一、实验介绍

1.1 实验内容

本课程使用 Node.js 实现一个简单的静态文件服务器，主要学习 Node.js 的 http 模块，熟悉 Node.js 创建 Web 服务器的过程。同时学习 ES6 的一些新语法，如今 Node.js 是极力推荐 ES6 写法的。

1.2 实验知识点：

- ECMAScript 6 语法
- Node.js 模块的掌握
- 简单 http 协议

1.3 实验环境

- Node.js 6.x

1.4 适合人群

本课程难度一般，属于初级级别课程，适合 JavaScript 新手学习 Node.js 创建 Web 服务器。

二、开发准备

开发静态文件服务器之前我们先来学习一下会用到的 ES6 语法。

let 和 const

`let 和 const 用于声明变量，与 var 类似。`

`let 声明的变量只在 let 所在的代码块内有效，也就让变量有了块级作用域，比如：`

```
{
    let a = 10;
    var b = 10;
}

console.log(a); // 这里会报错，因为 let 定义的变量在此处不可访问。
console.log(a);

// 这在循环中非常有用
for (let i = 0; i < 10; i++) {
    //...
}
console.log(i); // 报错，这里不能访问 i 变量
```

而且，`let 定义的变量不存在变量提升，比如：`

```
console.log(a); // 这里会报错，而不是打印出 undefined
let a = 10;
```

虽然不存在变量提升，但是在局部作用域，只要定义了变量，整个局部作用域内都不能访问到全局作用域中的变量，如：

```
var a = 10;
{
    console.log(a); // 这里会报错
    let a = 5;
}
```

再者， let 不允许相同作用域内重复声明一个变量，如：

```
{
    let a = 10;
    let a = 5; // 这里会报错
}
```

const 与 let 类似，只不过 const 用于声明常量，其他特性和 let 一样，如：

```
const a = 10;
a = 5;
console.log(a); // 10
```

箭头函数

ES6 中新增了箭头函数，如：

```
var fn = () => {
    console.log('hello');
};
// 等价于
var fn = function() {
    console.log('hello')
};
```

如果箭头后面没有大括号，则表示要返回的值，如：

```
var fn = (x) => 2 * x;
// 等价于
var fn = function(x) {
    return 2 * x;
};
```

ES6 中的 class

在 ES6 中，可以直接定义类（class），如：

```
function Person(name, age) {
    this.name = name;
    this.age = age;
}
Person.prototype.talk = function() {
    console.log('talk with me');
};
```

这样一个 Person 类，在 ES6 中可以这样写：

```
class Person {
    // 构造函数
    constructor(name, age) {
        this.name = name;
        this.age = age;
    }
    // class 中的方法可以省略 function 关键字
    talk() {
        console.log('talk with me');
    }
}
```

使用类的方法还是一样：

```
var p = new Pserson('Jack', 25);
p.talk();
```

Promise

Promise 用于异步编程，可以避免无限嵌套的回调函数，用法：

```
// 默认传入两个参数，都是函数
// 成功时执行 resolve
// 失败时执行 reject
var p = new Promise(function(resolve, reject) {
    // 这里进行异步操作

    if (/* 异步操作成功 */) {
        resolve(value);
    } else {
        // 异步操作失败
        reject(error);
    }
});
```

举个栗子：

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

var r = function() {
    return new Promise(function(resolve, reject) {
        // 读取文件
        fs.readFile('./test.txt', function(err, data) {
            if (err) {
                // 读取文件失败
                reject(err);
            } else {
                // 读取文件成功
                resolve(data);
            }
        });
    });
};

// then() 用于添加异步任务执行成功时需要执行的函数
// catch() 用于添加异步任务执行失败时需要执行的函数
r().then(function(data) {
    console.log(data);
}).catch(function(err) {
    console.log(err);
});
```

三、实验步骤

所谓静态文件服务器，其实就是指定服务器上面的一个文件夹，然后启动一个 **http** 服务，使得在这个文件夹中的文件，客户端（通常是浏览器）可以正常访问。

这里实现一个简单的静态文件服务器，实现的主要原理是，通过 fs 模块读取客户端请求的文件内容，然后把读取到的文件内容通过 http 模块发送给客户端。使用 **ES6** 语法，代码看起来会更清晰简洁。

3.1 编写代码

定义 StaticServer 类，新建 StaticServer.js 文件，输入如下代码：

```
'use strict';

// 引入模块
const http = require('http');
const fs = require('fs');
const url = require('url');
const mime = require('mime'); // 用于处理文件的 Conten-Type
                          // 需要安装模块：npm install mime

// 创建并导出 StaticServer 类
module.exports = class StaticServer {
    // 构造函数
    // 初始化时自动调用
    constructor(options) {
        this.currentServer = null;   // http 对象
        this.options = {
            port: 1337,             // 服务器启动的端口
            host: '127.0.0.1',      // 服务器启动的 ip
            filePath: './public',    // 静态文件根目录
            homePage: '/index.html'  // 指定首页文件
        };

        // 把传入构造函数的参数中的值加入到options中
        for (let key in options) {
            this.options[key] = options[key];
        }
    }

    // 服务器启动函数
    run() {
        let self = this;

        // 通过 http.createServer 创建 http 服务
        this.currentServer = http.createServer((req, res) => {
            let tmpUrl = url.parse(req.url).pathname; // 解析客户端请求访问的 url 地址
            let reqUrl = tmpUrl === '/' ? self.options.homePage : tmpUrl; // 如果用户访问的是 '/' 首页，则自动指定读取首页文
件，默认是 'index.html'
            let filePath = self.options.filePath + reqUrl; // 组装文件地址

            // Promise 支持链式调用
            // 这样会使代码的逻辑更加清晰
            // 而不必层层嵌套回调函数
            // Promise 链式调用的条件是
            // 每个 then() 方法都 return 一个 Promise 对象
            // 后面才能跟着调用 then() 方法或者 catch() 方法
            // catch() 方法也要 return 一个 Promise 对象
            // 后面才能接着调用 then() 方法或者 catch() 方法

            // 检测文件是否存在
            self.checkFilePromise(filePath).then(() => {
                // 文件存在则尝试读取文件
                return self.readFilePromise(filePath);
            }).then((data) => {
                // 文件读取成功
                // 发送文件数据
                self.sendData(res, data, reqUrl);
            }).catch(() => {
                // 统一处理错误
                // 文件不存在或者读取失败
                self.catch404(res);
            });

        }).listen(this.options.port, this.options.host, () => {
            console.log('Server is running on ' + this.options.host ':' + this.options.port);
        });
    }

    // 关闭服务
    close() {
        this.currentServer.close(() => {
            console.log('Server closed.');
        });
    }

    // 发送文件内容
    sendData(res, data, url) {
        res.writeHead(200, { 'Content-Type': mime.lookup(url) });
```

```
        res.write(data);
        res.end();
    }

    // 捕获404错误
    catch404(res) {
        res.writeHead(404, { 'Content-Type': 'text/plain' });
        res.write('Error 404. Resource not found.');
        res.end();
    }

    // 使用 promise 包装读取文件的方法
    readFilePromise(path) {
        return new Promise((resolve, reject) => {
            fs.readFile(path, (err, data) => {
                if (err) {
                    reject(err);
                } else {
                    resolve(data);
                }
            });
        });
    }

    // 使用 promise 包装见文件是否可读取的方法
    // fs.access 用于检测文件是否可读取
    checkFilePromise(path) {
        return new Promise((resolve, reject) => {
            fs.access(path, fs.R_OK, (err) => {
                if (err) {
                    reject(err);
                } else {
                    resolve('success');
                }
            });
        });
    }
};
```

OK，代码是不是很简单呢！下面我们来启动服务吧，新建 `test.js` 文件，输入下代码：

```
'use strict';

const StaticServer = require('./StaticServer');

// 创建对象
let server = new StaticServer();

// 启动服务
server.run();

// 停止服务
// server.close();
```

需要在项目目录下创建一个文件夹 `public` ，然后创建一个 `index.html` 文件：

```
<div>home page</div>
```

现在启动测试：

```
$ node test.js
```

然后打开浏览器，访问： `http://localhost:1337` ，看到浏览器上面显示 `home page` ，那么服务就启动成功了。

四、实验总结

通过本课程，我们完成了一个基于 **Node.js** 的静态文件服务器，主要学习了 **Node.js** 的 **http** 模块，同时对 **ECMAScript 6** 语法也有了更加深入的理解。

查看老师的所有课程 > (/teacher/45)

前置课程

NodeJS教程 (/courses/44)

Node.js包教不包会 (/courses/493)

进阶课程

Node.js实现简单爬虫 (/courses/448)

Node.js实现私人笔记 (/courses/446)

NodeJS 经典项目实战 (/courses/455)