1.0 版本**：npm 成为了 Node.js 生态系统中不可或缺的一部分，帮助开发者管理和分发代码。
- **较大公司开始采用 Node.js**：LinkedIn、Uber 等公司开始在生产环境中使用 Node.js，推动了 Node.js 的普及。
- **hapi 诞生**：hapi 是另一个 Node.js Web 应用框架，专注于提供丰富的插件系统和灵活性。

2013年

- **第一个使用 Node.js 的大型博客平台：Ghost**：Ghost 是一个开源的博客平台，完全基于 Node.js 构建。
- **Koa 诞生**：Koa 是一个由 Express 团队创建的下一代 Web 框架，旨在提供更简洁和优雅的 API 设计。

2014年

- **大分支：io.js**：io.js 是 Node.js 的一个主要分支，目的是引入 ES6 支持并加快开发速度。

2015年

- Node.js **基金会诞生**：为了更好地推动 Node.js 的发展，成立了 Node.js 基金会。
- IO.js **被合并回** Node.js：经过一段时间的独立发展，IO.js 最终被合并回 Node.js，带来了许多新特性和改进。
- npm **引入私有模块**：npm 开始支持私有模块，使得企业可以更安全地管理内部代码。
- Node.js 4 **发布**：这是 Node.js 的一个重要版本，尽管之前从未发布过 1、2 和 3 版本。

2016年

- leftpad **事件**：一个依赖问题导致许多 Node.js 项目受到影响，引发了对 npm 生态系统的广泛讨论。
- Yarn **诞生**：Yarn 是一个由 Facebook 推出的新的包管理器，旨在解决 npm 的一些性能和安全性问题。
- Node.js 6 **发布**：这个版本带来了许多新特性和改进。

2017年

- npm **更加注重安全性**：npm 团队开始重视安全性问题，推出了许多新功能来保护开发者。
- Node.js 8 **发布**：这个版本引入了对 HTTP/2 的支持，以及其他许多改进。
- V8 **在其测试套件中引入了** Node.js：除了 Chrome，Node.js 也正式成为 JS 引擎的标杆。
- **每周** 30 **亿次** npm **下载**：这显示了 npm 在开发者中的普及程度。

2018年

- Node.js 10 **发布**：这个版本带来了对 ES 模块 .mjs 实验支持。
- Node.js 11 **发布**：继续推动 Node.js 的发展和改进。

2019-2023年

Node.js 版本迭代发布。

2024年

- Node.js 22.5.0 **发布**：这是最新的一个版本，发布于 2024 年 7 月 17 日。
- Node.js 20.15.1 **发布**：代号为 Iron，发布于 2024 年 7 月 8 日。
- Node.js 18.20.4 **发布**：代号为 Hydrogen，发布于 2024 年 7 月 8 日。

Noc

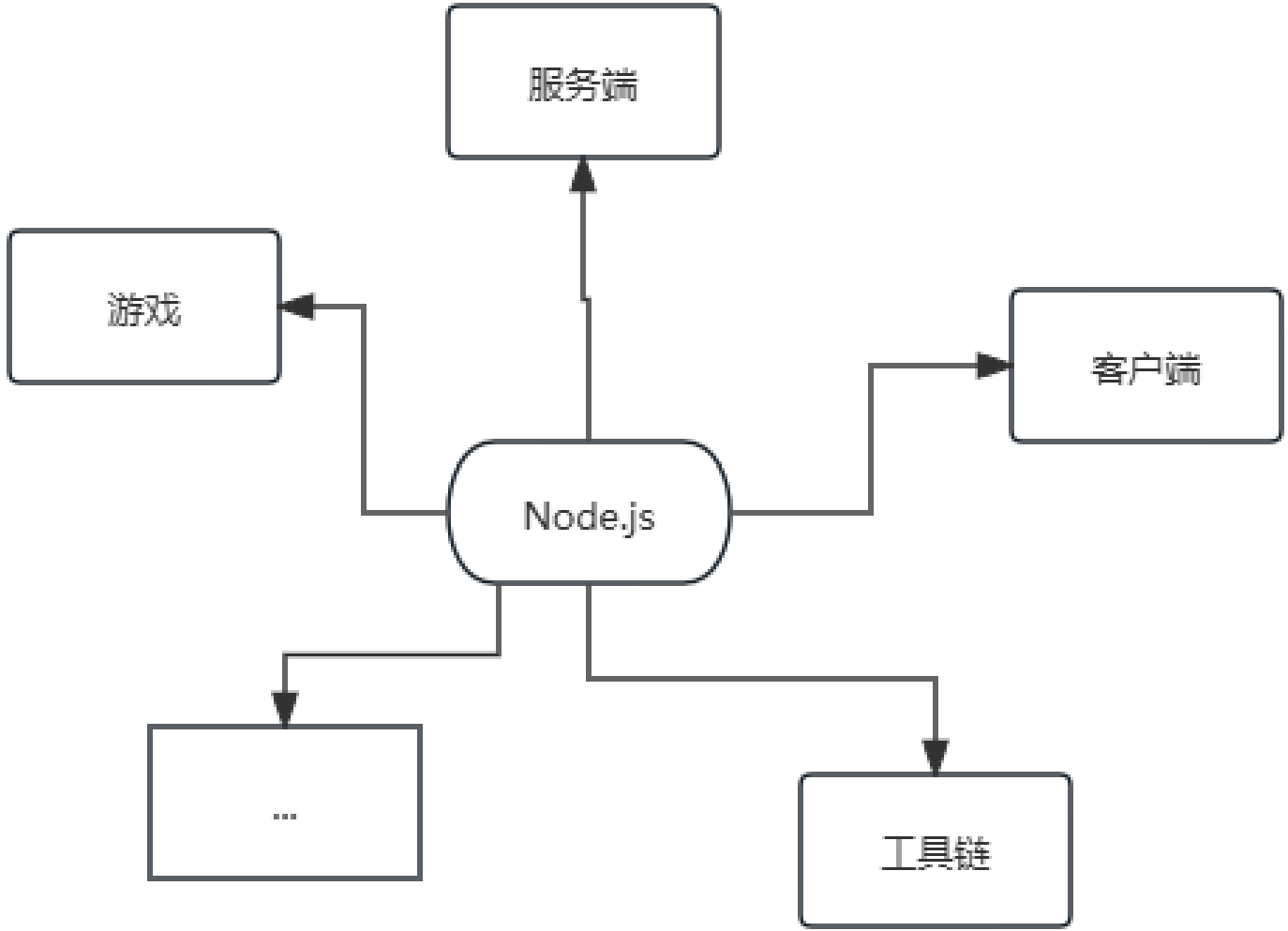
在 Node 考虑
各种锁

Node.js
TCP、U

除了服务 的能
力。

Node.js word、
钉钉等

目前来 .js 的趋
势越来

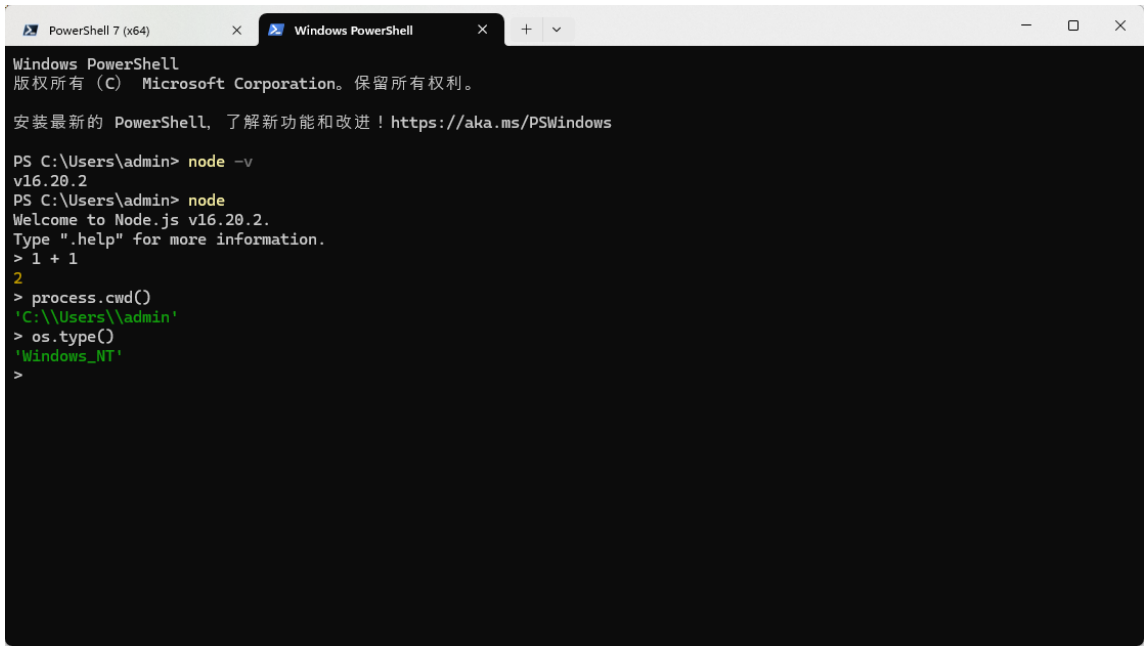


安装Node.js

可以直接在官网下载安装。

新版本的Node.js提供了nvm、fnm等工具，可以方便地管理多个Node.js版本。

在控制台输入`node -v`就可以看到是否安装成功了以及安装的node版本。



介绍npm

NPM (Node.js Package Manager) 是 Node.js 的包管理工具，它可以方便地安装、更新、卸载和管理开发中需要用到的各种包和模块。NPM 会随 Node.js 的安装一同被下载。

可以通过 npm -v 查看 npm 的版本信息。npm的版本通常独立于Node.js的版本，但是某些版本的npm可能需要特定版本的Node.js才能运行。

```
PS C:\Users\admin> npm -v
8.19.4
PS C:\Users\admin>
```

1. 配置国内镜像源

npm 默认的镜像源地址是`https://registry.npmjs.org/`，国内访问较慢，通常会使用淘宝开源的镜像站`https://registry.npmmirror.com/`。

可通过`npm config set registry https://registry.npmmirror.com/`手动切换镜像源；也可通过nrm、yrm等工具来管理。

```
yrm ls

  npm ---- https://registry.npmjs.org/
  cnpm --- http://r.cnpmjs.org/
* taobao - https://registry.npmmirror.com/
  nj ----- https://registry.nodejitsu.com/
  rednpm - http://registry.mirror.cqupt.edu.cn/
  npmMirror  https://skimdb.npmjs.com/registry/
  edunpm - http://registry.enpmjs.org/
  yarn --- https://registry.yarnpkg.com
```

2. npm常用指令

npm（Node Package Manager）是Node.js的包管理器，用于安装、卸载、更新和管理Node.js项目的依赖包。以下是一些常用的npm指令：

初始化项目

- `npm init`：创建一个新的`package.json`文件，用于记录项目元数据和依赖。
- `npm init -y`：快速初始化项目，使用默认值生成`package.json`文件。

安装依赖

- `npm install <package>`：安装指定的包到`node_modules`目录，并将其添加到`package.json`的`dependencies`列表中。
- `npm install <package> --save-dev`：将包添加到`devDependencies`列表中，用于开发阶段的依赖。
- `npm install <package> -g`：全局安装包，通常用于命令行工具。

更新依赖

- `npm update <package>`：更新指定的包到最新版本。
- `npm update`：更新所有已安装的包到它们各自的最新版本。

卸载依赖

- `npm uninstall <package>`：从`node_modules`目录和`package.json`文件中移除指定的包。

列出依赖

- `npm list`：显示当前项目的所有依赖。
- `npm list <package>`：显示指定包的信息。
- `npm ls -g`：列出全局安装的包。

检查依赖

- `npm outdated`：列出过时的包和它们的最新版本。
- `npm check`：验证`node_modules`中的包是否与`package.json`和`package-lock.json`文件中的条目匹配。

执行脚本

- `npm run <script>`：运行`package.json`中的预定义脚本。

清理缓存

- `npm cache clean --force`：清理npm缓存。

查看包信息

- `npm info/view <pkg>`：查看包的信息。

安装生产依赖

- `npm ci`：在CI/CD环境中使用，基于`package-lock.json`或`npm-shrinkwrap.json`文件安装依赖，不执行任何`package.json`中的脚本。

发布包

- `npm publish`：将本地包发布到npm仓库。

3. package.json

名字

- `name`: 指定项目的名称，通常遵循`@scope/project-name`格式，其中`scope`可选。

版本

- `version`: 定义项目当前版本，遵循语义化版本规则，例如`1.0.0`。

描述

- `description`: 简短描述项目功能或用途。

主文件

- `main`: 指定作为模块入口的主文件，默认为`index.js`。

脚本命令

- `scripts`: 定义一系列npm脚本命令，如`"start": "npm run dev"`。

依赖

- `dependencies`: 列出项目运行时所需的依赖包及其版本。

开发依赖

- `devDependencies`: 列出仅用于开发环境的依赖包，如测试框架和构建工具。

关键词

- `keywords`: 一组关键词，用于在npm上搜索此项目。

作者

- `author`: 项目作者的信息，可以是名字或邮箱。

许可证

- `license`: 项目使用的许可证类型，如MIT, GPL等。

关于version字段前面的符号，可以参考以下说明：

```
无符号：如果版本号前没有任何符号，那么这个依赖将被固定到确切的版本，例如 "vue": "2.6.11"。这意味着每次安装都将获取这个特定版本。

波浪号（~）：当版本号前有波浪号时，如 "autoprefixer": "~7.1.2"，这表示你可以接收与次要版本和补丁版本相关的更新，但不会升级到新的主要版本。

尖括号（^）：尖括号表示你可以接收与补丁版本相关的更新，并且还可以升级到新的次要版本，只要它们不改变主要版本号。例如，"autoprefixer": "^7.1.
```

这样做的目的是提高依赖包的兼容性。使用 ^ 或 ~ 符号，可以在不破坏 API 的情况下，获取到最新的修复和功能更新。

4. node_modules

这是Node.js项目中存放所有npm安装的依赖包的地方。每个依赖包都会有自己的目录，其中包含了该包的源代码、依赖项和其他元数据。

- **依赖管理**：自动解析`package.json`中的依赖树，并下载和安装所有必要的包。
- **隔离环境**：通过层级结构避免不同项目间的依赖冲突，每个项目都有自己的`node_modules`目录。
- **缓存机制**：首次安装的包会被缓存，后续相同包的安装会从缓存中读取，提高安装速度。

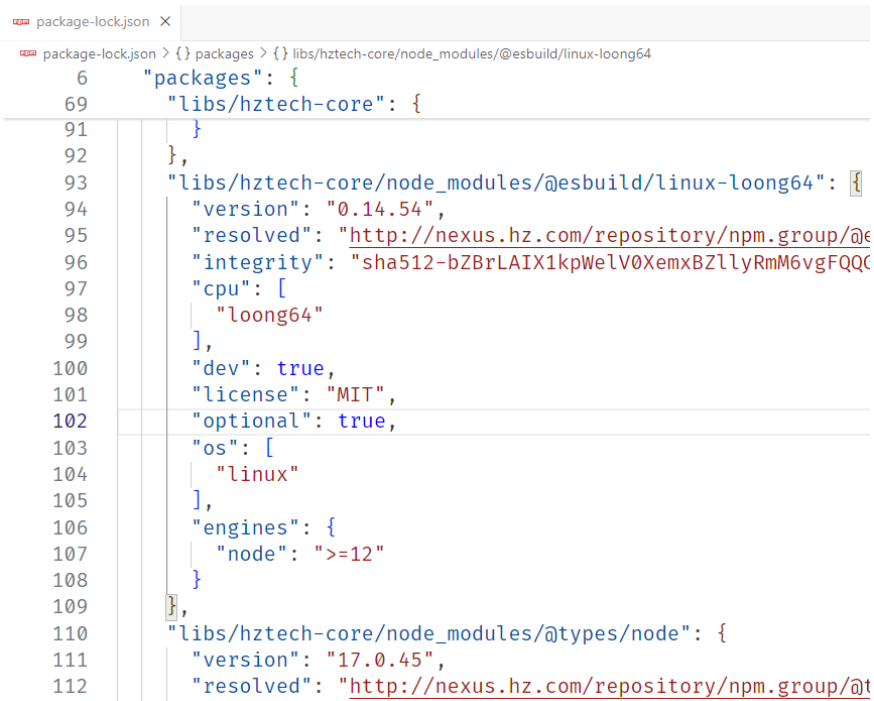
使用注意事项

- **避免提交到版本库**：由于`node_modules`目录通常非常大，包含大量文件，因此不应将其添加到版本控制系统中。相反，应提交`package.json`和`package-lock.json`或`yarn.lock`，以便其他开发者可以重现相同的依赖环境。
- **性能优化**：大型的`node_modules`目录可能会影响构建和部署时间，可以通过减少不必要的依赖或使用`npm prune`命令来清理未使用的包以优化性能。
- **安全检查**：定期使用如`npm audit`等工具检查`node_modules`中的依赖是否存在已知的安全漏洞。

5. package-lock.json

主要用于锁定项目依赖的版本号，以确保在不同的机器和环境中安装相同的依赖和版本。

拿基线版移动端项目来看。



`version`：实际版本号
`resolved`：包下载的地址
`cpu`：当前包支持的 CPU 架构
`dev`：是否为开发依赖
`license`：当前包的许可证类型
`optional`：是否为可选依赖
`os`：当前包支持的操作系统
`integrity`：用于安全校验的哈希值，确保下载的包与发布的包一致
`dependencies`：当前包依赖的所有包的版本信息
`engines`：当前包支持的 Node.js 版本

6. .npmrc 文件

.npmrc 文件是 NPM 的配置文件，它包含了一些 NPM 的配置信息，比如代理、镜像、命令别名等。通过修改 .npmrc 文件，可以更改 NPM 的默认行为。

例如修改 registry 配置，让项目协作同学不用主动设置镜像源的地址，也能和自己保持一致。

```
# .npmrc
registry=https://registry.npmjs.org/
```

7. npx

npx 是随 Node.js 安装附带的另一个指令，可以更方便的调用 Node.js 生态中的包 (通常是一些 Node CLI 工具)，使用 npx，可以在不全局安装一个命令行工具的情况下直接运行它，同时也不会污染全局环境。

比如 nodemon 这个 CLI 工具，可以在开发时替代 node 指令执行 js 文件，文件修改后自动重新执行。

```
# 直接通过npx 调用执行
npx nodemon test.js

# 等价于
# ① 全局安装CLI工具
npm i -g nodemon
# ② 调用执行
nodemon test.js
```

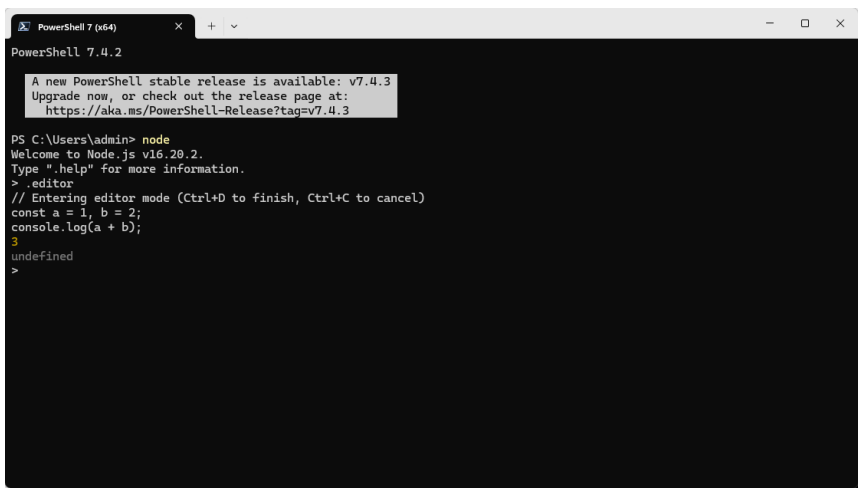
运行Node.js

有几种常用的运行方式。

1. REPL 环境

全称是 Read-Eval-Print Loop，它是一种编程语言交互式解释器的实现方式。

在终端里我们输入 node 即可进入。默认是单行输入，当然也可以输入 .editor，进行多行编辑。



2. 命令行执行

通过`node`命令直接执行脚本文件，如`node test.js`，`curl https://script.sugarat.top/js/tests/test.js | node`。

3. 直接执行

`Hashbang（也称为 shebang）是一种特殊的注释，以 #! 开头，通常出现在可执行的脚本文件的第一行，用于告诉系统要使用哪个解释器来执行该脚本文件。`

```
#!/usr/bin/env node

const hello = 'hello world'
console.log(hello)
```

然后在命令行执行`chmod +x test.js`，然后执行`./test.js`即可。