Node.js

Content

Reference

1. Node.js 介绍

1. 概念：Node.js 是一个开源的，跨平台的 JavaScript 运行环境。
2. 作用
    - 开发服务器应用
    - 开发工具类应用：如 Webpack、Vite、Babel
    - 开发桌面端应用：如 VSCode
3. Node.js 下载地址

    > p.s. 可以在命令行通过 `node -v` 检查是否安装成功

4. 使用 Node.js 运行 JavaScript 代码：`node xxx.js`
5. Node.js 中不能使用 BOM 和 DOM 对象
    - 浏览器中的 JavaScript
        - 核心语法：ECMAScript
        - Web API
            - DOM、BOM、AJAX、Storage、alert/confirm
            - console、定时器

        - .....
    - Node.js 中的 JavaScript
        - 核心语法：ECMAScript
        - Node API
            - fs、url、http、util、path
            - console、定时器
            - ......
    - 说明
        - Node.js 和 浏览器共有支持的 API 是 console 和定时器
        - Node.js 中的顶级对象是 `global` ，也可以使用 `globalThis` 访问；浏览器中的顶级对象是 `window`

2. Buffer

1. 概念：Buffer 是一个**类似于数组**的对象，表示**固定长度**的**字节（byte）序列**，其本质是一段内存空间，专用于处理**二进制数据。**
2. 特点
    - 大小固定无法修改
    - 性能较好，直接操作计算机内存
    - 每个元素的大小为 `1 byte`
3. Buffer 的简单使用
    - Buffer 的创建
        - `Buffer.alloc(n)`
        - `Buffer.allocUnsafe(n)`
        - `Buffer.from(obj)`
    - Buffer 与字符串转化
        - Buffer => String: `bufObj.toString()`
        - String => Buffer: `Buffer.from(strObj)`
    - Buffer 的读写
        - 读：`bufObj[index]`
        - 写：`bufObj[index] = newValue`

3. fs 模块

3.1 简要介绍

1. 概念：`fs` 全称为 `file system`，翻译为文件系统，是 `Node.js` 中的**内置模块**，可以实现与硬盘的交互，如文件的创建、删除、重命名、移动；文件内容的写入、读取；文件夹的相关操作
2. 相关操作
    - 文件写入
    - 文件读取
    - 文件移动与重命名

    - 文件删除
    - 文件夹操作
    - 查看资源状态
3. 模块的导入 `require('fs')`

3.2 文件的写入

文件写入就是将**数据**保存到**文件**中，可以使用如下四种方式来实现文件写入。

1. 异步写入 `fs.writeFile(file, data[, option], err => {}): Undefined`
2. 同步写入 `fs.writeFileSync(file, data[, options]): Undefined`
3. 追加写入
    - 异步追加写入 `fs.appendFile(file, data[, options], err => {}): Undefined`
    - 同步追加写入 `fs.appendFile(file, data[, options]): Undefined`
4. 流式写入
    - 创建流对象 `fs.createWriteStream(file[, option]): Object`
    - 向流中写入 `ws.write(data): Undefined`
    - 关闭流对象 `ws.end([() => {}]): Undefined`

3.3 文件的读取

文件读取就是从**文件**中提取出其中的**数据**，可以使用如下三种方式实现文件读取。

1. 异步读取 `fs.readFile(file[, options], (err, data) => {}): Undefined`
2. 同步读取 `fs.readFileSync(file[, options]): Object(Buffer)`
3. 流式读取
    - 读取流对象 `fs.createReadStream(file[, options]): Object`
    - 从流中读取 `rs.on('data', chunk => {}): Undefined`
    - 关闭流对象 `rs.on('end', () => {})): Undefined`

3.4 文件/文件夹的移动/重命名

关于移动/重命名文件/文件夹的操作，Node.js 有两种实现

- 异步实现 `fs.rename(oldPath, newPath, err => {})`
- 同步实现 `fs.renameSync(oldPath, newPath)`

3.5 文件删除

关于删除文件的操作，Node.js 也提供了两种实现

- 异步实现
    - `fs.unlink(path, err => {})`
    - `fs.rm(path, err => {})`
- 同步实现
    - `fs.unlinkSync(path)`

- `fs.rmSync(path)`

3.6 文件夹操作

1. 文件夹的创建
   - 异步实现 `fs.mkdir(path[, options], (err) => {})`
   - 同步实现 `fs.mkdirSync(path[, options])`
2. 文件夹的读取
   - 异步实现 `fs.readdir(path[, options], (err, data) => {})`
   - 同步实现 `fs.readdirSync(path[, options]): Array`
3. 文件夹的删除
   - 异步实现 `fs.rmdir(path[, options], err => {})`
   - 同步实现 `fs.rmdirSync(path[, options])`

> p.s. 我们可以通过设置 `options={recursive: true}` 的方式实现文件夹的递归创建与递归删除，不过实现递归删除时，更推荐使用 `rm/rmSync` 方法

3.7 查看资源状态

关于查看资源状态的操作，Node.js 提供了两种实现

- 异步实现 `fs.stat(path[, options], (err, data) => {})`
- 同步实现 `fs.stat(path[, options]): data`

3.8 相对路径与绝对路径

1. 在使用 fs 模块对资源进行操作时，路径通常有**绝对路径**和**相对路径**两种选择，规则如下
   - 相对路径
     - `./xxx.txt` 当前目录下的 xxx.txt 文件
     - `xxx.txt` 当前目录下的 xxx.txt 文件
     - `../xxx.txt` 当前目录下的上一级目录中的 xxx.txt 文件
   - 绝对路径
     - `D:/xxx.txt` 表示 D 盘根路径下的 xxx.txt 文件（windows 系统下的绝对路径）
     - `/xxx.txt` 表示当前根目录下的 xxx.txt 文件（Linux 系统下的绝对路径）

   > p.s. fs 中的绝对路径和相对路径的写法与 url 中的写法略有差异

2. 因为 fs 中的相对路径参考的不是当前文件的所在目录，而是命令行中的工作目录，所以在使用相对路径时，需要使用 `__dirname` 来对相对路径进行修正，使其变成绝对路径，从而避免 bug 的出现
   - `绝对路径 = __dirname + 相对路径`

4. path 模块

path 模块提供了操作路径的功能，有如下几个常用 API

API	解释
path.resolve(绝对路径, 相对路径)	根据绝对路径和相对路径拼接出一个规范的绝对路径

API	解释
path.sep	获取当前操作系统的路径分隔符
path.parse(路径)	解析当前路径，返回路径对象
path.basename(路径)	解析当前路径，返回路径对象的名称（文件名）
path.dirname(路径)	解析当前路径，返回路径对象的所在目录
path.extname(路径)	解析当前路径，返回路径对象的扩展名

5. http 模块

5.1 HTTP 协议

1. 概念：**HTTP**（hyper transport protocol）协议，译为**超文本传输协议**，是一种基于 TCP/IP 的应用层通信协议，详细规定了**浏览器和服务器**之间的通信规则。

   > p.s. 协议：双方必须共同遵从的一组约定

2. HTTP 协议的主要内容
   - **请求**：**浏览器向服务器**发送数据，该数据称之为**请求报文**
   - **响应**：**服务器向浏览器**发送数据，该数据称之为**响应报文**

   > p.s. 报文：简单理解为一堆字符串

3. 报文查看与举例
   - 可以使用 `fiddler` 软件查看浏览器向服务器发送的请求报文和服务器向浏览器发送的响应报文，该软件处于浏览器和服务器之间，接受浏览器的请求并发送给服务器，接受服务器的响应并发送给浏览器
   - 当我们在浏览器访问 www.baidu.com 时，第一个请求报文与对应的响应报文内容如下
     - 请求报文（Request）

```
   GET https://www.baidu.com/ HTTP/2
   host: www.baidu.com
   sec-ch-ua: "Google Chrome";v="123", "Not:A-Brand";v="8",
"Chromium";v="123"
   sec-ch-ua-mobile: ?0
   sec-ch-ua-platform: "Windows"
   upgrade-insecure-requests: 1
   user-agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/123.0.0.0
Safari/537.36
   accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,ima
ge/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
   sec-fetch-site: none
   sec-fetch-mode: navigate
   sec-fetch-user: ?1
   sec-fetch-dest: document
   accept-encoding: gzip, deflate, br, zstd
   accept-language: zh-CN,zh;q=0.9
```

```
- 响应报文
```html
HTTP/1.1 200 OK
Content-Encoding: gzip
Content-Security-Policy: frame-ancestors 'self' https://chat.baidu.com
http://mirror-chat.baidu.com https://fj-chat.baidu.com https://hba-chat.baidu.com
https://hbe-chat.baidu.com https://njjs-chat.baidu.com https://nj-chat.baidu.com
https://hna-chat.baidu.com https://hnb-chat.baidu.com http://debug.baidu-int.com;
Content-Type: text/html; charset=utf-8
Date: Sat, 13 Apr 2024 04:35:33 GMT
P3p: CP=" OTI DSP COR IVA OUR IND COM "
P3p: CP=" OTI DSP COR IVA OUR IND COM "
Server: BWS/1.1
Set-Cookie: BAIDUID=A8FE4EF62D60DAF270D0C07E9A4CEA5C:FG=1; expires=Thu,
31-Dec-37 23:55:55 GMT; max-age=2147483647; path=/; domain=.baidu.com
Set-Cookie: BIDUPSID=A8FE4EF62D60DAF270D0C07E9A4CEA5C; expires=Thu, 31-
Dec-37 23:55:55 GMT; max-age=2147483647; path=/; domain=.baidu.com
Set-Cookie: PSTM=1712982933; expires=Thu, 31-Dec-37 23:55:55 GMT; max-
age=2147483647; path=/; domain=.baidu.com
Set-Cookie: BAIDUID=A8FE4EF62D60DAF266BD0F16045E437C:FG=1; max-
age=31536000; expires=Sun, 13-Apr-25 04:35:33 GMT; domain=.baidu.com; path=/;
version=1; comment=bd
Traceid: 1712982933370247066618326354761719158631
X-Ua-Compatible: IE=Edge,chrome=1
X-Xss-Protection: 1;mode=block
Content-Length: 102530

<!DOCTYPE html><!--STATUS OK--><html><head><meta http-equiv="Content-
Type" content="text/html;charset=utf-8">
<!-- 以下内容省略 -->
```
```

请求报文

请求报文由**请求行、请求头、空行、请求体**组成

1. 请求行
    - 请求行包含三部分内容：**请求方法、请求 URL、HTTP 协议版本号**
    - 请求方法：用于说明请求目的，常用 **GET** 和 **POST**

方法	作用
GET	获取数据
POST	新增数据
PUT/PATCH	更新数据
DELETE	删除数据
HEAD/OPTIONS/CONNECT/TRACE	使用较少，不做介绍

    - 请求 URL：Uniform Resource Locator，翻译为统一资源定位符，用于定位服务器中的资源

        - URL 包括：**协议名称、主机名（可以是域名或 IP 地址）、端口号、路径、查询字符串、锚点链接**
        - URL 举例： http://www.baidu.com:80/index.html?a=100&b=200#logo
        - 协议名：如上述例子中的 `http`，协议名有 `http`、`https`、`ftp`、`ssh` 等
        - 主机名：如上述例子中的 `www.baidu.com`，除了域名，主机名也可以是 IP 地址，用于定位一台服务器
        - 端口号：如上述例子中的 `80`
        - 路径：如上述例子中的 `/index.html`，用于定位一台服务器中的某一个资源
        - 查询字符串：如上述例子中的 `a=100&b=200`，是键值字符串
        - 锚点链接：如上述例子中的 `#logo`，又称哈希
    - HTTP 协议版本号

版本号	发布时间
1.0	1996
1.1	1999
2	2015
3	2018

    - 请求行举例 `GET https://www.baidu.com/ HTTP/1.1`
        - 请求方法 `GET`
        - URL `https://www.baodu.com/`
        - HTTP 版本号 `HTTP/1.1`
2. 请求头：由一系列的键值对组成，关于请求头键值对的含义及取值见 https://developer.mozilla.org/en-US/docs/web/http/headers

请求头	解释
Host	主机名
Connection	连接的设置 keep-alive（保持连接）；close（关闭连接）
Cache-Control	缓存控制 max-age = 0 （没有缓存）
Upgrade-Insecure-Requests	将网页中的http请求转化为https请求（很少用）老网站升级
User-Agent	用户代理，客户端字符串标识，服务器可以通过这个标识来识别这个请求来自哪个客户端，一般在PC端和手机端的区分
Accept	设置浏览器接收的数据类型
Accept-Encoding	设置接收的压缩方式
Accept-Language	设置接收的语言 q=0.7 为喜好系数，满分为1
Cookie	略

3. 空行

4. 请求体：内容格式十分灵活，可以是任意内容，对于 GET 请求，请求体可以为空；对于 POST 请求，请求体可以是字符串或 JSON。

响应报文

与请求报文类似，响应报文由**响应行、响应头、空行、响应体**组成

1. 响应行
    - 响应行包含三部分内容：**HTTP 协议版本号、响应状态码、响应状态描述**
    - HTTP 协议版本号：与请求行中的 HTTP 协议版本号相同
    - 响应状态码：用于表示响应结果的状态，不同的响应状态码对应不同的含义，可以在 https://developer.mozilla.org/en-US/docs/Web/HTTP/status 中查询状态码的含义

状态码	含义	状态描述
200	请求成功	OK
403	禁止请求	Forbidden
404	找不到资源	Not Found
500	服务器内部错误	Internal Server Error
1??	信息响应	
2??	成功响应	
3??	重定向消息	
4??	客户端错误响应	
5??	服务端错误响应	

    - 响应状态描述：响应状态描述与响应状态码一一对应
    - 响应行举例 `HTTP/1.1 200 OK`
        - HTTP 协议版本号 `HTTP/1.1`
        - 响应状态码 `200`
        - 响应状态描述 `OK`
2. 响应头：记录了与服务器相关的内容，与请求头相同，由一系列的键值对组成，关于响应头键值对的含义及取值见 https://developer.mozilla.org/en-US/docs/web/http/headers

> p.s. 响应头中的键名可能无法在官方网站查询到，这是因为网站可以自定义一些键名

响应头	解释
Cache-Control	缓存控制，private 则只允许客户端缓存数据
Connection	链接设置
Content-Type:text/html;charset=utf-8	设置响应体的数据类型以及字符集，响应体为 html，字符集 utf-8
Content-Length	响应体的长度，单位为字节

3. 空行
4. 响应体：与请求体相同，内容格式十分灵活，常见类型有 HTML、CSS、JS、图片、JSON

5.2 IP 与端口

1. IP 是什么?
    - IP 又称 IP 地址，是一个数字标识，用于**标识网络中的计算机设备，实现设备间的通信**
    - IP 的本质是一个 **32** 位的二进制数字，以 8 位为一组，则可以得到 `xxx.xxx.xxx.xxx` 形式的通常情况下的十进制的表示
    - IP 的最大数量是 $2^{32} \approx 43$亿 < 80 亿的人口数量，因此为了解决 IP 不够用的问题，现常采用**区域共享和家庭共享**的方式来应对。
2. 家庭共享
    - 一个家庭中的所有设备处于一个局域网中，其 IP 属于**局域网 IP 或私网 IP**，局域网中的设备之间可以相互通信，但是无法与外界设备通信
    - 当连接互联网后，此时家庭中的设备才可以与外界通信，同时家庭被分配一个 IP 称之为**广域网 IP 或公网 IP**，家庭中的所有设备都共享这个公网 IP
3. IP 的分类
    - 本地回环地址（指向本机的 IP 地址）
        - `127.0.0.1 ~ 127.255.255.254`
    - 局域网 IP（私网 IP）
        - `192.168.0.0 ~ 192.168.255.255`
        - `172.16.0.0 ~ 172.31.255.255`
        - `10.0.0.0 ~ 10.255.255.255`
    - 广域网 IP（公网 IP）：除上述之外的 IP
4. 端口是什么?
    - 端口是计算机中应用程序的数字标识，用于实现**不同计算机设备应用程序之间的通信**
    - 一台计算机可以有 $2^{16} = 65536(0 - 65535)$ 个端口，一个应用程序可以使用**一个或多个端口**

5.3 创建 HTTP 服务

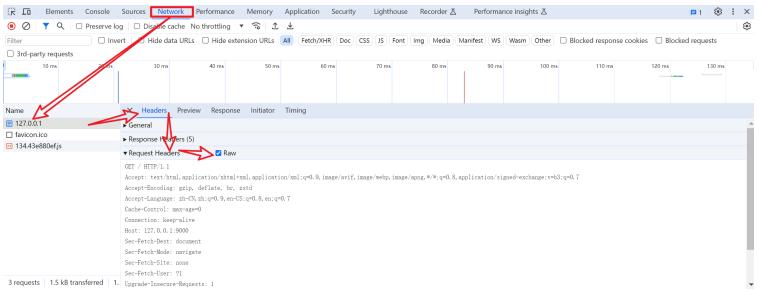
我们可以使用 Node.js 提供的 `http` 模块创建一个 http 服务，用于接收浏览器请求并返回响应，实现浏览器与服务之间的通信，不过首先，我们应该使用 `require('http')` 导入 `http` 模块

1. 创建 http 服务 `http.create((request, response) => {}): object`
    - `response.setHeader('content-type', 'text/html;charset=utf-8')`
    - `response.end(msg)`
2. 启动 http 服务 `server.listen(port, () => {})`
3. 测试 http 服务 浏览器中输入 `http://127.0.0.1:port`

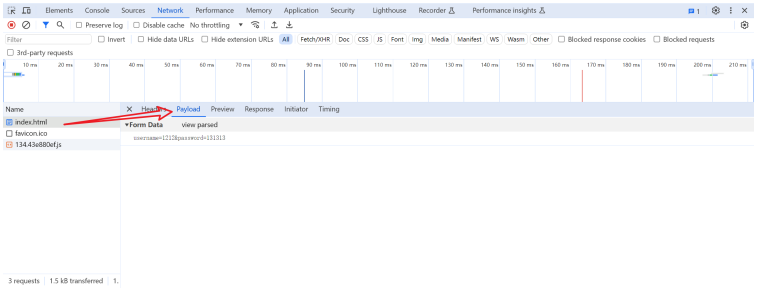
5.4 通过浏览器查看 HTTP 报文

`F12` 打开浏览器的检查，通过网络选项卡可以看到请求报文和响应报文的详细内容

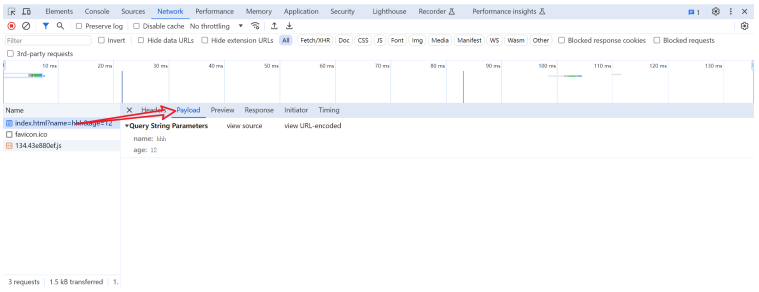
1. 请求报文
    - 请求行与请求头



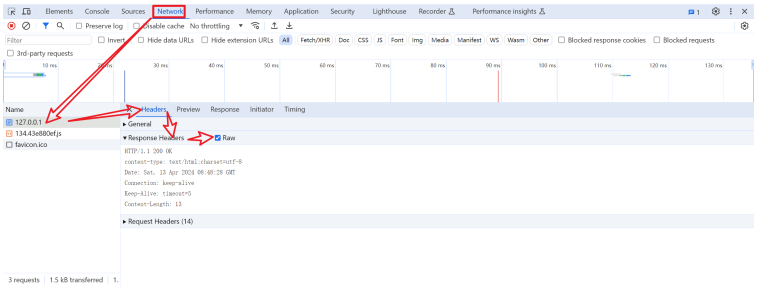
    - 请求体



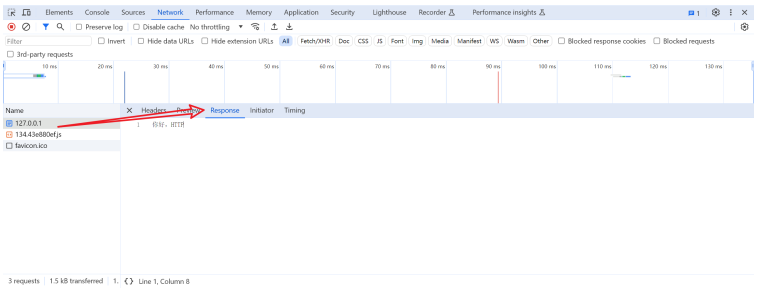
    - URL 中的查询字符串



2. 响应报文
    - 响应行与响应头



    - 响应体



5.5 获取 HTTP 请求报文

我们可以通过 `http.createServer` 中的回调函数中的 `request` 对象获取请求报文

1. 获取请求行和请求头
    - 请求行
        - 请求方法 `request.method`
        - 请求路径 `request.url`
        - 请求版本 `request.httpVersion`
    - 请求头 `request.headers`
2. 获取请求体
    - `request.on('data', chunk => {})`
    - `request.on('end', () => {})`
3. 获取（请求行中的请求路径中的）路径和查询字符串
    - 通过 url 模块：首先通过 `require('url')` 导入 url 模块
        - 路径 `url.parse(request.url).pathname`
        - 查询字符串 `url.parse(request.url, true).query`
    - 通过 URL 对象：首先通过 `new URL(request.url,'http://127.0.0.1:port')` 创建 URL 对象
        - 路径 `url.pathname`
        - 查询字符串 `url.searchParams`

5.6 设置 HTTP 响应报文

我们可以通过 `http.createServer` 中的回调函数中的 `request` 对象设置响应报文

1. 设置响应行
    - 响应状态码 `response.statusCode = xxx`
    - 响应状态信息 `response.statusMessage = xxx`
2. 设置响应头 `response.setHeader(头名, 头值)`
3. 设置响应体
    - `response.write(msg)`
    - `response.end(msg)`

5.7 网页资源的基本加载过程

- 在网页资源加载的过程中，浏览器**首先发送请求获取 HTML 内容**，然后在解析 HTML 的过程中，向服务器发送其他资源的请求，如 CSS，JavaScript，图片等等
- 多个请求是**并行**发送的。

5.8 静态资源与动态资源

1. **静态资源**

- 定义：内容长时间不发生改变的资源
- 举例：图片、视频、CSS、JS、HTML、字体文件等等

2. **动态资源**

- 定义：内容经常更新的资源
- 举例：网站首页、搜索列表页面等等

5.9 静态资源目录

1. **静态资源目录**（或称**网站根目录**）在一个 HTTP 服务器中是指**用于存放静态资源文件的文件夹或目录。**静态资源通常包括 HTML、CSS、JavaScript、图片、视频、字体文件等，它们在服务器上存储并被客户端（例如浏览器）请求和获取。
2. 当浏览器发送一个 HTTP 请求时，其中的 URL 路径（pathname）会告诉服务器需要请求的资源在服务器文件系统中的位置。服务器会根据这个路径来查找对应的资源文件。静态资源目录就是服务器用来**查找这些资源文件的基础路径。**

> 当我们使用 live server 打开 HTML 时，它启动的服务中的网站根目录是该 HTML 文件所在目录

5.10 网页中的 URL

1. 绝对路径
    - 形式一
        - 特点：（最完整）直接向目标资源发送请求，易理解，多用于网站的外链
        - 举例：`https://github.com/dashboard`
    - 形式二
        - 特点：（省略协议）与当前页面 URL 的协议拼接成完整的 URL 后再向目标资源发送请求，多用于大型网站
        - 举例：`//github.com/dashboard`
    - 形式三
        - 特点：（省略协议、主机名/域名、端口）与当前页面 URL 的协议、主机名/域名、端口拼接成完整的 URL 后再向目标资源发送请求，多用于中小型网站
        - 优势：当网页更改域名等信息后，不需要一一更改相关静态资源的路径
        - 举例：`/dashboard`
2. 相对路径：当使用相对路径发送请求时，需要**与当前页面的 URL 进行计算**，得到绝对路径后再向目标资源发送请求，假设当前网页的 URL 为 `https://github.com/dashboard/index.html`
    - 形式一：当前路径
        - 计算规则：以当前网页 URL 所在目录为基准
        - 举例1 `./css/app.css` => `https://github.com/dashboard/css/app.css`
        - 举例2 `css/app.css` => `https://github.com/dashboard/css/app.css`
    - 形式二：上一级路径
        - 计算规则：以当前网页 URL 所在目录的上一级目录为基准

        - 举例 `../img/logo.png` => `https://github.com/img/logo.png`
    - 形式三：上上一级路径
        - 计算规则：以当前网页 URL 所在目录的上上一级目录为基准
        - 举例：`../../img/logo.png` => `https://github.com/img/logo.png`
        - 注意事项：顶级路径进行 `../` 运算后的结果还是顶级路径

> p.s. 相对路径在项目运行时使用较少，因为相对路径需要参考当前网页的 URL 进行计算，如果当前网页的 URL 有问题，就会导致计算得到的绝对路径不可靠

3. 网页中 URL 的使用场景

```
<a href=""></a>
<link href="">
<script src=""></script>
<img src="">
<video src=""></video>
<audio src=""></audio>
<form action=""></form>
<!-- 还有 ajax 中的 get 请求 -->
```

5.11 设置资源类型

1. 资源类型：又称媒体类型、`MIME` 类型（multipurpose Internet Mail Extensions）。在网络通信中，服务器向浏览器传输数据时，会使用 MIME 类型（多用途互联网邮件扩展类型）来告诉浏览器数据的格式，以便浏览器能够以正确的方式处理这些数据。MIME 类型是一种标准，用于描述文件的格式和内容类型，例如文本、图片、音频等。
2. 常见文件的资源类型：`MIME` 类型的结构为 `[type]/[subtype]`

文件	资源类型	文件	资源类型
html	'text/html'	gif	'image/gif'
css	'text/css'	mp4	'video/mp4'
js	'text/javascript'	mp3	'audio/mpeg'
png	'image/png'	json	'application/json'
jpg	'image/jpeg'	未知类型时	'application/octet-stream'

3. 设置资源类型：我们可以在 http 服务中设置响应头 `content-type` 来表明响应体的 `MIME` 类型，此时浏览器会根据 `MIME` 类型决定如何处理对应的资源。

> p.s. 当我们不使用 `setHeader` 设置响应头 `content-type`，此时浏览器会根据其 `MIME` 嗅探功能，根据响应体的内容自动设置响应体的 `MIME` 类型

```
response.setHeader('content-type', '[type]/[subtype]')
```

4. 关于资源类型 `application/octet-stream`

- 当服务器返回的**资源类型对于浏览器来说是未知的**，或者服务器开发者并**不清楚具体应该使用哪种 MIME 类型来描述返回的内容**时，就可以选择使用 `application/octet-stream` 这个 MIME 类型。这个类型本质上是告诉浏览器："这些数据是一系列的字节流，我不会告诉你具体是什么类型，你也**不需要尝试去解析**它。"
- 浏览器在遇到 `application/octet-stream` 类型的响应时，通常不会尝试去解析这些数据。相反，浏览器会认为这是一种"原始"的数据格式，不知道如何直接展示给用户。因此，浏览器会选择一种安全的处理方式 —— 触发**文件下载**，将响应体的内容存储到用户的设备上，而不是尝试在浏览器中直接打开或展示这些内容。这就是我们通常所说的"下载效果"。
- 使用 `application/octet-stream` 类型的场景包括但不限于：
  - 服务器返回的数据确实是一种**应该被下载而不是展示**的文件，例如可执行文件、压缩包等。
  - 开发者未确切知道返回数据的具体类型，为**避免浏览器错误地处理**，选择使用这个类型**确保数据能被安全地传输至客户端。**
- `application/octet-stream` 是一种"万能"的、安全的 MIME 类型，用于处理未知或不确定类型的数据，确保数据能够**被安全地传输并提示用户下载，而不是在浏览器中直接展示。**

5. 关于使用 `setHeader` 设置响应体的字符集解决乱码的几点讨论
   - 使用 `response.setHeader('content-type', 'charset=utf-8')` 可以设置响应体的字符集，也可以使用 `<meta charset='utf-8'>` 设置浏览器解析 HTML 时采用的字符集，但是 `setHeader` 的优先级更高
   - 一般而言，我们只需要使用 `setHeader` 设置 `html` 文件的字符集，不用设置其中引入的各种资源（如 css、js、图片等）的字符集，因为这些资源引入网页后，会根据网页自动调整用于解析字符集

5.12 GET 和 POST 请求

1. 使用场景
   - GET

```
<link href="">
<script src=""></script>
<img src="">
<form method="get"></form>
<!-- 点击 a 标签 -->
<!-- 地址栏直接输入 url 访问 -->
<!-- ajax 中的 GET 请求 -->
```

   - POST

```
<form method="post"></form>
<!-- ajax 中的 POST 请求 -->
```

2. 区别
   - 功能不同：`GET` 主要用于获取数据；`POST` 主要用于提交数据
   - 参数位置不同：`GET` 的带参数请求的参数在 URL 中；`POST` 的带参数请求的参数在请求体中
   - 安全性不同：`POST` 相比 `GET` 相对更加安全，因为 `GET` 的带参数请求中的参数会暴露在地址栏中

> p.s. 安全是相对的，无论是 `POST` 还是 `GET` 请求，都可以通过 fiddler 等抓包工具获取请求内容

- 请求大小限制：`GET` 请求大小限制一般为 2K；`POST` 请求没有大小限制

6. 模块化

6.1 模块化简介

1. 模块化与模块
   - 将一个复杂的程序文件依据一定规则或规范拆分成多个文件的过程称之为**模块化**，其中拆分出的每个文件就是一个**模块**
   - 模块的内部数据是**私有的**，但是可以**暴露**内部数据，供其他模块使用
2. 模块化项目：编码时按照模块一个个编码的项目就是模块化项目
3. 模块化的好处
   - 防止命名冲突：不同模块之间可以声明相同名称的函数或变量，而不会产生冲突
   - 提高了代码的复用性：模块可以被多次引用，从而实现其中代码的复用
   - 代码更易于维护：某个模块的修改或扩展不会影响到其他模块

6.2 模块化的极简单使用

为了使用模块化，

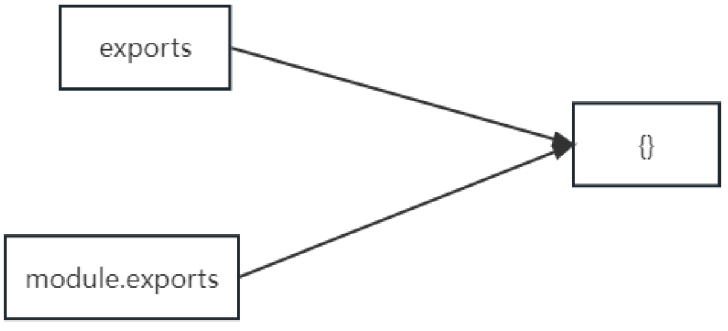
- 首先，我们需要创建一个模块（假定为 .js 文件），在其中编写相关代码，并通过 `module.exports` 向外界暴露数据
- 然后，在另一个文件中使用 `require` 函数导入该模块，函数的返回值就是我们在模块中暴露出的数据
- 总的来说，模块化的使用包含两个方面：向外暴露数据、模块的导入

6.3 向外暴露数据

1. exports 和 module.exports
   - 伪代码层面的解释

     ```
     let module = { exports: {} };
     let exports = module.exports;
     ```

   - 内存层面的解释

   - 关于 module.exports：当我们创建一个新的模块时，Node.js 会为我们创建了一个新的 `module` 对象，这个对象有一个属性 `exports`，该属性初始值为一个空对象 `{}`，即 `module = {exports: {}}`

- 关于 exports:：我们创建一个新的模块时，Node.js 会创建一个变量 `exports`，改变量初始化指向 `module` 对象的 `exports` 属性，即 `exports = module.exports`
- 注意：`module.exports` 和 `exports` 不是一个概念，前者是 `module` 对象的属性，后者是一个全局变量，二者初始情况下指向的是同一个空对象 `{}`

2. 讨论：当我们导入模块时，我们导入了什么?
   - 当我们使用 `require(模块路径)` 函数导入 `js` 模块时，函数的返回值本质上是 `module.exports` 所指向的对象
3. **summary**：模块向外暴露数据的两种方式
   - `module.exports = 值`
   - `exports.属性名=属性值`
   - 注意事项
     - 基于 2. 中的讨论，我们可以知道 `exports = 值` 的方式只会让 `exports` 变量不再指向 `require` 函数返回的对象，也就是 `module.exports` 所指向的对象
     - 如果想要添加或修改从模块中导出的对象中的属性或方法，则 `exports` 与 `module.exports` 都可以使用，不过通常使用前者，因为代码更简短
     - 如果希望导出的内容不是一个对象，而是一个函数、类或者其他数据，则只能通过 `module.exports` 的方式修改

6.4 模块的导入

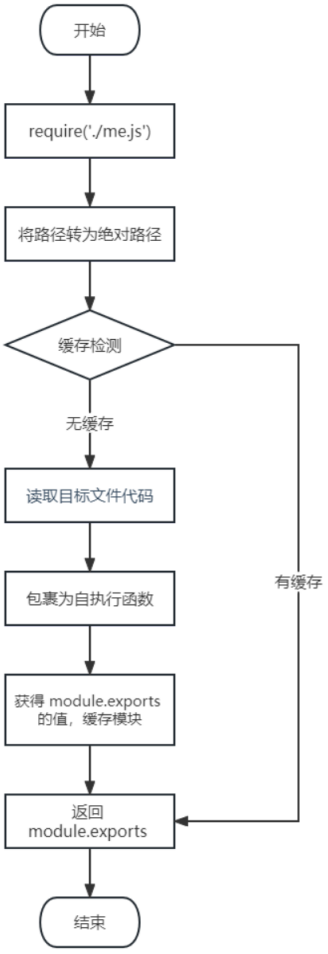
1. `.js` 文件中导入模块的语法 `const moduleName = require(modulePath)`
2. 模块导入的几点注意
   - 建议使用**相对路径**导入自定义模块，并且相对路径不可省略 `./` 或 `../`（不像 `fs` 模块中的函数，`require` 函数中的路径不受 `node` 工作目录的影响）
   - 导入 `.js`、`.json`、`.node` 文件时**可以省略后缀名**，但是如果同时存在 `xxx.js`、`xxx.json`、`xxx.node`，使用省略后缀名的方式 `require('xxx')` 导入模块时，则只会导入 `.js` 文件（`.js` > `.json` > `.node`）
     - 导入 `.js` 文件时，函数返回值是 `module.export` 暴露出的数据
     - 导入 `.json` 文件时，函数返回值是根据 `.json` 文件中的内容转换的对象
   - 导入**其他类型文件**时（如 `.abc`），会**当作** `.js` **文件**进行处理
   - 导入 `node.js` **内置模块**时，使用 `require(模块名)` 即可
3. 讨论：当 `require` 中的模块路径 `modulePath` 是个文件夹时，实际导入了什么?

```
if ('文件夹中存在 package.json 文件' && 'package.json 文件中存在 main 属性') {
   if ('main 属性对应的文件存在') {
      '导入该文件';
      return;
   } else {
      throw new Error('package.json 文件中 main 属性对应的模块不存在');
   }
} else {
   if ('文件夹中存在 index.js 文件') {
      '导入该文件';
      return;
   } else if ('文件夹中存在 index.json 文件') {
```

```
        '导入该文件';
        return;
    } else if ('文件夹中存在 index.node 文件') {
        '导入该文件';
        return;
    }
    else {
        throw new Error('找不到该文件夹中的可导入模块');
    }
}
```

6.5 模块导入的基本流程

当我们使用 `require` 函数导入自定义模块时，有以下流程



6.6 CommonJS 规范

1. CommonJS 是一个用于定义 JavaScript **模块化标准**的规范，它最初是为了在服务器端开发中使用 JavaScript 而产生的。CommonJS 规定了**如何编写模块代码，如何导出和引入模块**，以便于代码的组织和复用。
2. 在 CommonJS 规范中，`module.exports` 和 `exports` 是用于导出模块的接口，而 `require` 是用于导入其他模块的函数。具体来说：
    - `module.exports` 是一个对象，它是模块的公开接口。当其他文件需要引入该模块时，它们可以通过 `require` 函数获取 `module.exports` 所导出的内容。

- `exports` 是 `module.exports` 的一个引用，它用于暴露模块的属性或方法。你可以通过向 `exports` 对象上添加属性来导出它们，但如果你直接给 exports 赋值一个新对象，它将不会影响 `module.exports`。
- `require` 是一个函数，用于在一个模块中加载和使用另一个模块。通过 `require` 调用模块的路径，它会执行目标模块的代码，并返回目标模块的 `module.exports` 对象。

3. Node.js 是一个基于 Chrome V8 引擎的 JavaScript **运行时环境**，它允许开发者在服务器端运行 JavaScript 代码。Node.js 实现了 CommonJS 模块化规范，这意味着在 Node.js 中，你可以按照 CommonJS 规范来组织和管理你的代码模块。
4. ECMAScript 与 CommonJS
   - **ECMAScript 是 JavaScript 语言的规范**，它定义了语法、类型、语句、关键字等，而 JavaScript 是 ECMAScript 规范的实现。
   - 类似地，**CommonJS 是模块化的规范，而 Node.js 是这个规范的一种实现。**Node.js 不仅支持 CommonJS 规范，还提供了许多其他的内建模块和 API，使得它成为了一个功能齐全的服务器端开发环境。

7. 包管理工具

npm — commands

命令	含义
`npm -v`	查看 `npm` 的版本号
`npm init`	将当前文件夹初始化为一个包， 并交互式地创建 `package.json` 文件
`npm init -y` `npm init -yes`	将当前文件夹初始化为一个包， 并以默认值创建 `package.json` 文件
`npm s 关键字` `npm search 关键字`	搜索包名包含指定关键字的包
`npm i 包名` `npm install 包名`	安装指定包
`npm i -D 包名` `npm i --save-dev 包名`	安装指定包， 并设置依赖类型为开发依赖
`npm i -S 包名` `npm i --save 包名`	安装指定包， 并设置依赖类型为生产依赖
`npm i -g 包名`	全局安装指定包
`npm root -g`	查看全局安装位置
`npm i` `npm install`	一键安装项目依赖
`npm i 包名@版本号`	安装指定版本的包
`npm remove 包名` `npm r 包名`	局部删除包

命令	含义
`npm remove -g 包名`	全局删除包
`npm run 命令别名`	执行 `package.json` 的 `scripts` 属性中命令别名对应的命令

7.1 包管理工具简介

1. 包（package）：表示一组特定功能的**源码集合**
2. 包管理工具
    - 含义：管理包的应用软件，用于对包进行下载安装、更新、删除、上传等操作
    - 作用：提高项目开发效率
3. 前端常用的包管理工具
    - `npm`（最重要）
    - `yarn`
    - `cnpm`

7.2 npm 简介

1. npm：全称 Node Package Manager，译为 Node 包管理工具，是 `node.js` 官方**内置的包管理工具**，必须掌握
2. npm 的安装：安装 `node.js` 时，会自动安装 `npm`

> p.s. 我们可以在命令行中通过命令 `npm -v` 查看 `npm` 的版本号来测试 `npm` 是否安装成功

7.3 npm 的基本使用

1. **初始化包**：我们可以以一个**空目录**作为工作目录来启动命令行，使用 `npm init` 来将这个空文件夹初始化为一个包，并且交互式地创建 `package.json` 文件
    - 关于 `package.json`
        - `package.json` 是**包的配置文件**，每个包都必须要有 `package.json` 文件
        - `package.json` 的 `name` 属性表示**包名**，**不能使用中文和大写英文**；如果不指定，则**默认是所在文件夹的名称**，此时要求文件夹名称不能使用中文和大写英文
        - `package.json` 的 `version` 属性表示**版本号**，必须以 `x.x.x` 的形式来定义，其中 `x` 必须是数字，默认值为 `1.0.0`
        - `package.json` 的 `main` 属性对应交互设置 `package.json` 时指定的 `entry point` 值，用于**指定包的主入口文件**，默认值为 `index.js`
        - `package.json` 也可以手动创建与修改，此时所在文件夹也就成为了一个包
    - 我们也可以通过 `npm init -y` 或 `npm init -yes` 快速初始化包，**以默认值创建** `package.json` 文件
2. **搜索包**
    - 方式一 `npm s 关键字` 或 `npm search 关键字`，表示搜索包名包含指定关键字的包
    - 方式二 直接在 `https://www.npmjs.com/` 中搜索

3. **下载包**：我们可以在我们初始化的包中，使用 `npm i 包名` 或 `npm install 包名` 命令来安装指定包
    - 当我们使用命令安装指定包后，当前文件夹中会增加两项内容
        - `node_module` 该文件夹用于存放下载的包
        - `package-lock.json` 该文件称之为包的锁文件，用于锁定包的版本
    - 当我们安装了包 B 时，我们称包 B 是一个**依赖包**，或者说**依赖**
    - 当我们在创建的包 A 中安装了包 B，我们称**包 B 是 A 的一个依赖包**，或者说 **A 依赖 B**，此时我们通过 `npm init` 创建的包中的 `package.json` 中的 `dependencies` 属性记录包 A 的依赖信息，如包 B 的包名和版本信息等
4. 导入 npm 包的基本流程

    当我们使用 `require(包名)` 函数导入 `npm` 包时，`node`
    - 首先会在当前文件下的 `node_modules` 中寻找同名（包名）的文件夹，如果找到则根据其中 `package.json` 中 `main` 指定的入口去导入包
    - 如果当前文件夹中没有 `node_modules` 文件夹，就继续去上一级文件夹中的 `node_modules` 中寻找同名（包名）的文件夹，以此类推...

> 在 Node.js 中，`require` 函数是用于加载模块的主要方法，包括内置模块、第三方模块和自定义模块。

7.4 开发环境与生产环境

1. 开发环境
    - 程序员专门用来写代码的环境
    - 一般指程序员的电脑
    - 开发环境的项目一般只能程序员自己访问
2. 生产环境
    - 项目代码正式运行的环境
    - 一般指正式的服务器电脑
    - 生产环境的项目一般每个客户都可以访问

7.5 开发依赖与生产依赖

我们已经知道，我们在创建的包中安装 `npm` 包，又称安装**依赖**，或者说**创建的包依赖于安装的 `npm` 包**，或者说**安装的包是依赖。**依赖也有两种类型：**开发依赖**、**生产依赖**，我们可以在安装包时设置选项来区分依赖类型。

1. 开发依赖 `npm i -D 包名` 或 `npm i --save-dev`，此时包信息包含在 `package.json` 的 `devDependencies` 属性中
2. 生产依赖 `npm i -S 包名` 或 `npm i --save 包名`，**默认**情况下 `npm i` 命令安装的包属于生产依赖，包信息包含在 `package.json` 的 `dependencies` 属性中
3. 开发依赖与生产依赖区分：开发依赖**只是在开发阶段使用**的依赖包，而生产依赖是**开发阶段和最终上线运行阶段都使用**的依赖包

7.6 npm 全局安装

1. 全局安装：上述我们安装包的方式都属于**局部安装**，要求只能在包的安装目录所在的 `node_modules` 所在的目录或其子目录中通过 `require(包名)` 的方式导入和使用包；除此之外，我们也可以使用 `npm i -g 包名` **全局安装**包，全局安装完成后我们就可以在**任何目录的命令行**中运行**包名同名的命令。**
2. 示例：比如，`npm i -g nodemon` 全局安装了 `nodemon` 包，然后我们就可以在任何目录的命令行中通过 `nodemon .js 文件名` 运行 `js` 文件，`nodemon` 命令的作用是：一旦 `js` 文件中的代码改变，则重新运行 `js` 文件（自动重启 `node` 应用程序），可以使用 `nodemon` 运行开启 `http` 服务的 `js` 文件，一旦代码更新，服务也随之更新。
3. 使用说明
    - 全局安装的命令**不受工作目录位置的影响**
    - 可以通过 `npm root -g` 查看全局安装包的位置
    - **并不是所有的包都适合全局安装**，只有全局类的工具才适合
4. 问题解决：windows 的 `powershell` 默认不允许 `npm` 全局安装的命令执行脚本文件
    - 解决方式一 管理员身份打开 `powershell` => 键入命令 `set-ExecutionPolicy remoteSigned` => 键入 A，回车
    - 解决方式二：使用 `cmd`
5. 疑惑：为什么我们可以在任何地方使用全局安装的命令？
    - 当我们使用 `npm` 全局安装一个包时,这个包中如果包含一个可执行的命令,那么此命令就会被添加到系统的环境变量路径中(通常是 PATH 环境变量)。这意味着我们可以在命令行的任何位置直接运行这个命令,而无需输入完整的文件路径。
    - 我们可以在 `cmd` 中使用 `where 全局命令名称` 或在 `powershell` 中使用 `get-command 全局命令名称` 查找命令所在位置

7.7 npm 安装包的依赖

在项目协作时，当使用 `git push`，我们通常不会上传项目的依赖，即 `node_modules` 文件夹中的内容（或者说 `node_modules` 文件夹大多数情况不会存入版本库），因此当我们使用 `git pull` 拉取到项目代码后无法执行，这是因为缺乏依赖，此时我们可以使用 `npm i` 或 `npm install` 根据 `package.json` 中的内容，自动安装相关依赖

7.8 npm 安装指定版本的包

- 语法 `npm i 包名@版本号`
- 举例 `npm i jquery@1.11.2`

7.9 npm 删除包

- 删除包 `npm remove 包名` 或 `npm r 包名`
- 全局删除包 `npm remove -g 包名`

7.10 npm 配置别名

我们可以在创建的包（项目）中 `package.json` 文件的 `scripts` 属性配置命令别名以更简单的方式执行命令，如

```
{

   "scripts": {
      "server": "node server.js",
      "start": "node index.js",
      "命令别名": "实际执行的命令",
   },

}
```

配置完命令别名后，我们就可以使用 `npm run 命令别名` 的方式执行命令，如 `npm run server`，`npm run start`；其中，当执行 `start` 别名时，可以省略 `run`，即 `npm start`，该命令一般用于启动项目

`npm run 命令别名` 有自动向上级目录查找的特性，和 `require` 函数一样，现在当前文件夹中寻找 `package.json` 文件中的 `scripts` 属性，找到对应命令则执行，否则向当前文件夹的上一级文件夹去寻找

7.11 cnpm 简介及简单实用

1. cnpm
    - 是一个淘宝构建的 `npmjs.com` 的完整镜像，又称**淘宝镜像**，其服务部署在国内的阿里云服务器上，可以提高包的下载速度，网址：https://npmmirror.com/
    - 也是一个 `npm` 中的**全局工具包**，操作命令与 `npm` 的大体相同
2. cnpm 的使用方式
    - **作为镜像使用**：可以设置 `npm`、`yarn` 等包管理工具的源为淘宝镜像，具体见 npm 配置淘宝镜像、yarn 配置淘宝镜像 —— 推荐
    - **作为全局工具包使用**：安装 `cnpm` 工具包，使用 `cnpm` 作为包管理工具进行包的管理
        - `cnpm` 的安装 `npm install -g cnpm --registry=https://registry.npmmirror.com`
        - `cnpm` 的相关命令
    - 注：两种方式都可以提高包的下载速度

功能	命令
初始化	`cnpm init`

安装包	`cnpm i 包名`
	`cnpm i -S 包名`
	`cnpm i -D 包名`
	`cnpm i -g 包名`

安装项目依赖	`cnpm i`
删除包	`cnpm r 包名`

7.12 npm 配置淘宝镜像

1. 方式一：直接配置 `npm config set registry https://registry.npmmirror.com/`
2. 方式二：使用 `nrm`（npm registry manager）配置 —— 推荐
   - `nrm` 的安装 `npm i -g nrm`
   - `nrm` 的使用

命令	功能
`nrm ls`	列出所有支持的镜像地址
`nrm use 镜像名`	修改 `npm` 的镜像源
`npm config list`	查看 `npm` 的配置信息， 通过 `registry` 字段检查镜像源是否配置成功

   - `nrm` 支持的镜像

```
npm ---------- https://registry.npmjs.org/
yarn --------- https://registry.yarnpkg.com/
tencent ------ https://mirrors.cloud.tencent.com/npm/
cnpm --------- https://r.cnpmjs.org/
taobao ------- https://registry.npmmirror.com/
npmMirror ---- https://skimdb.npmjs.com/registry/
```

7.13 yarn 简介及简单使用

1. yarn
   - yarn 是一个由 FaceBook 在 2016 年推出的 JavaScript 包管理工具，官网 https://yarnpkg.com/
   - yarn 具有速度超快、超级安全、超级可靠的特点
2. yarn 的安装：`npm i -g yarn`
3. yarn 的简单使用

功能	命令
初始化包	`yarn init` `yarn init -y`
安装包	`yarn add 包名`（生产依赖） `yarn add 包名 --dev`（开发依赖） `yarn global add 包名`（全局安装）
删除包	`yarn remove 包名`（删除项目依赖包） `yarn global remove 包名`（全局删除包）
安装项目依赖	`yarn`
运行命令别名	`yarn 命令别名`

> p.s. `yarn` 全局安装的包不可用，因为没有将全局安装的包的位置添加进环境变量，我们需要通过 `yarn global bin` 来查看 `yarn` 全局安装包的位置，并将对应路径添加进环境变量

7.14 yarn 配置淘宝镜像

`yarn config set registry https://registry.npmmirror.com/`

> p.s. 我们可以通过 `yarn config list` 的 `registry` 查看 `yarn` 的镜像源是否设置成功

7.15 npm 管理发布包

1. 发布包
   - 创建文件夹，并创建文件 `index.js`， 在文件中声明函数，使用 `module.exports` 暴露
   - `npm` 初始化工具包， `package.json` 填写包的信息（**包的名字是唯一的**）
   - 注册账号 https://www.npmjs.com/signup 并激活
   - 修改为**官方镜像**（命令行中运行 `nrm use npm`）
   - 命令行下 `npm login` 填写相关用户信息
   - 命令行下 `npm publish` 提交包
2. 更新包
   - 更新包中的代码
   - 测试代码是否可用
   - 修改 `package.json` 中的**版本号**
   - 发布更新 `npm publish`
3. 删除包： `npm unpublish --force`

> p.s. 只有满足一定要求，才可以从 `npm` 服务中心删除包，见 https://docs.npmjs.com/policies/unpublish

7.16 其他包管理工具

除了**编程语言**领域有包管理工具之外，**操作系统**层面也存在包管理工具，不过这个包指的是『 **软件包** 』

1. 编程语言

语言	包管理工具
PHP	composer
Python	pip
Java	maven
Go	go mod
JavaScript	npm/yarn/cnpm/other
Ruby	rubyGems

2. 操作系统

操作系统	包管理工具	网址
Centos	yum	https://packages.debian.org/stable/
Ubuntu	apt	https://packages.ubuntu.com/
MacOS	homebrew	https://brew.sh/
Windows	chocolatey	https://chocolatey.org/

8. node 版本管理工具 nvm

1. nvm 介绍：`nvm` 全称为 `Node Version Manager`，是一个用于管理 `node` 版本的工具，便于切换不同版本的 `Node.js`
2. nvm 的下载
   - https://github.com/coreybutler/nvm-windows/releases
   - 下载 Assets 下的 `nvm-setup.exe`
3. nvm 的简单使用

命令	含义
`nvm list available`	显示所有支持可下载的 Node.js 版本
`nvm list`	显示所有已安装的 Node.js 版本
`nvm install 18.12.1`	安装 18.12.1 版本的 Node.js (安装指定版本的 Node.js)
`nvm install latest`	安装最新版本的 Node.js
`nvm uninstall 18.12.1`	删除 18.12.1 版本的 Node.js (删除指定版本的 Node.js)
`nvm use 18.12.1`	切换当前使用的 Node.js 版本为 18.12.1

9. Express 框架

1. express
   - express 是一个基于 Node.js 平台的极简、灵活的 **WEB 应用开发框架**，官方网址： https://www.expressjs.com.cn/
   - express 是一个**封装好的工具包**，封装了很多功能，便于我们开发 WEB 应用（**HTTP 服务**）
2. express 的下载和导入
   - 下载 `npm i express`
   - 导入 `const express = require('express');`
3. 注意事项
   - express 本身是一个 `npm` 包，因此可以使用 `npm i 包名` 的方式下载

- 对于使用 express 编写的代码，推荐使用 `nodemon` 包对应的命令 `nodemon 文件名` 的方式执行

9.1 Express 框架的简单使用

使用 express 编写创建一个 http 服务大致需要以下几个步骤

1. 导入 express `const express = require('express');`
2. 创建应用对象 `const app = express();`
3. 创建路由规则 `app.get(url, (req, res) => {})`
4. 监听端口，启动服务 `app.listen(9000, () => {})`

9.2 Express 路由

1. 什么是 Express 路由
   - Express 路由指的是 Express.js 框架中用于**定义应用程序如何响应特定 URL 请求的机制**
   - 路由用于**将 HTTP 请求（例如 GET、POST 等）映射到相应的处理程序函数**，以便执行特定的操作并生成相应的响应
   - Express 的路由系统**允许您定义多个路由**，当客户端发出 HTTP **请求时**，Express 会根据请求的 URL 和方法（GET、POST 等）来选择适当的路由，并执行相应的处理程序函数来处理请求
2. 路由的组成：请求方法、路径、回调函数

9.2.1 路由的使用

1. 路由的语法为 `app.<method>(path, callback)`
   - app 是由 express() 常见的应用对象
   - method 是 GET、POST 等请求方式
   - path 是网页请求 URL 的路径
   - callback 是当前路由的处理函数
2. 根据 method、path 的不同，路由的使用可以有多种形式，实现不同的功能

9.2.2 提取请求报文的参数

我们需要通过 request 这个封装了请求报文的对象，来提取请求报文中的数据；express 框架**封装了一些 API** 来方便获取请求报文中的数据，同时**兼容原生** http 模块中数据的获取方式

1. 原生操作（即 http 模块中使用的操作，express 兼容）
   - 请求方法 `request.method`
   - 请求路径 `request.url`
   - 请求 http 协议版本 `request.httpVersion`
   - 请求头 `request.headers`
2. express 封装的操作（express 独有）
   - 请求 URL 的路径 `request.path`
   - 请求 URL 的查询字符串 `request.query`
   - 获取 ip `request.ip`
   - 请求头 `value = request.get(headName)`

9.2.3 动态路由参数的使用

我们可以在路由的路径部分设定一个**动态路由参数**，即一个**使用冒号开头**的部分，该参数可以用于匹配 URL 中的任何值，并且在 `request.params` 对象中以键值对的形式存储，如以下示例

```
app.get('/:id.html', (request, response) => {
    console.log(`动态路由参数 id 的值为 ${request.params.id}`);
    response.end();
})
```

在该示例中，通过 `:id` 标识了一个动态路由参数 `id`，用于匹配路径中 '/' 和 '.html' 之间的任意内容，我们可以在回调函数中，通过 `request.params` 对象访问到 `id` 的内容；`request.params` 是一个对象，`id` 是属性名，`id` 匹配的值就是 `request.params.id`

9.3 Express 设置响应

与提取请求报文的参数类似，我们需要通过 response 这个封装了响应报文的对象，来设置响应报文中的数据；同时，express 框架也**封装了一些 API** 来方便设置响应报文中的数据，同时**兼容原生** http 模块中数据的获取方式

1. 原生操作（http 模块中使用的操作，express 中兼容）
    - 设置响应状态码 `response.statusCode = xxx`
    - 设置响应状态描述 `response.statusMessage = xxx`
    - 设置响应头 `response.setHeader('xxx', 'yyy')`
    - 设置响应体
        - `response.write('xxxx')`
        - `response.end('xxxx')`
2. express 封装的操作（express 独有）
    - 设置响应状态码 `response.status('xxx')`
    - 设置响应头 `response.set('xxx', 'yyy')`
    - 设置响应体 `response.send('xxxx')`（调用该方法时会自动添加响应头 `content-type: text/html; charset=utf-8`，因此中文不会乱码）
    - 重定向到一个新的网页 `response.redirect(url)`
    - 下载响应，自动下载内容 `response.download(path)`
    - JSON 响应，显示一个 json 内容 `response.json(json 字符串)`（调用该方法时会自动添加响应头 `content-type: application/json; charset=utf-8`）
    - 文件响应，显示一个文件（如 html） `response.sendFine(path)`

> p.s. express 封装的 response 对象可以链式调用来设置响应报文的内容，如 `res.status(404).set('content-type', 'text/html').end('Not Found')`

9.4 中间件

9.4.1 中间件简介

1. 中间件（Middleware）
   - 中间件本质是一个**回调函数**，又称**中间件函数**
   - 中间件函数可以像路由的回调函数一样访问请求对象 `request` 和响应对象 `response`
2. 中间件的作用：使用函数来封装公共操作，简化代码
3. 中间件的分类
   - 全局中间件：浏览器发送请求到服务端之后，**首先立即执行全局中间件函数**，然后再交给后续的路由回调函数处理
   - 局部中间件：只有**满足一定路由规则**之后，才执行对应的局部中间件函数
4. 中间件函数的声明

```
let middleware = function(request, response, next){
    /*
        这里是实现公共操作的代码
    */
    next(); // 如果在执行完当前中间件函数之后，仍希望执行后续的中间件函数或路由回调，则需要调用 next() 方法，否则会中断请求处理的过程
}
```

9.4.2 全局中间件

1. 我们可以使用 `app.use(callback)` 的方式定义一个全局中间件，也可以使用该方式定义多个全局中间件，即多次调用 `app.use` 方法，每次都定义不同的全局中间件
2. 我们可以利用全局中间件实现浏览器请求日志的记录

9.4.3 路由中间件

1. 我们可以使用 `app.请求方式(请求路径, 路由中间件函数1, 路由中间件函数2, ..., 路由中间件函数n, (request, response) => {})` 的方式为某些路由规则定义一个或多个路由中间件，执行顺序为 `路由中间件函数1` => `...` => `路由中间件函数n` => `路由回调函数`
2. 我们可以利用路由中间件封装多个路由的公共操作

9.4.4 静态资源中间件

1. 静态资源中间件
   - 我们可以通过 `express.static(rootPath)` 获取一个**静态资源中间件**，这是 express **内置**的处理静态资源的中间件，其中参数 `rootPath` 用于设置静态资源目录或网站根目录
   - 例如，当浏览器的请求路径为 `/js/app.js` 那么服务器就会在其本地的 `rootPath + /js/app.js` 中去寻找该静态资源，并将该资源作为响应返回给浏览器
   - 同时，当我们通过浏览器获取静态资源时，该中间件会**自动设置**资源的 `mime` 类型
   - 一般的，我们使用 `app.use(express.static(rootPath))` 来执行这个静态资源中间件
2. 静态资源与动态资源的处理方式
   - 静态资源：如视频、图片等，我们可以通过使用静态资源中间件，直接通过浏览器的请求路径获取到对应的静态资源的响应

    - 动态资源：如排行榜、新闻等，我们可以通过设置路由，匹配对应浏览器的请求路径，来实现动态资源的加载
3. 关于静态资源中间件的几点注意
    - 如果浏览器的请求路径是 `/`，则默认响应的是静态资源目录下的 `index.html` 文件；换句话说，直接打开某个网页，服务器直接响应静态资源目录下的 `index.html` 文件
    - 如果静态资源与路由规则同时匹配，则按照先后顺序，**谁先匹配谁就响应**；换句话说，如果静态资源目录下有 `index.html` 文件，同时服务中也设置了 `/index.html` 路由，则浏览器的请求路径为 `/index.html` 时，显示的是静态资源还是路由的响应，这是需要根据二者在服务中定义的先后顺序决定的
    - 一般而言，路由响应动态资源，静态资源中间件响应静态资源

9.5 获取请求体数据

express 可以基于 body-parser 包获取请求体的数据

- body-parser 可以通过 `npm install body-parser` 下载，通过 `const bodyParser = require('body-parser')` 导入
- body-parser 包含一系列的中间件，通常作为**路由中间件**使用，如
    - `let usrParser = bodyParser.urlencoded({extend: false})` 该中间件用于处理查询字符串格式的请求体
    - `let jsonParser = bodyParser.json()` 该中间件用于处理 json 格式的请求体
- 中间件函数 `usrParser`、`jsonParser` 执行完毕后，会向 `request` 对象上添加一个叫做 `body` 的属性，此时我们可以通过 `request.body` 获取请求体数据，该数据是一个键值对，存储着中间件函数处理好的请求体数据

9.6 防盗链

1. 防盗链是什么？防盗链技术通常用于保护网站的图片、视频或其他重要资源不被其他网站非法引用
2. 防盗链的作用
    - **保护网站资源**的唯一性和权益，防止被非法复制或滥用
    - **减轻服务器的压力**，如果太多的网站都链接到同一个资源，那么可能会消耗大量的服务器带宽和处理能力，影响网站的正常运行
3. 防盗链的实现方式
    - 实施防盗链的一种常见方法是使用服务器的 `Referer` 请求头。`Referer` 请求头可以告诉服务器这个请求是从哪个页面发出的。服务器可以检查这个 `Referer` 是否在允许的列表中。如果是，那么请求就被接受，资源可以被正常访问；如果不是，服务器可以拒绝这个请求，通常是返回一个404错误，表示请求的资源不存在。
    - 例如，假设您的网站是 example.com，您有一张图片，您不希望其他网站直接链接。那么您可以在服务器上设置一项规则，检查所有请求该图片的 `Referer` 头。如果 `Referer` 是 example.com，那么请求被接受；如果 `Referer` 是其他的，比如 other.com，那么服务器就拒绝请求，返回 404 错误。

```
const express = require('express');
const app = express();

const allowedDomains = ['https://www.example.com'];
```

```
app.use((req, res, next) => {
   const referer = req.headers.referer;

   if (!referer || allowedDomains.some(domain =>
referer.startsWith(domain))) {
      // If the referer header is empty or matches an allowed domain,
allow the request to proceed.
      next();
   } else {
      // If the referer header doesn't match an allowed domain, return a
404 response.
      res.sendStatus(404);
   }
});

// Routes and other middleware functions go here...

app.listen(3000, () => {
   console.log('Server listening on port 3000');
});
```

4. 关于 referer 请求头

   `Referer`（有时拼写为`Referrer`）是HTTP请求头中的一个字段，用于指示客户端（通常是浏览器）发起请求的源页面URL。它在以下场景中起作用：

   - **页面跳转**：当用户点击一个链接从一个页面跳转到另一个页面时，新页面的请求会带有`Referer`头，值为原始页面的URL。
   - **资源加载**：当HTML页面加载时，浏览器会解析页面中的引用资源（如CSS、JavaScript、图片等）。浏览器为这些资源的请求设置`Referer`头，值为加载这些资源的HTML页面的URL。
   - **安全与隐私**：`Referer`头可以用于网站分析用户来源，但同时也可能涉及用户隐私。因此，浏览器允许用户控制或禁用`Referer`头的发送，一些隐私模式或严格隐私设置可能不会发送`Referer`信息。
   - **安全应用**：`Referer`头有时用于防止恶意请求，比如验证用户是否从预期的页面发起请求，或者用于防止图片被盗链。

9.7 路由模块化

1. 路由模块化：express 中为了方便对路由规则的管理，使用 `Router` 实现路由的模块化，从而更好地管理路由
2. `Router`：是一个完整的中间件和路由系统，可以看作是一个小型的 `app` 对象
3. 如何使用 `Router` 实现路由模块化？
   - 第一步：创建独立的 `.js` 文件，存放路由规则，并通过 `module.exports` 将 `Router` 对外暴露
   - 第二步：在主文件中使用 `require` 导入外部文件暴露的 `Router`，并使用 `app.use` 使用该路由规则

9.8 EJS 模板引擎

1. EJS 的简单介绍
    - 模板引擎是分离**用户界面和业务数据**的一种技术
    - EJS 是一个高效的 JavaScript 的模板引擎
    - 我们可以使用 `npm install ejs` 命令安装 EJS 模板引擎
2. EJS 的基本使用
    - `ejs.render(模板字符串, 变量解释对象)`
        - 描述
            - 该方法是 EJS 模板引擎中用于**渲染模板**的核心方法
            - 该方法将**模板字符串和变量解释对象**结合起来，生成最终的 HTML 字符串
            - 该方法会**解析模板字符串**，找到所有 `<%= %>` 标记，并用变量解释对象中对应的值替换它们
        - 参数
            - **模板字符串**：字符串，可以使用 EJS 语法嵌入动态内容，如 `<%= 变量名 %>` 这个 EJS 语法表示将变量名对应的值插入到模板的这个位置
            - **变量解释对象**：对象，包含了模板中可能出现的变量名和对应的值，该对象用于提供数据给模板，以便模板可以根据数据动态生成内容
        - 返回值：字符串，即解析后的 HTML 内容
        - 示例

```
const template = '<h1>Hello, <%= name %>!</h1>';
const data = { name: 'John' };

const html = ejs.render(template, data);

console.log(html); // 输出: <h1>Hello, John!</h1>
```

        - EJS 模板字符串的常用语法
            - 变量嵌入 `<%= 变量名 %>`
            - 代码执行
                - `<% 代码 %>`：执行 JavaScript 代码，但不会将其结果输出到模板；通常用于**控制流程**，如条件语句、循环等
                - `<%= 代码 %>`：将 JavaScript 代码的结果**转义后插入**到 HTML 中；这样做可以**防止 HTML 注入攻击**，因为特殊字符会被转义成它们的 HTML 实体，从而使其成为安全的文本
                - `<%- 代码 %>`：直接将 JavaScript 代码的结果插入到 HTML 中，而**不进行转义**；这在需要输出 HTML 标签或其他 HTML 特性的情况下很有用，但也要小心，因为它可**能导致 XSS（跨站脚本攻击）安全漏洞**
3. EJS 渲染 HTML 举例：即将 HTML 作为一个模板字符串，然后通过 `ejs.render` 方法，根据变量解释对象解析 HTML 中使用 `<%= %>` 标记地方的变量，最后返回一个渲染后的 HTML
    - 示例1：变量匹配
    - 示例2：列表渲染

    - 示例3：条件渲染
4. 如何在 Express 中使用 EJS 模板引擎
    - 第一步：通过应用对象设置 Express 的模板引擎为 EJS以及模板文件的存放位置

```
app.set('view engine', 'ejs'); // 设置 Express 应用对象的模板引擎为 EJS
app.set('views', '路径'); // 设置 Express 应用对象的模板文件的存放位置
// 注1：'view engine', 'views' 是固定的键名，不可改变
// 注2：路径必须是绝对路径，因为相对路径不是相对于本文件，而是命令行的工作目录
// 注3：模板文件的后缀名是 .ejs
```

    - 第二步：使用 `response.render(模板文件名，变量解释对象)` 方法渲染模板文件夹下的指定模板，并使用变量解释对象解析这个模板，解析内容就是服务端的响应

9.9 Express application generator

- `express-generator` 是一个包，用于快速创建一个 Express 应用骨架，我们可以通过 `npm install -g express-generator` 全局安装这个包，之后会暴露一个 `express` 命令
- 我们可以通过 `express -e 路径` 在指定位置快速创建一个支持 ejs 模板引擎的 Express 应用骨架，然后需要使用 `npm i` 安装对应依赖

9.10 通过表单的文件上传与对应报文的处理

- 表单逻辑

```
<!-- 如果要通过表单上传一个文件（同时请求方法时 POST），此时必须要有属性
enctype="multipart/form-data" -->
  <form action="/portrait" method="post" enctype="multipart/form-data">
      Username: <input type="text" name="username"><br>
      Avatar: <input type="file" name="avatar"><br>
      <hr>
      <button>SUBMIT</button>
  </form>
```

- 路由逻辑

```
/* 文件上传 GET & POST */
// GET 请求：表单页面，提示用户提交个人信息
router.get('/portrait', (_, res) => {
res.render('portrait');
});

// POST 请求：对用户提交的个人信息（包括文件）进行处理
// 文件的处理使用 formidable 包，使用 npm i formidable 安装包
const { formidable } = require('formidable');

router.post('/portrait', (req, res) => {
// 表单对象，用于解析表单请求中的内容
const form = formidable({
   multiples: true, // 设置是否在本地存储上传的文件
   uploadDir: __dirname + '/../public/resources', // 设置上传文件的保存目录
   keepExtensions: true // 设置是否保存文件的后缀
});
```

```
form.parse(req, (err, fields, files) => {
    if (err) {
        next(err);
        return;
    }

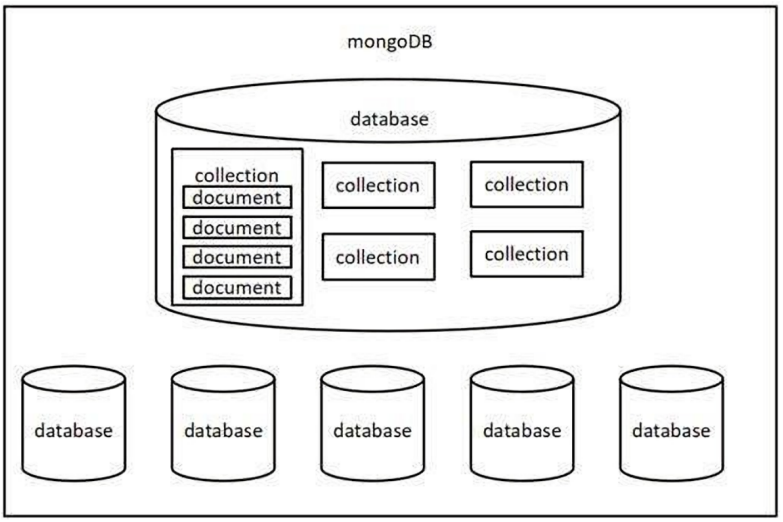
    /*
        在获取表单数据、保存文件到本地后，仍需要做一个操作：保存访问该图片的 URL（绝对路径，省略协议、域名、端口等信息），然后将该数
        据存放到数据库中，便于以后的访问
    */
    let fileUrl = '/resources/' + files.avatar[0].newFilename;
    console.log(fileUrl); // /resources/5868edc658fb373e76f7e9200.js
    console.log(123);

    res.json({ fields, files }); // fields 表示非文件数据信息；files 表示文件数据信息
});
});
```

10. MongoDB

10.1 MongoDB 简介

1. 数据库（Database）
    - 数据库是按照数据结构来组织、存储和管理数据的**应用程序**
    - 数据库的主要作用是**管理数据**，实现对数据的**增、删、改、查**
    - 相比于纯文件管理数据，数据库管理数据速度更快、扩展性更强、安全性更强
2. MongoDB
    - MongoBD 是一个**基于分布式文件存储的数据库**，点击跳转官网
    - MongoDB 语法与 JavaScript 相似，易上手，学习成本低
3. MongoDB 的核心概念
    - 数据库（database）
        - 数据库是一个数据仓库
        - **一个数据库服务中可以创建很多数据库**
        - **一个数据库中可以存放很多集合**
    - 集合（collection）
        - 集合类似 JS 中的数组
        - **一个集合中可以存放很多文档**
    - 文档（document）
        - **文档是数据库中的最小单位**，类似 JS 中的对象
    - 注意
        - 一般而言，**一个项目使用一个数据库**，一个集合存储**同一种类型的数据**
        - 对象中的属性又称为**字段**



10.2 MongoDB 的下载与配置

1. 下载：推荐在 https://www.mongodb.com/try/download/community 下载 zip 类型，然后再进行相关配置（这里下载的是 5.0.14 版本）
2. 配置
   - 压缩包移动到 `C:\Program Files` 并解压
   - 创建 `C:\data\db`，mongodb 默认将数据保存在该文件夹
   - 将 mongodb 的 bin 目录添加到环境变量
   - 【服务】运行 `mongod` 命令，启动 mongodb 服务，出现 waiting for connections 则表明服务已启动成功
   - 【客户端】运行 `mongo` 命令，启动 mongodb 客户端，用于连接本机的 mongodb 服务
   - 【客户端】可以向【服务】发送请求，【服务】返回给【客户端】响应
3. 注意：如果选中 mongodb 服务窗口的内容，则会停止服务，可以通过回车键取消选中

10.3 MongoDB 的操作命令

1. 数据库命令

命令	功能
`show dbs`	显示所有不为空的数据库
`use 数据库名`	切换到指定的数据库，如不存在则会自动创建
`db`	显示当前正在使用的数据库
`use 数据库名` `db.dropDatabase()`	删除当前正在使用的数据库

```
> show dbs
admin   0.000GB
config  0.000GB
local   0.000GB
```

```
> use admin
switched to db admin
```

```
> db
admin
```

```
> use bilibili
switched to db bilibili
> db.dropDatabase()
{ "ok" : 1 }
```

2. 集合命令

命令	功能
`db.createCollection('集合名称')`	创建集合
`show collections`	显示当前数据库中的所有集合
`db.集合名称.drop()`	删除特定集合
`db.集合名称.renameCollection('新的集合名称')`	重命名特定集合

```
> db.createCollection('users')
{ "ok" : 1 }
```

```
> show collections
books
pictures
users
```

```
> db.users.drop()
true
> db.pics.drop()
false
```

```
> db.pictures.renameCollection('pics')
{ "ok" : 1 }
```

3. 文档命令

命令	功能
`db.集合名称.insert(文档对象)`	在集合中插入一个文档
`db.集合名称.find()`	查询集合中的所有文档
`db.集合名称.find(查询条件)`	查询集合中满足查询条件的文档
`db.集合名称.update(查询条件, 新的文档对象)`	更新集合中满足查询条件的文档为新的文档（替换）

命令	功能
`db.集合名称.update(查询条件, {$set: {字段: 值}})`	更新集合集合中满足查询条件的文档的某些字段的值
`db.集合名称.remove(查询条件)`	删除集合中满足查询条件的文档

```
> db.books.insert({name: '钢铁是怎样炼成的', id: 1, number: 30})
WriteResult({ "nInserted" : 1 })
```

```
> db.books.find()
{ "_id" : ObjectId("6621d0385a0c900261df4c87"), "name" : "钢铁是怎样炼成的",
"id" : 1, "number" : 30 }
{ "_id" : ObjectId("6621d4c15a0c900261df4c88"), "name" : "简爱", "id" : 2,
"number" : 23 }
{ "_id" : ObjectId("6621d4d95a0c900261df4c89"), "name" : "傲慢与偏见", "id" :
3, "number" : 22 }
```

```
> db.books.find({id: 2})
{ "_id" : ObjectId("6621d4c15a0c900261df4c88"), "name" : "简爱", "id" : 2,
"number" : 23 }
```

```
> db.books.update({id:2},{name: "简爱2"})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.books.find()
{ "_id" : ObjectId("6621d0385a0c900261df4c87"), "name" : "钢铁是怎样炼成的",
"id" : 1, "number" : 30 }
{ "_id" : ObjectId("6621d4c15a0c900261df4c88"), "name" : "简爱2" }
{ "_id" : ObjectId("6621d4d95a0c900261df4c89"), "name" : "傲慢与偏见", "id" :
3, "number" : 22 }
```

```
> db.books.update({id:1},{$set:{name: "钢铁是怎样炼成的2"}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.books.find()
{ "_id" : ObjectId("6621d0385a0c900261df4c87"), "name" : "钢铁是怎样炼成的2",
"id" : 1, "number" : 30 }
{ "_id" : ObjectId("6621d4c15a0c900261df4c88"), "name" : "简爱2" }
{ "_id" : ObjectId("6621d4d95a0c900261df4c89"), "name" : "傲慢与偏见", "id" :
3, "number" : 22 }
```

```
> db.books.remove({id:3})
WriteResult({ "nRemoved" : 1 })
> db.books.find()
{ "_id" : ObjectId("6621d0385a0c900261df4c87"), "name" : "钢铁是怎样炼成的2",
"id" : 1, "number" : 30 }
{ "_id" : ObjectId("6621d4c15a0c900261df4c88"), "name" : "简爱2" }
```

10.4 Mongoose 简介

- Mongoose 是一个**对象文档模型库**，是 npm 中的一个包，可以通过 `npm install mongoose` 下载使用
- Mongoose 方便我们**使用代码操作 MongoDB 数据库**，点击跳转官网

10.5 Mongoose 连接数据库

连接数据库主要包含以下步骤

- 导入 mongoose `require('mongoose')`
- 连接数据库 `mongoose.connect('mongodb://127.0.0.1:27017/testdb')`
- 设置连接回调
  - 成功回调 `mongoose.connection.once('open', callback)`
  - 失败回调 `mongoose.connection.on('error', callback)`
  - 关闭回调 `mongoose.connection.on('close', callback)`

10.6 Mongoose 创建新文档

连接数据库后，我们通常在 `mongoose.connection.once('open', callback)` 的回调函数中创建新文档，主要包含以下步骤

- 创建文档结构对象 `new mongoose.Schema(obj)`
- 创建文档模型对象 `mongoose.model('集合的单数名称', 文档结构对象)`
- 使用文档模型对象创建新文档 `model.create(obj).then(data => {}).catch(err => {})`

10.7 字段类型和字段值验证

我们在创建一个文档对象前，必须要创建文档结构对象，用于约束文档对象的属性和属性值（属性又称字段，属性值又称字段值），例如

```
new mongoose.Schema({
   name: String,
   author: String,
   price: Number,
});
```

创建了一个文档结构对象，要求文档对象有三个字段，并对每个字段的取值类型做出了限制。

此外，还有其他可取的字段值，而且每个字段的值也可以是一个对象，约束字段值类型的同时，对字段进行其他限制。

- 字段值类型

字段值类型	含义
String	字符串
Number	数字
Boolean	布尔值

字段值类型	含义
Array	数组，也可以使用 [] 来标识
Date	日期
Buffer	Buffer 对象
mongoose.Schema.Types.Mixed	任意类型
mongoose.Schema.Types.ObjectId	对象 ID，一般用于存储其他文档对象的外键，联合查询使用
mongoose.Schema.Types.Decimal128	高精度数字

- 字段值验证：此时字段取值为一个对象，这里对这个对象可选属性及属性值进行介绍
  - type 表示字段值类型，取值为上述字段值类型
  - require 表示是否为必填项，取值 true / false
  - default 表示默认值，取值任意；如果创建文档对象时没指定对应字段的值，则使用默认值
  - enum 表示枚举值，取值必须是一个数组；创建文档对象时对应字段的值必须是枚举数组中的一个值
  - unique 表示是否是唯一值，取值 true / false；必须重建集合后 unique 才有效果

10.8 Mongoose 增删改查

我们可以使用**文档模型对象**实现对文档对象的增删改查等操作，具体语法如下

- 增
  - 添加一个文档 `model.create(文档对象).then(data => {}).catch(err => {})`
  - 添加多个文档 `model.insertMany([文档对象1, 文档对象2, ..., 文档对象m]).then(data => {}).catch(err => {})`
- 删
  - 删除一个文档 `model.deleteOne(条件对象).then(info => {}).catch(err => {})`
  - 删除多个文档 `model.deleteMany(条件对象).then(info => {}).catch(err => {})`
- 改
  - 更新一个文档 `model.updateOne(条件对象, 要更新的字段及其取值对象).then(info => {}).catch(err => {})`
  - 更新多个文档 `model.updateMany(条件对象, 要更新的字段及其取值对象).then(info => {}).catch(err => {})`
- 查
  - 查询一条数据
    - `model.findOne(条件对象).then(data => {}).catch(err => {})`
    - `model.findById('文档对象的 id').then(data => {}).catch(err => {})`
  - 查询多条数据
    - `model.find().then(data => {}).catch(err => {})`
    - `model.find(条件对象).then(data => {}).catch(err => {})`

10.9 Mongoose 条件对象语法

在对文档对象进行删改查时，我们需要使用条件对象匹配文档对象，除了**字段取值相等**匹配外，还有比较匹配、逻辑匹配、正则匹配，语法如下

- 比较匹配 `{字段名: {比较运算符: 数值}}`
- 逻辑匹配 `{逻辑运算符: [条件对象1, 条件对象2, ..., 条件对象n]}`
- 正则匹配 `{字段名: 正则表达式}`

10.10 Mongoose 个性化读取

在对文档模型进行读取时，我们还可以进行个性化读取操作，如字段筛选、数据排序、内容截取等，语法如下

- 字段筛选 `model.find().select({字段1: 0/1, 字段2: 0/1, ...,.字段n: 0/1}).exec().then(data => {}).catch(err => {})`
- 数据排序 `model.find().sort({字段名: 1/-1}).exec().then(data => {}).catch(err => {})`
- 内容截取 `model.find().skip(num1).limit(num2).exec().then(data => {}).catch(err => {})`

10.11 Mongoose 代码模块化

Mongoose 的模块化就是将原始的 js 文件根据其各部分功能拆分到多个 js 文件中，并通过 `module.exports` 的方式向外暴露，提高代码的可维护性，如连接数据库的参数可以编写到一个配置文件中；不同的文档模型对象可以归类为一个模型文件夹中；数据库连接的基本框架也可以抽离出来；在 index.js 中仅仅编写关于数据库相关的操作代码

未模块化时的代码组织

```
const mongoose = require('mongoose');
mongoose.connect('mongodb://127.0.0.1:27017/bilibili');
mongoose.connection.once('open', () => {
  // 创建文档结构对象
  // 创建文档模型对象
  // 使用文档模型对象对数据库进行操作，如增删改查等
});
mongoose.connection.on('error', () => {});
mongoose.connection.on('close', () => {});
```

模块化后代码组织

```
// dir: /index.js
const db = require('./db/db.js')
const 文档模型对象 = require('./models/文档模型对象.js');

db(() => {
    console.log('mongoose-mongodb 连接成功');
    // 使用文档模型对象对数据库进行操作，如增删改查等
})
```

```
// dir: /db/db.js
```

```
const mongoose = require('mongoose');
const config = require('../config/config.json')
module.exports = function (success, error = () => console.log('mongoose-mongodb
连接失败')) {
    let { DBHOST, DBPORT, DBNAME } = config;
    mongoose.connect(`mongodb://${DBHOST}:${DBPORT}/${DBNAME}`);

    mongoose.connection.once('open', () => {
        success();
    });

    mongoose.connection.on('error', () => {
        error();
    });

    mongoose.connection.on('close', () => {
        console.log('mongoose-mongodb 连接关闭');
    })
}
```

```
// dir: /config/config.json
{
    "DBHOST": "主机名或域名",
    "DBPORT": "端口",
    "DBNAME": "数据库名"
}
```

```
// dir: /models/文档模型对象.js
const mongoose = require('mongoose');

let 文档结构对象 = new mongoose.Schema({
    字段: 字段值类型,
    ......
});

let 文档模型对象 = mongoose.model('集合名', 文档结构对象);

module.exports = 文档模型对象;
```

10.12 MongoDB 图形化管理工具

图形化工具，如 Robo 3T（免费）、Navicat（付费） 与 mongoose、mongo 类似，用于与 MongoDB 进行交互

11. API 接口

11.1 接口的简介

1. 什么是接口?
    - 接口是**前后端通信**的桥梁，可以将一个接口理解为服务中的一个路由规则，**根据请求响应结果**
    - 接口为英文为 API（Application Program Interface），有时也称为 **API 接口**
    - 这里的接口指的是**数据接口**，与编程语言中的接口语法不同

- 接口大多由后端工程师开发，由前端工程师调用

2. 接口的组成：**请求方法、接口地址、请求参数、响应结果**
3. 举例：身份证查询接口
    - 请求方法：get/post
    - 接口地址：https://api.asilu.com/idcard/
    - 请求参数：id string 必填 身份证号码
    - 响应结果：json
        - addr string 籍贯
        - date string 出生年月
        - sex string 性别
    - 请求举例：https://api.asilu.com/idcard/?id=152502199405148245

11.2 RESTful API

1. 什么是 RESTful API?
    - RESTful API 是一种特殊风格的接口，其要求
        - 接口地址（URL）的路径表示资源
        - 资源操作必须与 **HTTP 请求方法**对应，如 POST 表示新增资源、GET 表示获取资源、DELETE 表示删除资源等
        - 资源操作结果必须与 **HTTP 响应状态码**对应
    - RESTful API 规则举例如下

操作	请求类型	URL	返回
新增歌曲	POST	/song	返回新生成的歌曲信息
删除歌曲	DELETE	/song/10	返回一个空文档
修改歌曲	PUT	/song/10	返回更新后的歌曲信息
修改歌曲	PATCH	/song/10	返回更新后的歌曲信息
获取所有歌曲	GET	/song	返回歌曲列表数组
获取单个歌曲	GET	/song/10	返回单个歌曲信息

2. RESTful API 服务的搭建 —— json-server 包
    - 介绍：json-server 是一个 JS 编写的工具包，可以快速搭建 RESTful API 服务
    - 安装：`npm i -g json-server`
    - 使用教程
        - 第一步：创建 json 文件，编写基本结构 { 资源名: [资源对象1, 资源对象2, ..., 资源对象n] }
        - 第二步：以 json 文件所在文件夹为工作目录，执行命令 `json-server --watch json文件名`，启动服务，默认监听端口为 3000
        - 第三步：使用接口测试工具（如 apipost、apifox、postman），测试 RESTful API

12. 会话控制

12.1 会话控制简介

1. 什么是会话控制?
    - 会话可以理解为浏览器与服务器之间的交互，会话控制就是对会话进行控制
    - HTTP 是一种**无状态的协议**，即没有办法区分多次请求是否来自于同一个客户端，因而无法区分用户，会话控制技术就是用来解决该问题的，借此我们可以在服务端中识别区分不同的用户发出的请求
2. 会话控制技术
    - `cookie`
    - `session`
    - `token`

12.2 cookie

1. 什么是 cookie?
    - cookie 是 HTTP **服务器发送到浏览器**并**保存在浏览器端**的**一小块数据**
    - cookie 是**按照域名划分保存**的
    - cookie 可以看作是一系列的键值对数据
2. cookie 的运行流程
    - cookie 的产生：用户填写账号密码登录某个网站，**服务器校验通过后下放 cookie 给浏览器**
    - cookie 的使用：有了 cookie 之后，浏览器向服务器发送请求时，会**自动将当前域名下的可用 cookie 设置在请求头中发送给服务器**（这个请求头的名字叫做 cookie，因此也可以将 cookie 理解为一个 HTTP 的请求头）
3. 浏览器对 cookie 的操作
    - 禁用所有 cookie：设置 => 隐私与安全 => 第三方 cookie
    - 删除 cookie：设置 => 隐私与安全 => 第三方 cookie
    - 查看 cookie：开发者工具 => application 页面查看
4. express 中对 cookie 的操作（增删改查） —— `cookie-parser`

12.3 session

1. 什么是 session?
    - session 是保存在**服务器端**的**一块数据**，表示当前访问用户的相关信息
    - session 可以用于**识别用户身份**，快速获取当前用户的相关信息
2. session 的运行流程
    - session 的产生：用户填写账号密码登录某个网站，**服务器校验通过后创建 session 信息**，然后将 **session_id 通过 cookie 返回给浏览器**
    - session 的使用：有了 cookie 之后，浏览器向服务器发送请求时，服务器**通过 cookie 中的 session_id 确定用户的身份**
3. express 中对 session 的操作（增删改查） —— `express-session`、`connect-mongo`
4. cookie Vs. session

区别	cookie	session
存放位置	浏览器端	服务器端
安全性	以明文存放在浏览器端，安全性较低	安全性**相对较好**
网络传输量	内容过多会增大报文体积，影响传输效率	只通过 cookie 传递 session_id，不影响传输效率
存储限制	单个 cookie 保存数据不超过 4K，单个域名下的 cookie 数量也有限制	无限制

12.4 token

1. 什么是 token?
    - token 是**由服务器端生成并返回给客户端**的一串**加密字符串**
    - token 中保存着**用户信息**，用于**识别用户身份**，实现会话控制
    - token 多用于移动端 app；cookie、session 多用于网页端
2. token 的运行流程
    - token 的产生：用户填写账号密码登录后，**服务端校验通过后创建 token**，并将 token 作为**响应体内容**返回给客户端
    - token 的使用：有了 token 以后，后续再向服务器端发送请求时，需要**手动**将 token 添加在请求报文中，一般是放在**请求头**中
3. token 的特点
    - **服务端压力更小**，因为数据都存储在客户端
    - **相对更加安全**，因为数据被加密，且因为需要手动发送给服务端，可以避免 CSRF（跨站请求伪造）
    - **扩展性更强**，因为服务器之间可以共享 token，增加服务器节点更加简单
4. 什么是 JWT？JWT（JSON Web Token）是一种流行的**跨域认证解决方案**，可用于基于 token 的身份验证，其使 token 的生成与校验更加规范
5. token 的操作（创建和解析） —— `jsonwebtoken`

12.5 本地域名

1. 什么是本地域名：只能在**本机使用的域名**，一般在开发阶段使用
2. 浏览器的域名解析流程：当我们在浏览器中输入域名，浏览器并不知道向哪里发送请求，此时浏览器会
    - 首先，**查询本地缓存**，看是否存在当前域名与对应 ip 的映射，如果存在，则向对应 ip 地址发送请求，否则
    - 然后，**查询 `host` 文件**，看是否存在当前域名与对应 ip 的映射，如果存在，则向对应 ip 地址发送请求，否则（注：`host` 文件的路径为 `C:\Windows\System32\drivers\etc\HOSTS`）
    - 最后，**进行 DNS(Domain Name System) 域名解析**，浏览器给 DNS 系统发送域名，DNS 系统返回对应的 ip 地址，此时浏览器向对应 ip 地址发送请求（注：可以通过 `ipconfig /all` 查看本机的 DNS 服务器）

3. 示例：当我们在本地 `host` 文件中添加了 `127.0.0.1 www.baidu.com`，我们在浏览器中输入 `www.baidu.com` 时，浏览器实际会向本机 `127.0.0.1` 发送请求（注意：需要在无痕模式下使用，避免浏览器缓存的影响；需要关闭 VPN，因为代理会修改 DNS）

13. 项目上线

项目上线的流程为：将代码上传到仓库 => 购买云服务器，并安装相关软件 => 云服务器上克隆仓库中的代码，启动代码服务 => 购买域名并备案解析 => 配置 HTTPS 证书

跳转至顶部