

Node.js 入门

JavaScript 运行时环境



黑马程序员
www.itheima.com

传智教育旗下
高端IT教育品牌

什么是 Node.js?

定义:

Node.js

Node.js 是一个跨平台 JavaScript 运行环境，使开发者可以搭建服务器端的 JavaScript 应用程序。

作用：使用 Node.js 编写服务器端程序

- ✓ 编写数据接口，提供网页资源浏览功能等等
- ✓ **前端工程化**：为后续学习 Vue 和 React 等框架做铺垫

什么是前端工程化?

前端工程化：开发项目直到上线，过程中集成的所有**工具和技术**

Node.js 是前端工程化的基础（因为 Node.js 可以主动读取前端代码内容）



Node.js 为何能执行 JS?

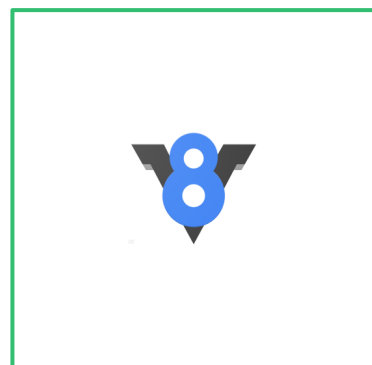
首先：浏览器能执行 JS 代码，依靠的是内核中的 **V8 引擎** (C++ 程序)

其次：Node.js 是基于 Chrome V8 引擎进行封装 (运行环境)

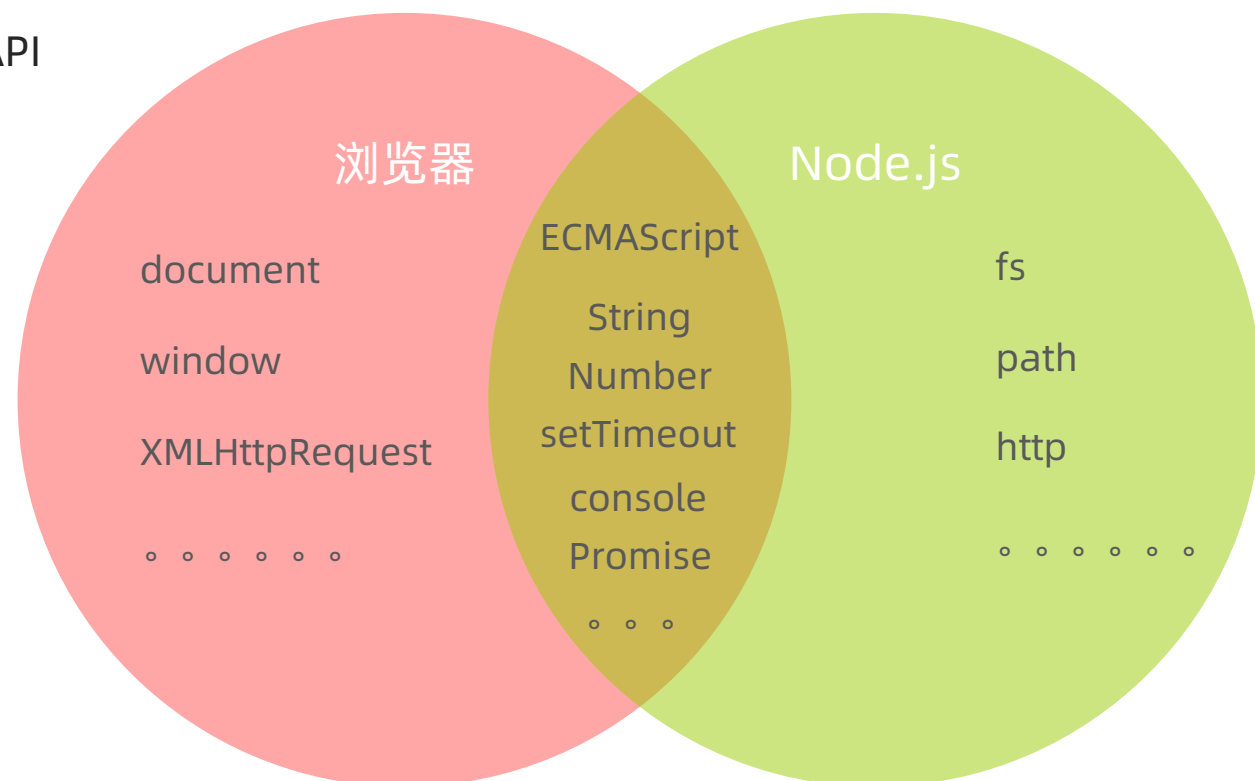
区别：都支持 ECMAScript 标准语法，Node.js 有独立的 API



浏览器



Node.js



注意：Node.js 环境没有 DOM 和 BOM 等

Node.js 安装

要求：下载 node-v16.19.0.msi 安装程序（指定版本：兼容 vue-admin-template 模板）

安装过程：默认下一步即可

注释事项：

1. 安装在非中文路径下
2. 无需勾选自动安装其他配套软件

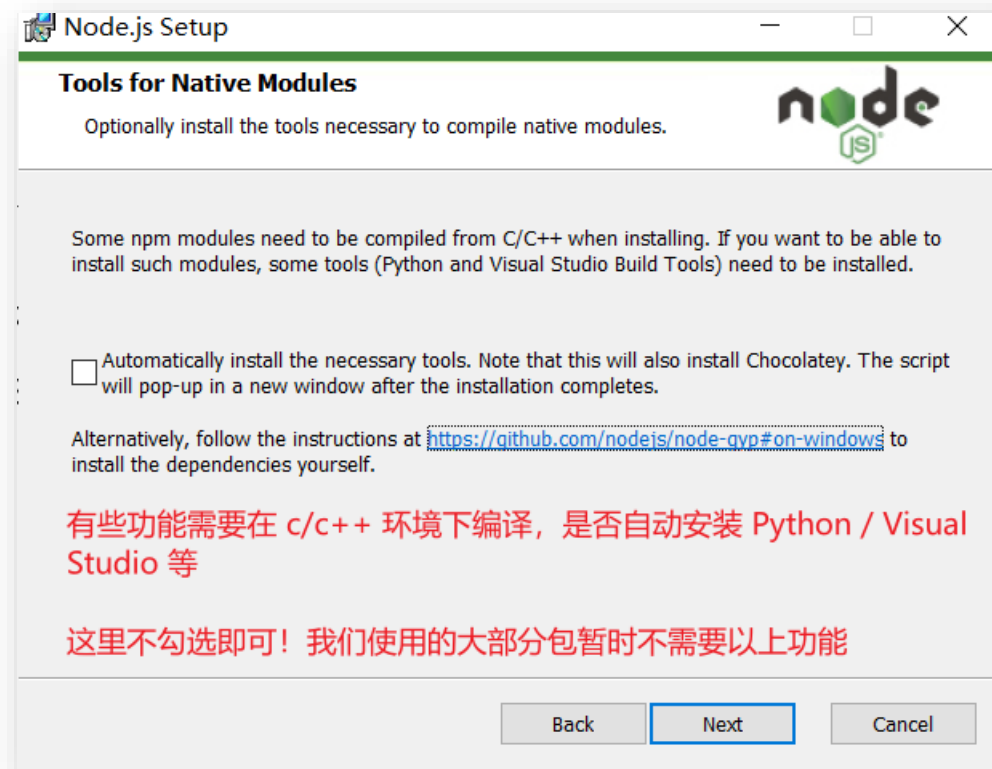
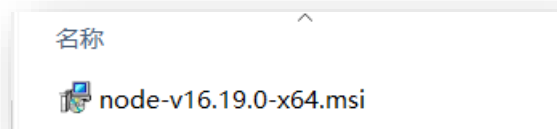
成功验证：

1. 打开 cmd 终端，输入 `node -v` 命令查看版本号
2. 如果有显示，则代表安装成功

```
命令提示符
Microsoft Windows [版本 10.0.19044.2604]
(c) Microsoft Corporation。保留所有权利。

C:\Users\lenovo>node -v
v16.19.0

C:\Users\lenovo>
```



使用 Node.js

需求：新建 JS 文件，并编写代码后，在 node 环境下执行

命令：在 VSCode 集成终端中，输入 `node xxx.js`，回车即可执行

```
console.log('Hello, World')
for (let i = 0; i < 3; i++) {
  console.log(6)
}
```

问题 输出 终端 调试控制台

```
D:\备课代码\3-B站课程\03_Node.js与Webpack\03-code>node 01.js
Hello, Node.js
0
1
2

D:\备课代码\3-B站课程\03_Node.js与Webpack\03-code>
```



总结

1. Node.js 是什么?
 - 基于 Chrome 的 **V8 引擎** 封装，独立执行 JavaScript 代码的环境
2. Node.js 与浏览器环境的 JS 最大区别?
 - Node.js 环境中**没有 BOM 和 DOM**
3. Node.js 有什么用?
 - 编写后端程序：提供数据和网页资源等
 - **前端工程化**：集成各种开发中使用的工具和技术
4. Node.js 如何执行代码?
 - 在 VSCode 终端中输入：**node xxx.js** 回车即可执行（注意路径）

fs 模块 - 读写文件

模块：类似插件，封装了方法/属性

fs 模块：封装了与本机文件系统进行交互的，方法/属性

语法：

1. 加载 fs 模块对象
2. 写入文件内容
3. 读取文件内容

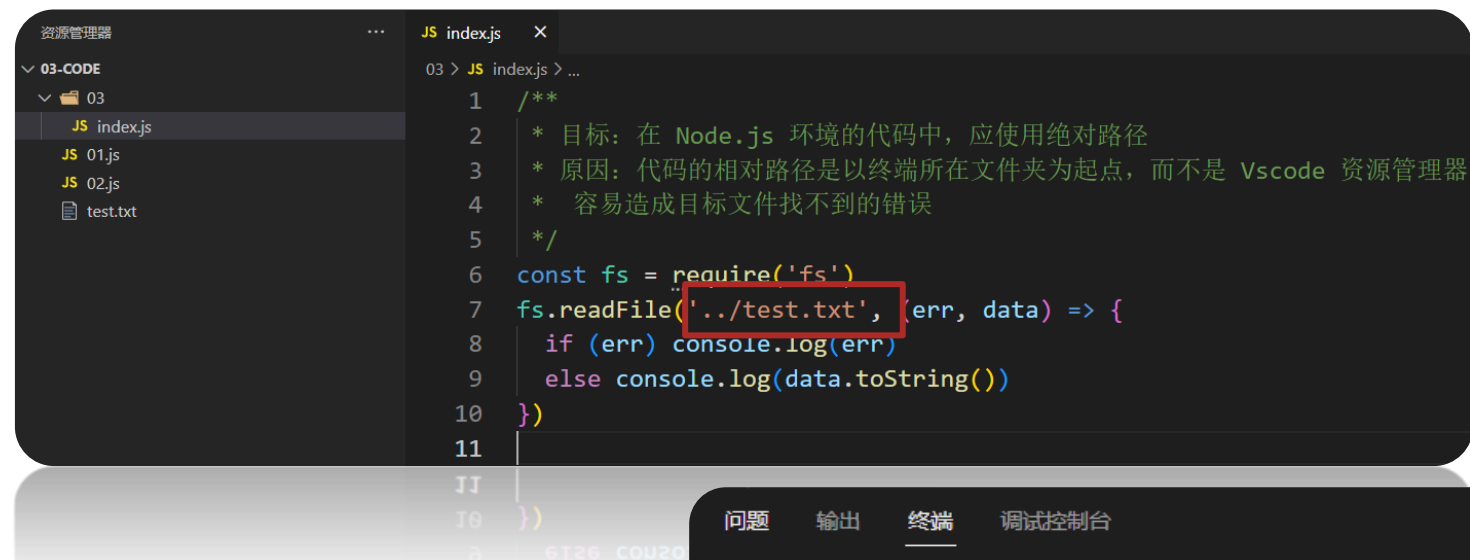
```
const fs = require('fs') // fs 是模块标识符：模块的名字
```

```
fs.writeFile('文件路径', '写入内容', err => {  
  // 写入后的回调函数  
})
```

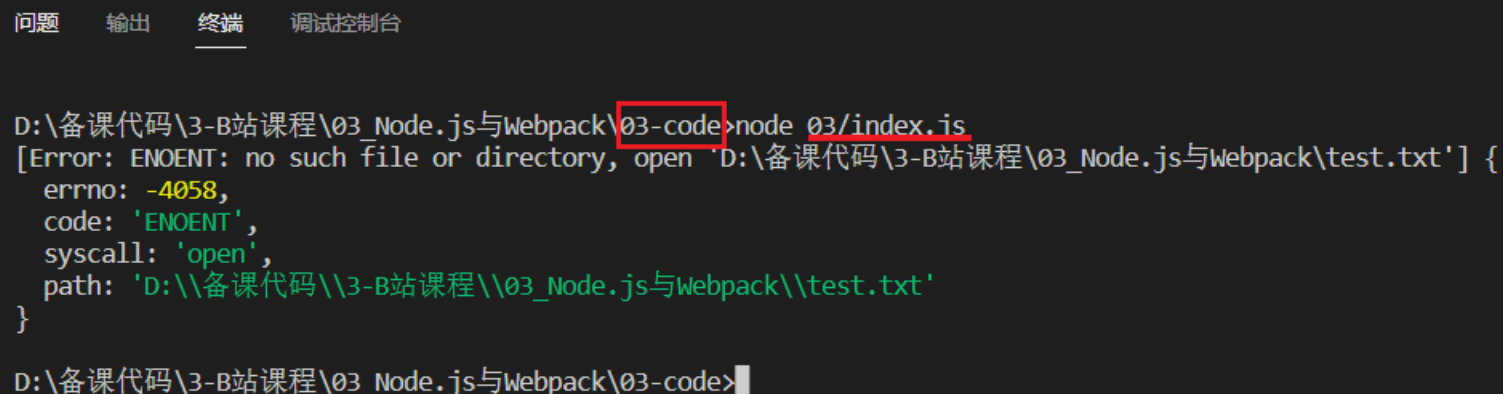
```
fs.readFile('文件路径', (err, data) => {  
  // 读取后的回调函数  
  // data 是文件内容的 Buffer 数据流  
})
```


path 模块 - 路径处理

问题：Node.js 代码中，相对路径是根据终端所在路径来查找的，可能无法找到你想要的文件



```
1  /**
2   * 目标：在 Node.js 环境的代码中，应使用绝对路径
3   * 原因：代码的相对路径是以终端所在文件夹为起点，而不是 Vscode 资源管理器
4   * 容易造成目标文件找不到的错误
5   */
6  const fs = require('fs')
7  fs.readFile('./test.txt', (err, data) => {
8    if (err) console.log(err)
9    else console.log(data.toString())
10 })
11
```



```
问题 输出 终端 调试控制台
D:\备课代码\3-B站课程\03_Node.js与Webpack\03-code>node 03/index.js
[Error: ENOENT: no such file or directory, open 'D:\备课代码\3-B站课程\03_Node.js与Webpack\test.txt'] {
  errno: -4058,
  code: 'ENOENT',
  syscall: 'open',
  path: 'D:\\备课代码\\3-B站课程\\03_Node.js与Webpack\\test.txt'
}
D:\备课代码\3-B站课程\03_Node.js与Webpack\03-code>
```

path 模块 - 路径处理

建议：在 Node.js 代码中，使用**绝对路径**

补充：__dirname 内置变量（获取当前模块目录-绝对路径）

- ✓ windows: D:\备课代码\3-B站课程\03_Node.js与Webpack\03-code\03
- ✓ mac: /Users/xxx/Desktop/备课代码/3-B站课程/03_Node.js与Webpack/03-code/03

注意：path.join() 会使用特定于平台的分隔符，作为定界符，将所有给定的路径片段连接在一起

语法：

1. 加载 path 模块
2. 使用 path.join 方法，拼接路径

```
path.join('03', 'dist/js', 'index.js')  
// windows: '03\\dist\\js\\index.js'  
// mac: '03/dist/js/index.js'
```

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

```
path.join('路径1', '路径2', ...)
```

案例 - 压缩前端 html

需求：把 回车符 (\r) 和换行符 (\n) 去掉后，写入到新 html 文件中

步骤：

1. 读取源 html 文件内容
2. 正则替换字符串
3. 写入到新的 html 文件中

```
04 > public > index.html > html
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4    <meta charset="UTF-8">
5    <meta name="viewport" content="width=device-width, initial-scale=1.0">
6    <meta http-equiv="X-UA-Compatible" content="ie=edge">
7    <title>时钟案例</title>
8    <style>
9      html,
10     body { ...
15     }
16     .box { ...
36     }
37   </style>
38 </head>
39 <body>
40   <div class="box">
41     <div id="HH">00</div>
42     <div>:</div>
43     <div id="mm">00</div>
44     <div>:</div>
45     <div id="ss">00</div>
46   </div>
47 </body>
48 </html>
```



```
04 > dist > index.html > html
1  <!DOCTYPE html><html lang="en"><head> <meta charset="UTF-8"> <meta name="viewport"
content="width=device-width, initial-scale=1.0"> <meta http-equiv="X-UA-Compatible"
content="ie=edge"> <title>时钟案例</title> <style> html, body { margin: 0;
padding: 0; height: 100%; background-image: linear-gradient(to bottom right, red,
gold); } .box { width: 400px; height: 250px; background-color: rgba(255,
255, 255, 0.6); border-radius: 6px; position: absolute; left: 50%; top:
40%; transform: translate(-50%, -50%); box-shadow: 1px 1px 10px #fff; text-shadow:
0px 1px 30px white; display: flex; justify-content: space-around; align-items:
center; font-size: 70px; user-select: none; padding: 0 20px;
-webkit-box-reflect: below 0px -webkit-gradient(linear, left top, left bottom, from(transparent),
color-stop(0%, transparent), to(rgba(250, 250, 250, .2))); } </style></head><body> <div
class="box"> <div id="HH">00</div> <div>:</div> <div id="mm">00</div> <div>:</div>
<div id="ss">00</div> </div></body></html> 5.3333rem, 3.3333rem, 0.08rem, 0.0133rem, 0.0133rem, 0.
```

URL 中的端口号

URL：统一资源定位符，简称网址，用于访问服务器里的资源

端口号：标记服务器里不同功能的**服务程序**

端口号范围：0-65535 之间的任意整数



钥匙



地址



房间号



物品位置

<http://hmajax.itheima.net:80/api/province>

注意：http 协议，**默认**访问 **80** 端口

常见的服务程序

Web 服务程序：用于提供网上信息浏览功能

注意：0-1023 和一些特定端口号被占用，我们自己编写服务程序请避开使用





总结

1. 端口号的作用?
 - 标记区分服务器里不同的服务程序
2. 什么是 Web 服务程序?
 - 提供网上信息浏览的程序代码

http 模块-创建 Web 服务

需求：创建 Web 服务并响应内容给浏览器

步骤：

1. 加载 **http 模块**，创建 Web 服务对象
2. 监听 **request** 请求事件，设置响应头和响应体
3. 配置**端口号**并**启动** Web 服务
4. 浏览器请求 **http://localhost:3000** 测试

(localhost: 固定代表本机的域名)

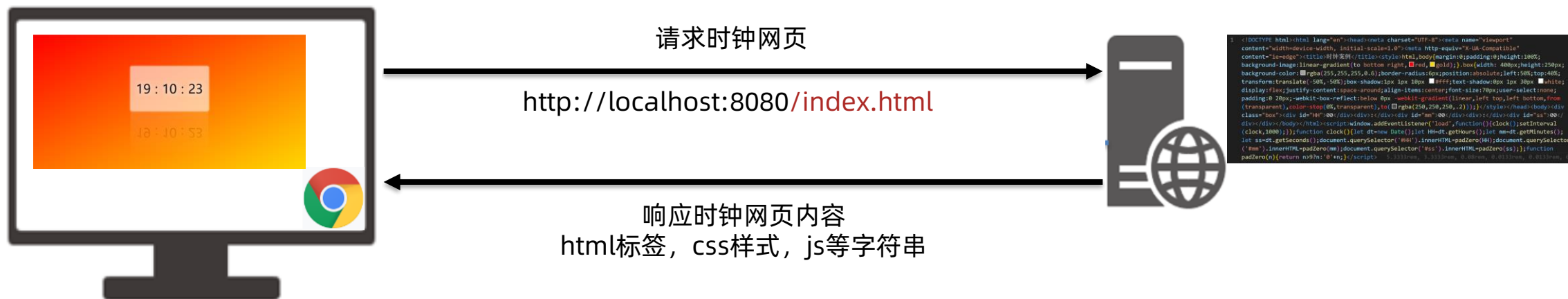
```
const http = require('http')
const server = http.createServer()

server.on('request', (req, res) => {
  // 设置响应头：内容类型，普通文本；编码格式为 utf-8
  res.setHeader('Content-Type', 'text/plain;charset=utf-8')
  res.end('您好，欢迎使用 node.js 创建的 Web 服务')
})

server.listen(3000, () => {
  console.log('Web 服务已经启动')
})
```

案例 浏览时钟

需求：基于 Web 服务，开发提供网页资源的功能



案例 浏览时钟

步骤:

1. 基于 http 模块，创建 Web 服务
2. 使用 `req.url` 获取请求资源路径，判断并读取 `index.html` 里字符串内容返回给请求方
3. 其他路径，暂时返回不存在的提示
4. 运行 Web 服务，用浏览器发起请求测试

```
server.on('request', (req, res) => {  
  if (req.url === '/index.html') {  
    fs.readFile(path.join(__dirname, 'dist/index.html'), (err, data) => {  
      res.setHeader('Content-Type', 'text/html; charset=utf-8')  
      res.end(data.toString())  
    })  
  } else {  
    res.setHeader('Content-Type', 'text/html; charset=utf-8')  
    res.end('你要访问的路径不存在')  
  }  
})
```

Node.js 模块化



黑马程序员
www.itheima.com

传智教育旗下
高端IT教育品牌

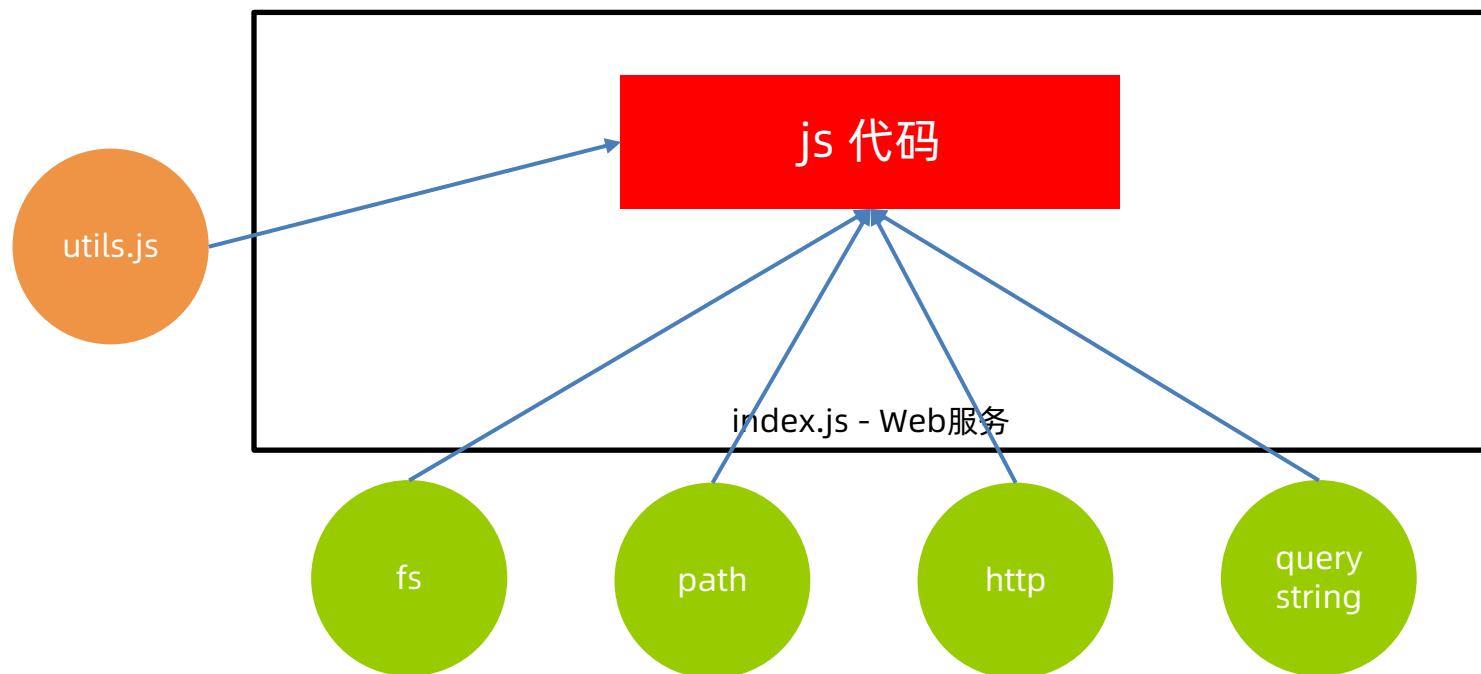
什么是模块化?

定义: CommonJS 模块是为 Node.js 打包 JavaScript 代码的原始方式。Node.js 还支持浏览器和其他 JavaScript 运行时使用的 **ECMAScript 模块标准**。
在 Node.js 中，每个文件都被视为一个单独的模块。

概念：项目是由很多个模块文件组成的

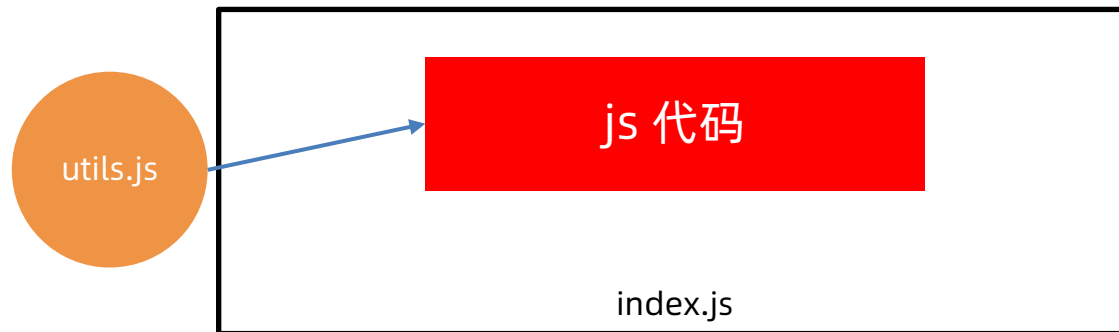
好处：提高代码复用性，按需加载，**独立作用域**

使用：需要标准语法**导出**和**导入**进行使用



CommonJS 标准

需求：定义 utils.js 模块，封装基地址和求数组总和的函数



使用：

1. 导出：`module.exports = {}`
2. 导入：`require('模块名或路径')`

模块名或路径：

- ✓ 内置模块：直接写名字（例如：fs, path, http）
- ✓ 自定义模块：写模块文件路径（例如：./utils.js）

```
const baseUrl = 'http://hmajax.itheima.net'
const getArraySum = arr => arr.reduce((sum, val) => sum += val, 0)

module.exports = {
  对外属性名1: baseUrl,
  对外属性名2: getArraySum
}
```

```
const obj = require('模块名或路径')
// obj 就等于 module.exports 导出的对象
```



总结

1. Node.js 中什么是模块化?

- 每个文件都是独立的模块

2. 模块之间如何联系呢?

- 使用特定语法，导出和导入使用

3. CommonJS 标准规定如何导出和导入模块呢?

- 导出: `module.exports = {}`
- 导入: `require('模块名或路径')`

4. 模块名/路径如何选择?

- 内置模块，直接写名字。例如: `fs`, `path`, `http`等
- 自定义模块，写模块文件路径。例如: `./utils.js`

ECMAScript 标准 - 默认导出和导入

需求：封装并导出基地址和求数组元素和的函数

默认标准使用：

1. 导出：`export default {}`
2. 导入：`import 变量名 from '模块名或路径'`

注意：Node.js 默认支持 CommonJS 标准语法

如需使用 ECMAScript 标准语法，在运行模块所在文件夹新建 `package.json` 文件，并设置 `{"type": "module"}`

```
const baseUrl = 'http://hmajax.itheima.net'
const getArraySum = arr => arr.reduce((sum, val) => sum += val, 0)

export default {
  对外属性名1: baseUrl,
  对外属性名2: getArraySum
}
```

```
import obj from '模块名或路径'
// obj 就等于 export default 导出的对象
```

```
package.json ×
D: > 备课代码 > 2_node_3天 > Node_代码 > Day03_we
1 { "type": "module" }
```



总结

1. ECMAScript 标准规定如何默认导出和导入模块呢?
 - 导出: `export default {}`
 - 导入: `import 变量名 from '模块名或路径'`
2. 如何让 Node.js 切换模块标准为 ECMAScript?
 - 运行模块所在文件夹，新建 package.json 并设置
 - `{"type": "module"}`

ECMAScript 标准 - 命名导出和导入

需求：封装并导出基地址和求数组元素和的函数

命名标准使用：

1. 导出：`export` 修饰定义语句
2. 导入：`import { 同名变量 } from '模块名或路径'`

```
export const baseUrl = 'http://hmajax.itheima.net'  
export const getArraySum = arr => arr.reduce((sum, val) => sum += val, 0)
```

```
import { baseUrl, getArraySum } from '模块名或路径'  
// baseUrl 和 getArraySum 是变量，值为模块内命名导出的同名变量的值
```

如何选择：

按需加载，使用命名导出和导入

全部加载，使用默认导出和导入



总结

1. Node.js 支持哪 2 种模块化标准?
 - CommonJS 标准语法（默认）
 - ECMAScript 标准语法
2. ECMAScript 标准，命名导出和导入的语法?
 - 导出：export 修饰定义的语句
 - 导入：import { 同名变量 } from '模块名或路径'
3. ECMAScript 标准，默认导出和导入的语法?
 - 导出：export default {}
 - 导入：import 变量名 from '模块名或路径'

包的概念

包：将模块，代码，其他资料聚合成一个文件夹

包分类：

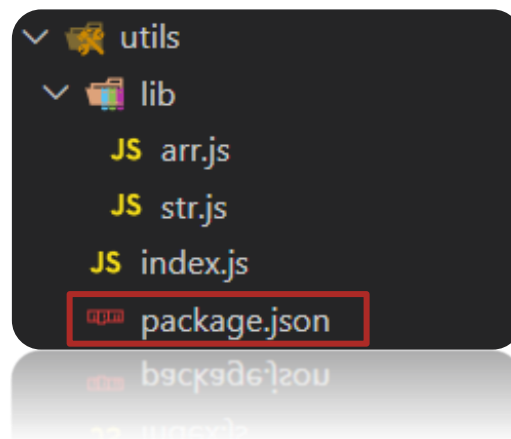
- ✓ 项目包：主要用于编写项目和业务逻辑
- ✓ 软件包：封装工具和方法进行使用

要求：根目录中，必须有 package.json 文件（记录包的清单信息）

注意：导入软件包时，引入的默认是 index.js 模块文件 / main 属性指定的模块文件

需求：封装数组求和函数的模块，判断用户名和密码长度函数的模块，形成成一个软件包

```
package.json > ...
1  {
2    "name": "cz_utils",    软件包名称
3    "version": "1.0.0",   软件包当前版本
4    "description": "一个数组和字符串常用工具方法的包",
5    "main": "index.js",   软件包入口点          软件包简短描述
6    "author": "itheima",  软件包作者
7    "license": "MIT"      软件包许可证（商用后可以用作者名字宣传）
8  }
```





总结

1. 什么是包?

- 将模块，代码，其他资料聚合成的**文件夹**

2. 包分为哪 2 类呢?

- 项目包：编写项目代码的文件夹
- **软件包**：封装工具和方法供开发者使用

3. package.json 文件的作用?

- 记录**软件包的名字**，作者，**入口**文件等信息

4. 导入一个包文件夹的时候，导入的是哪个文件?

- **默认 index.js 文件**，或者 main 属性指定的文件

npm - 软件包管理器

定义： npm 简介

npm 是 Node.js 标准的软件包管理器。

在 2017 年 1 月时，npm 仓库中就有超过 350000 个软件包，这使其成为世界上最大的单一语言代码仓库，并且可以确定几乎有可用于一切的软件包。

它起初是作为下载和管理 Node.js 包依赖的方式，但其现在也已成为前端 JavaScript 中使用的工具。

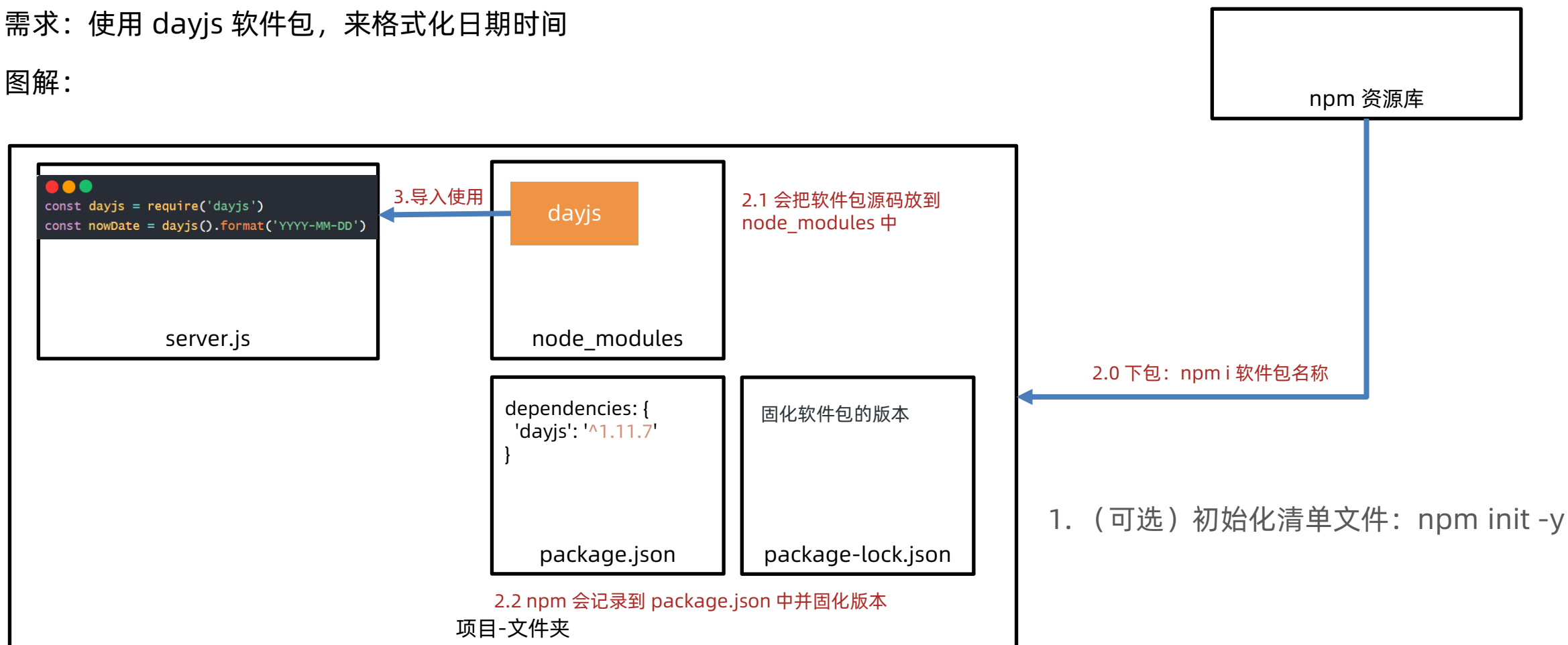
使用：

1. 初始化清单文件：npm init -y（得到 package.json 文件，有则略过此命令）
2. 下载软件包：npm i 软件包名称
3. 使用软件包

npm - 软件包管理器

需求：使用 dayjs 软件包，来格式化日期时间

图解：





总结

1. npm 软件包管理器作用?
 - 下载软件包以及管理版本
2. 初始化项目清单文件 package.json 命令?
 - `npm init -y`
3. 下载软件包的命令?
 - `npm i 软件包名字`
4. 下载的包会存放在哪里?
 - 当前项目下的 `node_modules` 中，并记录在 package.json 中

npm - 安装所有依赖

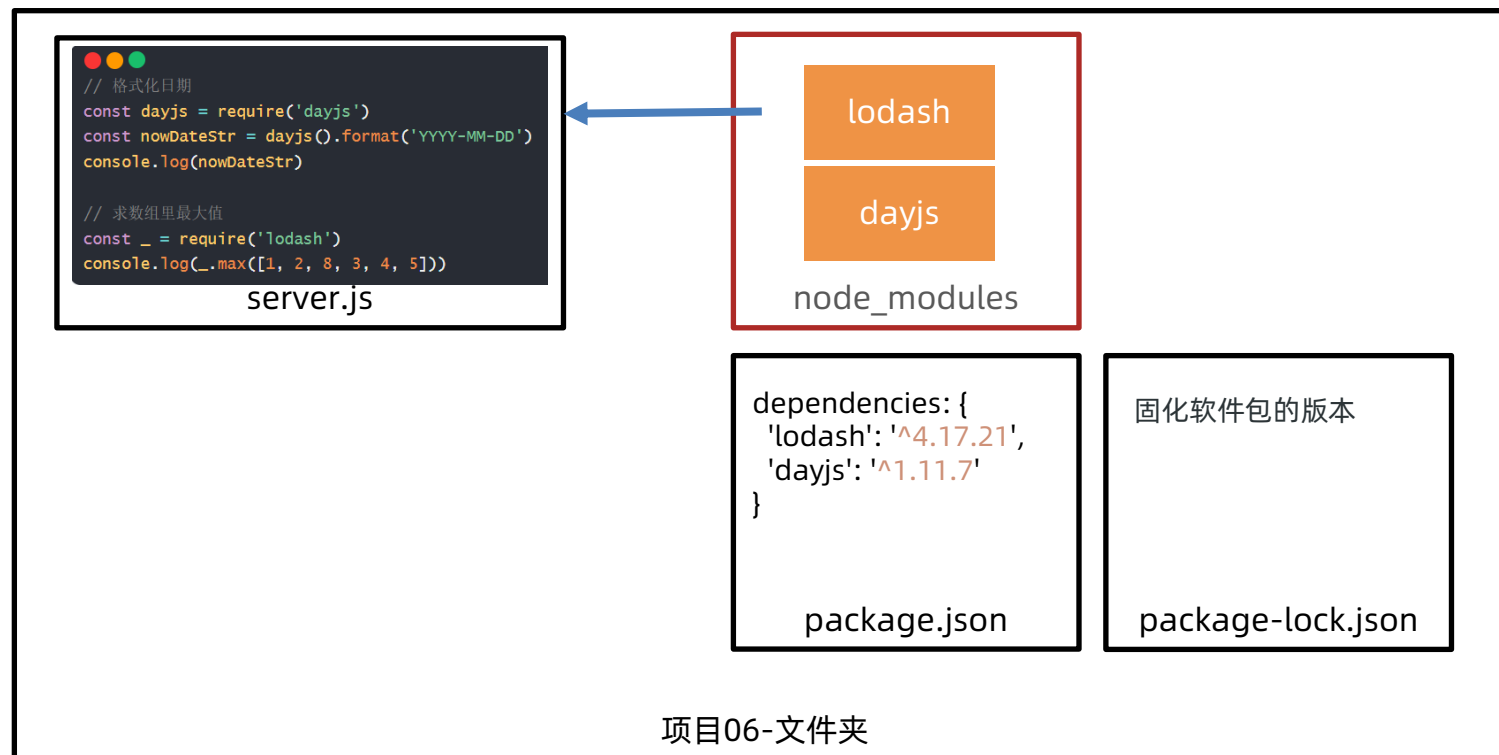
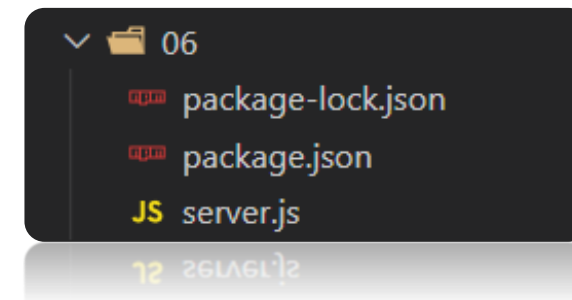
问题：项目中不包含 node_modules，能否正常运行？

答案：不能，缺少依赖的本地软件包

原因：因为，自己用 npm 下载依赖比磁盘传递拷贝要快得多

解决：项目终端输入命令：**npm i**

下载 package.json 中记录的所有软件包





总结

1. 当项目中只有 package.json 没有 node_modules 怎么办?
 - 当前项目下，执行 `npm i` 安装所有依赖软件包
2. 为什么 node_modules 不进行传递?
 - 因为用 npm 下载比磁盘传递要快

npm - 全局软件包 nodemon

软件包区别：

- 本地软件包：当前项目内使用，封装属性和方法，存在于 node_modules
- 全局软件包：本机所有项目使用，封装命令和工具，存在于系统设置的位置

nodemon 作用：替代 node 命令，检测代码更改，自动重启程序

使用：

1. 安装：npm i nodemon -g (-g 代表安装到全局环境中)
2. 运行：nodemon 待执行的目标 js 文件

需求：启动准备好的项目，修改代码保存后，观察自动重启应用程序



总结

1. 本地软件包和全局软件包区别?

- 本地软件包，作用在**当前项目**，**封装属性和方法**
- 全局软件包，**本机**所有项目使用，**封装命令和工具**

2. nodemon 作用?

- 替代 node 命令，检测代码更改，**自动重启**程序

3. nodemon 怎么用?

- 先确保安装 `npm i nodemon -g`
- 使用 **nodemon** 执行目标 **js 文件**

Node.js 总结

Node.js 模块化：

概念：每个文件当做一个模块，独立作用域，按需加载

使用：采用特定的标准语法导出和导入进行使用

CommonJS 标准

	导出	导入
语法	<code>module.exports = {}</code>	<code>require('模块名或路径')</code>

ECMAScript 标准

	导出	导入
默认	<code>export default {}</code>	<code>import 变量名 from '模块名或路径'</code>
命名	<code>export 修饰定义语句</code>	<code>import { 同名变量 } from '模块名或路径'</code>

CommonJS 标准：一般应用在 Node.js 项目环境中

ECMAScript 标准：一般应用在前端工程化项目中

Node.js 总结

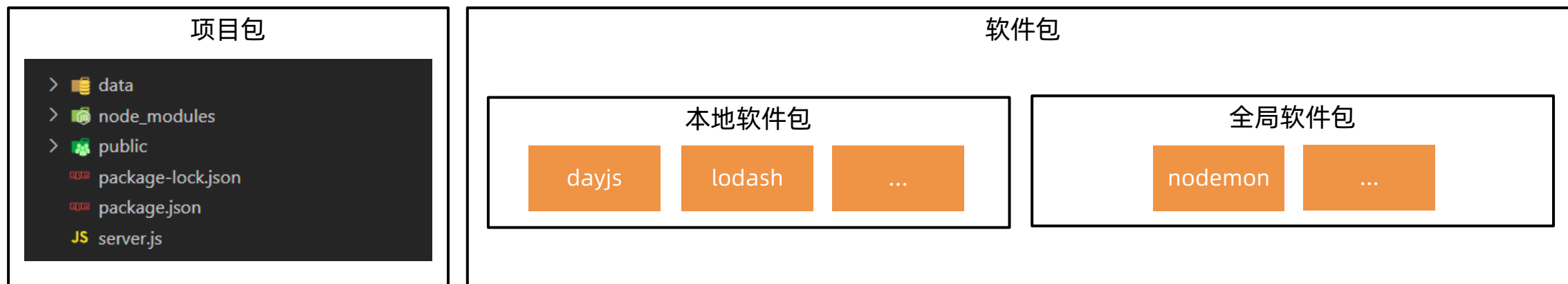
Node.js 包：

概念：把模块文件，代码文件，其他资料聚合成一个文件夹

项目包：编写项目需求和**业务逻辑**的文件夹

软件包：**封装工具和方法**进行使用的文件夹（一般使用 npm 管理）

- ✓ 本地软件包：作用在**当前**项目，一般封装的**属性/方法**，供项目调用编写业务需求
- ✓ 全局软件包：作用在**所有**项目，一般封装的**命令/工具**，支撑项目运行



Node.js 总结

常用命令：

功能	命令
执行 js 文件	<code>node xxx</code>
初始化 package.json	<code>npm init -y</code>
下载本地软件包	<code>npm i 软件包名</code>
下载全局软件包	<code>npm i 软件包名 -g</code>
删除软件包	<code>npm uni 软件包名</code>

Webpack



黑马程序员
www.itheima.com

传智教育旗下
高端IT教育品牌

什么是 Webpack?

定义:

本质上，webpack 是一个用于现代 JavaScript 应用程序的 静态模块打包工具。当 webpack 处理应用程序时，它会在内部从一个或多个入口点构建一个 依赖图(dependency graph)，然后将你项目中所需的每一个模块组合成一个或多个 *bundles*，它们均为静态资源，用于展示你的内容。

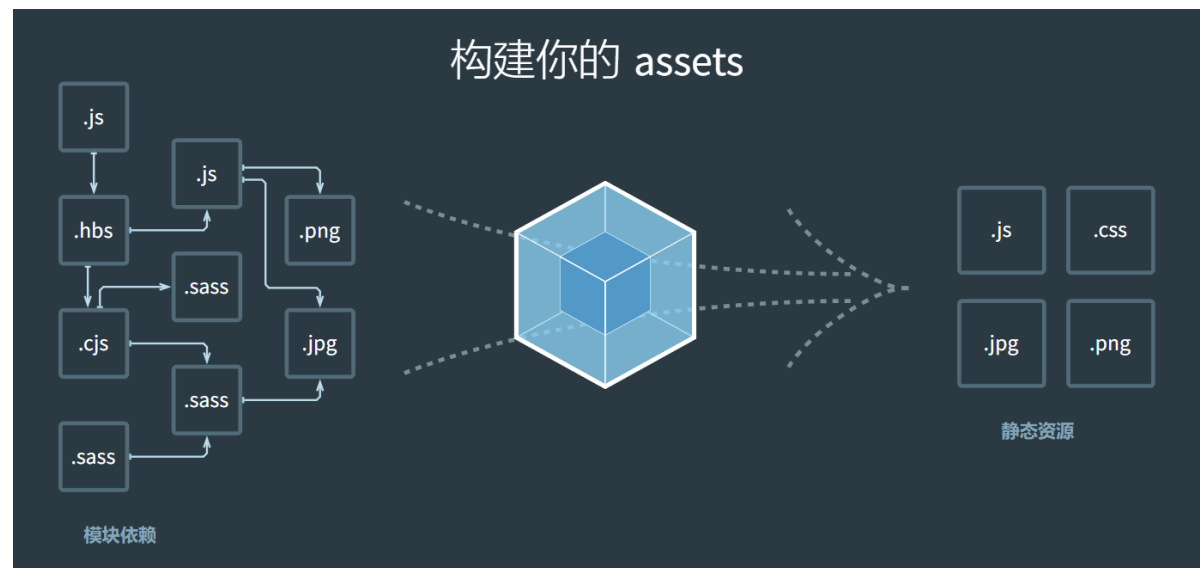
静态模块：指的是编写代码过程中的，html，css，js，图片等固定内容的文件

打包：把静态模块内容，压缩，整合，转译等（前端工程化）

- ✓ 把 less / sass 转成 css 代码
- ✓ 把 ES6+ 降级成 ES5
- ✓ 支持多种模块标准语法

问题：为何不学 vite ?

因为：很多项目还是基于 Webpack 构建，并为 Vue
React 脚手架使用做铺垫！



使用 Webpack

需求：封装 utils 包，校验手机号长度和验证码长度，在 `src/index.js` 中使用并打包观察

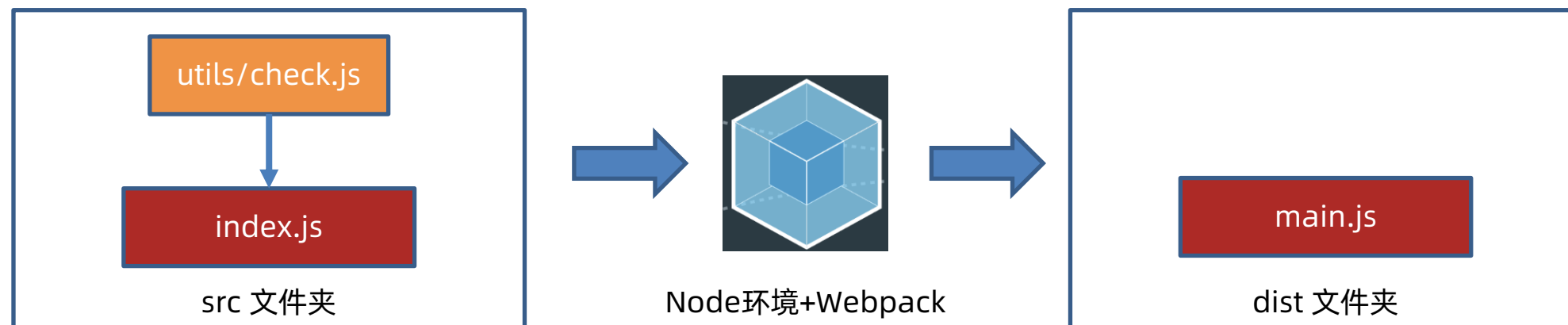
步骤：

1. 新建并初始化项目，编写业务源代码
2. 下载 webpack webpack-cli 到当前项目中（版本独立），并配置局部自定义命令
3. 运行打包命令，自动产生 dist 分发文件夹（压缩和优化后，用于最终运行的代码）

```
npm i webpack webpack-cli --save-dev
```

```
"scripts": {  
  "build": "webpack"  
},
```

```
npm run build
```



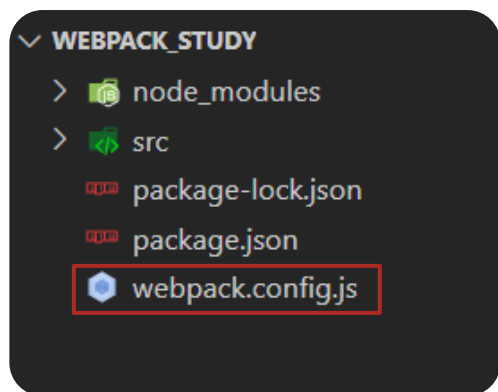
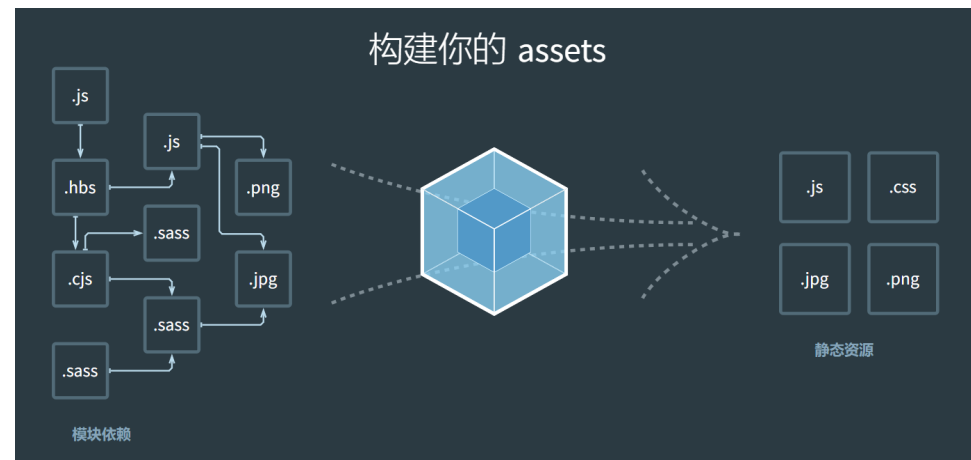
修改 Webpack 打包入口和出口

Webpack 配置: 影响 Webpack 打包过程和结果

步骤:

1. 项目根目录，新建 **webpack.config.js** 配置文件
2. 导出**配置对象**，配置入口，出口文件的路径
3. 重新打包观察

注意：只有和入口产生直接/间接的引入关系，才会被打包



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

module.exports = {
  entry: path.resolve(__dirname, 'src/login/index.js'),
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: './login/index.js'
  }
}
```

案例

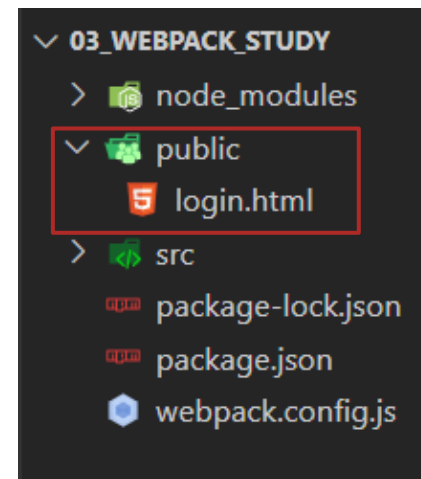
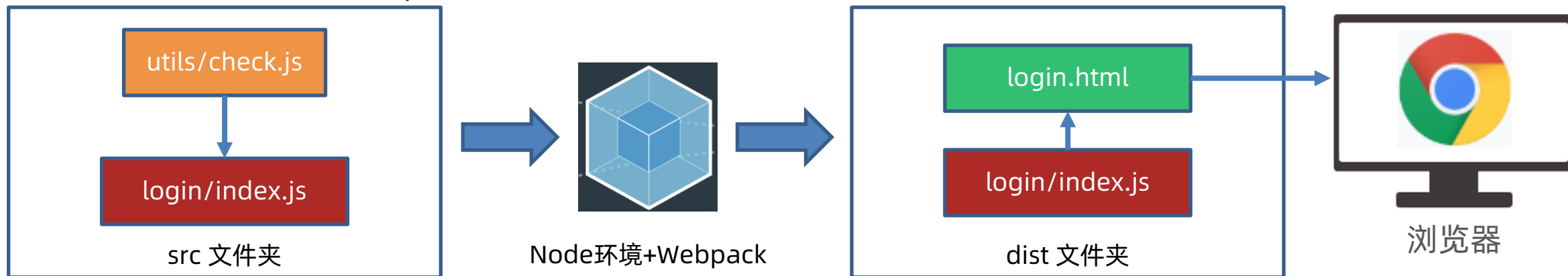
用户登录 - 长度判断

需求：点击登录按钮，判断手机号和验证码长度

步骤：

1. 准备用户登录页面
2. 编写核心 JS 逻辑代码
3. 打包并手动复制网页到 dist 下，引入打包后的 js，运行

核心：Webpack 打包后的代码，作用在前端网页中使用



自动生成 html 文件

插件 `html-webpack-plugin`: 在 Webpack 打包时生成 html 文件

步骤:

1. 下载 `html-webpack-plugin` 本地软件包
2. 配置 `webpack.config.js` 让 Webpack 拥有插件功能
3. 重新打包观察效果

```
npm i html-webpack-plugin --save-dev
```

```
// ...  
const HtmlWebpackPlugin = require('html-webpack-plugin')  
  
module.exports = {  
  // ...  
  plugins: [  
    new HtmlWebpackPlugin({  
      template: './public/login.html', // 模板文件  
      filename: './login/index.html' // 输出文件  
    })  
  ]  
}
```

打包 css 代码

[加载器 css-loader](#): 解析 css 代码

[加载器 style-loader](#): 把解析后的 css 代码插入到 DOM

步骤:

1. 准备 css 文件代码引入到 src/login/index.js 中（压缩转译处理等）
2. 下载 css-loader 和 style-loader 本地软件包
3. 配置 webpack.config.js 让 Webpack 拥有该加载器功能
4. 打包后观察效果

注意: Webpack 默认只识别 js 代码

```
npm i css-loader style-loader --save-dev
```

```
// ...  
  
module.exports = {  
  // ...  
  module: {  
    rules: [  
      {  
        test: /\.css$/i,  
        use: ["style-loader", "css-loader"],  
      },  
    ],  
  },  
};
```

优化-提取 css 代码

[插件 mini-css-extract-plugin](#): 提取 css 代码

步骤:

1. **下载** mini-css-extract-plugin 本地软件包
2. **配置** webpack.config.js 让 Webpack 拥有该插件功能
3. 打包后观察效果

注意: 不能和 style-loader 一起使用

好处: css 文件可以被浏览器缓存, 减少 js 文件体积

```
npm install --save-dev mini-css-extract-plugin
```

```
// ...
const MiniCssExtractPlugin = require("mini-css-extract-plugin")

module.exports = {
  // ...
  module: {
    rules: [
      {
        test: /\.css$/i,
        // use: ['style-loader', 'css-loader']
        use: [MiniCssExtractPlugin.loader, "css-loader"],
      },
    ],
  },
  plugins: [
    // ...
    new MiniCssExtractPlugin()
  ]
};
```

优化-压缩过程

问题：css 代码提取后没有压缩

解决：使用 [css-minimizer-webpack-plugin](#) 插件

步骤：

1. 下载 css-minimizer-webpack-plugin 本地软件包
2. 配置 webpack.config.js 让 webpack 拥有该功能
3. 打包重新观察

```
npm install css-minimizer-webpack-plugin --save-dev
```

```
// ...
const CssMinimizerPlugin = require("css-minimizer-webpack-plugin");

module.exports = {
  // ...
  // 优化
  optimization: {
    // 最小化
    minimizer: [
      // 在 webpack@5 中，你可以使用 `...` 语法来扩展现有的 minimizer（即
      // `terser-webpack-plugin`），将下一行取消注释（保证 JS 代码还能被压缩处理）
      `...`,
      new CssMinimizerPlugin(),
    ],
  },
};
```

打包 less 代码

加载器 less-loader: 把 less 代码编译为 css 代码

步骤:

1. 新建 less 代码（设置背景图）并引入到 src/login/index.js 中
2. 下载 less 和 less-loader 本地软件包
3. 配置 webpack.config.js 让 Webpack 拥有功能
4. 打包后观察效果

```
npm i less less-loader --save-dev
```

```
// ...  
  
module.exports = {  
  // ...  
  module: {  
    rules: [  
      // ...  
      {  
        test: /\.less$/i,  
        use: [MiniCssExtractPlugin.loader, "css-loader", "less-loader"]  
      }  
    ]  
  }  
}
```

注意: less-loader 需要配合 less 软件包使用

打包图片

资源模块: Webpack5 内置资源模块（字体，图片等）打包，无需额外 loader

步骤:

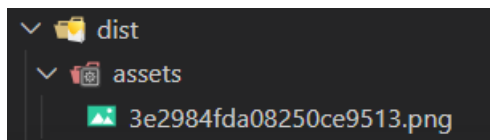
1. **配置** webpack.config.js 让 Webpack 拥有打包图片功能

- ✓ 占位符 **【hash】** 对模块内容做算法计算，得到映射的数字字母组合的字符串
- ✓ 占位符 **【ext】** 使用当前模块原本的占位符，例如: .png / .jpg 等字符串
- ✓ 占位符 **【query】** 保留引入文件时代码中查询参数（只有 URL 下生效）

2. 打包后观察效果和区别

注意: 判断临界值默认为 **8KB**

- ✓ 大于 8KB 文件: 发送一个单独的文件并导出 URL 地址
- ✓ 小于 8KB 文件: 导出一个 data URI (base64字符串)



```
body {  
  ☒ background: url(http://localhost:5501/dist/assets/3e2984f....png) no-repeat  
}
```

```
// ...  
  
module.exports = {  
  // ...  
  module: {  
    rules: [  
      // ...  
      {  
        test: /\. (png|jpg|jpeg|gif)$/i,  
        type: 'asset',  
        generator: {  
          filename: 'assets/[hash][ext][query]'  
        }  
      }  
    ]  
  }  
}
```

```
  
</div>  
<div class="login-wrap">  
  <!-- Code injected by live-  
<script></script>  
</div>  
</div>
```

渲染的大小:	436 × 44 px
渲染时的宽高比:	109:11
文件大小:	5.5 kB
当前来源:	data:image/png;base64,iVBORw0KGgoAAAABJRU5ErkJggg==

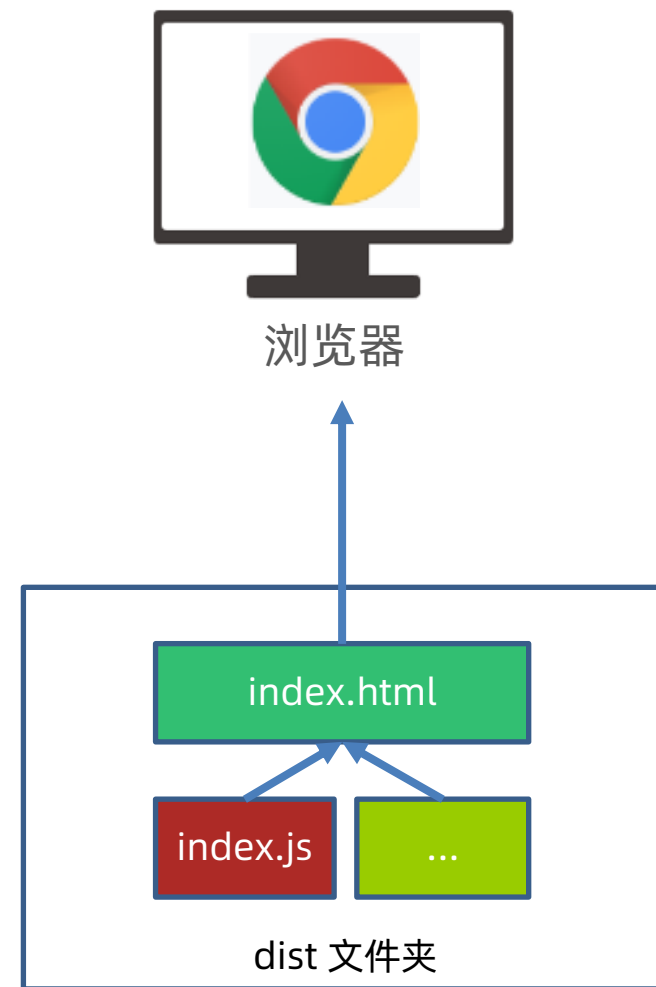
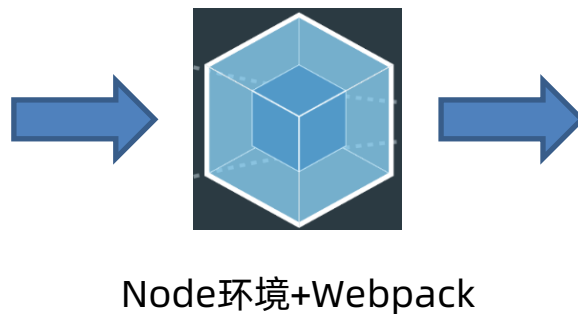
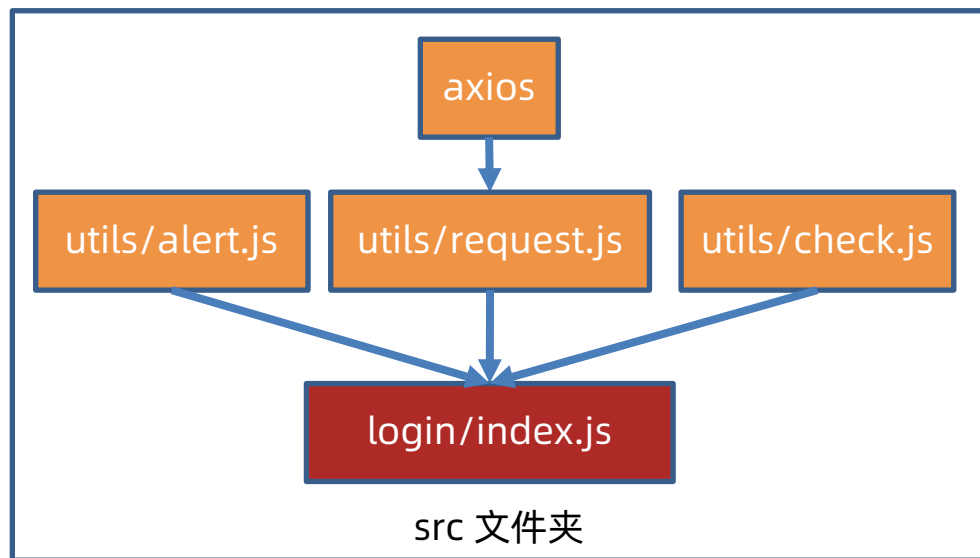
案例

用户登录 - 完成功能

需求：完成登录功能的核心流程，以及 Alert 警告框使用

步骤：

1. 使用 npm 下载 axios（体验 npm 作用在前端项目中）
2. 准备并修改 utils 工具包源代码导出实现函数
3. 导入并编写逻辑代码，打包后运行观察效果



搭建开发环境

问题：之前改代码，需重新打包才能运行查看，效率很低

开发环境：配置 webpack-dev-server 快速开发应用程序

作用：启动 Web 服务，**自动**检测代码变化，**热更新**到网页

注意：dist 目录和打包内容是在内存里（更新快）

步骤：

1. **下载** webpack-dev-server 软件包到当前项目
2. 设置模式为**开发模式**，并配置**自定义命令**
3. 使用 npm run dev 来启动开发服务器，试试热更新效果

```
npm i webpack-dev-server --save-dev
```

```
// ...  
  
module.exports = {  
  // ...  
  mode: 'development'  
}
```

```
"scripts": {  
  "build": "webpack",  
  "dev": "webpack serve --open"  
},
```

打包模式

打包模式：告知 Webpack 使用相应模式的内置优化

分类：

模式名称	模式名字	特点	场景
开发模式	development	调试代码，实时加载，模块热替换等	本地开发
生产模式	production	压缩代码，资源优化，更轻量等	打包上线

```
// ...  
  
module.exports = {  
  // ...  
  mode: 'production'  
};  
  
};  
  
mode: 'production'
```

设置：

方式1：在 webpack.config.js 配置文件设置 mode 选项

方式2：在 package.json 命令行设置 mode 参数

```
"scripts": {  
  "build": "webpack --mode=production",  
  "dev": "webpack serve --mode=development"  
},  
  
'
```

注意：命令行设置的优先级高于配置文件中的，推荐用命令行设置

打包模式的应用

需求：在开发模式下用 style-loader 内嵌更快，在生产模式下提取 css 代码

[方案1](#)：webpack.config.js 配置导出函数，但是局限性大（只接受 2 种模式）

方案2：借助 cross-env（跨平台通用）包命令，设置参数区分环境

步骤：

```
npm i cross-env --save-dev
```

1. **下载** cross-env 软件包到当前项目
2. **配置**自定义命令，传入参数名和值（会绑定到 `process.env` 对象下）
3. 在 webpack.config.js 区分不同环境**使用**不同配置
4. 重新打包观察两种配置区别

```
"scripts": {  
  "test": "echo \"Error: no test specified\" && exit 1",  
  "build": "cross-env NODE_ENV=production webpack --mode=production",  
  "dev": "cross-env NODE_ENV=development webpack serve --open --mode=development"  
},
```

[方案3](#)：配置不同的 webpack.config.js（适用多种模式差异性较大情况）

前端-注入环境变量

需求：前端项目中，开发模式下打印语句生效，生产模式下打印语句失效

问题：cross-env 设置的只在 Node.js 环境生效，前端代码无法访问 process.env.NODE_ENV

解决：使用 Webpack 内置的 DefinePlugin 插件

作用：在编译时，将前端代码中匹配的变量名，替换为值或表达式

```
// ...
const webpack = require('webpack')

module.exports = {
  // ...
  plugins: [
    // ...
    new webpack.DefinePlugin({
      // key 是注入到打包后的前端 JS 代码中作为全局变量
      // value 是变量对应的值（在 cross-env 注入在 node.js 中的环境变量字符串）
      'process.env.NODE_ENV': JSON.stringify(process.env.NODE_ENV)
    })
  ]
}
```

开发环境调错 - source map

问题：代码被压缩和混淆，无法正确定位源代码位置（行数和列数）

[source map](#)：可以准确追踪 error 和 warning 在原始代码的位置

设置：webpack.config.js 配置 [devtool 选项](#)

```
// ...  
  
module.exports = {  
  // ...  
  devtool: 'inline-source-map'  
};  
  
};
```

inline-source-map 选项：把源码的位置信息一起打包在 js 文件内

注意：source map 仅适用于开发环境，不要在生产环境使用（防止被轻易查看源码位置）

解析别名 alias

解析别名：配置模块如何解析，创建 import 引入路径的别名，来确保模块引入变得更简单

例如：原来路径如图，比较长而且相对路径不安全

解决：在 webpack.config.js 中配置解析别名 @ 来代表 src 绝对路径

```
// ...  
  
const config = {  
  // ...  
  resolve: {  
    alias: {  
      '@': path.resolve(__dirname, 'src')  
    }  
  }  
}
```

```
import { checkPhone, checkCode } from '../src/utils/check.js'
```

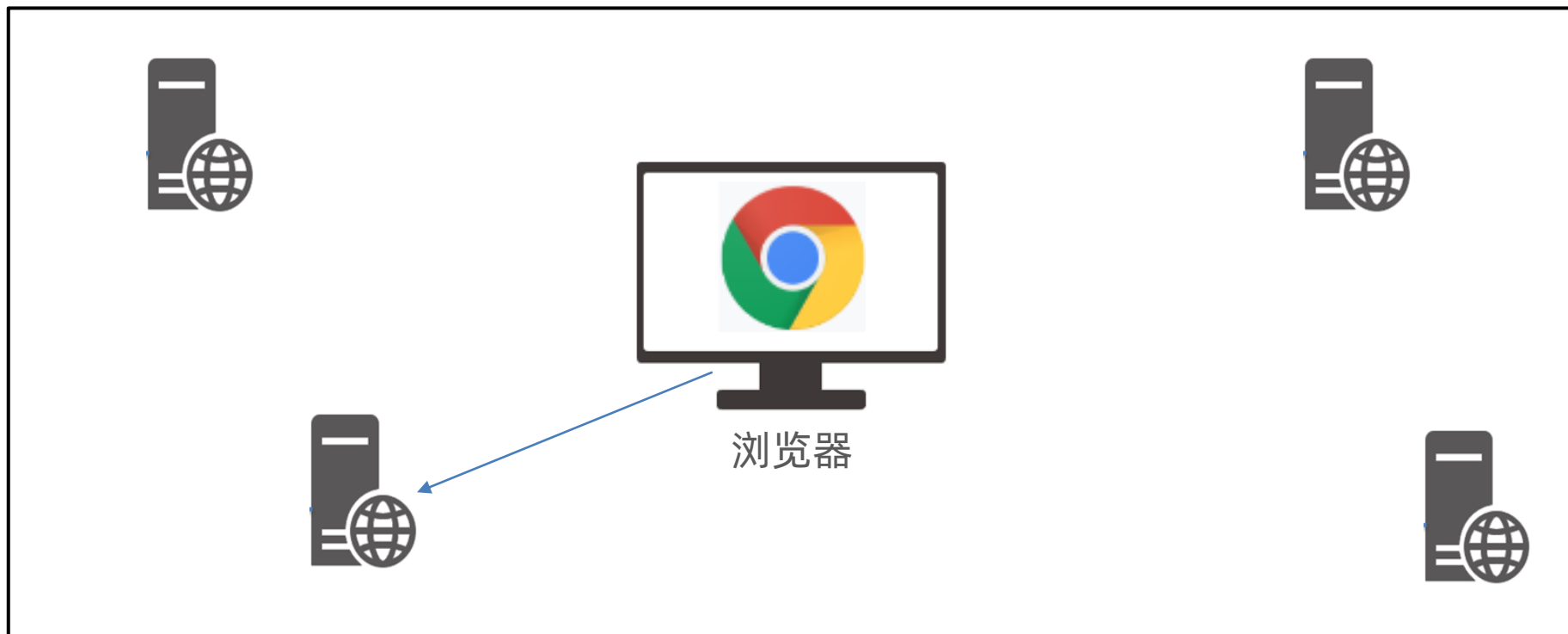
```
import { checkPhone, checkCode } from '@/utils/check.js'
```

优化-CDN使用

CDN定义：内容分发网络，指的是一组分布在各个地区的服务器

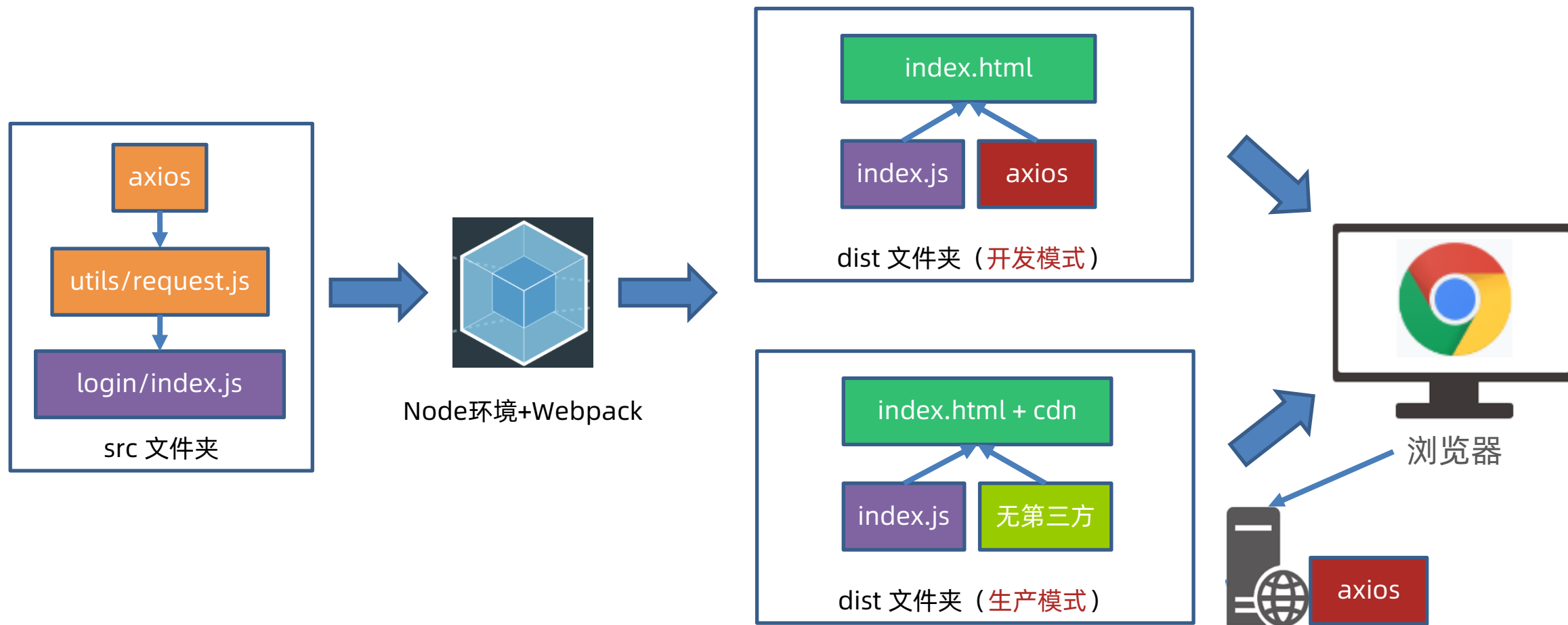
作用：把静态资源文件/第三方库放在 CDN 网络中各个服务器中，供用户就近请求获取

好处：减轻自己服务器请求压力，就近请求物理延迟低，配套缓存策略



优化-CDN使用

需求：开发模式使用本地第三方库，生产模式下使用 CDN 加载引入



优化-CDN使用

需求：开发模式使用本地第三方库，生产模式下使用 CDN 加载引入

步骤：

1. 在 html 中引入第三方库的 [CDN 地址](#) 并用模板语法判断
2. 配置 webpack.config.js 中 [externals](#) 外部扩展选项（防止某些 import 的包被打包）
3. 两种模式下打包观察效果

```
<% if(htmlWebpackPlugin.options.useCdn){ %>
  <link href="https://cdn.bootcdn.net/ajax/libs/twitter-bootstrap/5.2.3/css/bootstrap.min.css"
  rel="stylesheet">
<% } %>
```

```
// 生产模式下-使用
if (process.env.NODE_ENV === 'production') {
  config.externals = {
    // key: 代码中 import from 后面的模块标识字符串
    // value: 替换在原地的变量名（要和 cdn 暴露在全局的变量名一致）
    'axios': 'axios'
  }
}
```

```
// ...
const config = {
  // ...
  plugins: [
    new HtmlWebpackPlugin({
      // ...
      // 自定义属性，在 html 模板中 <%=htmlWebpackPlugin.options.useCdn%> 访问使用
      useCdn: process.env.NODE_ENV === 'production'
    })
  ]
}
```

多页面打包

单页面：单个 html 文件，切换 DOM 的方式实现不同业务逻辑展示，后续 Vue/React 会学到

多页面：多个 html 文件，切换页面实现不同业务逻辑展示

需求：把黑马头条-数据管理平台-内容页面一起引入打包使用

步骤：

1. 准备源码（html, css, js）放入相应位置，并改用模块化语法导出
2. 下载 form-serialize 包并导入到核心代码中使用
3. 配置 webpack.config.js 多入口和多页面的设置
4. 重新打包观察效果

```
// ...
const config = {
  entry: {
    '模块名1': path.resolve(__dirname, 'src/入口1.js'),
    '模块名2': path.resolve(__dirname, 'src/入口2.js'),
  },
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: './[name]/index.js'
  },
  plugins: [
    new HtmlWebpackPlugin({
      template: './public/页面2.html', // 模板文件
      filename: './路径/index.html', // 输出文件
      chunks: ['模块名2']
    })
    new HtmlWebpackPlugin({
      template: './public/页面2.html', // 模板文件
      filename: './路径/index.html', // 输出文件
      chunks: ['模块名2']
    })
  ]
}
```

案例-发布文章页面打包

需求：把发布文章页面一起打包

步骤：

1. 准备发布文章页面源代码，改写成模块化的导出和导入方式
2. 修改 webpack.config.js 的配置，增加一个入口和出口
3. 打包观察效果

优化-分割公共代码

需求：把 2 个以上页面引用的公共代码提取

步骤：

1. 配置 webpack.config.js 的 splitChunks 分割功能
2. 打包观察效果

```
// ...
const config = {
  // ...
  optimization: {
    // ...
    splitChunks: {
      chunks: 'all', // 所有模块动态非动态移入的都分割分析
      cacheGroups: { // 分隔组
        commons: { // 抽取公共模块
          minSize: 0, // 抽取的chunk最小大小字节
          minChunks: 2, // 最小引用数
          reuseExistingChunk: true, // 当前 chunk 包含已从主 bundle 中拆分出的模块，则它将被重用
          name(module, chunks, cacheGroupKey) { // 分离出模块文件名
            const allChunksNames = chunks.map((item) => item.name).join('~') // 模块名1~模块名2
            return `./js/${allChunksNames}` // 输出到 dist 目录下位置
          }
        }
      }
    }
  }
}
```

总结

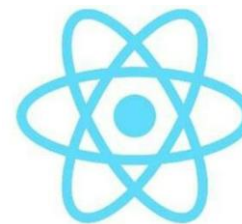
配套的笔记中有重点知识的记录和梳理，大家一定要看起来啊!



黑马头条



webpack





传智教育旗下高端IT教育品牌