Class10:前后端交互Node+Gulp

服务器基础

HTTP协议

创建web服务器

```
const http = require('http');
const app = http.createServer();
app.on('request', (req, res) => {
    res.end('<h2>hello user</h2>');
});

app.listen(3000);
console.log('网站服务器启动成功。');
```

超文本传输协议（英文：HyperText Transfer Protocol，缩写:**HTTP**）规定了如何从网站服务器传输超文本到本地浏览器，它基于客户端服务器架构工作，是客户端（用户）和服务器端（网站）请求和应答的标准。

在HTTP请求和响应的过程中传递的数据块就叫报文，包括要传送的数据和一些附加信息，并且要遵守规定好的格式。

报文

- 请求报文
  - 请求方式
    - GET
      - 获取数据
      - 在地址栏输入网址属于GET
    - POST
      - 添加数据，一般逻辑
      - POST相对于GET安全
  - 服务端通过 req.mehtod获得请求方式
  - 服务端通过req.url获取请求地址

```
console.log(req.headers['accept']);
```

req.headers获取请求报文信息

- 响应报文
  - HTTP状态码
    - 200 请求成功
    - 404 请求的资源没有被找到
    - 500 服务器端错误
    - 400 客户端请求语法错误

```
res.writeHead(400)
```

通过res.writeHead(状态码)设置

  - 内容类型
    - text/html
    - text/css
    - application/javascript
    - image/jpeg
    - application/json

```
res.writeHead(200, {
    'content-type': 'text/plain' //代表返回纯文本
});
```

```
res.writeHead(200, {
    'content-type': 'text/html;charset=utf8'
});
```

解决返回中文乱码问题

客户端向服务器端发送请求时，有时需要携带一些客户信息，客户信息需要通过请求参数的形式传递到服务器端，比如登录操作。用户输入的用户名和密码。

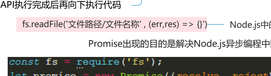
- 参数被放置在浏览器地址栏中，例如：http://localhost:3000/?**name=zhangsan&age=20**

请求参数

- GET请求参数
  - 在Node.js中提供了url内置模块获取请求参数

```
console.log(req.url);
let params = url.parse(req.url, true).query;
console.log(params.name);
console.log(params.age);
```

- POST请求参数



POST请求参数不在地址栏中而在请求报文From Data中

```
// post参数是通过事件触发的
//data  当请求参数传递的时候触发data事件
//end 当参数传递完成的时候触发end事件
let postParams = '';
req.on('data', (params) => {
    postParams += params;
});

req.on('end', () => {
    console.log(postParams);
});
```

```
const queryString = require('querystring');
```

Node.js提供了处理字符串的内置对象 querystring

```
queryString.parse(postParams)
```

通过queryString.parse()方法将字符串转换成对象的形式

路由是指客户端请求地址与服务器端程序代码的对应关系。简单的说，就是请求什么响应什么。

- 路由
  - 路由功能核心代码

```
const http = require('http');
const url = require('url');
const app = http.createServer();
app.on('request', (req, res) => {
    const method = req.method.toLowerCase();
    const pathname = url.parse(req.url).pathname;

    res.writeHead(200, {
        'content-type': 'text/html;charset=utf8'
    });
    if (method == 'get') {
        if (pathname == '/' || pathname == '/index') {
            res.end('欢迎来到首页');
        } else if (pathname == '/list') {
            res.end('欢迎来到列表页');
        } else {
            res.end('未找到页面');
        }
    } else if (method == 'post') {

    }
});

app.listen(3000);
console.log('服务器启动成功');
```

服务器端不需要处理，可以直接响应给客户端的资源就是静态资源，例如CSS、JavaScript、image文件。

- 静态资源
  - 静态资源访问

```
app.on('request', (req, res) => {
    let pathname = url.parse(req.url).pathname;
    //将用户的请求路径转换为服务器中的绝对路径
    let realPath = path.join(__dirname, pathname);
    //读取文件
    fs.readFile(realPath, (error, result) => {
        if (error != null) {
            res.writeHead(404, {
                'content-type': 'text/html;charset=utf8'
            });
            res.end('文件读取失败');
            return;
        }
        res.end(result);
    });
});
```

- 动态资源

相同的请求地址不同的响应资源，这种资源就是动态资源。

- HTTP请求与响应的处理
  - 同步API，异步API
    - 同步API：只有当前API执行完成后，才能继续执行下一个API
    - 异步API：当前API的执行不会阻塞后续代码的执行
    - 同步API和异步API区别（获取返回值）
      - 同步API可以从返回值中拿到API执行的结果, 但是异步API是不可以的
    - 同步API和异步API区别（代码指向顺序）
      - 同步API从上到下依次执行，前面代码会阻塞后面代码的执行
      - 异步API不会等待API执行完成后再向下执行代码
    - Node.js中的异步API
      - fs.readFile('文件路径/文件名称' , (err,res) => {})
  - Promise
    - Promise出现的目的是解决Node.js异步编程中回调地狱的问题。

```
const fs = require('fs');
let promise = new Promise((resolve, reject) => {
    fs.readFile('./10.txt', 'utf8', (err, result) => {
        if (err != null) {
            reject(err);
        } else {
            resolve(result);
        }
    });
});

promise.then((result) => {
    console.log(result);

}).catch((err) => {
    console.log(err);

})
```

    - 解决回调地狱问题

```
p1().then((result) => {
    console.log(result);
    return p2();
}).then((result) => {
    console.log(result);
    return p3();
}).then((result) => {
    console.log(result);

})
```

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

function p1() {
    return new Promise((resolve, reject) => {
        fs.readFile('./1.txt', 'utf8', (err, result) => {
            if (err != null) {
                reject(err);
            } else {
                resolve(result);
            }
        })
    })
}
```

  - 异步函数
    - 异步函数是异步编程语法的终极解决方案，它可以让我们将异步代码写成同步的形式，让代码不再有回调函数嵌套，使代码变得清晰明了。
    - 异步函数的定义就是在普通函数的前边加上async关键字

```
// 异步函数定义一
const fn = async (args) => {
    // function body
}
//异步函数定义二
async function(args) {
    //function body
}
```

```
async function fn() {
    return 123;
}
console.log(fn());//Promise { 123 }
```

```
async function fn() {

}
console.log(fn()); //Promise { undefined }
```

默认的返回值是promise对象

    - 异步函数then调用

```
fn().then(function(data) {
    console.log(data);

})
```

    - promise.catch的调用

```
async function fn() {
    throw '发生了一些错误';
    return 123;
}
```

throw关键字一旦执行，下边的代码不再执行

err输出的其实就是throw后边的内容

```
fn().then(function(data) {
    console.log(data);
}).catch(function(err) {
    console.log(err);
})
```

    - await关键字
      - 只能出现在异步函数中
      - 他可以暂停异步函数的执行，等待promise对象返回结果之后再向下执行函数。
    - 异步函数的应用
      - promisify 是用来改造node中现有的异步API的，使其返回promise对象，从而支持异步函数语法。被改造后返回一个新的方法。

```
const fs = require('fs');
const promisify = require('util').promisify;
// fs.readFile()
const readFile = promisify(fs.readFile);
async function run() {
    let r1 = await readFile('./1.txt', 'utf8')
    let r2 = await readFile('./2.txt', 'utf8')
    let r3 = await readFile('./3.txt', 'utf8')
    console.log(r1);
    console.log(r2);
    console.log(r3);
}

run()
```

Node基础

Node开发属于服务器开发

- 服务器端开发要做的事情
  - 1、实现网站业务逻辑
  - 2、数据的增删改查
- 为什么选择node
  - node使用JavaScript语言开发后端应用
  - 一些公司要求掌握node开发
  - 生态系统活跃，有大量开源库可以使用
  - 前端开发工具大多是基于Node开发
- Node是什么
  - Node是一个基于ChromeV8引擎的JavaScript代码运行环境
  - Chrome V8引擎是能够运行JavaScript的引擎环境

Node.js运行环境安装

- 去Node.js 官网下载
- 检查Node.js安装
  - 启动windows powershell
  - 在命令行中输入 node -v
  - PATH的作用
    - PATH 存放文件的可执行文件的目录，命令行工具根据PATH目录找到exe可执行文件打开软件。
  - 常见的安装错误
    - 2502 或 2503 用户权限不够 —— 使用管理员运行安装包即可解决
    - node -v验证出错 安装路径没能正确写入PATH变量 —— 手动写入安装目录，重启shell检查

Node.js快速入门

- Node.js的组成
  - JavaScript包含了 ECMAScript、DOM 、BOM
  - Node.js 包含了 ECMAScript、Node模块API。包括文件、网络、路径等等API
- Node.js 基础语法
  - 所有ECMAScript语法在Node中均适用。
- Node.js全局对象global
  - 在浏览器中全局对象是window，在Node中全局对象是global
  - Node中全局对象下有以下方法，可以在任何地方使用，global可以省略。
    - console.log
    - setTimeout
    - clearTimeout
    - setInterval
    - clearInterval

Node.js模块化开发

- JavaScript开发弊端
  - 文件依赖关系不明确的问题 —— 引入先后顺序的问题
  - 命名冲突的问题 —— 后续引入的文件中的变量覆盖了之前引入的变量的问题
  - 模块化开发就是可以解决这两个问题
- 生活中的模块化开发
  - 例如电脑的组装。
- 软件中的模块化开发

一个功能就是一个模块，多个模块可以组成完整的应用，抽离一个模块不会影响其他功能的运行。

  - 模块化开发规范
    - Node.js规定一个JavaScript文件就是一个模块,模块内部定义的变量和函数默认情况下在外部无法得到。


```
const add = (n1, n2) => n1 + n2;
exports.add = add;
```

模块对象的导出

```
const a = require('./03.modle-a');
console.log(a.add(10, 20));
```

模块对象的导入

模块的导入后缀可以省略

模块你不可以使用exports对象进行成员导出，使用require方法导入其他模块。

    - 模块成员导出的另一种方式
      - module.exports.version = vsesion
      - exports 是 module.exports 的别名，导出对象最终以module.exports为准
  - Node运行环境提供的API，因为这些API都是以模块化的方式开发的，所以我们又称Node运行环境提供的API为系统模块。
  - 系统模块fs 文件操作

```
const fs = require('fs');
fs.readFile('./03.modle-a.js', 'utf8', (err, doc) => {
    console.log(err);

    if (err == null) {
        console.log(doc);

    }
});
```

读取文件内容 fs.reaFile('文件路径/文件名称' , ['文件编码'] , callback);

```
const fs = require('fs');
fs.writeFile('./demo.txt', '即将写入的内容', (err) => {
    if (err != null) {
        console.log(err);
        return;
    }
    console.log('文件内容写入成功');
})
```

写入文件内容 fs.writeFile('文件路径/文件名称', '数据', callback);

  - 系统模块
    - 系统模块PATH
      - 路径拼接
        - 为什么要进行路径拼接
          - 1、不同的操作系统的路径分隔符不统一。
          - 2、windows上是 \ 或 /
          - 3、Linux上是 /

```
const path = require('path');
const finalPath = path.join('public', 'upload', 'avatar');
console.log(finalPath); //public\upload\avatar
```

path.join('路径','路径'...)

        - 相对路径VS绝对路径
          - 在Node开发中绝大多数使用的是绝对路径，因为相对路径有时候相对的是命令行工具的当前工作目录。例如readFile的路径时相对于命令行的。写相对路径不安全。
          - 在读取文件或者设置文件路径时都会选择绝对路径。

```
const fs = require('fs');
const path = require('path');
console.log(__dirname);

console.log(path.join(__dirname, 'hello.js'));

fs.readFile(path.join(__dirname, 'hello.js'), 'utf8', (err, doc) => {
    console.log(err);
    console.log(doc);

})
```

使用__dirname获取当前文件所在的绝对路径

    - 第三方模块
      - 什么是第三方模块
        - 别人写好的、具有特定功能的、我们能直接使用的模块即第三方模块，由于第三方模块通常都是由多个文件组成并且被放置在一个文件夹中，所以又名包。
        - 第三方模块有两种存在非方式
          - 1、以js文件形式存在，提供实现项目具体功能的API
          - 2、以命令行工具形式存在，辅助项目开发
      - 获取第三方模块
        - npmjs.com ： 第三方模块的存储和分发仓库 —— 提供了npm工具 （node package manager）node的第三方模块管理工具
        - 下载模块：npm install 模块名称 —— 下载的文件默认在命令行所在的工作目录下面
        - 卸载模块：npm unistall 模块名称
      - 本地安装和全局安装
        - 本地安装表示供当前的项目使用。
        - 全局安装是安装到公共目录，所有项目都可以使用。 —— 一般命令行工具全局安装，库文件本地安装。
      - nodemon
        - nodemon是一个命令行工具，用于辅助项目开发。
        - 在Node.js中，每次修改文件都要在命令行工具中重新执行该文件，非常繁琐。
        - 使用步骤
          - 1、使用npm install nodemon -g下载
          - 2、在命令行工具中用nodemon命令替代node命令执行文件
      - nrm （npm registry manager）
        - npm下载地址切换工具
        - 使用步骤
          - 1、使用npm install nrm -g
          - 2、查询可用的下载地址 nrm ls
          - 3、切换npm下载地址 nrm use 下载地址

Gulp

Gulp是基于node平台开发的前端构建工具

将机械化操作编写成任务, 想要执行机械化操作时执行一个命令行命令任务就能自动执行了 用机器代替手工，提高开发效率。

- gulp能做什么
  - 项目上线，HTML、CSS 、JS文件合并压缩
  - 语法转换es6 、 less
  - 公共文件抽离
  - 修改文件浏览器自动刷新
- gulp使用
  - 1、使用npm install gulp现在gulp库文件
  - 2、在项目更目录下建立gulpfile.js文件
  - 3、重构项目的文件夹结构src目录放置源代码文件dist目录放置构建后文件
  - 4、在gulpfile.js中编写任务
  - 5、在命令行工具中指向gulp任务
- gulp中提供的方法

```
const gulp = require('gulp');
gulp.task('first', () => {
    console.log('第一个gulp任务');

    gulp.src('./src/css/base.css')
        .pipe(gulp.dest('dist/css'));
    //要执行的任务写在pipe管道中
});
```

  - 1、gulp.src('文件路径/文件名称')：获取任务要处理的文件
  - 2、gulp.dest()：输出文件
  - 3、gulp.task()：建立gulp任务 —— gulp.task('name' , callback)
    - name :是任务的名称
    - callback: 回调函数是要执行的任务
  - 4、gulp.watch()：监控文件变化
- gulp插件
  - 1、gulp-htmlmin： html文件压缩
  - 2、gulp-csso： 压缩css
  - 3、gulp-babel:JavaScript语法转化
  - 4、gulp-less:less语法转化
  - 5、gulp-uglify: 压缩混淆JavaScript
  - 6、gulp-file-include 公共文件包含
  - 7、browsersync 浏览器 实时同步
- gulp插件的使用
  - 1、通过npm下载插件
  - 2、引入插件
  - 3、调用插件
  - 实例

```
const htmlmin = require('gulp-htmlmin');
gulp.task('htmlmin', () => {
    gulp.src('./src/*.html')
        .pipe(htmlmin({ collapseWhitespace: true }))
        .pipe(gulp.dest('dist'));
    //collapseWhitespace折叠空格
});
```

html文件的压缩

- package.json文件
  - node_moudules文件夹的问题
    - 1.文件夹以及文件过多过碎，当我们将项目整体拷贝给别人的时候，传输速度会很慢很慢。
    - 1.复杂的模块依赖关系需要被记录，确保模块的版本和当前保持一致，否则会导致当前项目运行报错
    - 解决方法
      - package.json文件
        - 项目描述文件，记录了当前项目信息，例如项目名称，版本、作者、github地址、当前项目依赖了哪些第三方模块等。
        - 使用npm init -y生成该文件
        - 在拿到没有node_moudules文件的项目之后 在终端输入npm install 自动下载相关依赖插件
  - 项目依赖
    - 在项目的开发阶段和线上运行阶段都需要依赖第三方包，称为项目依赖
    - 使用npm install 包名命令下载的文件会默认被添加到 package.json 文件的 dependencies 字段中
  - 开发依赖
    - 在项目开发阶段需要的依赖，线上运行阶段不需要依赖第三方，称为开发依赖。 例如 gulp
    - 使用npm install 包名 --save-dev命令将包添加到package.json文件的 devDependencies字段中
- package-lock.json
  - npm生成的文件详细的记录了模块与模块之间的依赖关系
  - 锁定包的版本，确保再次下载时不会因为包的版本不同而产生问题
  - 加快下载速度，因为该文件已经记录了项目所依赖的第三方包的树状结构和包的下载地址，重新安装只需下载即可，不需要额外的工作。
- Node.js中模块的加载机制（一）
  - 1、require方法根据模块路径查找模块，如果是完整路径，直接引入模块。
  - 2、如果模块后缀省略，先找同名js文件再找同名js文件夹
  - 3、如果找到了同名文件夹，找文件夹中的index.js
  - 4、如果文件夹中没有index.js就会去当前文件夹中的package.json文件中查找main选项中的入口文件。
  - 5、如果指定的入口文件不存在或者没有指定入口文件就会报错，模块没有找到。
- Node.js中模块加载机制（二）
  - 1、既没有路径也没有后缀，Node.js会假设他是系统模块
  - 2、Node.js会去node_moudules文件夹中
  - 4、再看是否有改名字的文件夹
  - 5、如果是文件夹看里面是否有index.js
  - 7、否则找不到报错。