# 初始Node.js与内置模块

## 1. 初始Node.js

### 1.1. Node.js简介

* 1. 什么是Node.js

Node.js是一个基于Chrome V8引擎的JavaScript运行环境。

* 1. Node.js中的JavaScript运行环境

浏览器是JavaScript的前端运行环境；

Node.js是JavaScript的后端运行环境；

Node.js中无法调用DOM和BOM等浏览器内置对象

### 1.2. 在Node.js环境中执行JavaScript代码

在终端中输入node js文件名，切换至js文件所处目录。

## 2. fs文件系统模块

### 2.1. 什么是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。提供一系列的方法和属性用来满足用户对文件的操作需求。

如果要在JavaScript代码中使用fs模块来操作文件，则需要使用const fs = require(‘fs’)来导入。

### 2.2. 读取指定文件中的内容

1. fs.readFile()可读取指定文件中的内容

fs.readFile(‘path’[, ‘options’], callback)

path：必选参数，字符串，表示文件路径(相对路径)；

options：可选参数，表示以什么编码格式来读取文件；

callback：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. 判断文件是否读取成功

判断err对象是否为null，若为null，表示读取成功；若不为null，则表示读取失败。

### 2.3. 向指定文件中写入内容

1. fs.writeFile()向指定文件中写入内容

fs.writeFile(‘file’, ‘data’[, ‘options’], callback)

1. 判断文件是否写入成功

判断err对象是否为null，若为null，表示写入成功；若不为null，则表示写入失败。

注意：①fs.writeFile()只能用来创建文件，能不能用来创建路径；②重复调用fs.writeFile()，会发生覆盖。

### 2.4. 路径动态拼接问题

执行fs.readFile()读取指定文件时，如果js执行文件中提供的被读取文件的路径是以./或../开头的相对路径，则会以执行node命令时所处的目录动态拼接js执行文件中提供的被读取文件的相对路径，形成完整路径。若执行node命令时所处目录不完整，则无法读取到指定路径。



将相对路径换成被读取文件的绝对路径可解决路径动态拼接问题，但代码可移植性差，不利于维护。

解决方法：\_\_dirname可获得js执行文件所在的目录，拼接上被读取文件的路径。



## 3. path路径模块

### 3.1. 什么是path路径模块

path模块是Node.js官方提供的、用来处理路径的模块。提供一系列的方法和属性用来满足用户对路径的处理需求。

如果要在JavaScript代码中使用path模块来处理路径，则需要使用const path = require(‘path’)来导入。

### 3.2. 路径拼接

path.join([…paths])可把多个路径片段拼接为完整的路径字符串

…paths：路径片段的序列

### 3.3. 获取路径中的文件名

1. path.basename(path, [.文件扩展名])可以从一个文件路径中获取文件的名称

.文件扩展名：可选参数，获取的文件名称无扩展名

1. path.extname(path)可以从文件路径中获取文件的扩展名部分

## 4. http模块

### 4.1. 什么是http模块

http模块是Node.js官方提供的、用来创建web服务器的模块。

如果要在JavaScript代码中使用http模块来创建web服务器，则需要使用const http= require(‘http’)来导入。

### 4.2. 创建web服务器

1. const http= require(‘http’)导入http模块；
2. const server = http.createServer()创建web服务器实例；
3. server.on(‘request’, function(req, res) {console.log(“Someone visit web server.”)})
4. 为服务器实例绑定request事件，监听客户端的请求，只要有客户端请求服务器就会触发request事件；
5. req是请求对象，包含了与客户端相关的数据和属性，例如：req.url, req.method等；
6. res是响应对象，res.end()向客户端发送指定内容，并结束这次请求的处理；
7. res.setH eader(‘Content-Type’, ‘text/html; charset=utf-8’)可解决res.end()响应内容有中文时的乱码问题。
8. server.listen(80, function() {console.log("Server running at 127.0.0.1:80");})启动服务器。

### 4.3. 根据不同的url响应不同的html内容

1. 获取请求的url地址；
2. 设置默认的响应内容为404 Not Found；
3. 判断用户请求的url，不同url对应不同的页面内容。

# 二、模块化

## 1. 模块化的基本概念

### 1.1. 什么是模块化

模块化就是遵守固定的规则，把一个大文件拆成独立并相互依赖的多个小模块。

模块化的好处：①提高了代码的复用性；②提高了代码的可维护性；③可以实现按需加载。

## 2. Node.js中的模块化

### 2.1. Node.js中模块的分类

1. 内置模块；
2. 自定义模块；
3. 第三方模块：使用前需先下载。

### 2.2. 加载模块

加载内置模块和第三方模块：const xxx = require(‘模块名’)；

加载自定义模块：const xxx = require(‘./自定义模块路径’)；

使用require()方法加载模块时，会执行被加载模块中的代码；可省略.js后缀名。

### 2.3. Node.js中模块作用域

1. 什么是模块作用域

在自定义模块中定义的变量、方法等，只能在当前模块内被访问。

1. 模块作用域的好处

防止全局变量污染和问题。

### 2.4. 向外共享模块作用域中的成员

1. module对象

每个.js自定义模块中都有一个module对象，其中存储了和当前模块有关的信息。

1. module.exports对象

module.exports对象默认为空，它可以将模块内的成员共享出去。

外界使用require()方法导入自定义模块时，得到的就是module.exports所指向的对象。

1. 共享成员时的注意点

永远以module.exports指向的对象为准。

1. export对象

默认情况下module.exports和exports指向同一个对象，最终共享的结果还是以module.exports指向的对象为准。

### 2.5. Node.js中的模块化规范

1. 每个.js模块中都有一个module对象，module代表当前模块；
2. module是一个对象，exports是module的属性，module.exports是对外的接口；
3. 导入模块/加载模块其实就是加载模块的module.exports。

## 3. npm与包

### 3.1. 什么是包

Node,js中的第三方模块就是包。包是基于内置模块封装出来的，可以极大的提高开发效率。

### 3.2. 使用包

1. 安装包

npm install 包的完整名称/npm i 包的完整名称

一次安装多个包是，以空格分隔。

1. 安装指定版本的包

npm i 包的完整名称@版本号

### 3.3. 包管理配置文件

npm规定，在项目根目录中，必须提供一个package.json的包管理配置文件，用来记录与项目有关的一些配置信息。

1. 多人协作问题

安装包后会产生node\_modules文件夹，用来存放所有已安装至项目的包。由于第三方包的体积过大，不方便使用Git共享，所以在共享时需要剔除node\_modules，只共享项目源代码。

1. 如何记录项目中已安装的包

由于只共享项目源代码，其他人使用时未安装包，无法运行项目，就需要知道项目使用了哪些包。

在项目根目录中，创建一个package.json的配置文件，用来记录运行项目需要安装哪些包。

1. 创建package.json配置文件

npm init -y

此命令只能在英文的目录下成功运行，故项目文件夹的名称应使用英文，并且不能出现空格。

运行此命令后再安装包时，npm包管理工具会自动把包的名称版本号记录到package.json中。

1. dependencies节点

package.json配置文件中的dependencies节点是用来记录已安装的核心依赖包包名和版本号(核心依赖包为开发阶段和项目上线后都会使用的包)。

1. 一次性下载项目所需要的包

从Git上拉取的项目无node\_module文件后，需要自行下载package.json配置文件中的包，才能运行项目。

npm install/npm i //此命令可一次性下载项目所需要的所有包

1. 卸载包

npm unistall 包的完整名称 //执行此命令后，会把指定的包从package.json配置文件中的dependencies节点中移除。

1. devDependencies节点

package.json配置文件中的devDependencies节点是用来记录已安装的开发依赖包包名和版本号(开发依赖包为开发阶段会使用，项目上线后不会使用的包)。

npm i 包名 -D/npm install 包名 –save-dev //此命令可将指定的包记录至devDependencies节点中

### 3.4. 解决下包速度慢的问题

默认直接从国外的服务器中下载包导致下载速度慢，淘宝NPM镜像服务器解决了下包速度慢的问题。

1. 方法1：npm config get registry //查看当前下包的镜像源

npm config set registry=https://registry.npm.taobao.org/ //将下包的镜像源切换为淘宝镜像源

npm config get registry //检查镜像源是否切换成功

1. 方法2：npm i nrm -g //通过npm管理器，将nrm安装为全局可用的工具

nrm ls //查看所有所有可用的镜像源

nrm use taobao //将下包的镜像源切换为淘宝镜像源

### 3.5. 包的分类

npm包管理工具下载的包可分为两类，分别是项目包和全局包。

1. 项目包：被安装到项目的node\_modules目录中的包。项目包又分为开发依赖包和核心依赖包，开发依赖包被记录到devDependencies节点中，核心依赖包被记录到dependencies节点中。
2. 全局包：执行npm install命令安装包时，添加了-g参数，安装的包即为全局包。
3. i5ting\_toc：可以将.md文档转换为.html页面的全局工具

npm i -g i5ting\_doc //执行此命令安装i5ting\_doc全局包

i5ting\_toc -f 要转换的.md文件路径 -o

### 3.6. 规范的包结构

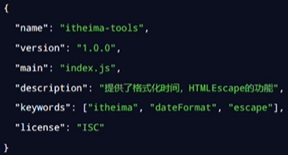
1. 包必须以单独的目录而存在；
2. 包的顶级目录下必须包含package.json这个包管理配置文件；
3. package.json中必须包含name, version, main这三个属性。main为包的入口。

### 3.7. 开发自己的包

1. 初始化包的基本结构
2. 新建一个文件夹为包的根目录；
3. 在根目录下新建package.json(包管理配置文件), index.js(包的入口文件), README.md文件。

注：在导入包时，未写明具体文件时，requir()方法会寻找包文件夹下的pachage.json文件，找到文件中的main属性。

1. 初始化package.json



name: 包名具有唯一性，使用前需在npm中查验是否唯一。

1. 在index.js中定义实现功能的方法
2. 在index.js中定义转义HTML的方法



转义后代码被直接呈现，防止用户通过嵌入js代码攻击网站。

1. 在index.js中定义还原HTML的方法



1. 将不同的功能进行模块化拆分
2. 将功能模块拆分为一个.js文件；
3. 将处理HTML字符串的功能拆分为一个.js文件；
4. 在index.js中导入模块，得到需要向外共享的方法；
5. 在index.js中使用module.exports把对应的方法共享出去。
6. 编写包的README.md文档

### 3.8. 发布包

1. 注册npm账号；
2. 登录npm账号；在终端中执行npm login命令，根据提示输入用户名、密码、邮箱(在执行npm login命令前，必须先把下包的服务器地址切换至npm官方服务器地址)；
3. 把包发布到npm上；将终端路径切换至包的根目录后执行npm publish命令(包名唯一)。
4. 删除已发布的包；在终端中执行npm unpublish 包名 –force命令。此命令只能删除72小时内发布的包，已删除的包，24小时内不允许重复发布。

## 4. 模块的加载机制

### 4.1. 优先从缓存中加载

不论内置模块、用户自定义模块或第三方模块在第一次加载以后都会被缓存，导致require()多次导入模块不会被多次执行，从而提高模块的加载效率。

### 4.2. 内置模块的加载机制

内置模块的加载优先级最高。

### 4.3. 自定义模块的加载机制

加载自定义模块：const xxx = require(‘./自定义模块路径’)，可省略.js文件扩展名。

加载自定义模块时，必须以./或../开头的路径标识符，否则node会把它当作内置模块或第三方模块进行加载。

省略.js文件扩展名后，Node.js会按照一下顺序加载自定义模块，否则加载失败，终端报错。

1. 带有文件扩展名的确切文件；
2. 补全.js；
3. 补全.json；
4. 补全.node；

### 4.4. 第三方模块的加载机制

如果require()加载的模块不是内置模块和自定义模块，则Node.js会从当前模块的父目录开始，从node\_modules文件夹中加载第三方模块。如果没有找到对应的第三方模块，则移动至再上一层父目录中，进行加载，直到文件系统的根目录。

### 4.5. 目录作为模块

1. 在被加载的目录下查找package.json文件，寻找其中的main属性作为加载的入口；
2. 如果目录中没有package.json文件或main属性不存在或无法解析，则加载目录下的index.js文件；
3. 如果以上都失败，则终端报错。

# 三、express

## 1. express

### 1.1. express简介

express是基于Node.js平台，快速、开放、极简的Web开发框架。本质为npm上的一个第三方包，提供创建Web服务器的便捷方法。

Express是基于内置模块http进一步封装出来的，可以提高开发效率。

Express可以方便快捷的创建Web网站的服务器或API接口的服务器。

### 1.2. Express基本使用

1. 安装express包

在终端中执行npm i express@版本号命令

1. 创建基本的Web服务器

const express = require(‘express’) //导入express包

const app = express() //创建web服务器

app.listen(80, function() { //调用app.listen(端口号， 启动成功后的回调函数)启动服务器

console.log(‘express server running at <http://127.0.0.1:80>’)

})

1. 监听GET请求

app.get(‘请求URL’, function(req, res) {/\*处理函数\*/})

1. 监听POST请求

app.post(‘请求URL’, function(req, res) {/\*处理函数\*/})

1. 把内容响应给客户端

app.get(‘请求URL’, function(req, res) {

res.send({/\*向客户端发送JSON对象\*/})

})

app.post(‘请求URL’, function(req, res) {

res.send(‘/\*向客户端发送文本内容\*/’)

})

1. 获取URL中携带的查询参数

req.query //默认req.query是个空对象

1. 获取URL中的动态参数

app.get(‘请求URL/:id’, function(req, res)

res.send(res.params) // res.params默认是一个空对象，里面存放着通过:匹配到的参数值

/})

注：id是一个不固定的参数名，可自主命名。

### 1.3. 托管静态资源

1. app.use(express.static(‘文件’))

express.static()可创建一个静态资源服务器

express在指定的静态目录中查找文件并对外提供资源的访问路径。因此存放静态文件的目录名不会出现在URL中。

1. 托管多个静态资源文件

多次调用express.static()。访问静态资源文件时，会根据目录的添加顺序查找所需要的文件。

1. 挂载路径前缀

app.use(‘/public’, express.static(‘public’)) //在托管的静态资源访问路径前挂载路径前缀，可通过带有/public前缀地址的来访问public目录中的文件

1. nodemon

nodemon能够监听项目文件的变动，当代码被修改后，nodemon会自动重启项目。

执行npm i -g nodemon 安装nodemon全局包，安装成功后执行项目时使用nodemon xxxx执行。

## 2. Express路由

### 2.1. 路由的概念

1. 路由就是映射关系。在express中路由指的是客户端的请求与服务器处理函数之间的映射关系。

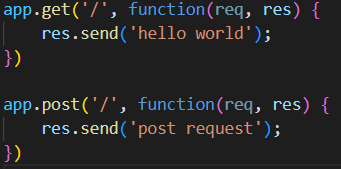
express中的路由由请求的类型、请求的URL地址和处理函数组成。

1. 路由的匹配过程

每当一个请求到达服务器之后，需要先经过路由的匹配，匹配成功后，才会调用对应的处理函数。

在匹配时，按照路由的顺序进行匹配，如果请求类型和请求的URL地址同时匹配成功，则express会将这次请求转交给对应的function函数进行处理。

### 2.2. 路由的使用



1. 模块化路由

express不建议将路由直接挂载在app上，推荐将路由抽离为单独的模块。

模块化路由的步骤

1. 创建路由模块对应的.js文件；
2. 调用exprss.Router()函数创建路由对象；
3. 向路由对象上挂载具体的路由；
4. 使用modual.exports向外共享路由对象；
5. 使用app.use()函数注册路由模块。
6. 注册路由模块

注册路由模块步骤

1. 导入路由模块；
2. 使用app.use()函数注册路由模块。
3. app.use()函数

app.use()函数是用来注册全局中间件。

app.use(express.static(‘文件’)) //托管静态资源文件

app.use(路由对象) //注册路由模块

1. 为路由模块添加前缀

app.use(‘/api’, 路由对象)­­

## 3. Express中间件

### 3.1. 中间件的概念

中间件特指业务流程中的中间处理环节。当一个请求到达Express的服务器之后，可以连续调用多个中间件，从而对这次请求进行预处理。

Express中间件本质是一个function函数。

app.get(‘/’, function(req, res, next) {

next();

})

中间件函数的形参列表中包含next参数，而路由处理函数中不包含next参数。

next()函数是实现中间件连续调用的关键，它表示把流转关系转交给下一个中间件或路由。

### 3.2. Express中间件初体验

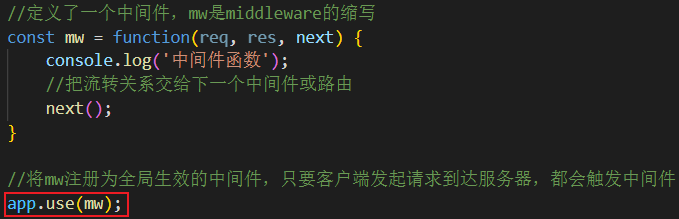
1. 定义中间件函数



1. 全局生效的中间件

客户端发起的任何请求，到达服务器之后，都会触发的中间件，叫做全局生效的中间件。

通过调用app.use(中间件函数)，即可定义一个全局生效的中间件。



1. 定义全局中间件的简化形式



1. 中间件的作用

多个中间件之间共享一份req和res，基于此特性，可以在上游的中间件中，统一为req和res对象添加自定义的属性或方法，供下游的中间件或路由使用。

1. 定义多个全局中间件

客户端请求达到服务器后，会按照中间件定义的先后顺序一次进行调用。

1. 局部生效的中间件

不使用app.use()定义的中间件，叫做局部生效的中间件。



1. 定义多个局部生效的中间件



1. 中间件的使用注意事项
2. 一定要在路由前注册中间件；
3. 客户端给发送过来的请求，可以连续调用多个中间件进行处理；
4. 执行完中间件的业务代码以后，不要忘记调用next()函数；
5. 为了防止代码逻辑混乱，调用next()函数后不要再写额外的代码，next()函数代表这个中间件的业务逻辑已经处理完；
6. 连续调用多个中间件时，多个中间件之间共享req和res对象。

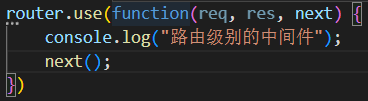
### 3.3. 中间件的分类

1. 应用级别的中间件

绑定到app实例上的中间件，叫做应用级别的中间件。

1. 路由级别的中间件

绑定到router实例上的中间件，叫做路由级别的中间件。



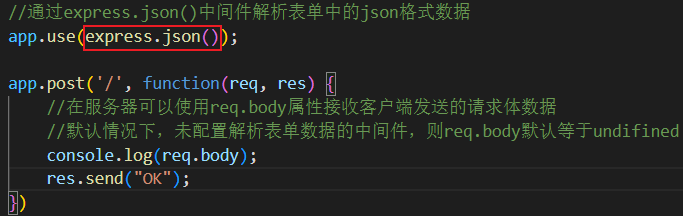
1. 错误级别的中间件

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。

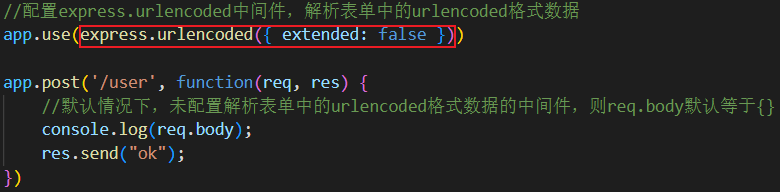


注意：错误级别的中间件必须注册在所有路由之后。

1. express内置的中间件
2. express.static快速托管静态资源的内置中间件，无兼容性；
3. express.json解析JSON格式的请求体数据，有兼容性，仅支持4.16.0+；



1. express.ulrencoded解析URL-encoded格式的请求体数据，仅支持4.16.0+。



1. 第三方的中间件
2. 执行npm install bode-parser命令安装中间件；
3. 使用require导入中间件；
4. 调用app.use()注册并使用中间件。



## 4. 使用Express写接口

### 4.1. 创建基本的服务器

### 4.2. 创建API路由模块



### 4.3. 编写GET接口

