

模块化

目录 Contents

◆ 模块化的基本概念

◆ Node.js 中模块化

◆ npm与包

◆ 模块的加载机制



1. 模块化的基本概念

1.1 什么是模块化

模块化是指解决一个复杂问题时，自顶向下逐层把系统划分成若干模块的过程。对于整个系统来说，模块是可组合、分解和更换的单元。



黑马程序员
www.itheima.com

1. 模块化的基本概念

1.1 什么是模块化

1. 现实生活中的模块化



■ 1. 模块化的基本概念

1.1 什么是模块化

1. 现实生活中的模块化





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**



1. 模块化的基本概念

1.2 模块化规范

模块化规范就是对代码进行模块化的拆分与组合时，需要遵守的那些规则。

例如：

- 使用什么样的语法格式来[引用模块](#)
- 在模块中使用什么样的语法格式[向外暴露成员](#)

模块化规范的好处：大家都遵守同样的模块化规范写代码，降低了沟通的成本，极大方便了各个模块之间的相互调用，利人利己。

目录 Contents

◆ 模块化的基本概念

◆ Node.js 中模块化

◆ npm与包

◆ 模块的加载机制

2. Node.js 中的模块化

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. Node.js 中的模块化

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')
```

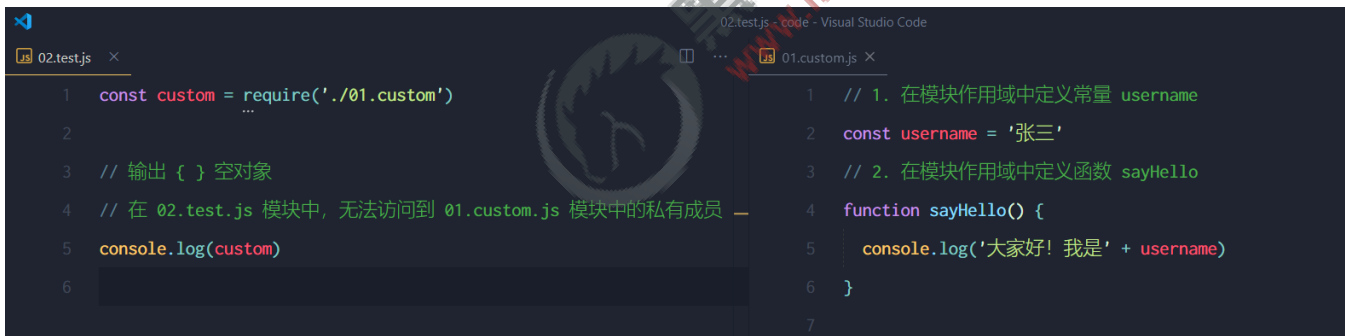
注意：使用 `require()` 方法加载其它模块时，会执行被加载模块中的代码。

2. Node.js 中的模块化

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**。



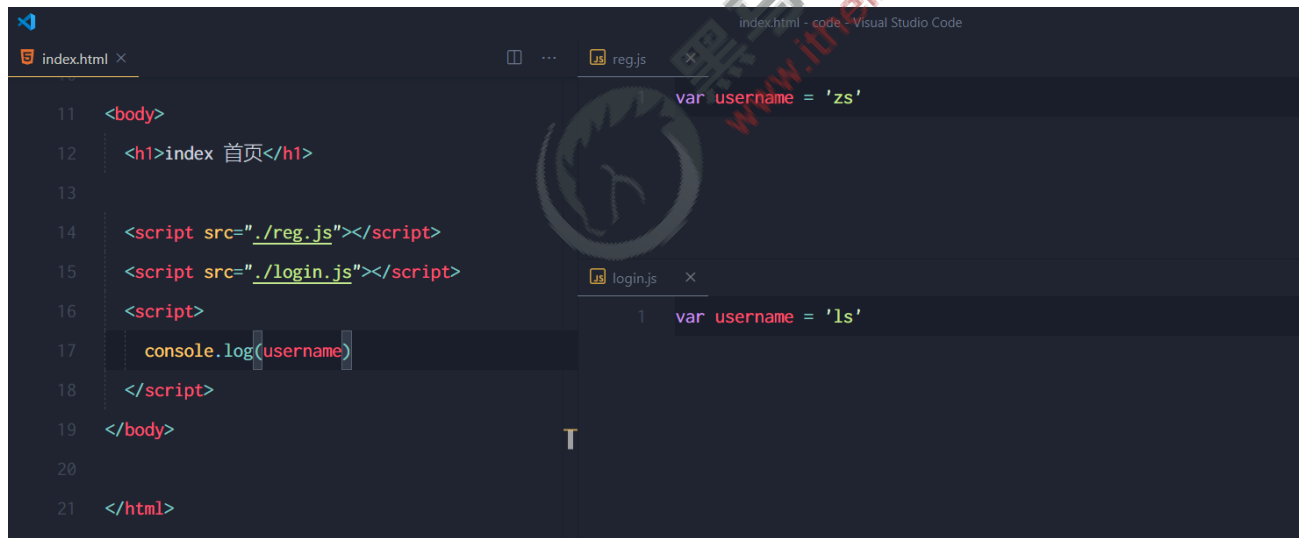
```
02.test.js - code - Visual Studio Code
01.test.js x
02.test.js x
1 const custom = require('./01.custom')
2
3 // 输出 { } 空对象
4 // 在 02.test.js 模块中，无法访问到 01.custom.js 模块中的私有成员
5 console.log(custom)
6
01.custom.js x
1 // 1. 在模块作用域中定义常量 username
2 const username = '张三'
3 // 2. 在模块作用域中定义函数 sayHello
4 function sayHello() {
5     console.log('大家好! 我是' + username)
6 }
7
```

2. Node.js 中的模块化

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



```
index.html - code - Visual Studio Code
index.html x
11 <body>
12 <h1>index 首页</h1>
13
14 <script src="./reg.js"></script>
15 <script src="./login.js"></script>
16 <script>
17   console.log(username)
18 </script>
19 </body>
20
21 </html>

reg.js
1 var username = 'zs'

login.js
1 var username = 'ls'
```

2. Node.js 中的模块化

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：



```
03.module对象.js - code - Visual Studio Code
console.log(module)

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:\\node_modules'
  ]
}
```

2. Node.js 中的模块化

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。

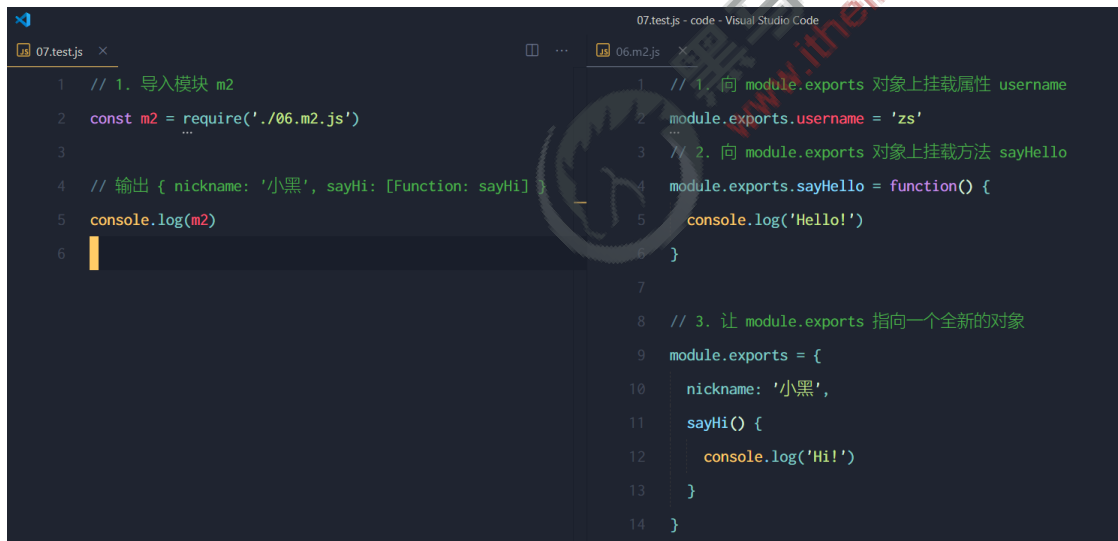


2. Node.js 中的模块化

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。**



```
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

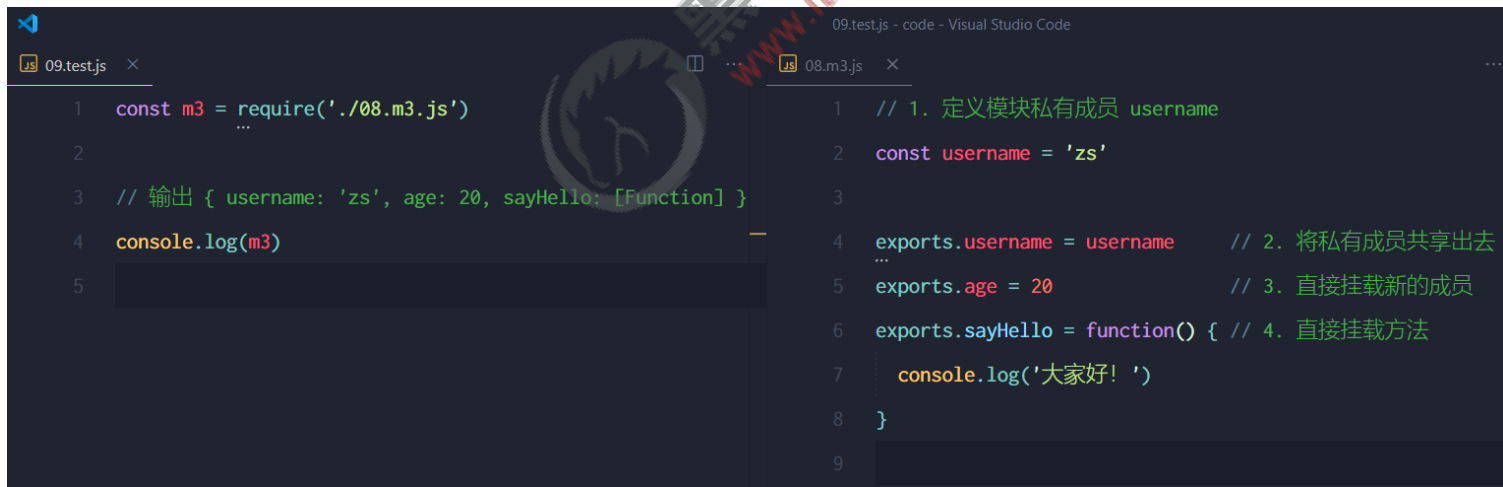
06.m2.js
1 // 1. 向 module.exports 对象上挂载属性 username
2 module.exports.username = 'zs'
3 // 2. 向 module.exports 对象上挂载方法 sayHello
4 module.exports.sayHello = function() {
5     console.log('Hello!')
6 }
7
8 // 3. 让 module.exports 指向一个全新的对象
9 module.exports = {
10     nickname: '小黑',
11     sayHi() {
12         console.log('Hi!')
13     }
14 }
```

2. Node.js 中的模块化

2.4 向外共享模块作用域中的成员

4. exports 对象

由于 `module.exports` 单词写起来比较复杂，为了简化向外共享成员的代码，Node 提供了 `exports` 对象。默认情况下，`exports` 和 `module.exports` 指向同一个对象。最终共享的结果，还是以 `module.exports` 指向的对象为准。



```
09.test.js  ×
1  const m3 = require('./08.m3.js')
2
3  // 输出 { username: 'zs', age: 20, sayHello: [Function] }
4  console.log(m3)
5

08.m3.js  ×
1  // 1. 定义模块私有成员 username
2  const username = 'zs'
3
4  exports.username = username    // 2. 将私有成员共享出去
5  exports.age = 20               // 3. 直接挂载新的成员
6  exports.sayHello = function() { // 4. 直接挂载方法
7      console.log('大家好! ')
8  }
9
```


2. Node.js 中的模块化

2.4 向外共享模块作用域中的成员

4. exports 和 module.exports 的使用误区

时刻谨记，require() 模块时，得到的永远是 `module.exports` 指向的对象：

```
exports.username = 'zs'  
...  
module.exports = {  
  gender: '男',  
  age: 22  
}
```

{ gender: '男', age: 22 }

```
module.exports.username = 'zs'  
  
exports = {  
  gender: '男',  
  age: 22  
}
```

{ username: 'zs' }

```
exports.username = 'zs'  
module.exports.gender = '男'
```

{ username: 'zs', gender: '男' }

```
exports = {  
  username: 'zs',  
  gender: '男'  
}  
module.exports = exports  
module.exports.age = '22'
```

{ username: 'zs', gender: '男', age: '22' }

注意：为了防止混乱，建议大家不要在同一模块中同时使用 `exports` 和 `module.exports`

2. Node.js 中的模块化

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。

目录 Contents

- ◆ 模块化的基本概念
- ◆ Node.js 中模块化
- ◆ npm与包
- ◆ 模块的加载机制

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. npm与包

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. npm与包

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的时，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. npm与包

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. npm与包

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

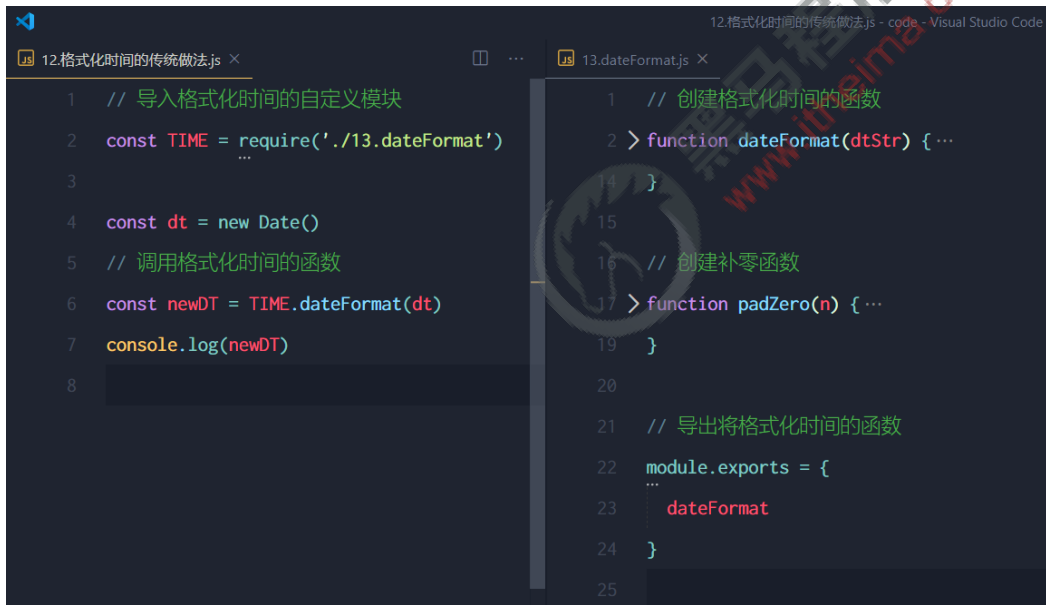
C:\Users>npm -v
6.13.2

C:\Users>
```


3. npm与包

3.2 npm 初体验

1. 格式化时间的传统做法



```
12.格式化时间的传统做法.js
1 // 导入格式化时间的自定义模块
2 const TIME = require('./13.dateFormat')
3
4 const dt = new Date()
5 // 调用格式化时间的函数
6 const newDT = TIME.dateFormat(dt)
7 console.log(newDT)
8

13.dateFormat.js
1 // 创建格式化时间的函数
2 > function dateFormat(dtStr) { ...
14 }
15
16 // 创建补零函数
17 > function padZero(n) { ...
19 }
20
21 // 导出将格式化时间的函数
22 module.exports = {
23   ...
24   dateFormat
25 }
```

- 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. npm与包

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. npm与包

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. npm与包

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。

注意：程序员不要手动修改 `node_modules` 或 `package-lock.json` 文件中的任何代码，npm 包管理工具会自动维护它们。

3. npm与包

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：

```
1 npm i moment@2.22.2
```

3. npm与包

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. npm与包

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：

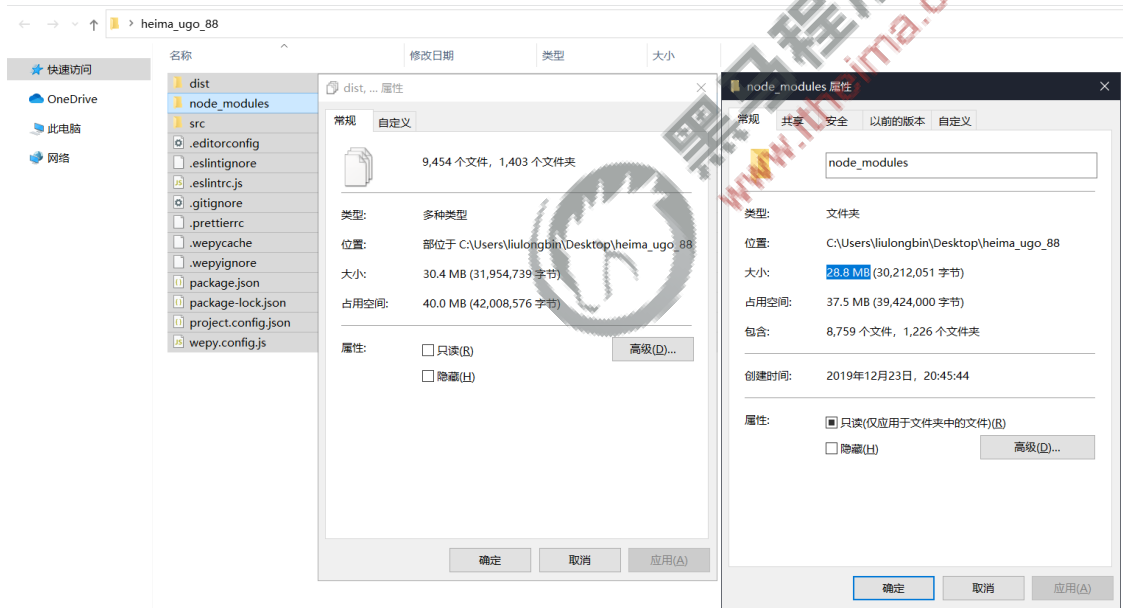
- 项目的名称、版本号、描述等
- 项目中都用到了哪些包
- 哪些包只在**开发期间**会用到
- 那些包在**开发**和**部署**时都需要用到



3. npm与包

3.3 包管理配置文件

1. 多人协作的问题



整个项目的体积是 30.4M

第三方包的体积是 28.8M

项目源代码的体积 1.6M

遇到的问题：第三方包的体积过大，不方便团队成员之间共享项目源代码。

解决方案：共享时剔除node_modules

3. npm与包

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. npm与包

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. npm与包

3.3 包管理配置文件

4. **dependencies** 节点



```
1 {
2   "name": "my-project",
3   "version": "1.0.0",
4   "description": "",
5   "main": "",
6   "scripts": {},
7   "keywords": [],
8   "author": "",
9   "license": "ISC",
10  "dependencies": {
11    "moment": "^2.24.0"
12  }
13 }
```

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. npm与包

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. npm与包

3.3 包管理配置文件

5. 一次性安装所有的包

可以运行 `npm install` 命令（或 `npm i`）一次性安装所有的依赖包：

```
1 // 执行 npm install 命令时, npm 包管理工具会先读取 package.json 中的 dependencies 节点,
2 // 读取到记录的所有依赖包名称和版本号之后, npm 包管理工具会把这些包一次性下载到项目中
3 npm install
```

3. npm与包

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. npm与包

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
```

3. npm与包

3.4 解决下包速度慢的问题

1. 为什么下包速度慢

在使用 npm 下包的时候，默认从国外的 <https://registry.npmjs.org/> 服务器进行下载，此时，网络数据的传输需要经过漫长的海底光缆，因此下包速度会很慢。

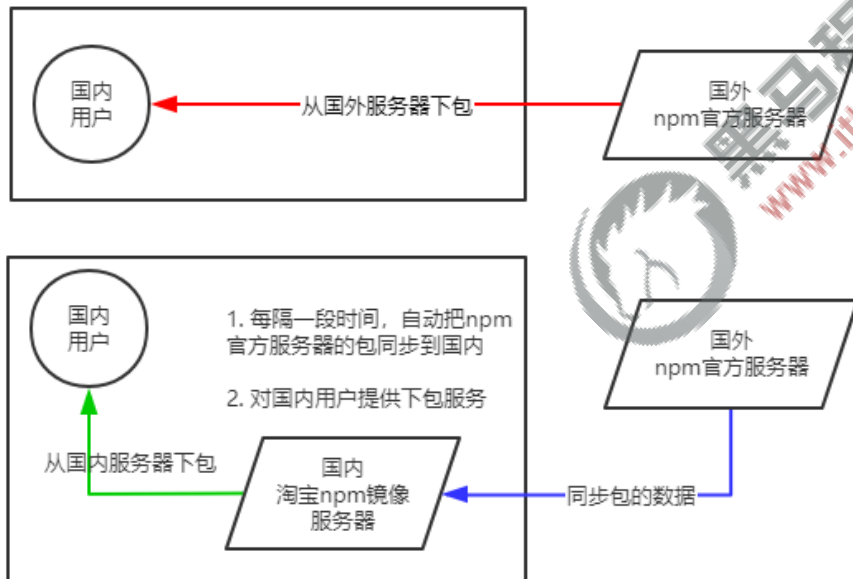
扩展阅读 - 海底光缆：

- <https://baike.baidu.com/item/%E6%B5%B7%E5%BA%95%E5%85%89%E7%BC%86/4107830>
- <https://baike.baidu.com/item/%E4%B8%AD%E7%BE%8E%E6%B5%B7%E5%BA%95%E5%85%89%E7%BC%86/10520363>
- <https://baike.baidu.com/item/APG/23647721?fr=aladdin>

3. npm与包

3.4 解决下包速度慢的问题

2. 淘宝 NPM 镜像服务器



淘宝在国内搭建了一个服务器, 专门把国外官方服务器上的包同步到国内的服务器, 然后在国内提供下包的服务。从而极大的提高了下包的速度。

扩展:

镜像 (Mirroring) 是一种文件存储形式, 一个磁盘上的数据在另一个磁盘上存在一个完全相同的副本即为镜像。

3. npm与包

3.4 解决下包速度慢的问题

3. 切换 npm 的下包镜像源

下包的镜像源，指的就是下包的服务器地址。

```
1 # 查看当前的下包镜像源
2 npm config get registry
3 # 将下包的镜像源切换为淘宝镜像源
4 npm config set registry=https://registry.npm.taobao.org/
5 # 检查镜像源是否下载成功
6 npm config get registry
```

3. npm与包

3.4 解决下包速度慢的问题

4. nrm

为了更方便的切换下包的镜像源，我们可以安装 **nrm** 这个小工具，利用 nrm 提供的终端命令，可以快速查看和切换下包的镜像源。

```
1 # 通过 npm 包管理器, 将 nrm 安装为全局可用的工具
2 npm i nrm -g
3 # 查看所有可用的镜像源
4 nrm ls
5 # 将下包的镜像源切换为 taobao 镜像
6 nrm use taobao
```

3. npm与包

3.5 包的分类

使用 npm 包管理工具下载的包，共分为两大类，分别是：

- 项目包
- 全局包



3. npm与包

3.5 包的分类

1. 项目包

那些被安装到项目的 `node_modules` 目录中的包，都是项目包。

项目包又分为两类，分别是：

- **开发依赖包**（被记录到 `devDependencies` 节点中的包，只在开发期间会用到）
- **核心依赖包**（被记录到 `dependencies` 节点中的包，在开发期间和项目上线之后都会用到）

```
1 npm i 包名 -D # 开发依赖包（会被记录到 devDependencies 节点下）
2 npm i 包名    # 核心依赖包（会被记录到 dependencies 节点下）
```

3. npm与包

3.5 包的分类

2. 全局包

在执行 `npm install` 命令时，如果提供了 `-g` 参数，则会把包安装为全局包。

全局包会被安装到 `C:\Users\用户目录\AppData\Roaming\npm\node_modules` 目录下。

```
1 npm i 包名 -g      # 全局安装指定的包
2 npm uninstall 包名 -g # 卸载全局安装的包
```

注意：

- ① 只有工具性质的包，才有全局安装的必要性。因为它们提供了好用的终端命令。
- ② 判断某个包是否需要全局安装后才能使用，可以参考官方提供的使用说明即可。

3. npm与包

3.5 包的分类

3. i5ting_toc

i5ting_toc 是一个可以把 md 文档转为 html 页面的小工具，使用步骤如下：

```
1 # 将 i5ting_toc 安装为全局包
2 npm install -g i5ting_toc
3 # 调用 i5ting_toc, 轻松实现 md 转 html 的功能
4 i5ting_toc -f 要转换的md文件路径 -o
```

3. npm与包

3.6 规范的包结构

在清楚了包的概念、以及如何下载和使用包之后，接下来，我们深入了解一下包的内部结构。

一个规范的包，它的组成结构，必须符合以下 3 点要求：

- ① 包必须以单独的目录而存在
- ② 包的顶级目录下必须包含 `package.json` 这个包管理配置文件
- ③ `package.json` 中必须包含 `name`，`version`，`main` 这三个属性，分别代表包的名字、版本号、包的入口。

注意：以上 3 点要求是一个规范的包结构必须遵守的格式，关于更多的约束，可以参考如下网址：

<https://yarnpkg.com/zh-Hans/docs/package-json>

■ 3. npm与包

3.7 开发属于自己的包

1. 需要实现的功能

- ① 格式化日期
- ② 转义 HTML 中的特殊字符
- ③ 还原 HTML 中的特殊字符



3. npm与包

3.7 开发属于自己的包

1. 需要实现的功能

```
1 // 1. 导入自己的包
2 const itheima = require('itehima-utils')
3
4 // ----- 功能1: 格式化日期-----
5 const dt = itheima.dateFormat(new Date())
6 // 输出 2020-01-20 10:09:45
7 console.log(dt)
```

3. npm与包

3.7 开发属于自己的包

1. 需要实现的功能

```
1 // 1. 导入自己的包
2 const itheima = require('itheima-utils')
3
4 // ----- 功能2: 转义 HTML 中的特殊字符-----
5 const htmlStr = '<h1 style="color: red;">你好! &copy;<span>小黄! </span></h1>'
6 const str = itheima.htmlEscape(htmlStr)
7 // &lt;h1 style=&quot;color: red;&quot;&gt;你好! &amp;copy;&lt;span&gt;小黄! &lt;/span&gt;&lt;/h1&gt;
8 console.log(str)
```

3. npm与包

3.7 开发属于自己的包

1. 需要实现的功能

```
1 // 1. 导入自己的包
2 const itheima = require('itehima-utils')
3
4 // ----- 功能3: 还原 HTML 中的特殊字符-----
5 const rawHTML = itheima.htmlUnEscape(str)
6 // 输出 <h1 style="color: red;">你好! &copy;;<span>小黄! </span></h1>
7 console.log(rawHTML)
```

3. npm与包

3.7 开发属于自己的包

2. 初始化包的基本结构

- ① 新建 itheima-tools 文件夹，作为包的根目录
- ② 在 itheima-tools 文件夹中，新建如下三个文件：
 - package.json （包管理配置文件）
 - index.js （包的入口文件）
 - README.md （包的说明文档）

3. npm与包

3.7 开发属于自己的包

3. 初始化 package.json

```
1 {  
2   "name": "itheima-tools",  
3   "version": "1.0.0",  
4   "main": "index.js",  
5   "description": "提供了格式化时间, HTMLEscape的功能",  
6   "keywords": ["itheima", "dateFormat", "escape"],  
7   "license": "ISC"  
8 }
```

关于更多 license 许可协议相关的内容, 可参考 <https://www.jianshu.com/p/86251523e898>

3. npm与包

3.7 开发属于自己的包

4. 在 index.js 中定义格式化时间的方法

```
1 // 格式化时间的方法
2 function dateFormat(dateStr) { /* 省略其余代码 */ }
3
4 // 补零的方法
5 function padZero(n) {
6   return n > 9 ? n : '0' + n
7 }
8
9 module.exports = {
10   dateFormat
11 }
```

3. npm与包

3.7 开发属于自己的包

5. 在 index.js 中定义转义 HTML 的方法

```
1 function htmlEscape(htmlStr) {  
2   return htmlStr.replace(/<|>|"|'/g, (match) => {  
3     switch (match) {  
4       case '<':  
5         return '&lt;';  
6       case '>':  
7         return '&gt;';  
8       case '"':  
9         return '&quot;';  
10      case "'":  
11        return '&apos;';  
12      }  
13    })  
14 }
```


3. npm与包

3.7 开发属于自己的包

6. 在 index.js 中定义还原 HTML 的方法

```
1 function htmlUnEscape(str) {  
2   return str.replace(/&lt;|&gt;|&quot;|&amp;/g, (match) => {  
3     switch (match) {  
4       case '&lt;':  
5         return '<'  
6       case '&gt;':  
7         return '>'  
8       case '&quot;':  
9         return '"'  
10      case '&amp;':  
11        return '&'  
12      }  
13    })  
14 }
```

3. npm与包

3.7 开发属于自己的包

7. 将不同的功能进行模块化拆分

- ① 将格式化时间的功能，拆分到 src -> `dateFormat.js` 中
- ② 将处理 HTML 字符串的功能，拆分到 src -> `htmlEscape.js` 中
- ③ 在 `index.js` 中，导入两个模块，得到需要向外共享的方法
- ④ 在 `index.js` 中，使用 `module.exports` 把对应的方法共享出去

■ 3. npm与包

3.7 开发属于自己的包

8. 编写包的说明文档

包根目录中的 **README.md** 文件，是**包的使用说明文档**。通过它，我们可以事先把包的使用说明，以 markdown 的格式写出来，方便用户参考。

README 文件中具体写什么内容，没有强制性的要求；只要能够清晰地把包的作用、用法、注意事项等描述清楚即可。

我们所创建的这个包的 README.md 文档中，会包含以下 6 项内容：

安装方式、导入方式、格式化时间、转义 HTML 中的特殊字符、还原 HTML 中的特殊字符、开源协议

■ 3. npm与包

3.8 发布包

1. 注册 npm 账号

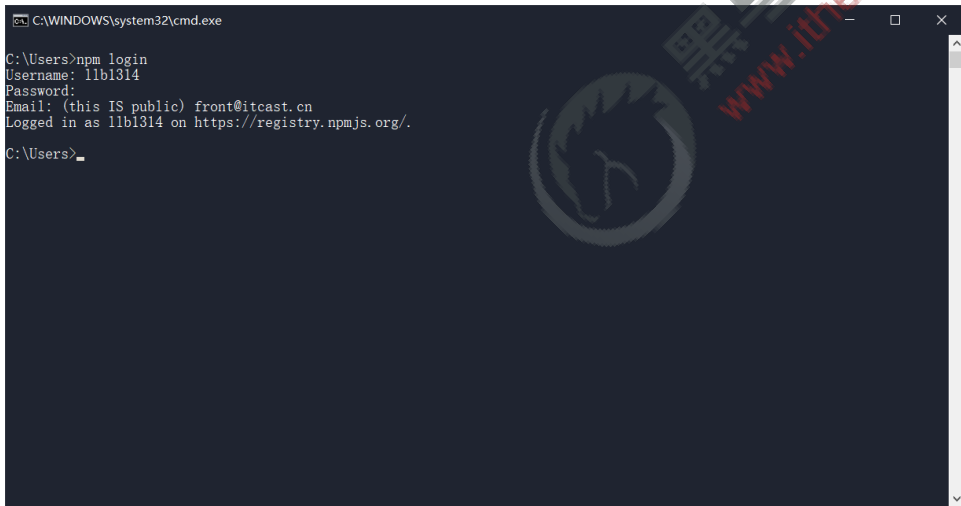
- ① 访问 <https://www.npmjs.com/> 网站，点击 **sign up** 按钮，进入注册用户界面
- ② 填写账号相关的信息：Full Name、Public Email、Username、Password
- ③ 点击 **Create an Account** 按钮，注册账号
- ④ 登录邮箱，**点击验证链接**，进行账号的验证

3. npm与包

3.8 发布包

2. 登录 npm 账号

npm 账号注册完成后，可以在终端中执行 **npm login** 命令，依次输入用户名、密码、邮箱后，即可登录成功。



```
C:\WINDOWS\system32\cmd.exe
C:\Users>npm login
Username: llb1314
Password:
Email: (this IS public) front@itcast.cn
Logged in as llb1314 on https://registry.npmjs.org/.
C:\Users>_
```

注意：在运行 npm login 命令之前，必须先把**下包的服务器**地址切换为 **npm 的官方服务器**。否则会导致发布包失败！

3. npm与包

3.8 发布包

3. 把包发布到 npm 上

将终端切换到包的根目录之后，运行 `npm publish` 命令，即可将包发布到 npm 上（注意：包名不能雷同）。

```
C:\Users\liulongbin\Desktop\itheima-utils1>npm publish
npm notice
npm notice package: itheima-utils1@1.0.1
npm notice === Tarball Contents ===
npm notice 677B src/dateFormat.js
npm notice 741B src/htmlEscape.js
npm notice 349B index.js
npm notice 229B package.json
npm notice 816B README.md
npm notice === Tarball Details ===
npm notice name: itheima-utils1
npm notice version: 1.0.1
npm notice package size: 1.4 kB
npm notice unpacked size: 2.8 kB
npm notice shasum: 4683fd9e9f14e8a8656a7ebfa46c59e576525dcf
npm notice integrity: sha512-b0mS3ZPe2vxAu[...]g2MNxaJLYePFA==
npm notice total files: 5
npm notice
+ itheima-utils1@1.0.1
C:\Users\liulongbin\Desktop\itheima-utils1>
```

3. npm与包

3.8 发布包

4. 删除已发布的包

运行 `npm unpublish 包名 --force` 命令，即可从 npm 删除已发布的包。

```
C:\Users\liulongbin\Desktop\itheima-utils>npm unpublish itheima-utils --force
npm WARN using --force I sure hope you know what you are doing.
- itheima-utils
```

注意：

- ① `npm unpublish` 命令只能删除 72 小时以内发布的包
- ② `npm unpublish` 删除的包，在 24 小时内不允许重复发布
- ③ 发布包的时候要慎重，尽量不要往 npm 上发布没有意义的包！

目录 Contents

- ◆ 模块化的基本概念
- ◆ Node.js 中模块化
- ◆ npm与包
- ◆ 模块的加载机制

4. 模块的加载机制

4.1 优先从缓存中加载

模块在第一次加载后会被缓存。这也意味着多次调用 `require()` 不会导致模块的代码被执行多次。

注意：不论是内置模块、用户自定义模块、还是第三方模块，它们都会优先从缓存中加载，从而提高模块的加载效率。



4. 模块的加载机制

4.2 内置模块的加载机制

内置模块是由 Node.js 官方提供的模块，**内置模块的加载优先级最高。**

例如，`require('fs')` 始终返回内置的 `fs` 模块，即使在 `node_modules` 目录下有名字相同的包也叫做 `fs`。



4. 模块的加载机制

4.3 自定义模块的加载机制

使用 `require()` 加载自定义模块时，必须指定以 `./` 或 `../` 开头的**路径标识符**。在加载自定义模块时，如果没有指定 `./` 或 `../` 这样的路径标识符，则 node 会把它当作**内置模块**或**第三方模块**进行加载。

同时，在使用 `require()` 导入自定义模块时，如果省略了文件的扩展名，则 Node.js 会**按顺序**分别尝试加载以下的文件：

- ① 按照**确切的文件名**进行加载
- ② 补全 **.js** 扩展名进行加载
- ③ 补全 **.json** 扩展名进行加载
- ④ 补全 **.node** 扩展名进行加载
- ⑤ 加载失败，终端报错



4. 模块的加载机制

4.4 第三方模块的加载机制

如果传递给 `require()` 的模块标识符不是一个内置模块，也没有以 `'/'` 或 `'../'` 开头，则 Node.js 会从当前模块的父目录开始，尝试从 `/node_modules` 文件夹中加载第三方模块。

如果没有找到对应的第三方模块，则移动到再上一层父目录中，进行加载，直到文件系统的根目录。

例如，假设在 `'C:\Users\itheima\project\foo.js'` 文件里调用了 `require('tools')`，则 Node.js 会按以下顺序查找：

- ① `C:\Users\itheima\project\node_modules\tools`
- ② `C:\Users\itheima\node_modules\tools`
- ③ `C:\Users\node_modules\tools`
- ④ `C:\node_modules\tools`

4. 模块的加载机制

4.5 目录作为模块

当把目录作为模块标识符，传递给 `require()` 进行加载的时候，有三种加载方式：

- ① 在被加载的目录下查找一个叫做 `package.json` 的文件，并寻找 `main` 属性，作为 `require()` 加载的入口
- ② 如果目录里没有 `package.json` 文件，或者 `main` 入口不存在或无法解析，则 Node.js 将会试图加载目录下的 `index.js` 文件。
- ③ 如果以上两步都失败了，则 Node.js 会在终端打印错误消息，报告模块的缺失：Error: Cannot find module 'xxx'





黑马程序员

www.itheima.com

传智播客旗下高端IT教育品牌

