# Node.js基础

## 模块化开发

b.js

const add = (x, y) => x + y

exports.add = add;

a.js

var ae = require("./b.js");

console.log(ae.add(10, 20));

**导出方法还有module.exports**

**两者是地址引用关系，同时使用，以module.exports为主**

### 具体实例

**加载(require)**

var obj=require("./b.js")

console.log(obj.name)

console.log(obj.add(10,20))

**导出**

**单个成员：**  
module.exports.a=231

module.exports.add=function(a.b){

var c=a+b

console.log(c)

}

Module.export={

A:1,

Function add(a,b){

Var c=a+b

}

}

**多个成员**：exports默认是个空对象

exports.name="小明"

exports.add=function (a,b){

    var c=a+b

    console.log(c)

}

Exports() 默认是个空对象

## 系统模块

### 读取文件内容

fs.readFile("文件名及地址",【“编码”】,回调函数)

var fs=require("fs")

fs.readFile("./data.txt",function(err,data){

    //读取成功时，err是null;data中是读取的数据

    if(err){

        console.log("读取文件失败")

        return

    }else{

        console.log(data.toString())

    }

})

### 向文件写入内容

fs.writeFile("文件名及地址","写入内容",回调函数)

var fs = require("fs");

fs.writeFile("./ceshi2", "你好！tom11你好！tom11", err => {

    if (err != null) {

        console.log(err)

        console.log("写入失败")

        return

    }

    console.log("写入成功")

})

### Path模块

#### 1. 路径拼接

var path = require("path")

var filePath = path.join("tom", "act", "aaa")

console.log(filePath)

#### 2.使用绝对路径来读取文件

var fs = require("fs")

var path = require("path")

var filePath = path.join(\_\_dirname, "ceshi2")

fs.readFile(filePath, "utf-8", (err, data) => {

    if (err) {

        console.log("err")

        return

    }

    console.log(data)

})

### 开启一个服务器

var http = require("http")

var url = require("url")

    //创建一个web服务器

var server = http.createServer()

//req可以获取，客户端的请求地址

//res可以向客户端发送信息

server.on("request", function(req, res) {

        var { pathname, query } = url.parse(req.url, true)

        console.log(query)

        console.log(query.name)

        console.log(query.age)

        if (pathname === "/index") {

            res.end("index")

        } else {

            console.log("404 Not Find")

        }

    })

    //绑定端口号

server.listen(3000, function() {

    console.log("服务器已开启，通过http://127.0.0.1:3000 进行访问")

})

## 第三方模块

#### 通过npm install 下载的模块

#### 一些js文件

##### nodemon

##### nrm (切换npm下载地址)

##### glup(前端构建工具),公共文件抽离，自动刷新浏览器（本身功能较少，依靠插件）

用于文件压缩，es6转es5，less

## 常见的问题

1. 变量具有模块作用域
2. 通过require() 可以调用其他模块

3..toString()可以将所有的的数据都转换为字符串，但是要排除null 和 undefined

**4.JSON 的常规用途是同 web 服务器进行数据交换。**

**在向 web 服务器发送数据时，数据必须是字符串。**

**通过**JSON.stringify()**把 JavaScript 对象转换为字符串。**

5.ctrl+f oschina

6. indexOf() 方法可返回某个指定的字符串值在字符串中首次出现的位置。

<script type="text/javascript">

var str="Hello world!"

document.write(str.indexOf("Hello") + "<br />")

document.write(str.indexOf("World") + "<br />")

document.write(str.indexOf("world"))

</script>

以上代码的输出：

0

-1

6

## 服务器与客户端交流

#### 1.http状态码

200 请求成功

400 客服端请求有语法错误

404 请求的资源没有被找到

500 服务器端错误

#### 2. res.setHeader()和res.writeHead()

在node.js中，http协议如果不设置响应头，中文的话在客户端浏览器是会乱码的

设置响应头的目的是为了告诉浏览器我发送的数据是什么类型的，你应该用什么格式来编码显示。如果不设置，会自动生成一个响应头，但中文的话浏览器会乱码。

在http协议中， Content-type就是用来告诉对方我给你发送的数据内容是什么类型

如何设置响应头，我认为有两种方法：

1. res.setHeader()
2. res.writeHead()

res.setHeader()的写法：

|  |  |
| --- | --- |
| 1 | res.setHeader('Content-type', 'text/plain;charset=utf-8') |

res.writeHead()的写法：

|  |  |
| --- | --- |
| 1 2 3 4 | 参数1：必选，三位数的http状态码 参数2：可选，可有可无 参数3：可选，告诉浏览器我发给你的数据是什么类型的 res.writeHead(200, { 'Content-type': 'text/html;charset=utf-8' }) |

**注意：**

1. res.writeHead()必须在res.end()之前调用
2. 如果两者同时存在（没必要），要先写res.setHeader()，后写res.writeHead()，且res.writeHead()优先

#### url请求参数(get)

get请求数据在url中

var http = require("http")

var url = require("url")

    //创建一个web服务器

var server = http.createServer()

//req可以获取，客户端的请求地址

//res可以向客户端发送信息

server.on("request", function(req, res) {

        var { pathname, query } = url.parse(req.url, true)

        console.log(query)

        console.log(query.name)

        console.log(query.age)

        if (pathname === "/index") {

            res.end("index")

        } else {

            console.log("404 Not Find")

        }

    })

    //绑定端口号

server.listen(3000, function() {

    console.log("服务器已开启，通过http://127.0.0.1:3000 进行访问")

})

#### url请求参数(post)

post请求在请求报文中

var http = require("http")

var querystring = require("querystring") //可将参数转化为对象模式

    //创建一个web服务器

var server = http.createServer()

//req可以获取，客户端的请求地址

//res可以向客户端发送信息

server.on("request", function(req, res) {

        let postParse = ""

        req.on("data", params => { //监听参数传输事件，data表示事件在传输，end表示结束

            postParse += params

        })

        req.on("end", () => {

            var parse = querystring.parse(postParse)

            console.log(parse.username)

            console.log(parse.password)

        })

        res.end("ok")

    })

    //绑定端口号

server.listen(3001, function() {

    console.log("服务器已开启，通过http://127.0.0.1:3000 进行访问")

})

## Promise解决回调地狱

### Promise的基本使用

var fs = require("fs")

    //promise就是利用回调函数，来将原来的数据，转到promise对象上来解决

var promise = new Promise((res, ref) => { //res成功时调用，ref失败时调用

    fs.readFile("./1.txt", "utf-8", (err, data) => {

        if (err) {

            ref(err) //将错误导出

        } else {

            res(data) //将成功导出

        }

    })

})

promise.then(data => {

    console.log(data)

}).catch(err => {

    console.log(err)

})

### 利用Promise解决回调地狱

var fs = require("fs")

function p1() { //通过函数来控制执行顺序

    return promise = new Promise((res, ref) => { //通过return ,函数p()才能.then

        fs.readFile("1.txt", "utf-8", (err, data) => {

            res(data)

        })

    })

}

function p2() {

    return promise = new Promise((res, ref) => {

        fs.readFile("2.txt", "utf-8", (err, data) => {

            res(data)

        })

    })

}

function p3() {

    return promise = new Promise((res, ref) => {

        fs.readFile("3.txt", "utf-8", (err, data) => {

            res(data)

        })

    })

}

p1().then(data => {

    console.log(data)

    return p2()

}).then(data => {

    console.log(data)

    return p3()

}).then(data => {

    console.log(data)

})

## 异步函数

异步函数是异步编程的终极方案，以同步的形式来解决一部编程

### async

1. 在普通函数前加async关键字，普通函数就变成异步函数
2. 默认返回值是一个promise对象
3. 有 await 关键字，可以暂停异步函数执行，等待本次异步操作返回结果再执行

### async的基本使用

用法一：当成功时 .then(()=>{})

async function getData(){

return "syy";

}

getData().then(data=>{

console.log(data) //syy

});

用法二：async函数内部抛出错误，会导致返回的 Promise 对象变为reject状态，抛出的错误对象会被catch方法回调函数接收到 .catch(()=>{})

async function getData(){

throw new Error('出错了');

console.log(data) //这句将无法输出

}

getData()

.then(v=>{

console.log(v)

})

.catch(e=>{

console.log(e) //Error: 出错了

});

用法三：使用await

async function p1() {

    return "1"

}

async function p2() {

    return "2"

}

async function p3() {

    return "3"

}

async function run() {

    var r1 = await p1()

    var r2 = await p2()

    var r3 = await p3()

    console.log(r1)

    console.log(r2)

    console.log(r3)

}

run()

### async在node中的应用

const fs = require("fs")

const promisify = require("util").promisify //用于改造现有异步API，让其返回PROMISE对象，从而支持异步语法

const readFile = promisify(fs.readFile)

async function run() {

    let r1 = await readFile("1.txt", "utf-8")

    let r2 = await readFile("2.txt", "utf-8")

    let r3 = await readFile("3.txt", "utf-8")

    console.log(r1)

    console.log(r2)

    console.log(r3)

}

run()