模块化

模块化概念

- 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为若干模块的过程，模块是可组合、分解和更换的单元。
- 模块化可提高代码的复用性和可维护性，实现按需加载。
- 模块化规范是对代码进行模块化拆分和组合时需要遵守的规则，如使用何种语法格式引用模块和向外暴露成员。

Node.js 中的模块化

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)

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. 加载内置的fs模块
const fs = require('fs')
//2. 加载用户的自定义模块。需要相对路径
const custom = require(' ./custom.js')
// 3. 加载第三方模块(关于第三方模块的下载和使用，后面详讲)
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。

Node.js中的模块作用域

- 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。
- 能够防止全局变量污染的问题

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。
    - 在js文件中写入`console.log(module)`即可打印module。
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。

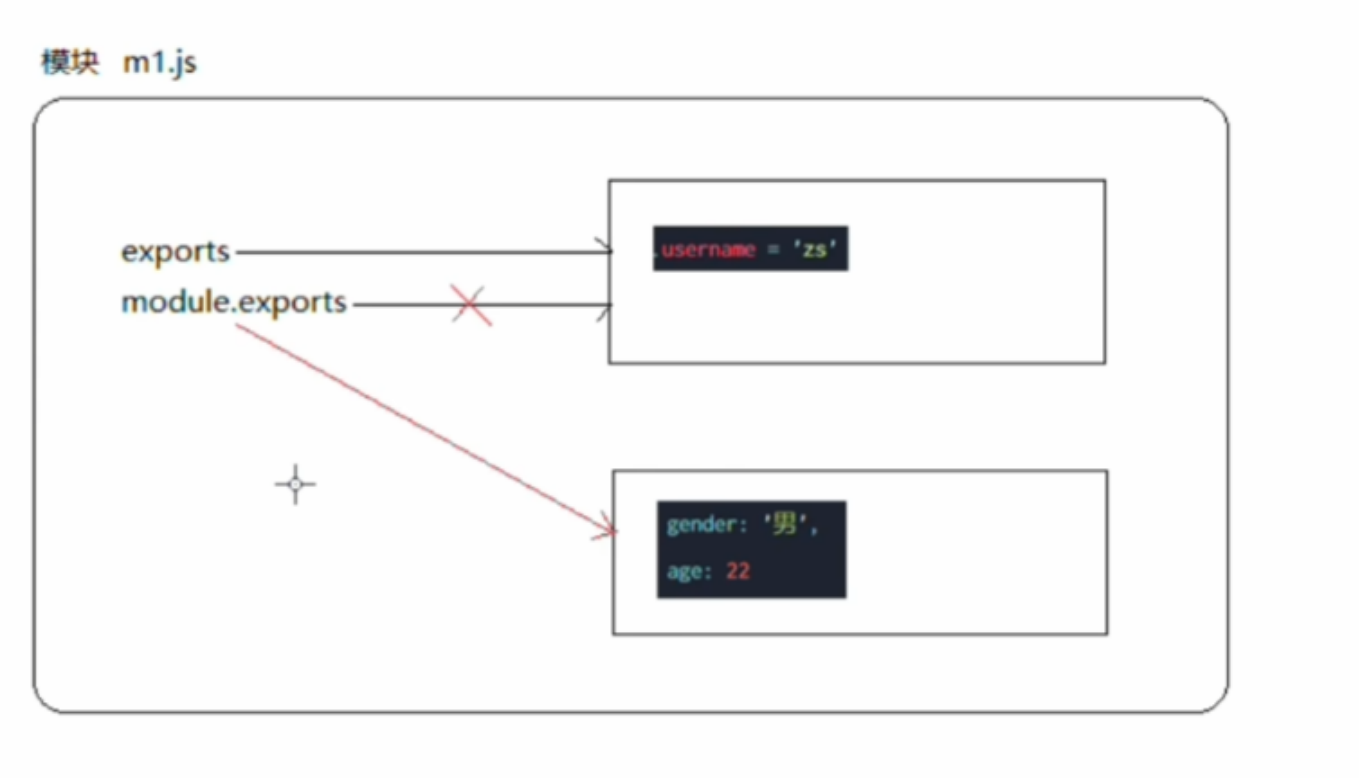
- 外界用`require()`方法导入自定义模块时，得到的就是`module.exports`所指向的对象。
- 使用`require()`方法导入模块时，导入的结果，永远以`module.exports`指向的对象为准。

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。

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。



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模块化规范：

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，`module.exports`是该对象的属性也是对外的接口
3. 加载某个模块即加载该模块的`module.exports`属性。`require()`方法用于加载模块。

npm与包

包

- Node.js 中的第三方模块又叫做包。
- 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。
- 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。 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!到目前为止，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。

包下载

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://reqistry.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。

- 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

```
//终端指令 查看版本号
npm -v
```

在项目中安装包

```
npm install 包的名称
npm i 包的名称
```

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。

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:

```
npm i moment@2.22.2
```

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的含义如下:

- 第1位数字：大版本（底层）
- 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。

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
- 哪些包在开发和部署时都需要用到 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。

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。在新建文件夹后先不要写代码，先执行生成这个文件。

```
//作用:在执行命令所处的目录中，快速新建package.json文件
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。

dependencies节点

`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。

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

```
//执行npm install命令时，npm包管理工具会先读取package.json 中的dependencies节点，
//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
npm install
```

卸载包

可以运行`npm uninstall`命令，来卸载指定的包。 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。

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。

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

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。

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。 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。

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。

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

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。

```
#通过npm包管理器，将nrm安装为全局可用的工具
npm i nrm -g
#查看所有可用的镜像源
nrm ls
```

```
#将下包的镜像源切换为taobao镜像
nrm use taobao
```

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(那些被安装到项目的node_modules目录中的包，都是项目包。)
  - 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)
- 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。) 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。

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:

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

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:包必须以单独的目录而存在 包的顶级目录下要必须包含package.json这个包管理配置文件 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。

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。包的README.md文档中，一般会包含以下内容:安装方式、导入方式、具体功能、开源协议。

模块的加载机制

模块第一次加载后会被缓存，即多次调用require()不会导致模块的代码被执行多次，提高模块加载效率。

内置模块加载

内置模块加载优先级最高。

自定义模块加载

加载自定义模块时，路径要以`./`或`../`开头，否则会作为内置模块或第三方模块加载。

导入自定义模块时，若省略文件扩展名，则 Node.js 会按顺序尝试加载文件：

- 按确切的文件名加载
- 补全 `.js` 扩展名加载
- 补全 `.json` 扩展名加载
- 补全 `.node` 扩展名加载
- 报错

第三方模块加载

- 若导入第三方模块,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\bruce\project\foo.js`文件里调用了`require('tools')`，则 Node.js 会按以下顺序查找：
- `C:\Users\bruce\project\node_modules\tools`
- `C:\Users\bruce\node_modules\tools`
- `C:\Users\node_modules\tools`
- `C:\node_modules\tools`

目录作为模块加载

当把目录作为模块标识符进行加载的时候，有三种加载方式：

1. 在被加载的目录下查找`package.json`的文件，并寻找`main`属性，作为`require()`加载的入口。
2. 如果没有`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. 若失败则报错。