LUO JIN 2020-05-14

4. 第三方模块

4.5 第三方模块-gulp

- 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任务就能自动执行了
- 用机械代替手工，提高开发效率

4.6 gulp能做什么？

- 项目上线，HTML,CSS,JS文件压缩合并
- 语法转换（ES6，LESS）
- 公共文件抽离（方便修改）
- 修改文件浏览器自动刷新

4.7 gulp的使用

① 下载：npm install gulp

② 在项目根目录下建立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-放置构建后的文件

④ 在gulpfile.js文件中编写任务

⑤ 在命令行工具执行gulp任务

4.8 gulp中提供的方法

Gulp.src(): 获取任务处理的文件

Gulp.dest(): 输出文件

Gulp.task(): 建立gulp任务

Gulp.watch(): 监控文件的变化

```
const gulp = require('gulp');
// 使用gulp.task()方法建立任务
gulp.task('first', () => {
    // 获取要处理的文件
    gulp.src('./src/css/base.css')
    // 将处理后的文件输出到dist目录
    .pipe(gulp.dest('./dist/css'));
});
```

执行:

Node命令执行gulpfile.js会执行整个文件

想执行其中的first文件

Gulp给我们提供了一个命令行工具：先去下载：npm install gulp-cli -g

这样执行： gulp first(要执行的任务名字)，他会自动你取当前项目根目录下去找gulpfile.js，在文件中招first,执行这个任务的回调函数

基本使用:

```
//1, 引入gulp模块
const gulp =require('gulp');

//2, 使用gulp.task()建立任务
//参数一，任务名称
//参数二，任务回调函数
gulp.task('first', async() => {
    console.log("人生中第一个gulp文件执行了");

    //3, 使用gulp.src()获取要处理的文件
    gulp.src("./src/css/base.css")

    //4, 将处理后的文件输出到dist下的css目录,没有会帮我们创建
    .pipe(gulp.dest('dist/css'));

});

/*1,  done => {
  console.log('Hello World!');
  done();

  2, async() =>{
  }
 解决： The following tasks did not complete: first
 Did you forget to signal async completion?
});*/
```

4.9 gulp插件

Gulp轻内核的第三方模块，提供方法非常少，所有的功能用插件实现

Gulp-htmlmin: html文件压缩

Gulp-csso: 压缩css

Gulp-babel: JavaScript语法转换： ES6-ES5

Gulp-less: less语法转换

Gulp-uglify: 压缩混淆JS

Gulp-file-include：公共文件包含

Browsersync: 浏览器实时同步

插件的使用：

1） 下载

2） 引入

3） 调用

```
//html任务
$ npm install --save gulp-htmlmin
$ npm install --save-dev gulp-file-include
$ npm install gulp-less
$ npm i gulp-csso
$ npm install --save-dev gulp-babel @babel/core @babel/preset-env
$ npm i gulp-uglify
//建立build任务，构建任务
gulp.task('default', gulp.series('htmlmin','cssmin','jsmin','copy',done => done()));
// gulp 4 解决方法
// gulp.task('default', gulp.series('script', 'html', done => done()))

/* gulp AssertionError [ERR_ASSERTION]: Task function must be specified
 // gulp3 可以 gulp4 不可以
gulp.task('default', ['script', 'html'], done => {
 console.log('default')
```

```
    done()
});  */
```

6. package.json 文件

6.1 node_modules文件夹问题：

1）. 文件过多过碎，当我们将项目整体拷贝给别人的时候，传输速度慢很慢

2），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，否则会导致前前项目运行报错

6.2 package.json文件的作用

- Npm 为我们提供了这个项目描述文件，当别人拿到这个项目时，根据这个json文件中所记载的依赖项去下载这个第三方模块。这个项目在别人的电脑就可以运行

- package.json文件主要记录当前项目信息，例如名称，作者，github地址，当前项目依赖了那些那三方模块

- 使用npm init -y命令生成 y: yes, 不填写任何信息，使用默认值

```
{
  "name": "description",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",    // 项目的主入口文件
  "scripts": {    // 命令的别名，当执行命令很长，在命令行使用时很麻烦
    "test": "echo ₩"Error: no test specified₩" && exit 1"
  },
  “keywords”: [],   // 关键字，可以用关键字描述
  "author": "",
  "license": "ISC"  // 开放源代码的协议
}
```

【解决问题1】：

- 下载多个模块，空格隔开： npm i formidable mime
- 当我们给别人传文件的时候，并不会传node_modules文件：
- 别人去命令行： npm install 回车就好， node_modules文件会自己回来

6.3 项目依赖

在项目的开发阶段和线上运营阶段，都需要依赖得到第三方包，成为项目依赖

使用npm install 安装的文件默认会被添加到package.json文件的depedices

6.4 开发依赖

在项目开发阶段需要依赖，线上运营阶段不需要依赖的第三方包

使用npm install 包名 –save-dev 命令 添加到package.json文件的devDepedices

npm install gulp --save-dev: gulp 是开发依赖

npm install 会全部下载

npm install –production 生产环境 运行依赖= depedices

6.5 packjson-lock.json的作用

模块模块的依赖关系，下载地址，版本

锁定包的版本，确保再次下载不会因版本不同产生问题

加快下载速度，因为该文件中已经记录了i项目所依赖第三方包的树状结构和包的下载地址，重新安装只需下载就好，不需额外工作

5. Node.js 模块的加载机制

5.1 模块查找规则-当模块中拥有路径但灭有后缀时

- require('./find.js');
- require('./find');
- require方法根据模块路径查找模块，完整路径的话，直接顺着路径找到引入
- 后缀省略，先找同名JS文件，再找同名JS文件夹，
- 如果找到同名文件夹，找到其中的index.js

- index.js没有的话，去package.js文件去找main选项中的入口文件
- 如果指定入口文件不存在，或者没有就会报错，模块没有找到

5.2 模块查找规则 – 模块没有路径也灭有后缀 require('find');

- node.js会假设他是系统模块，是就执行，不是得话
- 回去node_modules文件夹去找
- 首先看是否有同名字的JS文件
- 再看是否有同名字的文件夹
- 如果是文件夹，看有没有index.js
- 如果没有，看package.js文件去找main选项中的入口文件
- 还没有，报错

5.3. 解决localhost 和127.0.0.1 不一致问题

win10 localhost 解析为::1 的解决办法

.输入命令 netsh interface ipv6 show prefixpolicies，查询ipv6优先级

```
C:\Windows\system32>netsh interface ipv6 show prefixpolicies
Querying active state...

Precedence  Label  Prefix
----------  -----  ----------------------------------
        50      0  ::1/128
        40      1  ::/0
        35      4  ::ffff:0:0/96
        30      2  2002::/16
         5      5  2001::/32
         3     13  fc00::/7
         1     11  fec0::/10
         1     12  3ffe::/16
         1      3  ::/96
```

请注意，IPv6地址（:: / 0）优先于IPv4地址（:: / 96，:: ffff：0：0/96），因此我们可以制定策略，使IPv6不会比任何IPv4地址有利。其中，标签表示优先级，0表示优先级最高，依次类推。

现在需要设置使::/96、::ffff:0:0/96的优先级高于::/0和::1/128，在命令行中依次设置优先级：

netsh int ipv6 set prefix ::/96 50 0

netsh int ipv6 set prefix ::ffff:0:0/96 40 1

netsh int ipv6 set prefix 2002::/16 35 2

netsh int ipv6 set prefix 2001::/32 30 3

netsh int ipv6 set prefix ::1/128 10 4

netsh int ipv6 set prefix ::/0 5 5

netsh int ipv6 set prefix fc00::/7 3 13

netsh int ipv6 set prefix fec0::/10 1 11

netsh int ipv6 set prefix 3ffe::/16 1 12

```
C:\Windows\system32>netsh int ipv6 set prefix ::ffff:0:0/96 40 1
Ok.


C:\Windows\system32>netsh int ipv6 set prefix 2002::/16 35 2
Ok.


C:\Windows\system32>netsh int ipv6 set prefix 2001::/32 30 3
Ok.


C:\Windows\system32>netsh int ipv6 set prefix ::1/128 10 4
Ok.


C:\Windows\system32>netsh int ipv6 set prefix ::/0 5 5
Ok.


C:\Windows\system32>netsh int ipv6 set prefix fc00::/7 3 13
Ok.


C:\Windows\system32>netsh int ipv6 set prefix fec0::/10 1 11
Ok.


C:\Windows\system32>netsh int ipv6 set prefix 3ffe::/16 1 12
Ok.
```

在查询优先级

```
C:\Windows\system32>netsh interface ipv6 show prefixpolicies
Querying active state...

Precedence  Label  Prefix
----------  -----  ----------------------------------
        50      0  ::/96
        40      1  ::ffff:0:0/96
        35      2  2002::/16
        30      3  2001::/32
        10      4  ::1/128
         5      5  ::/0
         3     13  fc00::/7
         1     12  3ffe::/16
         1     11  fec0::/10
```

可以看到，此时:: / 96，:: ffff：0：0/96优先级高于::/0了。ping一下localhost,

```
C:\Windows\system32>ping localhost

Pinging DESKTOP-RJORK2A [127.0.0.1] with 32 bytes of data:
Reply from 127.0.0.1: bytes=32 time<1ms TTL=128
Reply from 127.0.0.1: bytes=32 time<1ms TTL=128
Reply from 127.0.0.1: bytes=32 time<1ms TTL=128
Reply from 127.0.0.1: bytes=32 time<1ms TTL=128

Ping statistics for 127.0.0.1:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 0ms, Maximum = 0ms, Average = 0ms
```

1， 服务器端基础概念

1.1 网站组成

客户端：用户看到并与之交互的界面程序

服务器端：存数据，处理应用逻辑

1.2Node网站服务器

能够提供网站服务的机器，能够接收客户端请求，能够对其做出响应

1.3 IP地址

设备的唯一标识，网络协议地址-家庭地址

1.4 域名

上网作用的网址，与IP一一对应关系，将域名转为IP访问的

1.5 端口

食堂：窗口=计算机与外界通讯的出口，用来区分服务器电脑中提供的不同服务

1.6 URL

（uNIform Resource Locator）：统一资源标识符，是专为Internet网上资源位置而设的一种编制方式 = 网页地址

URL 组成

传输协议://服务器IP或域名:端口/资源所在位置标识

http://www.itcast.cn/news/20181018/09152238514.html

http：超文本传输协议，提供了一种发布和接收HTML页面的方法。

1.7 开发过程中客户端和服务器端说明

- 在开发阶段，客户端（浏览器）和服务器（node）使用同一台电脑，就是开发人员电脑
- 本地域名：localhost
- 本地IP：127.0.0.1

2，创建 web 服务器

2.1 创建服务器代码

```
//引用系统模块
const http = require('http');
//创建web服务器
const app = http.createServer();
//当客户端发送请求的时候
app.on('request', (req, res) => {
    // 响应
    res.end('<h1>hi, user</h1>');
});
//监听3000端口
app.listen(3000);
console.log('服务器已启动，监听3000端口，请访问localhost:3000')
```

访问：localhost：3000

命令行：node app.js / nodemon app.js(自动监听)

```
// 1, 引入创建web服务器模块
const http = require("http");
// app 对象就是网站服务器对象
const app = http.createServer();

// 2, 当客户端有请求来的时候, 执行回调函数
app.on('require', (req, res) => {
    // 3. 请求来了调用res.end方法响应请求
    res.end('<h2> hi!  luojin! i recieved your request!
});

// 4, 监听端口, 向外界提供服务
app.listen(8000);
console.log("web server 启动成功啦！");
```

3. HTTP 协议

3.1 HTTP协议的概念

和客户端和服务器端沟通的协议：超文本（HTML）传输协议（请求和响应的标准），

3.2 报文

在HTTP请求和响应的过程中传递的数据块（说明文）就叫报文。包括传的数据和附加信息，并且要遵守规定好的格式

3.3 请求报文

1，请求方式（request method）

Get 请求数据 - 获取数据

Post 发送数据 – 添加数据，一般登陆，安全post

2，请求地址：

RequestURL

```
app.on('request', (req, res) => {
    req.headers  //获取请求报文
    req.url      //获取请求地址
    req.method   //获取请求方法
});
```

3.4 响应报文

1，HTTP状态码

① 200 正常，请求成功

② 404 请求资源没有找到

③ 500 服务器端错误

④ 400 客户端请求有语法错误

2，内容类型

① text/html

② text/css

③ application/javascript

④ images/jpeg

⑤ application/json

4. HTTP 请求与响应处理

4.1 请求参数

客户端向服务器发送请求时，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，比如登陆操作。

4.2 GET请求参数

参数被放在浏览器的地址栏中：

- http://localhost:3000/?name=changsan&age=20：键值对

- 如何获取请求参数？

4.3 POST请求参数

- 被放在请求体中进行传输
- 获取POST 参数需要使用data 事件和end对象
- 使用querystring 系统模块将参数转化为对象格式

```
// 导入系统模块querystring用于将HTTP参数转换为对象格式
const querystring = require('querystring');
app.on('request', (req, res) => {
    let postData = '';
    //监听参数传输事件
    req.on('data', (chunk) => postData += chunk;);
    //监听参数传输完毕事件
    req.on('end', () => {
        console.log(querystring.parse(postData));
    });
});
```

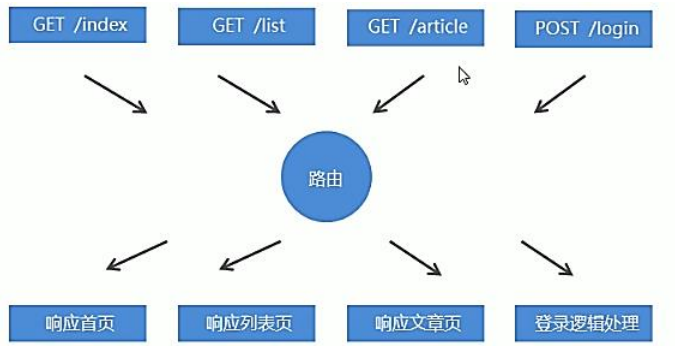
4.4 路由

http://localhost:3000/index

http://localhost:3000/login

访问那个请求地址就会登陆到那个页面的内容，通过路由就可以做到

路由：客户端请求地址和服务器程序代代码的对应关系，就是请求什么响应什么



```
// 当客户端发来请求的时候
app.on('request', (req, res) => {
    //获取客户端的请求路径
    let { pathname } = url.parse(req.url);
    if (pathname == '/' || pathname == '/index') {
        res.end('欢迎来到首页');
    } else if (pathname == '/list') {
        res.end('欢迎来到列表页页');
    } else {
       res.end('抱歉,您访问的页面出游了');
    }
});
```

这个代码中，忽略了来请求方式的判断；请求地址—请求逻辑-开发人员决定的

4.5 静态资源

服务器端不需要处理，可以直接响应给客户端的资源，例如CSS，JS，image文件

不论谁访问这个网站得到的结果都是一样的

- 静态资源访问功能：
- 在服务器端专门创建一个文件夹，放这些静态资源，客户端请求来时，直接响应给她
- 双击只能说明文件在自己的电脑上，被用户访问，就需要实现静态资源访问功能

4.6 动态资源

不同的请求地址不同的响应资源；比如亡之后的id=1,id=2

5. Node.js 异步编程

5.1 同步API，异步API

同步API ：只有当前API执行完成后，才能执行下一个API，代码只能一行一行的执行

异步API：当前API的执行，不会印象后续代码的执行

```
//路径拼接
const public = path.join(__dirname, 'public');
//请求地址解析
const urlObj = url.parse(req.url);
//读取文件
fs.readFile('./demo.txt', 'utf8', (err, result) => {
    console.log(result);
});
```

5.2 区别：获取返回值

同步API 可以从返回值中拿到API执行结果，异步API不可以

```
//同步
function sum (n1, n2) {
    return n1 + n2;
}
const result = sum (10, 20);

//异步
function getMsg () {
    setTimeout(function () {
        return { msg: 'Hello Node.js' }
    }, 2000);
}
const msg = getMsg ();
```

5.3 回调函数

自己定义，别人调用

Function getData (callback) { }

getData (() = {})

5.5 区别：代码执行顺序

- 同步API 从上到下依次执行，前面到代码会阻塞后边代码的执行

for (var i=0; i<10; i++){ console.log(i);}

console.log("for 循环后的代码")

- 异步不会等待API执行完后在执行代码： 先将所有的同步执行完，在执行异步API；定时器属于异步API，会放在异步代码执行区，里边的回调函数放在回调函数队列，不会执行；

继续找同步API

5.7 Node.js中的异步API

1）读取文件的操作：也是需要花时间的，不能通过返回值获取，通过回调函数参数的形式传递 Fs.readFile('./demo.txt'，（err,result） => {})

2）事件监听的API：事件处理函数=回调函数，自己不用调用，系统调用var server = http.createServe(); server.on('request',(req, res) => {})

3） 如果异步API 后边代码的执行结果依赖当前异步API的执行结果，但实际上后续代码在执行的时候异步API还没有返回结果，这个问题怎么解决呢？

- fs. readFile(....)

- Console.log("文件读取成功")

- 前边还没执行，后边已经执行了。那我们吧 Console.log("文件读取成功")放在回调函数里不就完了

4）但是需要依次读取：A， B， C 文件，需要通过都要写到回调函数里，回调函数嵌套的问题。

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

fs.readFile('./1.txt', 'utf8',(err, result1) => {
    console.log(result1);
    fs.readFile('./2.txt', 'utf8', (err, result2) => {
        console.log(result2);
        fs.readFile('./3.txt', 'utf8', (err, result3) => {
            console.log(result3);
        });
    });
});
```

如果这样嵌套很多层，就不好维护，自己都不想看了。= 回调地狱，一大问题

5.8 Node.js promise 来解决上边回调地狱问题

```
let promise = new Promise((resolve, reject) => {
    setTimeout(() => {
        if (true) {
            resolve({name: '张三'})
        }else {
            reject('失败了')
        }
    }, 2000);
});
promise.then(result => console.log(result); // {name: '张三'})
        .catch(error => console.log(error); // 失败了)
```

是一个构造函数

5.9 异步函数

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，

```
Const fn = async() => {}

Async function fn() { }

// 1，在普通函数定义前加上async关键字，就变成异步函数了

// 2，异步函数默认返回值是promise对象

// 3，在异步函数内部使用throw关键字抛出错误


// await 关键字

// 1，只能出现在异步函数内部中

// 2， 后边跟 promise 对象，暂停异步函数的执行，等待promise对象返回结果后在向下执行


/*async function fn() {
        throw 'error is made'
        return 123; // promise { 123 }
}
fn().then(function (){
        console.log(data)  //123
}).catch(function(err){
        console.log(err)
})*/


async function p1(){
        return 'p1'
}
async function p2(){
```

```
        return 'p2'
}
async function p3(){
        return 'p3'
}
// 保证顺序执行
async function run() {
        let r1 = await p1()
        let r2 = await p2()
        let r3 = await p3()
        console.log(r1)
        console.log(r2)
        console.log(r3)
}
run()
```

async关键字

1. 普通函数定义前加async关键字 普通函数变成异步函数
2. 异步函数默认返回promise对象
3. 在异步函数内部使用return关键字进行结果返回 结果会被包裹的promise对象中 return关键字代替了resolve方法
4. 在异步函数内部使用throw关键字抛出程序异常
5. 调用异步函数再链式调用then方法获取异步函数执行结果
6. 调用异步函数再链式调用catch方法获取异步函数执行的错误信息

await关键字

1. await关键字只能出现在异步函数中
2. await promise await后面只能写promise对象 写其他类型的API是不不可以的
3. await关键字可是暂停异步函数向下执行 直到promise返回结果

```
const fs = require('fs')
// 1, 改造现有异步api,让其返回promise对象，从而支持异步函数语法
const promisify = require('util').promisify
// 2, 调用promisify方法改造现有异步API，让其返回promise对象
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()
```

3. Nodejs 快速入门全局对象 global

3.3 Node.js全局对象是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() 在控制台中输出
- setTimeout() 设置超时定时器
- clearTimeout() 清除超时时定时器
- setInterval() 设置间歇定时器
- clearInterval() 清除间歇定时器