

Node.js

目标

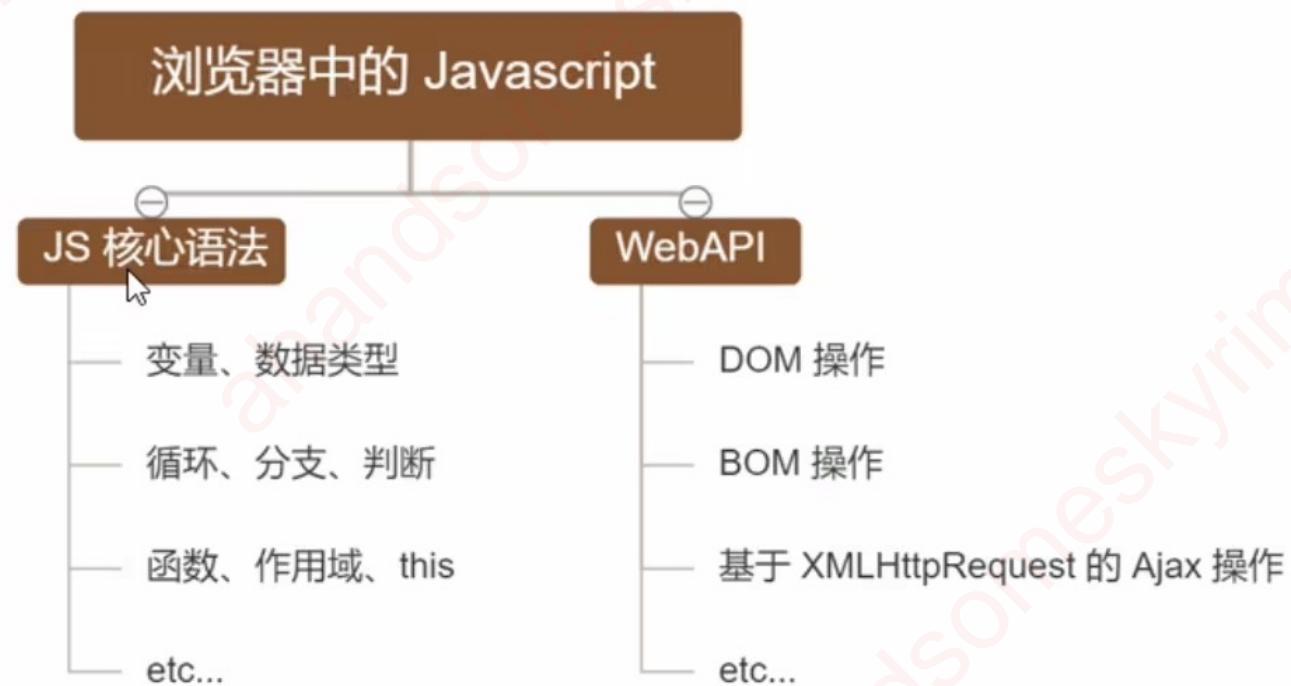
TARGET

- ◆ 能够知道什么是 Node.js
- ◆ 能够知道 Node.js 可以做什么
- ◆ 能够说出 Node.js 中的 JavaScript 的组成部分
- ◆ 能够使用 fs 模块读写操作文件
- ◆ 能够使用 path 模块处理路径
- ◆ 能够使用 http 模块写一个基本的 web 服务器



1.1 回顾与思考

2. 浏览器中的 JavaScript 的组成部分



1.1 回顾与思考

3. 思考：为什么 JavaScript 可以在浏览器中被执行



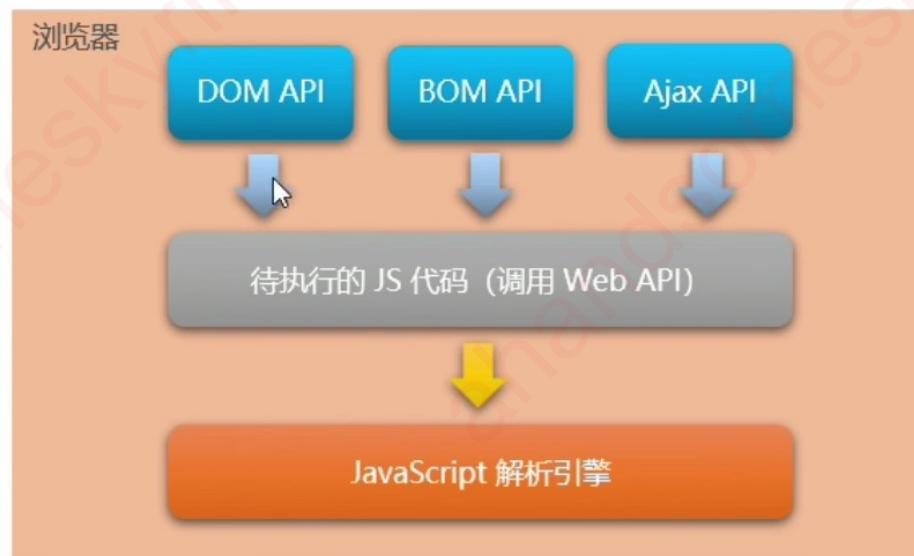
不同的浏览器使用不同的 JavaScript 解析引擎：

- Chrome 浏览器 => V8
- Firefox 浏览器 => OdinMonkey (奥丁猴)
- Safri 浏览器 => JSCore
- IE 浏览器 => Chakra (查克拉)
- etc...

其中，Chrome 浏览器的 V8 解析引擎性能最好！

1.1 回顾与思考

4. 思考：为什么 JavaScript 可以操作 DOM 和 BOM

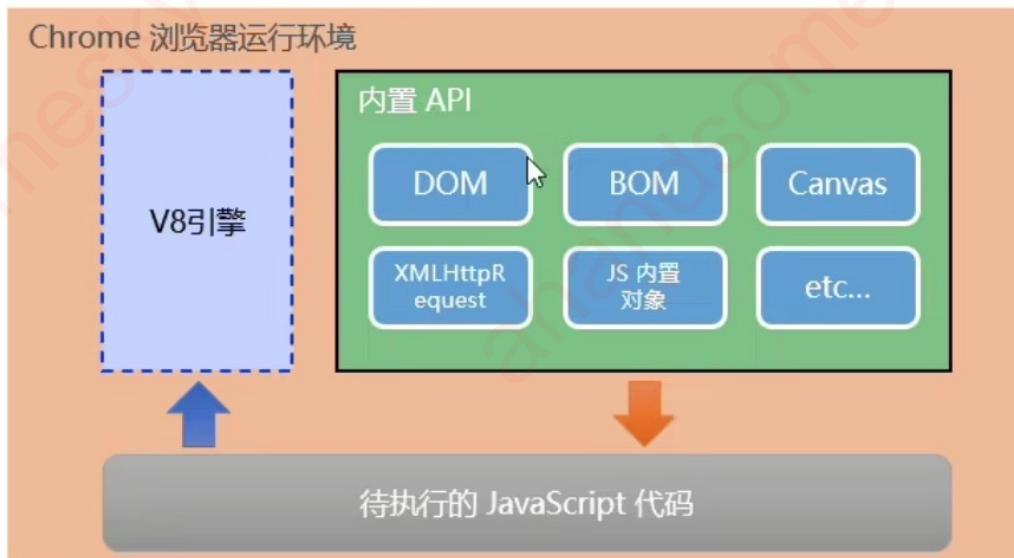


每个浏览器都**内置了** DOM、BOM 这样的 API 函数，因此，浏览器中的 JavaScript 才可以调用它们。

1.1 回顾与思考

5. 浏览器中的 JavaScript 运行环境

运行环境是指代码正常运行所需的必要环境。



总结:

- ① V8 引擎负责解析和执行 JavaScript 代码。
- ② 内置 API 是由运行环境提供的特殊接口，
只能在所属的运行环境中被调用。

1.1 回顾与思考

6. 思考：JavaScript 能否做后端开发



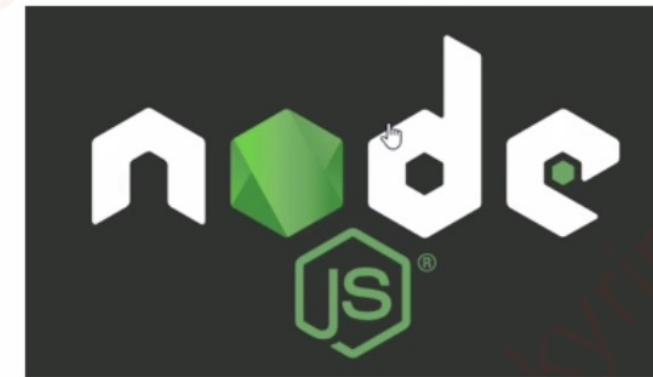
Java



Python



PHP



Node.js

1.2 Node.js 简介

1. 什么是 Node.js

Node.js® is a **JavaScript runtime** built on Chrome's V8 JavaScript engine.

Node.js 是一个基于 Chrome V8 引擎的 **JavaScript 运行环境**。

Node.js 的官网地址: <https://nodejs.org/zh-cn/>



Node.js® 是一个基于 Chrome V8 引擎的 JavaScript 运行时。

下载平台为：Windows (x64)

12.16.1 长期支持版

推荐多数用户使用 (LTS)

[其它下载](#) | [更新日志](#) | [API 文档](#)

13.11.0 当前发布版

含最新功能

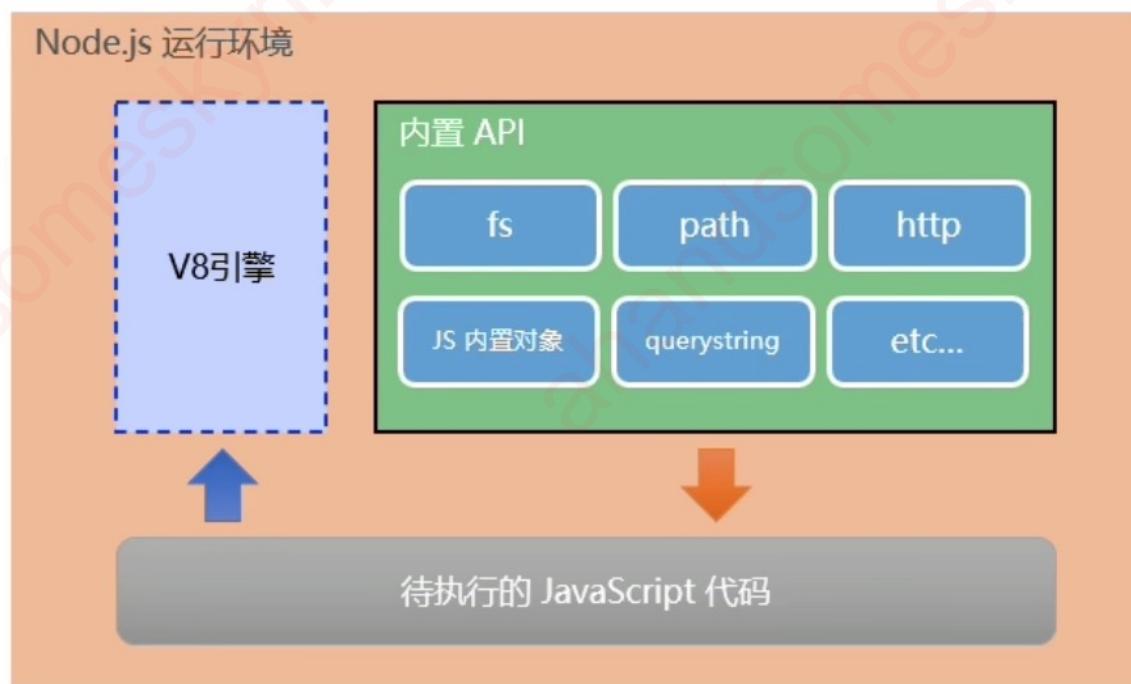
[其它下载](#) | [更新日志](#) | [API 文档](#)

可参考 [LTS 日程](#)。

订阅 [Node.js Everywhere](#), Node.js 官方的新闻月报。

1.2 Node.js 简介

2. Node.js 中的 JavaScript 运行环境



注意：

- ① 浏览器是 JavaScript 的前端运行环境。
- ② Node.js 是 JavaScript 的后端运行环境。
- ③ Node.js 中无法调用 DOM 和 BOM 等
浏览器内置 API。

1.2 Node.js 简介

3. Node.js 可以做什么

Node.js 作为一个 JavaScript 的运行环境，仅仅提供了基础的功能和 API。然而，基于 Node.js 提供的这些基础能，很多强大的工具和框架如雨后春笋，层出不穷，所以学会了 Node.js，可以让前端程序员胜任更多的工作和岗位：

- ① 基于 Express 框架 (<http://www.expressjs.com.cn/>)，可以快速构建 Web 应用
- ② 基于 Electron 框架 (<https://electronjs.org/>)，可以构建跨平台的桌面应用
- ③ 基于 restify 框架 (<http://restify.com/>)，可以快速构建 API 接口项目
- ④ 读写和操作数据库、创建实用的命令行工具辅助前端开发、etc...

总之：Node.js 是**大前端时代**的“大宝剑”，有了 Node.js 这个超级 buff 的加持，前端程序员的**行业竞争力**会越来越强！

1.2 Node.js 简介

4. Node.js 怎么学

浏览器中的 JavaScript 学习路径:

JavaScript 基础语法 + 浏览器内置 API (DOM + BOM) + 第三方库 (jQuery、art-template 等)

Node.js 的学习路径:

JavaScript 基础语法 + **Node.js 内置 API 模块** (fs、path、http等) + **第三方 API 模块** (express、mysql 等)

1.3 Node.js 环境的安装

如果希望通过 Node.js 来运行 Javascript 代码，则必须在计算机上安装 Node.js 环境才行。

安装包可以从 Node.js 的官网首页直接下载，进入到 Node.js 的官网首页 (<https://nodejs.org/en/>)，点击绿色的按钮，下载所需的版本后，双击直接安装即可。

1.3 Node.js 环境的安装

1. 区分 **LTS** 版本和 **Current** 版本的不同

- ① LTS 为长期稳定版，对于**追求稳定性**的**企业级项目**来说，推荐安装 LTS 版本的 Node.js。
- ② Current 为新特性尝鲜版，对**热衷于尝试新特性**的用户来说，推荐安装 Current 版本的 Node.js。但是，Current 版本中可能存在隐藏的 Bug 或安全性漏洞，因此不推荐在企业级项目中使用 Current 版本的 Node.js。

1.3 Node.js 环境的安装

2. 查看已安装的 Node.js 的版本号

打开[终端](#)，在终端输入命令 `node -v` 后，按下回车键，即可查看已安装的 Node.js 的版本号。

Windows 系统快速打开终端的方式：

使用快捷键 ([Windows徽标键 + R](#)) 打开运行面板，输入 `cmd` 后直接回车，即可打开终端。

查看node版本 `node -v`

windows 打开终端

Windows徽标键 + R 打开运行面板， 输入cmd后直接回车， 即可打开终端

打开文件所处目录

在文件所处目录， 右击打开powershell

浏览器提供了执行环境

Node也有类似的执行环境，在cmd中type node开始执行命令，退出用`.exit`

使用node运行js file代码 `node 1.js`

1. 模块化的基本概念

1.1 什么是模块化

2. 编程领域中的模块化

编程领域中的模块化，就是遵守固定的规则，把一个大文件拆成独立并互相依赖的多个小模块。

把代码进行模块化拆分的好处：

- ① 提高了代码的复用性
- ② 提高了代码的可维护性
- ③ 可以实现按需加载

2.1 Node.js 中模块的分类

Node.js 中根据模块来源的不同，将模块分为了 3 大类，分别是：

- 内置模块（内置模块是由 Node.js 官方提供的，例如 fs、path、http 等）
- 自定义模块（用户创建的每个 .js 文件，都是自定义模块）
- 第三方模块（由第三方开发出来的模块，并非官方提供的内置模块，也不是用户创建的自定义模块，使用前需要先下载）

2.2 加载模块

使用强大的 `require()` 方法，可以加载需要的内置模块、用户自定义模块、第三方模块进行使用。例如：

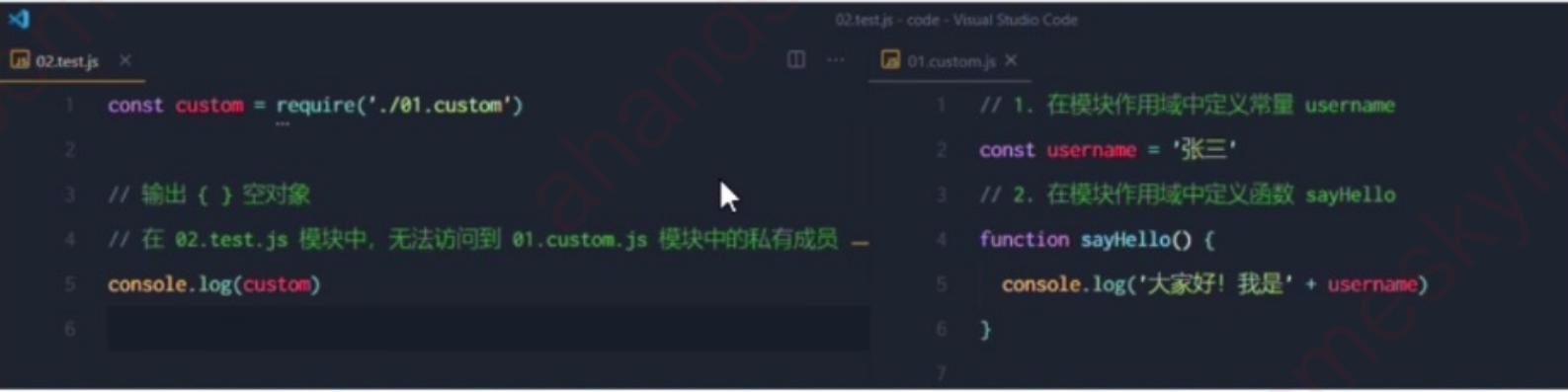
```
1 // 1. 加载内置的 fs 模块
2 const fs = require('fs')
3
4 // 2. 加载用户的自定义模块
5 const custom = require('./custom.js')
6
7 // 3. 加载第三方模块 (关于第三方模块的下载和使用，会在后面的课程中进行专门的讲解)
8 const moment = require('moment')
```

注意： 使用 `require()` 方法加载其它模块时，会执行被加载模块中的代码。

2.3 Node.js 中的模块作用域

1. 什么是模块作用域

和函数作用域类似，在自定义模块中定义的变量、方法等成员，只能在当前模块内被访问，这种模块级别的访问限制，叫做模块作用域。



```
02.test.js - code - Visual Studio Code
01.custom.js - code - Visual Studio Code

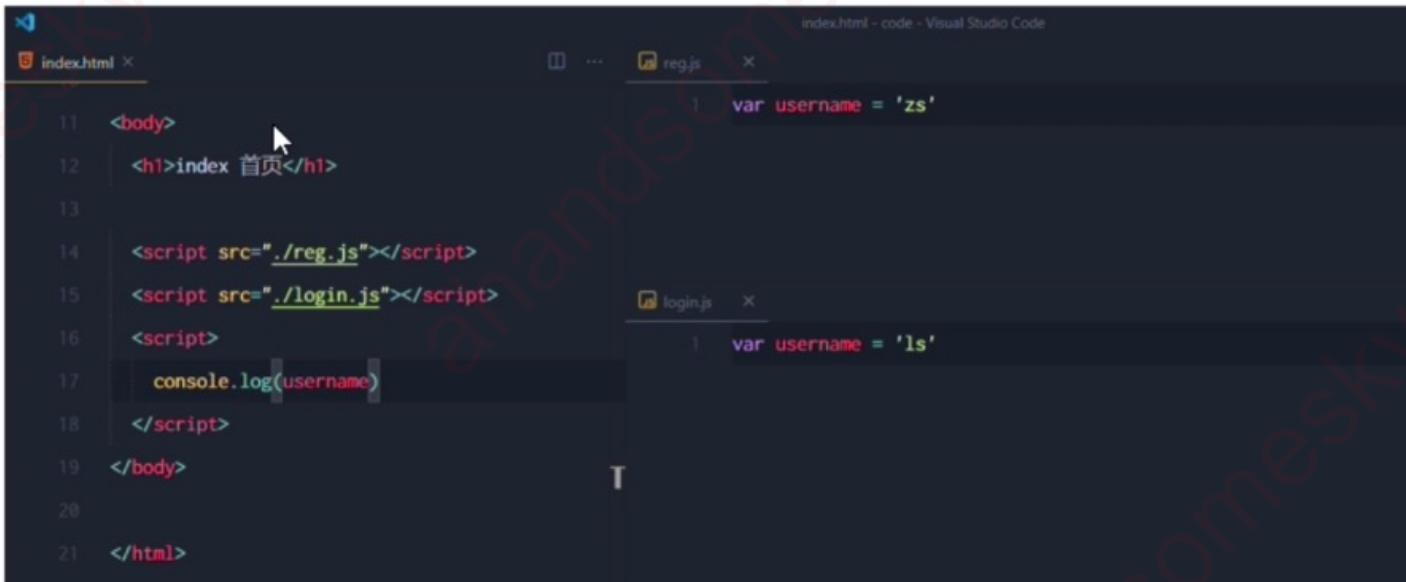
02.test.js
1 const custom = require('./01.custom')
...
2 // 输出 {} 空对象
3 // 在 02.test.js 模块中，无法访问到 01.custom.js 模块中的私有成员 -
4 console.log(custom)
5
6

01.custom.js
1 // 1. 在模块作用域中定义常量 username
2 const username = '张三'
3 // 2. 在模块作用域中定义函数 sayHello
4 function sayHello() {
5   console.log('大家好！我是' + username)
6 }
```

2.3 Node.js 中的模块作用域

2. 模块作用域的好处

防止了全局变量污染的问题



The screenshot shows three files in Visual Studio Code:

- index.html:** Contains HTML code with script tags pointing to `reg.js` and `login.js`. It also contains a `script` block with `console.log(username)`.
- reg.js:** Contains the code `var username = 'zs'`.
- login.js:** Contains the code `var username = 'ls'`.

The code in `reg.js` and `login.js` does not affect the variable in the `script` block of `index.html`, demonstrating module scope.

2.4 向外共享模块作用域中的成员

1. module 对象

在每个 .js 自定义模块中都有一个 module 对象，它里面**存储了和当前模块有关的信息**，打印如下：



The screenshot shows a Visual Studio Code interface. On the left is a code editor with a file named "03.module对象.js" containing the line "console.log(module)". To the right is a terminal window titled "03.module对象.js - code - Visual Studio Code". The terminal displays the output of running the script: a Module object with properties like id, path, exports, parent, filename, loaded, children, and paths. The "exports" property is highlighted with a red box. The terminal also shows the current working directory as "C:\Users\liulongbin\Desktop\node\day2\code>node 03.module对象.js".

```
03.module对象.js x
03.module对象.js - code - Visual Studio Code
1: cmd + ...
```

```
console.log(module)

问题 输出 调试控制台 编辑

C:\Users\liulongbin\Desktop\node\day2\code>node 03.module对象.js
Module {
  id: '',
  path: 'C:\\Users\\liulongbin\\Desktop\\node\\day2\\code',
  exports: {},
  parent: null,
  filename: 'C:\\Users\\liulongbin\\Desktop\\node\\day2\\code\\03.module对象.js',
  loaded: false,
  children: [],
  paths: [
    'C:\\Users\\liulongbin\\Desktop\\node\\day2\\code\\node_modules',
    'C:\\Users\\liulongbin\\Desktop\\node\\day2\\node_modules',
    'C:\\Users\\liulongbin\\Desktop\\node\\node_modules',
    'C:\\Users\\liulongbin\\Desktop\\node_modules',
    'C:\\Users\\liulongbin\\node_modules',
    'C:\\Users\\node_modules',
    'C:\\node_modules'
  ]
}
```

2.4 向外共享模块作用域中的成员

2. module.exports 对象

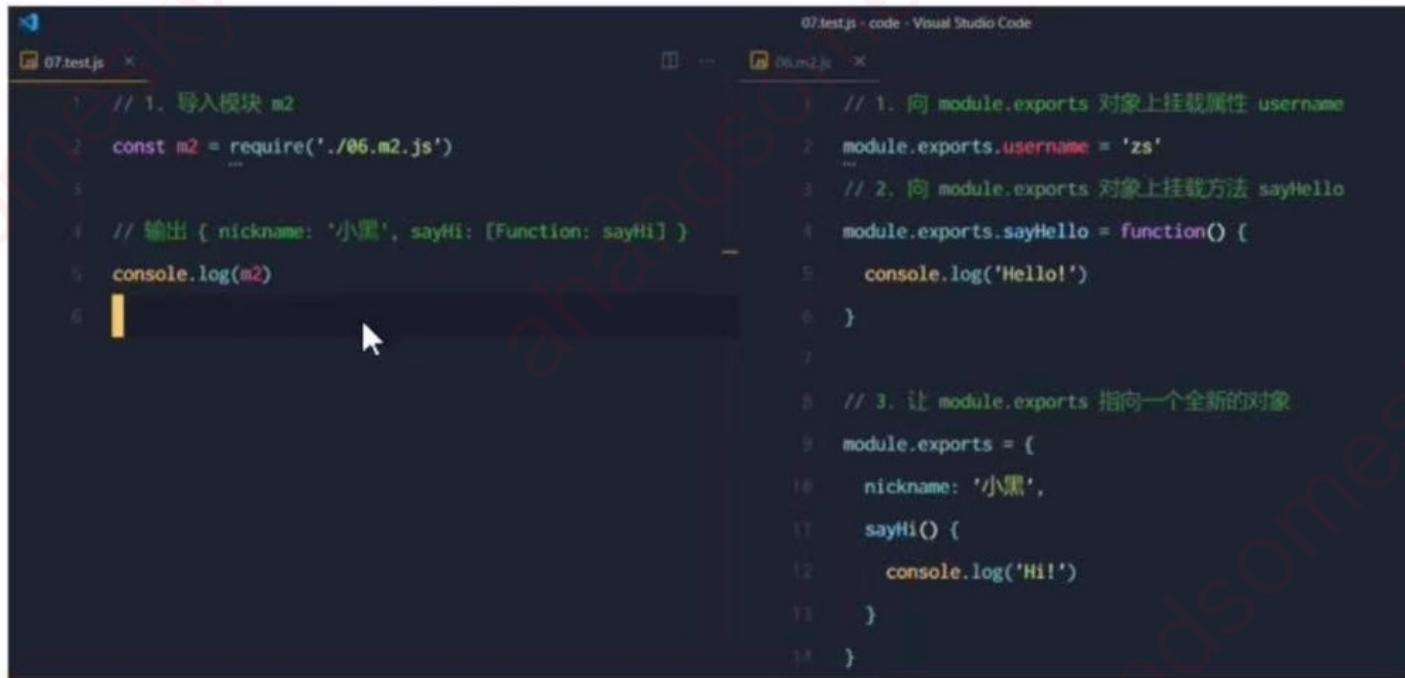
在自定义模块中，可以使用 `module.exports` 对象，将模块内的成员共享出去，供外界使用。

外界用 `require()` 方法导入自定义模块时，得到的就是 `module.exports` 所指向的对象。

2.4 向外共享模块作用域中的成员

3. 共享成员时的注意点

使用 require() 方法导入模块时，导入的结果，**永远以 module.exports 指向的对象为准**。



The screenshot shows two files open in Visual Studio Code:

- 07.test.js**:

```
1 // 1. 导入模块 m2
2 const m2 = require('./06.m2.js')
3
4 // 输出 { nickname: '小黑', sayHi: [Function: sayHi] }
5 console.log(m2)
6
```
- 06.m2.js**:

```
1 // 1. 向 module.exports 对象上挂载属性 username
2 module.exports.username = 'zs'
3
4 // 2. 向 module.exports 对象上挂载方法 sayHello
5 module.exports.sayHello = function() {
6     console.log('Hello!')
7 }
8
9 // 3. 让 module.exports 指向一个全新的对象
10 module.exports = {
11     nickname: '小黑',
12     sayHi() {
13         console.log('Hi!')
14     }
15 }
```

The code demonstrates three ways to share module members:

- Sharing a property (`username`)
- Sharing a method (`sayHello`)
- Sharing a new object that overrides the original `module.exports`

2.5 Node.js 中的模块化规范

Node.js 遵循了 CommonJS 模块化规范，CommonJS 规定了模块的特性和各模块之间如何相互依赖。

CommonJS 规定：

- ① 每个模块内部，`module` 变量代表当前模块。
- ② `module` 变量是一个对象，它的 `exports` 属性（即 `module.exports`）是对外的接口。
- ③ 加载某个模块，其实是加载该模块的 `module.exports` 属性。`require()` 方法用于加载模块。

2.1 什么是 fs 文件系统模块

fs 模块是 Node.js 官方提供的、用来操作文件的模块。它提供了一系列的方法和属性，用来满足用户对文件的操作需求。

例如：

- `fs.readFile()` 方法，用来读取指定文件中的内容
- `fs.writeFile()` 方法，用来向指定的文件中写入内容

如果要在 JavaScript 代码中，使用 fs 模块来操作文件，则需要使用如下的方式先导入它：

```
1 const fs = require('fs')
```

2.2 读取指定文件中的内容

1. fs.readFile() 的语法格式

使用 `fs.readFile()` 方法，可以读取指定文件中的内容，语法格式如下：

```
1 fs.readFile(path[, options], callback)
```

参数解读：

- 参数1：必选参数，字符串，表示文件的路径。
- 参数2：可选参数，表示以什么编码格式来读取文件。
- 参数3：必选参数，文件读取完成后，通过回调函数拿到读取的结果。

2.2 读取指定文件中的内容

2. fs.readFile() 的示例代码

以 utf8 的编码格式，读取指定文件的内容，并打印 err 和 dataStr 的值：

```
1 const fs = require('fs')
2 fs.readFile('./files/11.txt', 'utf8', function(err, dataStr) {
3   console.log(err)
4   console.log('-----')
5   console.log(dataStr)
6 })
```

2.2 读取指定文件中的内容

3. 判断文件是否读取成功

可以判断 err 对象是否为 null，从而知晓文件读取的结果：

```
1 const fs = require('fs')
2 fs.readFile('./files/1.txt', 'utf8', function(err, result) {
3   if (err) {
4     return console.log('文件读取失败! ' + err.message)
5   }
6   console.log('文件读取成功，内容是: ' + result)
7 })
```

2.3 向指定的文件中写入内容

1. fs.writeFile() 的语法格式

使用 fs.writeFile() 方法，可以向指定的文件中写入内容，语法格式如下：

```
1 fs.writeFile(file, data[, options], callback)
```

参数解读：

- 参数1：必选参数，需要指定一个文件路径的字符串，表示文件的存放路径。
- 参数2：必选参数，表示要写入的内容。
- 参数3：可选参数，表示以什么格式写入文件内容，默认值是 utf8。
- 参数4：必选参数，文件写入完成后的回调函数。

2.3 向指定的文件中写入内容

2. fs.writeFile() 的示例代码

向指定的文件路径中，写入文件内容：

```
1 const fs = require('fs')
2 fs.writeFile('./files/2.txt', 'Hello Node.js!', function(err) {
3   console.log(err)
4 })
```

2.3 向指定的文件中写入内容

3. 判断文件是否写入成功

可以判断 err 对象是否为 null，从而知晓文件写入的结果：

```
1 const fs = require('fs')
2 fs.writeFile('F:/files/2.txt', 'Hello Node.js!', function(err) {
3   if (err) {
4     return console.log('文件写入失败！' + err.message)
5   }
6   console.log('文件写入成功！')
7 })
```

3. npm与包

3.1 包

1. 什么是包

Node.js 中的第三方模块又叫做包。

就像电脑和计算机指的是相同的东西，第三方模块和包指的是同一个概念，只不过叫法不同。

3.1 包

2. 包的来源

不同于 Node.js 中的内置模块与自定义模块，**包是由第三方个人或团队开发出来的**，免费供所有人使用。

注意：Node.js 中的包都是免费且开源的，不需要付费即可免费下载使用。

3.1 包

3. 为什么需要包

由于 Node.js 的内置模块仅提供了一些底层的 API，导致在基于内置模块进行项目开发时，效率很低。

包是基于内置模块封装出来的，提供了更高级、更方便的 API，极大的提高了开发效率。

包和内置模块之间的关系，类似于 jQuery 和 浏览器内置 API 之间的关系。

3.1 包

4. 从哪里下载包

国外有一家 IT 公司，叫做 **npm, Inc.** 这家公司旗下有一个非常著名的网站：<https://www.npmjs.com/>，它是**全球最大的包共享平台**，你可以从这个网站上搜索到任何你需要的包，只要你有足够的耐心！

到目前位置，全球约 **1100 多万** 的开发人员，通过这个包共享平台，开发并共享了超过 **120 多万个包** 供我们使用。

npm, Inc. 公司提供了一个地址为 <https://registry.npmjs.org/> 的服务器，来对外共享所有的包，我们可以从这个服务器上下载自己所需要的包。

注意：

- 从 <https://www.npmjs.com/> 网站上搜索自己所需要的包
- 从 <https://registry.npmjs.org/> 服务器上下载自己需要的包

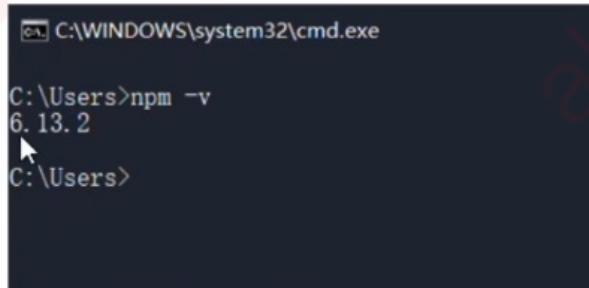
3.1 包

5. 如何下载包

npm, Inc. 公司提供了一个包管理工具，我们可以使用这个包管理工具，从 <https://registry.npmjs.org/> 服务器把需要的包下载到本地使用。

这个包管理工具的名字叫做 **Node Package Manager**（简称 **npm 包管理工具**），这个包管理工具随着 Node.js 的安装包一起被安装到了用户的电脑上。

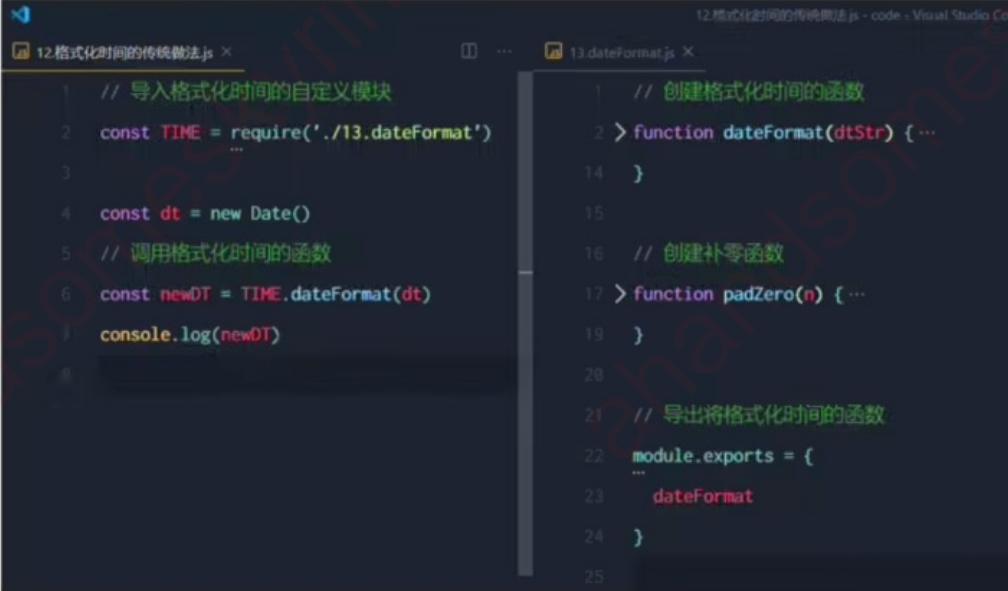
大家可以在终端中执行 **npm -v** 命令，来查看自己电脑上所安装的 npm 包管理工具的版本号：



```
C:\WINDOWS\system32\cmd.exe
C:\Users>npm -v
6.13.2
C:\Users>
```

3.2 npm 初体验

1. 格式化时间的传统做法



The screenshot shows two code files in Visual Studio Code:

- 12.dateFormat.js:** This file imports a module named TIME from './13.dateFormat'. It creates a new Date object named dt, calls the dateFormat function on it, and logs the result to the console.
- 13.dateFormat.js:** This file exports a module named TIME. It contains a dateFormat function that takes a date string as input. It also includes a padZero function to handle padding for digits. The module exports the dateFormat function.

```
// 12.dateFormat.js
// 导入格式化时间的自定义模块
const TIME = require('./13.dateFormat')

// 调用格式化时间的函数
const dt = new Date()
const newDT = TIME.dateFormat(dt)
console.log(newDT)

// 13.dateFormat.js
// 创建格式化时间的函数
function dateFormat(dtStr) {
}

// 创建补零函数
function padZero(n) {
}

// 导出将格式化时间的函数
module.exports = {
  ...
  dateFormat
}
```

- ① 创建格式化时间的自定义模块
- ② 定义格式化时间的方法
- ③ 创建补零函数
- ④ 从自定义模块中导出格式化时间的函数
- ⑤ 导入格式化时间的自定义模块
- ⑥ 调用格式化时间的函数

3.2 npm 初体验

2. 格式化时间的高级做法

- ① 使用 npm 包管理工具，在项目中安装格式化时间的包 moment
- ② 使用 require() 导入格式化时间的包
- ③ 参考 moment 的官方 API 文档对时间进行格式化

```
1 // 1. 导入 moment 包
2 const moment = require('moment')
3
4 // 2. 参考 moment 官方 API 文档，调用对应的方法，对时间进行格式化
5 // 2.1 调用 moment() 方法，得到当前的时间
6 // 2.2 针对当前的时间，调用 format() 方法，按照指定的格式进行时间的格式化
7 const dt = moment().format('YYYY-MM-DD HH:mm:ss')
8
9 console.log(dt) // 输出 2020-01-12 17:23:48
```

3.2 npm 初体验

3. 在项目中安装包的命令

如果想在项目中安装指定名称的包，需要运行如下的命令：

```
1 npm install 包的完整名称
```

上述的装包命令，可以简写成如下格式：

```
1 npm i 完整的包名称
```

3.2 npm 初体验

4. 初次装包后多了哪些文件

初次装包完成后，在项目文件夹下多一个叫做 `node_modules` 的文件夹和 `package-lock.json` 的配置文件。

其中：

`node_modules` 文件夹用来存放所有已安装到项目中的包。`require()` 导入第三方包时，就是从这个目录中查找并加载包。

`package-lock.json` 配置文件用来记录 `node_modules` 目录下的每一个包的下载信息，例如包的名字、版本号、下载地址等。

3.2 npm 初体验

5. 安装指定版本的包

默认情况下，使用 `npm install` 命令安装包的时候，**会自动安装最新版本的包**。如果需要安装指定版本的包，可以在包名之后，通过 `@` 符号指定具体的版本，例如：



A screenshot of a terminal window with a black background and white text. At the top left, there are three colored dots (red, yellow, green). Below them, the command `1 npm i moment@2.22.2` is displayed, with the cursor pointing at the first character of the command.

3.2 npm 初体验

6. 包的语义化版本规范

包的版本号是以“点分十进制”形式进行定义的，总共有三位数字，例如 **2.24.0**



其中每一位数字所代表的含义如下：

第1位数字：大版本

第2位数字：功能版本

第3位数字：Bug修复版本

版本号提升的规则：只要前面的版本号增长了，则后面的版本号归零。

3.3 包管理配置文件

npm 规定，在项目根目录中，必须提供一个叫做 `package.json` 的包管理配置文件。用来记录与项目有关的一些配置信息。例如：

- 项目的名称、版本号、描述等 
- 项目中都用到了哪些包
- 哪些包只在开发期间会用到
- 那些包在开发和部署时都需要用到

3.3 包管理配置文件

2. 如何记录项目中安装了哪些包

在项目根目录中，创建一个叫做 `package.json` 的配置文件，即可用来记录项目中安装了哪些包。从而方便剔除 `node_modules` 目录之后，在团队成员之间共享项目的源代码。

注意：今后在项目开发中，一定要把 `node_modules` 文件夹，添加到 `.gitignore` 忽略文件中。

3.3 包管理配置文件

3. 快速创建 package.json

npm 包管理工具提供了一个**快捷命令**，可以在**执行命令时所处的目录中**，快速创建 package.json 这个包管理配置文件：

```
1 // 作用：在执行命令所处的目录中，快速新建 package.json 文件
2 npm init -y
```

注意：

- ① 上述命令**只能在英文的目录下成功运行！**所以，项目文件夹的名称一定要使用英文命名，**不要使用中文，不能出现空格。**
- ② 运行 npm install 命令安装包的时候，npm 包管理工具会自动把**包的名称和版本号**，记录到 package.json 中。

3.3 包管理配置文件

4. **dependencies** 节点



```
package.json
{
  "name": "my-project",
  "version": "1.0.0",
  "description": "",
  "main": "",
  "scripts": {},
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "moment": "^2.24.0"
  }
}
```

A screenshot of a code editor showing a package.json file. The file contains configuration for a project named 'my-project' with version '1.0.0'. It includes fields for description, main file, scripts, keywords, author, license, and dependencies. The 'dependencies' field contains a dependency on 'moment' with a range '^2.24.0'. A cursor is visible at the end of the 'dependencies' object.

package.json 文件中，有一个 **dependencies** 节点，专门用来记录您使用 npm install 命令安装了哪些包。

3.3 包管理配置文件

5. 一次性安装所有的包

当我们拿到一个剔除了 `node_modules` 的项目之后，需要先把所有的包下载到项目中，才能将项目运行起来。

否则会报类似于下面的错误：

```
1 // 由于项目运行依赖于 moment 这个包，如果没有提前安装好这个包，就会报如下的错误：  
2 Error: Cannot find module 'moment'
```

3.3 包管理配置文件

6. 卸载包

可以运行 `npm uninstall` 命令，来卸载指定的包：

```
1 // 使用 npm uninstall 具体的包名 来卸载包
2 npm uninstall moment
```

注意：`npm uninstall` 命令执行成功后，会把卸载的包，自动从 `package.json` 的 `dependencies` 中移除掉。

3.3 包管理配置文件

7. devDependencies 节点

如果某些包只在项目开发阶段会用到，在项目上线之后不会用到，则建议把这些包记录到 devDependencies 节点中。

与之对应的，如果某些包在开发和项目上线之后都需要用到，则建议把这些包记录到 dependencies 节点中。

您可以使用如下的命令，将包记录到 devDependencies 节点中：

```
1 // 安装指定的包，并记录到 devDependencies 节点中
2 npm i 包名 -D
3 // 注意：上述命令是简写形式，等价于下面完整的写法：
4 npm install 包名 --save-dev
```

- 练习
- 格式化时间
- 1.传统做法
- 2.使用包moment.js

2. 格式化时间的高级做法

- ① 使用 npm 包管理工具，在项目中安装格式化时间的包 moment
- ② 使用 require() 导入格式化时间的包
- ③ 参考 moment 的官方 API 文档对时间进行格式化

4.1 什么是 http 模块

回顾：什么是客户端、什么是服务器？

在网络节点中，负责消费资源的电脑，叫做客户端；负责对外提供网络资源的电脑，叫做服务器。

http 模块是 Node.js 官方提供的、用来创建 web 服务器的模块。通过 http 模块提供的 `http.createServer()` 方法，就能方便的把一台普通的电脑，变成一台 Web 服务器，从而对外提供 Web 资源服务。



如果要希望使用 http 模块创建 Web 服务器，则需要先导入它：

```
1 const http = require('http')
```

4.2 进一步理解 http 模块的作用

服务器和普通电脑的**区别**在于，服务器上安装了 **web 服务器软件**，例如：IIS、Apache 等。通过安装这些服务器软件，就能把一台普通的电脑变成一台 web 服务器。

在 Node.js 中，我们**不需要使用 IIS、Apache 等这些第三方 web 服务器软件**。因为我们可以基于 Node.js 提供的 **http 模块**，**通过几行简单的代码，就能轻松的手写一个服务器软件**，从而对外提供 web 服务。

4.3 服务器相关的概念

1. IP 地址

IP 地址就是互联网上**每台计算机的唯一地址**，因此 IP 地址具有唯一性。如果把“个人电脑”比作“一台电话”，那么“IP 地址”就相当于“电话号码”，只有在知道对方 IP 地址的前提下，才能与对应的电脑之间进行数据通信。

IP 地址的格式：通常用“**点分十进制**”表示成 (a.b.c.d) 的形式，其中，a,b,c,d 都是 0~255 之间的十进制整数。例如：用点分十进表示的 IP 地址 (192.168.1.1)

注意：

- ① **互联网中每台 Web 服务器，都有自己的 IP 地址**，例如：大家可以在 Windows 的终端中运行 `ping www.baidu.com` 命令，即可查看到百度服务器的 IP 地址。
- ② 在开发期间，自己的电脑既是一台服务器，也是一个客户端，为了方便测试，可以在自己的浏览器中输入 127.0.0.1 这个 IP 地址，就能把自己的电脑当做一台服务器进行访问了。

4.3 服务器相关的概念

2. 域名和域名服务器

尽管 IP 地址能够唯一地标记网络上的计算机，但 IP 地址是一长串数字，**不直观**，而且**不便于记忆**，于是人们又发明了另一套**字符型的地址方案**，即所谓的**域名 (Domain Name) 地址**。

IP 地址和域名是**一对多的关系**，这份对应关系存放在一种叫做**域名服务器 (DNS, Domain name server)** 的电脑中。使用者只需通过好记的域名访问对应的服务器即可，对应的转换工作由域名服务器实现。因此，**域名服务器就是提供 IP 地址和域名之间的转换服务的服务器**。

注意：

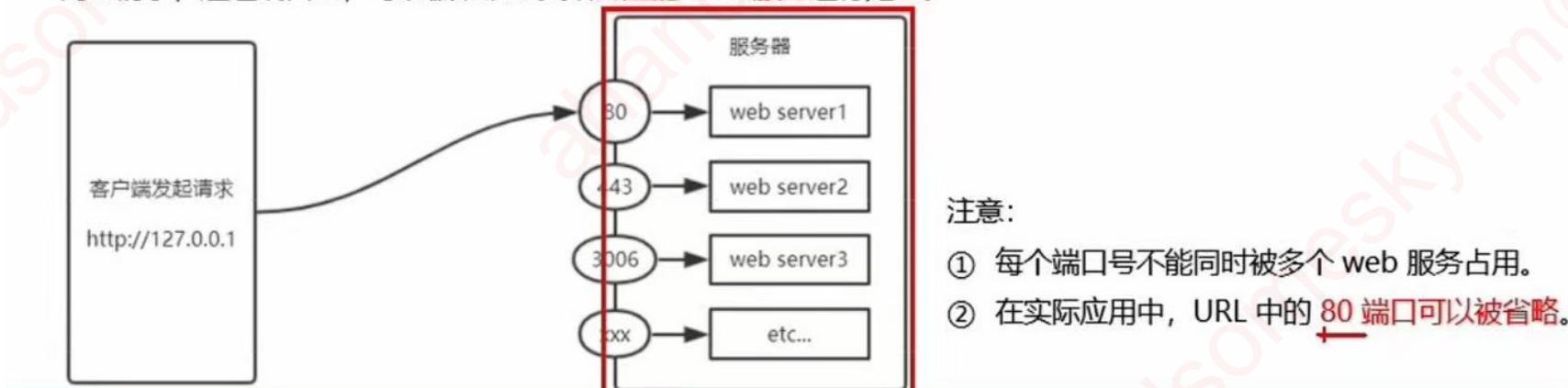
- ① 单纯使用 IP 地址，互联网中的电脑也能够正常工作。但是有了域名的加持，能让互联网的世界变得更加方便。
- ② 在开发测试期间，**127.0.0.1** 对应的域名是 **localhost**，它们都代表我们自己的这台电脑，在使用效果上没有任何区别。

4.3 服务器相关的概念

3. 端口号

计算机中的端口号，就好像是现实生活中的门牌号一样。通过门牌号，外卖小哥可以在整栋大楼众多的房间中，准确把外卖送到你的手中。

同样的道理，在一台电脑中，可以运行成百上千个 web 服务。每个 web 服务都对应一个唯一的端口号。客户端发送过来的网络请求，通过端口号，可以被准确地交给**对应的 web 服务**进行处理。



注意：

- ① 每个端口号不能同时被多个 web 服务占用。
- ② 在实际应用中，URL 中的 80 端口可以被省略。
†

4.4 创建最基本的 web 服务器

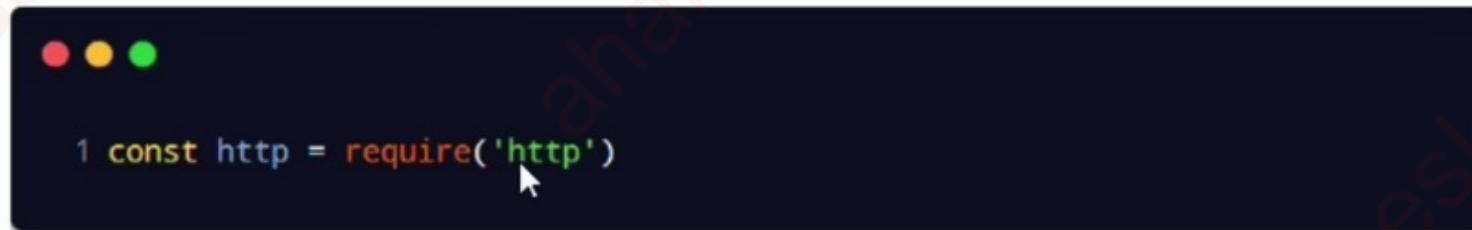
1. 创建 web 服务器的基本步骤

- ① 导入 http 模块
- ② 创建 web 服务器实例
- ③ 为服务器实例绑定 **request** 事件，监听客户端的请求
- ④ 启动服务器

4.4 创建最基本的 web 服务器

2. 步骤1 - 导入 http 模块

如果希望在自己的电脑上创建一个 web 服务器，从而对外提供 web 服务，则需要导入 http 模块：



```
1 const http = require('http')
```

A screenshot of a dark-themed terminal window. In the top-left corner, there are three small colored dots (red, yellow, green). The main area of the terminal contains a single line of code: '1 const http = require('http')'. A white cursor arrow points to the word 'http' at the end of the line.

4.4 创建最基本的 web 服务器

2. 步骤2 - 创建 web 服务器实例

调用 `http.createServer()` 方法，即可快速创建一个 web 服务器实例：

```
1 const server = http.createServer()
```

4.4 创建最基本的 web 服务器

2. 步骤3 - 为服务器实例绑定 request 事件

为服务器实例绑定 request 事件，即可监听客户端发送过来的网络请求：

```
1 // 使用服务器实例的 .on() 方法，为服务器绑定一个 request 事件
2 server.on('request', (req, res) => {
3     // 只要有客户端来请求我们自己的服务器，就会触发 request 事件，从而调用这个事件处理函数
4     console.log('Someone visit our web server.')
5 })
```

4.4 创建最基本的 web 服务器

2. 步骤4 - 启动服务器

调用服务器实例的 `.listen()` 方法，即可启动当前的 web 服务器实例：

```
1 // 调用 server.listen(端口号, cb回调) 方法, 即可启动 web 服务器
2 server.listen(80, () => {
3   console.log('http server running at http://127.0.0.1')
4 })
```

4.4 创建最基本的 web 服务器

3. req 请求对象

只要服务器接收到了客户端的请求，就会调用通过 `server.on()` 为服务器绑定的 `request` 事件处理函数。

如果想在事件处理函数中，访问与客户端相关的**数据**或**属性**，可以使用如下方式：

```
1 server.on('request', (req) => {  
2   // req 是请求对象，它包含了与客户端相关的数据和属性，例如：  
3   // req.url 是客户端请求的 URL 地址  
4   // req.method 是客户端的 method 请求类型  
5   const str = `Your request url is ${req.url}, and request method is ${req.method}`.  
6   console.log(str)  
7 })
```

4.4 创建最基本的 web 服务器

4. res 响应对象

在服务器的 request 事件处理函数中，如果想访问与服务器相关的**数据或属性**，可以使用如下的方式：

```
1 server.on('request', (req, res) => {
2   // res 是响应对象，它包含了与服务器相关的数据和属性，例如：
3   // 要发送到客户端的字符串
4   const str = `Your request url is ${req.url}, and request method is ${req.method}`
5   // res.end() 方法的作用：
6   // 向客户端发送指定的内容，并结束这次请求的处理过程
7   res.end(str)
8 })
```

4.4 创建最基本的 web 服务器

5. 解决中文乱码问题

当调用 `res.end()` 方法，向客户端发送中文内容的时候，会出现乱码问题，此时，需要手动设置内容的编码格式：

```
1 server.on('request', (req, res) => {
2   // 发送的内容包含中文
3   const str = `您请求的 url 地址是 ${req.url}, 请求的 method 类型是 ${req.method}`
4   // 为了防止中文显示乱码的问题，需要设置响应头 Content-Type 的值为 text/html; charset=utf-8
5   res.setHeader('Content-Type', 'text/html; charset=utf-8')
6   // 把包含中文的内容，响应给客户端
7   res.end(str)
8 })
```

4.5 根据不同的 url 响应不同的 html 内容

1. 核心实现步骤

- ① 获取请求的 url 地址
- ② 设置默认的响应内容为 404 Not found
- ③ 判断用户请求的是否为 / 或 /index.html 首页
- ④ 判断用户请求的是否为 /about.html 关于页面
- ⑤ 设置 Content-Type 响应头，防止中文乱码
- ⑥ 使用 res.end() 把内容响应给客户端

4.5 根据不同的 url 响应不同的 html 内容

2. 动态响应内容

```
1 server.on('request', function(req, res) {  
2   const url = req.url // 1. 获取请求的 url 地址  
3   let content = '<h1>404 Not found!</h1>' // 2. 设置默认的内容为 404 Not found  
4   if (url === '/') || url === '/index.html') {  
5     content = '<h1>首页</h1>' // 3. 用户请求的是首页  
6   } else if (url === '/about.html') {  
7     content = '<h1>关于页面</h1>' // 4. 用户请求的是关于页面  
8   }  
9   res.setHeader('Content-Type', 'text/html; charset=utf-8') // 5. 设置 Content-Type 响应头, 防止中文乱码  
10  res.end(content) // 6. 把内容发送给客户端  
11 })
```