nodejs

命令

node -v 查看版本 node js文件路径 执行js文件

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
- 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：
  - 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。
  - ```
    // 1.导入fs模块,来操作文件
    const fs = require('fs')
    // 2.调用fs.readFile()方法读取文件
    fs.readFile('./file/1.txt', 'utf8', function (err, dataStr) {
        // 打印失败的结果，如果读取成功，则err的值为null
        // 如果读取失败，则err的值为错误对象，dataStr的值为undefined
        if (err) {
            return console.log('读取文件失败' + err.message)
        }
        console.log('读取文件成功' + dataStr)
    })
    ```
- fs.writeFile()的语法格式
  - 使用该方法，可以向指定的文件中写入内容，语法格式如下
  - 参数1：必选参数，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，表示以什么编码格式来读取文件，默认值是utf8
  - 参数4：必选参数，文件写入完成后的回调函数。
    ```

- ```
  // 1. 导入 fs 文件系统模块
  const fs = require('fs')
  fs.writeFile('./files/3.txt', 'ok123', function(err) {
    // 2.1 如果文件写入成功，则 err 的值等于 null
    // 2.2 如果文件写入失败，则 err 的值等于一个 错误对象
    if (err) {
      return console.log('文件写入失败！' + err.message)
    }
    console.log('文件写入成功！')
  })
  ```

path路径模块

- path.join()语法格式
  - 使用path.join方法，可以把多个路径片段拼接为较为完整的路径字符串
  - 注意：今后凡是涉及路径拼接的操作，都是使用path.join()方法进行处理，不要直接使用+进行字符串的拼
  - ```
    const path = require('path')
    const fs = require('fs')
    // 注意：  ../ 会抵消前面的路径
    const pathStr = path.join('/a', '/b/c', '../../', './d', 'e')
    console.log(pathStr)  // \a\b\d\e
    fs.readFile(__dirname + '/files/1.txt')
    fs.readFile(path.join(__dirname, './files/1.txt'), 'utf8', function(err, dataStr) {
      if (err) {
        return console.log(err.message)
      }
      console.log(dataStr)
    })
    ```
- path.basename（） 获取文件名
  - path:必选参数，表示一个路径的字符串
  - ext可选参数，表示文件拓展名
  - 返回：表示路径中的最后一部分
  - ```
    const path = require('path')
    // 定义文件的存放路径
    const fpath = '/a/b/c/index.html'
    const nameWithoutExt = path.basename(fpath, '.html')  //index
    ```
- path.extname() 获取拓展名
    ```

- ```
  const path = require('path')
  // 这是文件的存放路径
  const fpath = '/a/b/c/index.html'
  const fext = path.extname(fpath)
  console.log(fext)    //  . html
  ```

HTTP

- 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。

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 sever.on()
- 4. 启动服务器 sever.listen()
- ```
  // 1. 导入 http 模块
  const http = require('http')
  // 2. 创建 web 服务器实例
  const server = http.createServer()
  // 3. 为服务器实例绑定 request 事件，监听客户端的请求
  server.on('request', function (req, res) {
    console.log('Someone visit our web server.')
  })
  // 4. 启动服务器
  server.listen(8080, function () {
    console.log('server running at http://127.0.0.1:8080')
  })
  ```

req请求对象 和响应

- req.url 是客户端请求的 URL 地址
- res.end() 方法，向客户端响应一些内容
- ```
  server.on('request', (req, res) => {
    // req.url 是客户端请求的 URL 地址
    const url = req.url
    // req.method 是客户端请求的 method 类型
    const method = req.method
    const str = `Your request url is ${url}, and request method is ${method}`
    console.log(str)
    // 调用 res.end() 方法，向客户端响应一些内容
    res.end(str)
  })
  ```
  ```

解决中文乱码

- 调用 res.setHeader() 方法，设置 Content-Type 响应头，解决中文乱码的问题
- ```
  res.setHeader('Content-Type', 'text/html; charset=utf-8')
  ```

根据不同的url响应不同的html内容

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

  server.on('request', (req, res) => {
    // 1. 获取请求的 url 地址
    const url = req.url
    // 2. 设置默认的响应内容为 404 Not found
    let content = '<h1>404 Not found!</h1>'
    // 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
    if (url === '/' || url === '/index.html') {
      content = '<h1>首页</h1>'
    } else if (url === '/about.html') {
      content = '<h1>关于页面</h1>'
    }
    // 5. 设置 Content-Type 响应头，防止中文乱码
    res.setHeader('Content-Type', 'text/html; charset=utf-8')
    // 6. 使用 res.end() 把内容响应给客户端
    res.end(content)
  })
  server.listen(80, () => {
    console.log('server running at http://127.0.0.1')
  })
  ```

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
- 内置模块 自定义模块 第三方模块

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

- 

```
// 1. 加载内置的 fs 模块
const fs = require('fs')

// 2. 加载用户的自定义模块
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，会在后面的课程中进行专门的讲解）
const moment = require('moment')
```

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。
- module.exports 对象
  - 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
  - 默认情况下，exports 和 module.exports 指向同一个对象。
- 时刻谨记，require() 模块时，得到的永远是 module.exports 指向的对象
-

```
module.exports.username = 'zs'
module.exports = {
  nickname: '小黑',
  sayHi() {
    console.log('Hi!')
  }
}
module.exports.username = username
exports.sayHello = function() {
  console.log('大家好！')
}
```

包和NPM

- Node.js中的第三方模块又叫做包
- 安装包的 命令 npm install 包的完整名称 || npm i 包的完整名称 @版本号
- 快速创建 package.json 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。
- 一次性安装所有的包 npm i || npm install package.json他里面存在的包名
- npm uninstall 卸载指定包
- 如果 只在开发阶段用到的 包 最后 添上-d
-

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。

```
# 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://registry.npm.taobao.org/
# 检查镜像源是否下载成功
npm config get registry
```

-

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。

```
# 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

- 项目包
    - 开发依赖包　-d
    - 核心依赖包
- 全局包 -g

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
```

Express

- Express 是基于 Node.js 平台，快速、开放、极简的 Web 开发框架。
- Express 的作用和 Node.js 内置的 http 模块类似，是专门用来创建 Web 服务器的。
- Express 的本质：就是一个 npm 上的第三方包，提供了快速创建 Web 服务器的便捷方法。
- 安装 npm i express@4.17.1

创建基本的web服务器

- res.send() 方法，向客户端响应一个 文本字符串
- :id 是一个动态的参数　req.params可以获取：后面的参数
- .express.static() 托管静态资源
- 如果希望在托管的静态资源访问路径之前，挂载路径前缀，则可以使用如下的方式：

```
// 1. 导入 express
const express = require('express')
// 2. 创建 web 服务器
const app = express()

// 4. 监听客户端的 GET 和 POST 请求，并向客户端响应具体的内容
app.get('/user', (req, res) => {
  // 调用 express 提供的 res.send() 方法，向客户端响应一个 JSON 对象
  res.send({ name: 'zs', age: 20, gender: '男' })
})
app.post('/user', (req, res) => {
  // 调用 express 提供的 res.send() 方法，向客户端响应一个 文本字符串
  res.send('请求成功')
})
app.get('/', (req, res) => {
  // 通过 req.query 可以获取到客户端发送过来的 查询参数
  // 注意：默认情况下，req.query 是一个空对象
  console.log(req.query)
  res.send(req.query)
})
// 注意：这里的 :id 是一个动态的参数
app.get('/user/:ids/:username', (req, res) => {
  // req.params 是动态匹配到的 URL 参数，默认也是一个空对象
  console.log(req.params)

  res.send(req.params)
```

```
})

// 3. 启动 web 服务器
app.listen(80, () => {
  console.log('express server running at http://127.0.0.1')
})


// 在这里，调用 express.static() 方法，快速的对外提供静态资源
app.use('/files', express.static('./files'))
app.use(express.static('./clock'))

app.listen(80, () => {
  console.log('express server running at http://127.0.0.1')
})
```

nodemon

- npm i -g nodemon 它能够监听项目文件的变动，当代码被修改后，nodemon 会自动帮我们重启项目，极大方便了开发和调试
- 使用 nodemon 文件名

Express 路由

- 最简单的路由使用

```
// 挂载路由
app.get('/', (req, res) => {
  res.send('hello world.')
})
app.post('/', (req, res) => {
  res.send('Post Request.')
})

app.listen(80, () => {
  console.log('http://127.0.0.1')
})

```

- 模块化路由

```
router.js
// 这是路由模块
// 1. 导入 express
const express = require('express')
// 2. 创建路由对象
const router = express.Router()

// 3. 挂载具体的路由
router.get('/user/list', (req, res) => {
  res.send('Get user list.')
```

```
})
router.post('/user/add', (req, res) => {
  res.send('Add new user.')
})

// 4. 向外导出路由对象
module.exports = router
const express = require('express')
const app = express()

// app.use('/files', express.static('./files'))

// 1. 导入路由模块
const router = require('./03.router')
// 2. 注册路由模块
app.use('/api', router)
// 注意： app.use() 函数的作用，就是来注册全局中间件
app.listen(80, () => {
  console.log('http://127.0.0.1')
})
```

中间件

- 中间件（Middleware），特指业务流程的中间处理环节。
- 定义中间件函数

```
// 这是定义全局中间件的简化形式
app.use((req, res, next) => {
  console.log('这是最简单的中间件函数')
  next()
})
```

- 定义多个中间件

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

// 定义第一个全局中间件
app.use((req, res, next) => {
  console.log('调用了第1个全局中间件')
  next()
})
// 定义第二个全局中间件
app.use((req, res, next) => {
  console.log('调用了第2个全局中间件')
  next()
})

// 定义一个路由

app.get('/user', (req, res) => {
```

```
  res.send('User page.')
})
```

- 局部生效的中间件 一个及多个

```
// 导入 express 模块
const express = require('express')
// 创建 express 的服务器实例
const app = express()

// 1. 定义中间件函数
const mw1 = (req, res, next) => {
  console.log('调用了第一个局部生效的中间件')
  next()
}

const mw2 = (req, res, next) => {
  console.log('调用了第二个局部生效的中间件')
  next()
}

// 2. 创建路由
app.get('/', [mw1, mw2], (req, res) => {
  res.send('Home page.')
})
app.get('/user', (req, res) => {
  res.send('User page.')
})

// 调用 app.listen 方法，指定端口号并启动web服务器
app.listen(80, function () {
  console.log('Express server running at http://127.0.0.1')
})
```

中间件分类

- ① 应用级别的中间件
  - 通过 app.use() 或 app.get() 或 app.post() ，绑定到 app 实例上的中间件，叫做应用级别的中间件，
- ② 路由级别的中间件
  - 绑定到 express.Router() 实例上的中间件，叫做路由级别的中间件。
- ③ 错误级别的中间件
  - 错误级别中间件的作用：专门用来捕获整个项目中发生的异常错误，从而防止项目异常崩溃的问题。
  - 格式：错误级别中间件的 function 处理函数中，必须有 4 个形参，形参顺序从前到后，分别是 (err, req, res, next)。
- ④ Express 内置的中间件
  - 自 Express 4.16.0 版本开始，Express 内置了 3 个常用的中间件，极大的提高了 Express 项目的开发效率和体验：

    - ① express.static 快速托管静态资源的内置中间件，例如： HTML 文件、图片、CSS 样式等（无兼容性）
    - ② express.json 解析 JSON 格式的请求体数据（有兼容性，仅在 4.16.0+ 版本中可用）
    - ③ express.urlencoded 解析 URL-encoded 格式的请求体数据（有兼容性，仅在 4.16.0+ 版本中可用）
- ⑤ 第三方的中间件
    - ① 运行 npm install body-parser 安装中间件
    - ② 使用 require 导入中间件
    - ③ 调用 app.use() 注册并使用中间件

Express 写接口

- ```
  // 导入 express
  const express = require('express')
  // 创建服务器实例
  const app = express()

  // 配置解析表单数据的中间件
  app.use(express.urlencoded({ extended: false }))

  // 必须在配置 cors 中间件之前，配置 JSONP 的接口
  app.get('/api/jsonp', (req, res) => {
    // TODO: 定义 JSONP 接口具体的实现过程
    // 1. 得到函数的名称
    const funcName = req.query.callback
    // 2. 定义要发送到客户端的数据对象
    const data = { name: 'zs', age: 22 }
    // 3. 拼接出一个函数的调用
    const scriptStr = `${funcName}(${JSON.stringify(data)})`
    // 4. 把拼接的字符串，响应给客户端
    res.send(scriptStr)
  })

  // 一定要在路由之前，配置 cors 这个中间件，从而解决接口跨域的问题
  const cors = require('cors')
  app.use(cors())

  // 导入路由模块
  const router = require('./16.apiRouter')
  // 把路由模块，注册到 app 上
  app.use('/api', router)

  // 启动服务器
  app.listen(80, () => {
    console.log('express server running at http://127.0.0.1')
  })
  ```

解决跨域问题

- ① CORS（主流的解决方案，推荐使用）
- ② JSONP（有缺陷的解决方案：只支持 GET 请求）
  ```

mysql

```sql
-- 通过 * 把 users 表中所有的数据查询出来
 select * from users

-- 从 users 表中把 username 和 password 对应的数据查询出来
select username, password from users

-- 向 users 表中，插入新数据，username 的值为 tony stark  password 的值为 098123
 insert into users (username, password) values ('tony stark', '098123')
 select * from users

-- 将 id 为 4 的用户密码，更新成 888888
update users set password='888888' where id=4
--select * from users

-- 更新 id 为 2 的用户，把用户密码更新为 admin123  同时，把用户的状态更新为 1
 update users set password='admin123', status=1 where id=2
-- select * from users

-- 删除 users 表中，  id 为 4 的用户
 delete from users where id=4
-- select * from users

-- 演示 where 子句的使用
select * from users where status=1
 select * from users where id>=2
select * from users where username<>'ls'
select * from users where username!='ls'

-- 使用 AND 来显示所有状态为0且id小于3的用户
select * from users where status=0 and id<3

-- 使用 or 来显示所有状态为1 或 username 为 zs 的用户
 select * from users where status=1 or username='zs'

-- 对users表中的数据，按照 status 字段进行升序排序
select * from users order by status

-- 按照 id 对结果进行降序的排序  desc 表示降序排序   asc 表示升序排序（默认情况下，就是升序排序的）
 select * from users order by id desc

-- 对 users 表中的数据，先按照 status 进行降序排序，再按照 username 字母的顺序，进行升序的排序
 select * from users order by status desc, username asc

-- 使用 count(*) 来统计 users 表中，状态为 0 用户的总数量
 select count(*) from users where status=0

-- 使用 AS 关键字给列起别名
select count(*) as total from users where status=0
 select username as uname, password as upwd from users
```