

[前言 · 千锋大前端小册-GP19-Node.js \(gitee.io\)](#)

[Node.js专业中文社区 \(cnodejs.org\)](#)

Node.js

Node.js

- node: 基于v8引擎的javascript运行环境, 提供了一些特定的api;
- 可以实现的功能:
 - web 服务器
 - 命令行工具
 - 网络爬虫
 - 桌面应用程序开发
 - app
 - 游戏
 - 嵌入式 (物联网设备)
 - 读写和操作数据库、创建使用的命令行工具辅助前端开发
 - 作为硬件控制工具代替C/C++, Node.js允许经由websocket或串来凝结实现arduino访问
 - 在树莓派上使用node.js
- 常用框架
 - [基于Express框架](#), 可以快速构建web应用
 - 基于electron框架, 可构建跨平台的桌面应用
 - 基于restify框架, 可快速构建api接口项目
- 内置API
 - fs、path、http、js内置对象、querystring、ect.....
- Node.js特性
 - 可解析js代码(没有浏览器安全级别限制) 提供很多系统级别的API
 - 浏览器自带安全沙箱, 存在跨域等一些列问题,
 - 文件的读写
 - 进程的管理
 - 网络通信, 不存在跨域的问题
- node.js特点
 - 借助谷歌V8引擎, 更高效的处理并发、异步等性能问题
 - 单线程、事件驱动、异步输入和输出
 - 主线程为单线程, 阻塞部分由线程池处理。大多数的I/O请求都为异步请求, 从而避免上下文频繁切换的耗时操作, 具有高并发的特点
 - 能够支持跨平台功能
 - 异步队列
 - 使用异步队列, 通过不断地循环队列从而等待程序进程的处理
 - 例如SQL查询、涉及硬盘、网络等延时较长的操作, php代码会产生阻塞, 从而影响后续代码的执行; 而Node.js使用异步队列的回调函数, 只有当SQL执行完毕, 才会执行相应的回调函数, 避免阻塞程序执行的问题
 - 基于事件与回调

- Node的异步操作是基于事件和回调完成的，例如：磁盘I/O、网络通讯、数据库查询
 - 核心：拥有一个事件循环，判断是否有事件，有则执行；然后再判断是否有相关的回调，有则执行；最后事件执行完毕。
 - 优点：增加了Node.js应用程序运行的健壮性和可用性
 - 缺点：不符合正常的线性开发思路，容易出错
- 绝大多数API都是基于事件驱动、异步风格、非阻塞
 - 代码的执行并不依赖他们出现的位置、而是等待相应的事件被触发
 - 异步执行的回调无须阻塞的等待其他操作，充分利用系统资源
- 注意事项：
 - 可以使用：ES6这些js语法、node提供的API，但没有DOM和BOM这些浏览器中的东西;jquery也不能直接用
 - 实际开发的文件命名建议使用全英文
- 学习路线
 - js基础语法+node内置api模块（fs、http、path..）+第三方api模块（mysql、express。。。）
- 安装Node.js
 - 下载nvm（node版本控制工具）
 - 下载node（使用nvm进行下载）
 - 使用nvm命令选择并使用相应的node版本

项目结构

- app.js:项目的入口和程序启动文件
- public:项目的静态文件(css,js,img)
 - 就是一些图片或者音频文件,还有一些js插件
- routes:项目的路由
 - 里面是一些路由,也就是自己封装的一些接口,响应前端的一些请求
- views:视图目录文件(相当有[mvc](#)里的v)
 - 其实就是html文件 模板引擎
- /src 存放源码，一般会对这个文件进行编译打包
 - /src/common 存放写的一些轮子，通用的代码，导入逻辑层被调用
 - /src/dbs 存放项目所需所有数据库接入文件和配置文件
 - /src/models 存放orm层的文件或者dao层文件
 - /src/middleware 存放自己写的中间件，这个相当于功能组件轮子，导入控制层做中间件
 - /src/services 存放逻辑层文件，这个是控制层的功能函数，导入控制层被调用
 - /src/controllers 存放控制层文件
 - /src/views 存放前端页面模板
 - /src/access 存放别的服务接入代码，例如接入机器学习，把数据接入处理，导出数据到C层输出
 - /src/app.js 项目入口文件
 - /src/deploy.js 项目自动化部署文件
 - /test 测试文件夹，存放功能测试、健壮性测试、性能测试、压测等文件
 - .gitignore git忽略文件

.eslintrc.json 代码格式检查配置文件

.pm2.json 集群配置文件

package.json 项目依赖信息说明文件

webpack.config.js 项目打包编译配置文件

Buffer数据类型

js语言自身只有字符串数据类型，没有二进制数据类型。但在处理文件流相关内容是，必须使用到二进制数据。因此Node.js中定义Buffer类，用来创建专门存放二进制数据的缓冲区 (类似于一个整数数组)

方法	描述
Buffer.from()	创建一个buffer对象
Buffer.alloc(10)	创建一个可以存放10个字符的buffer对象，用不到位置它默认会由00占位
buf3.write('abc')	往buffer对象中写入信息(转2再转16存起来)
.toString()	将16进制buffer内容 转为可识别的内容

创建buffer对象

```
let buf1 = Buffer.from([97,98,99]); // 根据一个数组创建Buffer对象
console.log(buf1); // <Buffer 61 62 63> 以16进制形式存储在Buffer对象中
console.log(buf1.toString()); // abc 将buffer内容转为可识别的内容

let buf2 = Buffer.from('nodejs'); // 根据一个字符串创建Buffer对象

let buf3 = Buffer.alloc(10); // 创建可以存放10个字符的buffer对象
buf3.write('abc'); // 按照ASCLL表的价值，转16进制，存在Buffer中
```

windows命令行

- PowerShell cmd 两种window版本工具，powershell为新版的，相比于cmd的功能更完善
- PowerShell的使用，在目录文件夹下，鼠标右键 在windows终端下打开 即可
- cmd 的使用，win+R cmd进入

NVM

- [Releases · coreybutler/nvm-windows · GitHub](#)
- Node版本的管理工具 当需要多个版本的node时，使用Nvm可以同时管理过个版本；
- 安装Nvm之前，需要删除现有的node
- 加快node下载的速度
 - nvm安装后，打开安装的目录文件setting.txt
 - C:\Program Files\nodejs
 - 我的安装目录：C:\Users\wzt\AppData\Roaming\nvm

- 加入: node_mirror:<https://npm.taobao.org/mirrors/node/>
npm_mirror:<https://npm.taobao.org/mirrors/npm/>

常用命令

• node相关

```
node          # 在控制台进入node开发环境
node -v       # 查询当前node版本
node 文件名   # 执行node文件

ctrl + c      # 退出node的运行
```

• nvm相关 (管理员身份)

```
# 安装node时被一起安装
nvm -v          # 查看当前nvm版本
nvm list        # 查看当前机器上安装了哪些node版本
nvm install 版本号      # 安装对应版本的node (例: nvm install 16.0.1)
nvm install latest      # 安装最新版本的node
nvm uninstall 版本号    # 卸载指定版本的node
nvm use 版本号          # 选择使用指定版本的node
nvm on             # 启用node.js版本管理, 即nvm
nvm off            # 禁用nvm, 但不卸载任何版本
```

• Yarn——新的包管理工具

```
# 由Facebook、Google等联合发布的全新JS包管理器, 用于替代 NPM的快速、可靠、安全的包管理工具
# 离线模式: 已经下载过的包, 使用时无需重复下载
# 网络恢复: 下载失败后, 会自动重新请求, 避免整个安装过程失败
# 多个注册表: 能从NPM或Bower安装任何包, 也能保证各平台安装的一致性
# 扁平化模式: 将不匹配的安装包解析为同一个版本, 避免重复创建
# 确定性: 无论安装顺序如何, 在不同机器上会以完全相同的方式进行安装
# 在初次安装依赖包时生成 yarn.lock文件, 用于确保安装包的确定性
npm i -g yarn # 使用npm 全局安装yarn
```

```
yarn --version    # 查看yarn版本
yarn init         # 生成项目package.json文件
yarn add 包名     # 安装依赖包
yarn add 包名 -dev # 安装依赖包并记录在 dependencies节点, 开发依赖包
yarn upgrade 包名@版本    # 升级指定的包的指定版本
yarn remove 包名@版本    # 移除指定的指定版本的依赖包
yarn global add 包名     # 全局安装指定包
```

```
yarn install      # 一次性安装package.json 中的依赖包
yarn install -flat    # 安装一个包的单一版本时添加 -flat
yarn install -force   # 强制重新下载使用 -force
yarn install --production # 只安装生产环境依赖的包 --production
```

```
yarn global bin    # 查看yarn安装的路径
```

```
yarn run dev       # 运行项目
yarn run build     # 编译项目
```

```
yarn config set registry https://registry.npm.taobao.org # 切换yarn下载源
```

- cnpm

- PM2

```
# Node.js 应用进程管理器
npm i pm2 -g # 安装PM2
pm2 start app.js # 以守护进程的方式运行app.js文件
pm2 -h # 查看所有的 pm2命令
pm2 list # 查看当前运行的所有应用 及 对应id
pm2 stop id # 停止id对应的应用
pm2 restart id # 重启node应用
pm2 restart all # 重启所有node应用
```

- 其他命令

```
ping 网址(www.baidu.com) # 查看对应网址(百度)服务器的ip地址
exit # 退出命令行
d: # 进入D盘
dir # 查看此文件下一级的目录
Tab 键 # 自动补全内容
cd 文件名 # 进入文件
esc # 快速清空当前输入的命令
↑ # 快速定位到上一行命令
```

(查看文件中：显示

表示它是个文件目录；时间表示此文件最后一次的修改时间)

全局对象

- 在浏览器中，window是全局对象
- Node.js中全局对象 global,所有全局变量都是 global
- Node.js中声明的变量，不会直接挂载到global上，但global上挂载的内容可以在任何地方使用
- 交互模式下
 - this 指向 global
 - 交互模式下只有module.exports，没有exports
- js文件下
 - this指向exports，即js模块导出的对象
 - exports 是 module.exports的引用，只有文件中才存在exports
 - 在js文件下执行Node的 全局this≠global,this指向的是当前这个js模块

module 对象

- 每个 .js 文件(模块) 都有一个 module对象；存储当前模块相关的信息
- module.exports 对象
 - 自定义模块中，可使用 module.exports对象，将模块内的成员共享出去，供外界使用。
 - 使用 require() 方法导入自定义模块时，导入的结果，以 module.exports 指向的对象为准。
 - 为简化向外共享成员的代码，Node提供了 exports对象，与 module.exports 等价。
 - 默认情况下，exports和module.exports 指定同一个对象
- 注意 :得到的永远是 module.exports对象



- 图一： { gender:男;age:22 }
- 图二： { username:zs }
- 图三： {gender:男 }
- 图四： { username:zs; gender男; age:22 }
- 为防止冲突，不要在同一个模块中同时使用 exports 和 module.exports



```
03.module对象.js x
1 console.log(module)

C:\Users\liulongbin\Desktop\node\day2\code>node 03.module对象.js
Module {
  id: '.',
  path: 'C:\\Users\\liulongbin\\Desktop\\node\\day2\\code',
  exports: {},
  parent: null,
  filename: 'C:\\Users\\liulongbin\\Desktop\\node\\day2\\code\\03.module对象.js',
  loaded: false,
  children: [],
  paths: [
    'C:\\Users\\liulongbin\\Desktop\\node\\day2\\code\\node_modules',
    'C:\\Users\\liulongbin\\Desktop\\node\\day2\\node_modules',
    'C:\\Users\\liulongbin\\Desktop\\node\\node_modules',
    'C:\\Users\\liulongbin\\Desktop\\node_modules',
    'C:\\Users\\liulongbin\\node_modules',
    'C:\\Users\\node_modules',
    'C:\\node_modules'
  ]
}
```

process对象

process 对象是一个全局变量，提供了有关当前 Node.js 进程的信息并对其进行控制。作为全局变量，它始终可供 Node.js 应用程序使用，无需使用 require()。也可以使用 require() 显式地访问。

- process.env 属性会返回包含用户环境的对象

```
// process.argv 获取到执行文件时，命令行执行的所有参数，作为元素放在这个的数组中(字符串形式)
// 使用场景之一：可以用来做命令行工具(拿到命令行输入的内容，再执行一些操作)
console.log(process.argv)    // 打印一个数组
// 例：拼接字符串
console.log(process.argv[1]+process.argv[2])

// process.arch 得到执行环境的系统位数 x64 x32 ...
console.log(process.arch)
```

console对象

- 支持：
 - 格式化输出
 - 清空控制台 console.clear()
 - 对输出的内容进行计数 console.count()
 - 打印堆栈踪迹 console.trace()

- ```

console.log("我爱你"); // 在控制台打印我爱你

// 支持格式化输出 %s为字符串 %d为相关的数字
console.log('我的%s已经%d','猫',2); // 输出: 我的猫已经2岁了

console.clear(); // 清空控制台
console.count(); // 对输出的内容进行计数 结果形式 1: 内容 下次还是指针的内容时 2: 内容

```

- console.log()

```

// 用法与传统中一致
// 可以利用占位符 定义输出的格式, %d 表示数字 %s表示字符串
// 使用%d占位后, 对应位置如果不是数字, 会输出NaN
console.log("%s%s", 'node.js', 'is', 'powerful'); // node.js is owerful
console.log('%d', 'node,js'); //NaN

```

- console.info(); console.warn(); console.error() 输出的结果与console.log()输出的内容一致
- console.dir()

```

// console.dir() 用于将一个对象的信息打印到控制台
const obj = {
 name: 'node.js',
 get: function(){
 console.log('get');
 },
 post: function(){
 console.log('post');
 }
}
console.dir(obj); // { name: 'node.js', get: [Function: get], post: [Function: post] }

```

- console.time() 和 console.timeEnd()

```

// 用于统计一段代码的运行时间
// 使用方法: 在代码块的开始和结尾分别放置console.time() 和 console.timeEnd()
// 并传入同一个参数, 再一段代码中可以存在多个计时代码对

console.time('one');
console.time('two');
for(var i=0;i<10;i++){
 var a = i;
}
console.timeEnd('two');
console.time('three');
for(var i=0;i<10;i++){
 var a = i;
}
console.timeEnd('three');

```



```
console.timeEnd('one');
// 控制台输出:
// two: 0.102ms
// three: 0.01ms
// one: 8.835ms
```

- console.trace()

```
// 输出当前位置的栈信息，需要传入任意参数作为标识
? ? ?
```

- console.table()

```
// 用于将数组格式的信息以表格的形式输出
// 参数：任意结构形式的数组信息，譬如对象、数组等
const arr={
 A: { name:'wu',id:1},
 B: { name:'wu',id:2},
 C: { name:'wu',id:3},
}

console.table(arr); // 输出内容为表格 行; index name id
 // A wu 1
```

## 内置模块

### fs 文件系统模块

| 方法                | 描述                               |
|-------------------|----------------------------------|
| fs.readFile()     | 读取指定文件中的内容，异步读取                  |
| fs.readFileSync() | 同步读取指定文件内容，读取完毕后再执行后面的代码，不需要回调函数 |
| fs.writeFile()    | 向指定的文件中写入内容，异步写入                 |
| fs.renameSync()   | 修改指定文件的文件名                       |
| fs.readdirSync()  | 获取指定路径下的文件列表，数组形式(包含文件和文件夹)      |
|                   |                                  |

- fs.readFile(path[, options], callback) 异步的，读取指定文件中的内容
  - 参数 path：字符串格式，表示文件的路径
  - 参数 options：可选参数，表示用什么编码格式读取文件,字符串格式
  - 参数 callback：回调函数，文件读取后，通过回调函数拿到读取的结果

```
// 导入fs模块
const fs =require('fs')
fs.readFile('./one.txt','utf-8',function(err,dataStr){
 console.log(err); //当接收到的err 为 null 时，表示文件读取成功；
 console.log(dataStr); //当文件读取失败时，值为null；成功时返回文件的内容
})
```

- fs.writeFile( file,data[,options],callback)      用于向指定的文件中写入内容
  - 参数 file：指定文件路径的字符串
  - 参数 data：表示写入的内容
  - 参数 options：表示写入的编码格式，默认utf-8；
  - 参数 callback：文件写入完成后的回调函数
- 当文件不存在时，fs.writeFile() 方法可以创建文件;但不能用来创建文件夹(路径)
- 重复调用 fs.writeFile() 写入同一个文件，新写入的内容会覆盖之前的旧内容

```
const fs =require('fs');
fs.writeFile('./two.txt','hello node,js!',function(err){
 console.log(err); //当返回值 err 为 null 时，表示写入成功；
 // 判断文件是否写入成功
 if(err){
 return console.log('文件写失败'+ err.message)
 }
 console.log('文件写入成功! ');
})
```

- fs.renameSync(旧文件名,新文件名)      修改指定文件的文件名
- fs.readdirSync(\_\_dirname)      获取指定文件夹路径下的文件名组成的数组
  - 返回值：[旧文件名1,旧文件名2,旧文件名3.....]

**小练习：批量修改文件的命名**

## 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。

| 内容                                    | 描述                             |
|---------------------------------------|--------------------------------|
| <code>__dirname</code>                | 当前文件所处的绝对路径，不包括文件名             |
| <code>__filename</code>               | 当前文件的绝对路径，包含文件名                |
| <code>path.extname()</code>           | 获取路径中 文件的扩展名                   |
| <code>path.basename()</code>          | 获取路径中的 文件名                     |
| <code>path.dirname(__filename)</code> | 去除路径中的文件名                      |
| <code>path.parse(__filename)</code>   | 将路径解析为一个对象：所在盘符   所在路径   文件名后缀 |
|                                       |                                |
| <code>path.join()</code>              | 将多个路径片段 拼接 成一个完整的路径字符串         |
|                                       |                                |

```
// 导入path模块
const path = require('path');

// 得到当前文件所处的绝对路径，不包括文件名 __dirname
console.log(__dirname);
// 得到当前文件的绝对路径，包括当前文件的文件名 __filename
console.log(__filename);

// 获取路径中 文件的扩展名 path.extname(path)
// 参数：文件路径字符串 返回值：文件的扩展名
// 例：
const fpath = '/a/b/c/d/index.html';
const fext = path.extname(fpath);
console.log(fext); //输出 .html

// 获取路径中的 文件名(包含后缀) path.basename(path[,ext])
// 参数1：文件路径字符串 返回值：路径中的最后一部分
// 参数2：可选，文件的扩展名 带上参数2时，返回值中则不包含文件的扩展名
// 例：
const fpath = '/a/b/c/d/index.html';
var fullName = path.basename(fpath);
console.log('fullName'); //输出：index.html
var Name = path.basename(fpath, '.html');
console.log('Name'); //输出：index

// 去除路径中的文件名 path.dirname(__filename);
// 将路径解析为一个对象 （所在盘符、所在路径、文件名后缀）
path.parse(__filename);
```

- `path.join(路径字符串1、字符串2、.....)`
- 该方法能够在不同操作系统中，转化成不同的路径格式，具有跨平台的兼容性

凡是涉及路径拼接的操作，都建议使用 `path.join()` 方法进行处理。不要直接使用 `+` 进行字符串的拼接，因为加号不支持 `./` 的识别，且路径的符号会被当做转译符号

```
// 将任意多个路径片段拼接成一个完整的路径字符串 path.join()
// 例：
const path = require('path') // 导入 path 模块
const pathStr = path.join('/a', '/b', '/c');
console.log(pathStr); // 输出路径： \a\b\c

const pathStr2 = path.join(__dirname + './one.txt');
console.log(pathStr2);
```

- `path.resolve()` 方法

### 路径拼接问题

- 原理：代码在运行的时候，会以执行 node 命令时所处的目录，动态拼接出被操作文件的完整路径。
- 解决方案：操作文件时直接提供完整的路径，避免使用 `./` 或 `../` 开头的相对路径，从而防止路径动态拼接的问题。
- 使用 `__dirname` 表示当前文件所处的绝对路径
- 例：`fs.readFile(__dirname + '/one.txt')` // 表示写入当前文件同级目录下的 `one.txt`

## http 模块

- 用于创建 web 服务器的模块。
- 什么是服务器：
  - 服务器和普通电脑的区别在于，服务器上安装了 web 服务器软件，例如：IIS、Apache 等。通过安装这些服务器软件，就能把一台普通的电脑变成一台 web 服务器。
  - 在 Node.js 中，我们不需要使用 IIS、Apache 等这些第三方 web 服务器软件。因为我们可以基于 Node.js 提供的 http 模块，通过几行简单的代码，就能轻松的手写一个服务器软件，从而对外提供 web 服务。
  - 通过 http 模块的 `http.createServer()` 方法，能方便的把电脑变成 Web 服务器，从而对外提供 Web 资源服务。
  - 在一台电脑中，可以运行成百上千个 web 服务。每个 web 服务都对应一个唯一的端口号。客户端发送过来的网络请求，通过端口号，可以被准确地交给对应的 web 服务进行处理。  
端口号范围：0——65536
  - 在开发测试期间，`127.0.0.1` 对应的域名是 `localhost`，它们都代表我们自己的这台电脑

## 服务器端

```
// 导入 http 模块
const http = require('http');
const port = 8001;
// 方法1: 先创建 web服务器实例, 再监听回调事件
const server = http.createServer();
// 为服务器实例绑定 request事件, 监听客户端的请求
// 使用服务器的 .on()方法, 为服务器绑定 request事件; 只要有客户端向服务器发送请求, 就触发 request事件
server.on('request', (req, res) => {
 console.log('有人访问我们的服务器 Someone visit our web server');
});

//方法2: 在创建实例时直接绑定监听事件
const server2= http.createSrever(function(res, req){
 res.writeHead(200, {'content-type': 'text/plain'});
 res.write('hello word'); // 向浏览器中写入内容
 res.end('hello word'); // 响应数据, 并断开连接
});

// 调用服务器实例的 .listen(端口号, 回调函数) 方法, 即可启动当前的 web服务器实例
server.listen(port, (error) => {
 //服务器启动成功即调用回调函数
 console.log(`http server running at http://127.0.0.1:${port}`);
});
```

- request: 当收到客户端请求时, 触发该事件, 提供res和req两个参数, 表示请求和相应信息
  - 回调函数中, 第一个参数 req 包含了客户端相关的数据或属性;
  - 回调函数中, 第二个参数 res 包含了服务器相关的数据和属性;

| req 事件 | 描述                                  |
|--------|-------------------------------------|
| data   | 使用回调函数, 监听post请求发来的数据, 值默认为buffer类型 |
| end    | 当post 请求的数据传输完毕时触发该事件, 此后不会再有数据     |
| close  | 用户当前请求结束时, 触发该事件                    |

| req参数           | 描述                            |
|-----------------|-------------------------------|
| req.url         | 得到 客户端请求的 url地址               |
| req.method      | 客户端的 method请求类型(post/get/...) |
| req.headers     | HTTP请求头信息                     |
| req.httpVersion | HTTP协议版本                      |

- res.writeHead(): 向请求的客户端发送响应头
- res.write(): 向请求发送内容

- `res.end()` 方法：向客户端发送指定的内容，并结束本次请求的处理

```
server.on('request',(req,res) => {
 // 监听post请求的参数，返回值为buffer类型，可以用.toString()转为字符串
 req.on('data',function(data){
 console.log(data);
 })
 // 使用ES6 模板语法时，必须使用反单引号
 const str = `Your request url is ${req.url},and request method is
${req.method}`;
 console.log(str);
 res.writeHead(200,{ 'content-type': 'text/plain' });
 res.end('hello word');
})
```

## 注意事项

- 需要对客户端发来的url请求地址进行操作，对应到服务器中实际的文件位置
  - 客户端请求的根目录为 /
  - 客户端发来的请求例： /index.html
  - 服务器接收到路径后需要根据文件在磁盘中的实际位置进行 path路径操作  
例： C:\Users\wzt\Desktop\knowledge\node.js\练习\node服务器/index.html (`--dirname +req.url`);
- 在fs模块文件的读取和写入时 进行及时的判断，观察请求和响应是否成功；
- 代码更改后，需要重启服务器并刷新页面才能生效
- 解决 `res.end()` 发送给客户端数据的中文乱码问题，此时需要手动设置内容的编码格式
- 设置响应头，允许跨域

```
server.on('request',(req,res) => {
 const str = '我是响应给客户端的内容';
 res.setHeader('Access-Control-Allow-Origin','*'); //允许跨域
 res.setHeader('Access-Control-Allow-Headers','*'); //允许向服务器发送任意的请求头信息
 res.setHeader('Content-Type','text/html;charset=utf-8'); // 为防止中文乱码问题，设置响应头
 res.end(str);
})
```

## 客户端

- `http.request(option,[callback])`：option为json对象，主要字段有host、port(默认为80)、methods(默认为get)、path(请求相对与根的路径，默认 /)、headers等
- `response`：当接受到响应时触发该事件
- `request.write()`：发送请求数据
- `res.end()`：请求发送完毕，应当始终指定这个方法

- ```

const http=require('http');
let reqData='';
http.request({
  'host':'127.0.0.1',
  'port':'8003',
  'method':'post',
},function(res){
  res.on('data',function(chunk){
    // 保存请求来的数据
    reqData+=chunk;
  });
  res.on('end',function(){
    // 打印请求来的数据
    console.log(reqData);
  });
}).end();

```

http2模块

- node对http2协议的实现，该协议是一个二进制复用协议
- 实现并行请求可以在同一个链接中处理，移除http1.1中关于顺序和阻塞的约束、压缩了headers
- 允许服务器在客户端缓存中填充数据，并通过服务器推送机制来提前请求.....

说明：

- http2模块需要依赖于ssh安全证书来实现
- 配合fs文件模块来引入证书文件
-

创建http2服务器

- 创建的服务器server拥有两个事件，一个方法
 - 参数：stream 数据流，实现向客户端发送信息
 - 参数：headers 文件响应头信息
 - stream事件：当请求体数据来到时触发该事件，
 - 通过 respond()方法向客户端设置响应头信息
 - 通过 end()方法向客户端发送文本内容，并结束请求
 - error事件：
 - 错误信息
 - listen方法：监听客户端请求

```

const http2 = require('http2');
const fs = require('fs');
const serve = http2.createSecureServer({
  key:fs.readFileSync('./ssl/localhost-cert.pem')
  cert:fs.readFileSync('./ssl/localhost-cert.pem')
});
server.on('error',(err) => console.log(err);)
server.on('stream',(stream,headers) => {
  //stream is a Duplex (两部分)

```

```
stream.respond({
  'content-type': 'text/html',
  ':status': 200
});
stream.end('<h1>Hello http2</h1>')
})
server.listen(8003);
```

客户端向服务器发http2请求

util模块

- util.promisify() 方法
 - 将异步回调风格的方法，转变成 promise 风格的方法

```
const fs = require('fs');
const util = require('util');
// 将 fs.readFile方法 转成 promise风格的方法 并返回
let mineReadFile = util.promisify(fs.readFile);
mineReadFile('./resource/content.txt').then(value=>{
  // 事件成功的回调
  console.log(value.toString());
})
```

url模块

- 用于分析、解析url

dns模块

- 域名处理和域名解析

网络通信模块

net模块

- 用于创建TCP服务器、TCP客户端

http模块

- 用于创建HTTP服务

dgram模块

- 用于创建UDP服务器、UDP客户端

Node模块化

- node遵循 CommonJS模块化规范，规定了模块的特性、各模块之间如何相互依赖。
 - 准确：Node.js使用的是轻微修改版本的CommonJS，它不需要考虑网络延迟问题
- CommonJS规定：
 - 每个模块内部，module变量 代表当前模块。
 - module 变量是一个对象，它的 exports 属性是对外的接口。(即 module.exports)
 - 加载某个模块，其实是加载该模块的 module.exports属性。require() 方法用于加载模块
- Node.js模块的分类
 - 内置模块（由Node.js官方提供的，例：http、path、fs等）
 - 自定义模块（由用户自行创建的 .js文件，都属于自定义模块）
 - 第三方模块（由第三方开发出来的模块、使用前需要先下载）
- 模块作用域 (Node.js)
 - 一个js文件就是一个模块，模块的作用域是私有的，内部定义的函数、变量名只能在当前文件(模块)使用
- 导出数据

```
// 方式1: 分别导出 可以是对象、数组、方法、字符串
exports.nmb=123;
exports.sum=sum;
// 方式2: 一次导出
module.exports
```

- 导入模块 require() 方法 (可省略 .js)
 - 可以是 绝对路径 或 相对路径

```
// 导入内置 fs模块
const fs = require('fs');
// 导入自定义模块
const custom = require('./custom.js');
const custom = require('./custom');
// 导入第三方模块
const moment = require('moment');
```

模块的加载机制

- node.js中可以使用require导入模块，使用exports方法导出模块
- 模块在第一次加载后会被缓存。因此多次调用 require() 并不会导致模块代码被执行多次，
- 引入模块的查询机制：逐级向上级文件查询依赖的包
- 内置模块
 - 内置模块加载的优先级最高
 - 例如：require('fs') 始终返回的是内置fs模块，即使node_modules 目录下存在名字相同的包也叫fs。
- 自定义模块
 - 使用require()，加载自定义模块时，必须指定 ./ 或 ../ 开头的路径标识符。否则 node会将其视为 内置模块 或 第三方模块 进行加载。

- 在导入自定义模块时，省略文件扩展名，则node会按顺序进行如下尝试
 - 按照确切的文件名加载
 - 补全 `.js` 扩展名进行加载
 - 补全 `.json` 扩展名进行加载
 - 补全 `.node` 扩展名进行加载
 - 加载失败、终端报错
- 第三方模块加载机制
 - 当传递给 `require()` 的模块标识符不是内置模块，也没有 `./` 或 `../` 开头，
 - 则node.js会从当前模块的父目录开始，尝试从 `/node_modules` 文件夹中加载第三方模块
 - 如果没有找到第三方模块，则移动到再上一层父目录中进行加载，直到文件系统的根目录

使用ES6模块化

- Node.js 默认不支持ES6模块发规范
- 可以借助插件，将代码进行转化，把ES6规范转化为Commonjs规范
- 实现插件：Bable-cli 和 Browserify
- 解决方法：

```
# 1.在项目下生成package.json文件
yarn init -y 或 npm init -y
# 2.全局 安装插件
yarn global add bable-cli browserify 或 npm i bable-cli browserify -g
# 3.在根目录下新建 .babelrc 文件，写入以下内容：
{
  "presets":[
    "es2015"
  ]
}

# 4.在src目录下写完代码之后，执行命令进行转化
# lib是src文件夹的同级文件夹，将src文件夹下的内容编译后放在lib文件夹下
babel src -d lib
# 5.此时，就可以使用es6的模块化语法在src文件夹下书写代码，编译后，运行lib文件夹下的对应代码即可
```

npm 与 包

- 概念
 - Node.js中的 第三方模块 又叫做 包，是同一个概念。
 - 在node下载安装时，npm命令工具一同被安装在系统中，可以直接使用
- 包的来源
 - 不同于node内置模块和自定义模块，包由第三方个人或团队开发出来，免费供所有人使用。
 - 注：node.js的包都是免费且开源的，不需要付费即可免费下载使用。

- 包的作用
 - 基于内置模块的封装，提供了更高效、便捷的API，提高开发效率。

nrm切换源

- 著名npm, Inc旗下
 - 网站: <https://www.npmjs.com/> (全球最大的包公享平台, 找寻需要的包)
 - 下载地址: <https://registry.npmjs.org/> (从这个服务器上 下载自己所需要的包)
- 包的下载优化
 - npm下载的默认地址是国外的 <https://registry.npmjs.org/>服务器, 需要经过[海底光缆](#)跨度较大导致时间较慢。
 - 解决方案: 淘宝npm镜像服务器
 - 镜像(Mirroring): 是一种文件存储形式, 一个磁盘上的数据在另一个磁盘上存在完全相同的副本;
- 切换npm的下载包的镜像源 //必须在英文目录下使用, 否则报错!
 - 淘宝镜像源: `https://registry.npm.taobao.org/`
 - 原始npm源: `https://registry.npmjs.org`
 - 或者用 `nrm ls` 查看可用的镜像源有哪些

```
npm config get registry # 查看当前的下包镜像源
npm congig set registry=http://registry.npm.taobao.org/ #切换下包镜像源
为淘宝镜像源
npm config get registry # 检查镜像源是否下载成功
```

npm包管理工具

由npm, Inc公司提供, 在node.js安装时已经一起安装在用户电脑(全名: Node Package Manager)

基于Node.js的包管理器, 初衷: JavaScript开发人员更容易分享和复用代码

命令操作

```
npm -v      或 npm version      # 查看npm包管理工具的版本号
npm i       # 项目初始时,根据package.json中的配置 进行所有
依赖的安装
npm i --production      # 项目初始时,根据package.json中的配置 只安装开
发和部署都依赖的开发包

npm install 包名 或 npm i 包名  # 在项目中安装指定的包,核心依赖包 dependencies
npm i 包名@版本号          # 通过@符, 下载指定版本, 默认最新版 例: npm i
moment@2.22.2
npm view 包名 versions      # 查看指定包的所有版本
npm i npm --global          # 更新npm版本到最新版

npm link 包名                # 将全局安装的包 链接到当前目录下; 才能使用require()引入
npm i 包名 -g               # 安装指定的包并设置为 全局包 -g; 会将npm安装在c盘指定
文件中
```

```

npm root -g # 查看全局安装的路径
npm i 包名 -D 或 npm install 包名 --save -dev # 安装指定的包并记录在
dependencies节点, 开发依赖包 save 表示只生产环境使用,可简写为S,是默认值 dev开发和部
署都依赖的包, 最终需要上传到项目服务器

npm init # 在项目中使用npm时, 进行npm初始化 (应当在项目根目
录下)
npm init -y # 同上 快速创建 package.json包管理配置文件(目录
文件必须是英文命名)

npm install 或 npm i # 一次安装所有的依赖包
npm uninstall 具体的包名 # 从当前目录卸载指定的包, 并从package.json中移除

npm login # 依次输入用户名、密码; 登录npm账号, 必须在官方服务
器上进行

npm list # 显示当前目录下安装的模块
npm list -g # 查看全局安装包
npm outdated # 查看依赖的包的版本新旧信息, 根据package.json判断是否需要更新
npm update # 根据配置的依赖包信息, 执行该命令进行更新

npm cache clean --force # 清除npm缓存, 解决一次安装出错后, 下次再报错

# npm 安装git或gitee上发布的包
npm install git+https://git@github.com:lurongtao/gp-project.git # 这样适合安
装内部的git服务器上的项目
npm install git+ssh://git@github.com:lurongtao/gp-project.git # 或者以ssh
的方式

npm help #查看帮助

```

包的语义化版本规范

包的版本以: '点分十进制' 进行定义,共三位数字, 例: 2.23.0

版本号提升规则: 只要前一位的数字增长了, 后面的数字都归零。

- 主版本号 major: 大的更新
- 次版本号 minor: 小的更新, 功能版本更新
- Bug修复版本 patch: 奇数表示不稳定, 一般都以偶数结尾

版本号的标识:

- ^2.2.0 表示第一位锁定
- ~2.2.0 表示前两位锁定
- 2.2.0 表示版本号都锁定
- * 表示最新版本
- 注: 未锁定的版本位, 默认会选择最新的版本

注意事项

初次包安装完成后，会多两个新的文件

不建议手动修改自动生册文件中代码，npm包管理工具会自动维护他们

- node_modules 文件夹，用来存放所有已经安装到项目中的包。
// require() 导入第三方包时，就是从这个目录中查找并加载包。
- package-lock.json 配置文件，// 用来记录 node_modules 目录下的每个包的下载信息。(包名、版本号、下载地址等)

package.json包管理配置文件(自动生成和维护)

npm规定，在**项目根目录**中，**必须**提供一个叫 **package.json**的包管理配置文件。

用来记录与项目相关的一些配置信息。

- 项目的名称name:如果要发布到npm平台,需要拥有全网唯一的名称
- repository: 保存放的仓库地址
- keywords: 包的关键字，有利于其他人搜索
- 版本号version、描述description
- 项目的作者author、项目所安装的依赖 dependencies、项目的版权分发方式license...
- 项目都用到了那些包
- **scripts**: npm 脚本,定义一些快捷指令之类的内容,可以方便的在控制台进行调用
- 哪些包只在开发期间使用 记录在: devDependencies 节点 生产包
- 哪些包在开发和部署时都需要使用 记录在: dependencies节点 开发包
- main: 记录项目的主文件，项目的启动文件
- license: 遵循的协议，常配置为MIT，表示开源，可以任意使用

局部与全局安装

- 全局安装后，可以在任意位置调用命令
- 局部安装调用
 - 方法一：在下载包文件 node_modules/.bin 路径对应的包下运行相关命令
 - 方法二：在项目文件中配置好相应的 scripts 脚本命令才行；

```
package.json X
node-basics > 02-tools > 02-npm > package.json > {} scripts > dev
1  {
2    "name": "02-npm",
3    "version": "1.0.0",
4    "description": "",
5    "main": "01-require-jquery-global.js",
6    "scripts": {
7      "dev": "gulp -v"
8    },
9    "keywords": [],
10   "author": "",
11   "license": "ISC",
12   "dependencies": {
13     "gulp": "^4.0.2"
14   }
15 }
```

npm脚本

- 写在项目package.json中的 scripts部分 用于自定义命令行中的脚本命令
- 控制台使用时必须 npm run 自定义的命令
-
- 关于脚本的串并行
 - 使用 & 符号分隔时，多个命令同时执行的，依据执行完成先后顺序输出
 - 使用 && 符号分隔时，多个命令按书写顺序执行，依次输出
- 脚本代码的简写
 - 解释：就是在执行的时候，可以不用写run，直接写 npm 自定义命令 即可
 - 可以使用的命令
 - start
 - test

```
> Debug
"scripts": {
  "dev": "gulp -v",
  "greeting": "echo hello",
  "runjs": "node ./03-scripts/script1.js && node ./03-scripts/script2.js",
  "start": "node ./03-scripts/script.js",
  "test": "node ./03-scripts/test.js"
},
```

解决的问题

1、解决多人协作问题

由于协作项目可能导致包的重复，引发项目体积过大，不方便团队成员之间共享项目源代码

解决方案：共享时剔除node_modules；即自己安装的包

2、记录项目中安装了哪些包

// 在上传gitup 时不用上传这些赘余的包，即node_modules文件，把它添加到.gitignore忽略文件中

3、快速创建package.json

`npm init -y` // 在执行命令的目录中，快速新建package.json文件
只能在英文目录下成功运行，不能有空格

// 运行 `npm install` 命令 安装包时，npm 包管理工具会自动把包的名称和版本号，记录到package.json 中。

4、dependencies 节点

package.json文件中，dependencies节点，专门用来记录使用 `npm install`命令安装过的包
表示在项目开发和实际上线后都需要使用的包，建议存放在dependencies节点中。

5、一次性安装所有的包

当拿到剔除了 node_modules 的项目后，需要先把所有的包下载到项目中，才能将项目运行起来。否则报错：

```
# 由于项目运行依赖于 moment 这个包，如果没有提前安装好这个包，就会报如下的错误：
Error: Cannot find module 'moment'
```

安装命令： `npm install` 或者 `npm i`

6、卸载指定的包

`npm uninstall 具体的包名` // 命令执行后，会把卸掉的包自动从package.json的dependencies中移除

7、devDependencies 节点

表示只在开发过程中使用，在实际线上项目中不使用的包，建议存放在devDependencies节点中。

```
# 安装指定的包，并记录到 devDependencies 节点中
npm i 包名 -D
# 上述命令的完整写法：
npm install 包名 --save-dev
```

包的分类

①、项目包

安装到node_modules项目目录中的包，都属于项目包

- 开发依赖包(被记录在 devDependencies节点中，只在开发期间使用)
- 核心依赖包(被记录在 dependencies节点中，在开发期间和项目上线后都使用)

②、全局包

安装在其他文件中，而不在项目文件。

`npm i 包名 -g` // 安装时使用 -g 参数，则会把安装包设置为全部包。

- 只有工具性质的包，才有安装在全局的必要性。
- 判断是否有全局安装的必要根据个人需要、官方文档、项目的实际需求综合考虑。

规范化的包结构

- 包必须以单独的目录而存在
- 包的顶级目录下必须包含package.json这个包管理配置文件
- package.json 中必须包含 name、version、main 这三个属性，分别代表 包的名字、版本号、包的入口。
- 以上三点是最基本的规范要求，还更多关于包的规范，自行寻找。

开发属于自己的包

1.初始化包的基本结构

2、将不同的功能进行模块化拆分

3、编写包的说明文档

4、发布包

• 注册 npm 账号

①访问 <https://www.npmjs.com/> 网站，点击 sign up 按钮，注册账号

④登录邮箱，点击验证链接，进行账号的验证

• 登录 npm 账号

可以在终端中执行 `npm login` 命令(或者 `npm adduser`)，依次输入用户名、密码、邮箱后，即可登录成功。

注意：注意：在运行 `npm login` 命令之前，必须先把下载包的服务器地址切换为 npm 的官方服务器。否则会导致发布包失败！

• 发布包

- 修改好包的package.json信息后上传
- 将终端切换到包的根目录、运行 `npm publish` 命令，即可将包发布到npm上
- 注意：包名不能出现雷同！


```
C:\Users\liulongbin\Desktop\itheima-utils1>npm publish
npm notice
npm notice package: itheima-utils1@1.0.1
npm notice === Tarball Contents ===
npm notice 677B src/dateFormat.js
npm notice 741B src/htmlEscape.js
npm notice 349B index.js
npm notice 229B package.json
npm notice 816B README.md
npm notice === Tarball Details ===
npm notice name:          itheima-utils1
npm notice version:       1.0.1
npm notice package size:  1.4 kB
npm notice unpacked size: 2.8 kB
npm notice shasum:        4683fd9e9f14e8a8656a7ebfa46c59e576525dcf
npm notice integrity:     sha512-b0mS3ZPe2vxAu[...]g2MNxaJLYePFA==
npm notice total files:   5
npm notice
+ itheima-utils1@1.0.1
C:\Users\liulongbin\Desktop\itheima-utils1>
```

5、删除已发布的包

- 运行 `npm unpublish 包名 --force` 命令
- 注意
 - `npm unpublish` 命令只能删除72小时以内发布的包
 - `npm unpublish` 删除的包，在24小时内不允许重复发布
 - 发布时应慎重，尽可能不要发布没有意义的包

其他第三方包

5ting_toc

可以把 md 文档转为 html 页面的 工具。

生成的 html页面(包含css和js文件) 会直接放在本地的 preview文件夹中

```
# 将 5ting_toc 安装为全局包
npm i 5ting_toc -g
# 调用5ting_toc ,轻松实现md 转html的功能
# 5ting_toc -f 要转换的md文件路径 -o
```

nrm源管理

- 为了更方便的切换下包的镜像源，可以安装nrm工具，能够快速的查看和切换下包的镜像源
- 必须在管理员模式下，解决无法运行脚本问题

```
+ CategoryInfo          : SecurityError: (:) [], PSSecurityException
+ FullyQualifiedErrorId : UnauthorizedAccess

PS E:\GitHub\reactstudy\my-app> get-ExecutionPolicy
Restricted
PS E:\GitHub\reactstudy\my-app> set-ExecutionPolicy RemoteSigned
PS E:\GitHub\reactstudy\my-app> get-ExecutionPolicy
RemoteSigned
```

```
//解决 安装后的警告：找到报错路径，替换掉报错的代码
//const NRMC = path.join(process.env.HOME, '.nrmrc'); (注掉)
const NRMC = path.join(process.env[(process.platform == 'win32') ?
'USERPROFILE' : 'HOME'], '.nrmrc');
```

```
# 常用nrm命令
npm i nrm -g           # 将nrm包下载为全局工具
nrm ls                 # 查看所有可用的镜像源
nrm use 镜像源名       # 切换镜像源路径

nrm test               # 测试并返回各个镜像源的速度
```

mongoose

- 用于连接Node.js与MongoDB的中间层，

```
# 在项目中安装mongoose 包
npm install mongoose --save -dev
```

nodemon

- 当修改并保存node可执行文件时，自动重启node服务

```
# 全局安装nodemon
npm install -g nodemon
# 启动运行文件 不再使用:node 文件名
nodemon 文件名
```

http-server

- 快捷创建本地服务器
- 命令行窗口：任意文件路径下 http-server
 - 创建本地地址
 - 局域网地址

multer

- Multer 是一个用于处理 multipart/form-data 的 node.js 中间件，主要用于上传文件。它被写在 busboy 的顶部以获得最大的效率。
- 在项目中安装 npm install multer
- 注意：用户获取服务器图片需要处理对应的静态资源请求

```
// 导入multer
const multer = require('multer');
// 配置文件保存的目录
const upload = multer({ dest: '设置图片上传的路径' })

// upload.single('avatar') 配置上传单一文件时的name属性 avatar
// 在执行回调函数之前执行，把图片存储到本地，再执行回调函数
// req.file得到上传的文件信息
app.post('/api', upload.single('avatar'), (req, res) => {
  console.log(req.file);
  res.send(req.file)
});

// 得到的信息格式 req.file
{
  fieldname: 'avatar',
  originalname: '1F36A2AFF47749C43C6665AC1E674EFE.jpg',
  encoding: '7bit',
  mimetype: 'image/jpeg',
  destination: './a',
  filename: '0c53ee9af2827ba7cdb7b55cdabc31c2',
  path: 'a\\0c53ee9af2827ba7cdb7b55cdabc31c2',
  size: 1096
}
```

```
// 更好的使用方式:
const multer = require('multer');
const storage = multer.diskStorage({
  // 上传的文件目录
  destination: function(req, file, cd) {
    cd(null, path.join(__dirname, 'public', 'images', 'goods'))
  },
  // 上传文件的名称
  filename: function(req, file, cd) {
    cd(null, file.originalname)
  }
});
// 应用配置
const upload = multer({ storage })
```

[alipay-sdk](#)

- [支付宝开放平台](#)
- [SDK 配置](#)
- 使用步骤

```
○ // 下载alipay-sdk
npm i alipay-sdk
// 建立单独文件配置支付基本信息
const AlipaySdk = require('alipay-sdk').default;
const alipaySdk = new AlipaySdk({
  // 应用id
  appId: '2021003129602062',
  // 签名算法
  signType: 'RSA2',
  // 支付宝网关地址
  gateway: 'https://openapi.alipay.com/gateway.do',
  // 支付宝公钥 在支付包软件中生成应用公钥 复制到支付宝网站中配置，生成支付宝公钥
  alipayPublicKey: '',
  // 应用私钥 在支付包软件中生成应用私钥
  privateKey: ''
})
module.exports = alipaySdk;

// 在主文件中 导入相关模块
const AlipayFormData = require('alipay-sdk/lib/form').default;
const alipay = require('./alipay/index.js');

// 创建订单
app.post('/alipay', async function(req, res) {
  console.log(req.body);
  const formData = new AlipayFormData();
  formData.setMethod('get');
  // 成功之后的异步通知地址 notifyUrl
  // formData.addField('notifyUrl', 'http://wudetian.top');
  // 成功之后的同步通知地址
  // formData.addField('returnUrl', 'http://wudetian.top');
  formData.addField('bizContent', {
    outTradeNo: req.body.id, // 商户订单号，确保商户端唯一
    productCode: 'FACE_TO_FACE_PAYMENT', // 固定的，不同支付方式该值不同
    totalAmount: '0.01', //总金额
    subject: '定制开发', //商品标题
    body: req.body.name, //商品详情
  });
  // 生成支付信息
  const result = await alipay.exec('alipay.trade.precreate', {}, { formData });
  // 从官方文档看到，result 包含 tradeNo、outTradeNo 2 个 key
  await axios({
    // 请求方法
    method: "get",
    // 请求url
```

```

        url: result,
    }).then(response => {
        let url =
response.data.alipay_trade_precreate_response.qr_code
        res.send(url);
    }).catch(err => {
        // 处理错误信息
        console.error(err);
    })

    })

    // 交易查询
    app.post('/alipayOver', async function(req, res) {
        const formData = new AlipayFormData();
        formData.setMethod('get');
        formData.addField('bizContent', {
            out_trade_no: req.body.id,
        });
        // 生成支付信息
        const result = await alipay.exec('alipay.trade.query', {}, {
formData });
        // 从官方文档看到, result 包含 tradeNo、outTradeNo 2 个 key
        await axios({
            // 请求方法
            method: "get",
            // 请求url
            url: result,
        }).then(response => {
            let msg =
response.data.alipay_trade_query_response.sub_msg;
            let code = response.data.alipay_trade_query_response.code;
            if (code == 10000) {
                let status =
response.data.alipay_trade_query_response.trade_status;
                switch (status) {
                    case 'WAIT_BUYER_PAY':
                        reqmsg = '未付款';
                        break;
                    case 'TRADE_CLOSED':
                        reqmsg = '交易超时,已退款';
                        break;
                    case 'TRADE_SUCCESS':
                        reqmsg = '支付成功';
                        break;
                    case 'TRADE_FINISHED':
                        reqmsg = '支付成功';
                        break;
                    default:
                        reqmsg = '交易失败';
                }
            } else {
                reqmsg = msg;
            }
        })
    })

```

```

        console.log(reqmsg);
        res.send(reqmsg);
    }).catch(err => {
        // 处理错误信息
        console.error(err);
        res.send('出错了')
    })
})
})

```

Express

- 快速、开发、极简的**web开发框架**，可以用来创建web服务
- 用途：
 - **Web 网站服务器**：专门对外提供web网页资源的服务器
 - **API接口**：专门对外提供 API 接口的服务器。
- 使用

1. 安装 Express `npm i express`
2. 创建基本服务器

```

//导入 express
const express = require('express');
//创建 应用对象
const app = express();

//创建路由规则 见3、4
//利用 app.listen(端口号、回调函数)：监听端口 启动服务器
app.listen(80, () => {
    console.log('express server running at http://127.0.0.1')
    console.log('80默认端口监听中')
})

```

- `app.get()` 方法，监听客户端的 get请求。
 - 配置路由规则，可以响应的访问路径、请求的方式（如果不配置 前端请求时会报错）
- `app.post()` 方法，监听客户端的 post请求。
- `app.all()` 方法，监听所有类型的请求。

```

// request: 对请求报文的封装，包含与请求相关的属性和方法
// response: 是对响应报文的封装，包含与响应相关的属性和方法
// 参数1: 客户端请求的 url 地址      参数2: 请求的回调函数

app.get('请求的url', function(request, response){
    // 设置响应头，允许跨域 *表示所有的url请求都支持
    response.setHeader('Access-Control-Allow-Origin', '*');
    // 表示接受任意的请求头信息，针对前端的自定义请求头信息
    response.setHeader("Access-Control-Allow-Headers", "*");
    /*处理函数*/
}

```

```

    response.send('hello express');
  })

  app.post('请求的url',function(req,res){
    /*处理函数*/
    res.send('hello express'); //设置响应体
  })

  app.all('请求的url',function(req,res){
    /*处理函数*/
    res.send('hello express'); //设置响应体
  })

```

- 向客户端发送相应信息
 - 通过 res.send() 方法，可以把处理好的内容，发送给客户端
 - res.send() 应该写在监听请求处理函数的末尾
 - send中的数据必须是字符串类型
 - 如果是对象或其他数据格式，需要使用js方法进行转化

```

app.get('/url',(req,res) => {
  //向客户端发送 json 对象
  res.send({
    name:'zs',
    age:20,
    gender:'男'
  })
})

app.post('/url',(req,res) => {
  //向客户端发送文本内容
  res.send('请求成功');
})

```

req请求参数

- req.query对象
 - 获取 url中携带的查询参数
 - 可以访问到客户端通过查询字符串的形式，发送到服务器的参数：
- req.params 对象
 - 获取url的动态参数
 - 访问到 URL 中，匹配到的动态参数：

```

1 app.get('/', (req, res) => {
2   // req.query 默认是一个空对象
3   // 客户端使用 ?name=zs&age=20 这种查询字符串形式, 发送到服务器的参数,
4   // 可以通过 req.query 对象访问到, 例如:
5   // req.query.name    req.query.age
6   console.log(req.query)
7 })

```

```

1 // URL 地址中, 可以通过 :参数名 的形式, 匹配动态参数值
2 app.get('/user/:id', (req, res) => {
3   // req.params 默认是一个空对象
4   // 里面存放着通过 : 动态匹配到的参数值
5   console.log(req.params)
6 })

```

解决跨域问题

```

app.all('*', function(req, res, next) {
  // 允许跨域
  res.header("Access-Control-Allow-Origin", "*");
  // 允许携带自定义请求头信息
  res.header("Access-Control-Allow-Headers", "X-Requested-With");
  // 支持接收的请求类型
  res.header("Access-Control-Allow-Methods", "PUT,POST,GET,DELETE,OPTIONS");
  res.header('Access-Control-Allow-Headers', 'Content-Type, Content-Length, Authorization, Accept, X-Requested-With , yourHeaderFeild');
  res.header("Content-Type", "application/json;charset=utf-8");
  next();
});

```

静态资源及post请求问题

```

// 解决POST请求参数无法解析的问题
app.use(express.json());
app.use(express.urlencoded({ extended: true }));
// 设置静态资源路径
app.use(express.static(path.join(__dirname, 'public')))

```

[指数 \(mongodb.github.io\)](https://github.com/mongodb)

Node概述

编译过程区别

- Java
 - 先把源码 编译为 字节码
 - JVM优化执行 字节码
 - JRE(Java环境) 运行
- V8引擎
 - JS源码 抽象为 语法树
 - 转化为本地机器码
 - 直接运行

Node.js安装

- linux环境:
 - 使用库安装 (epel仓库, 为linux及衍生版提供高质量的软件安装源)
 - ```
yum install epel-release # 安装epel仓库
yum install nodejs # 安装node.js
npm yum install npm # 下载相关 NPM 的主要核心依赖包
```
  - 使用源代码安装
    - ```
wget https://npmirror.com/mirrors/node/v14.18.2/node-
v14.18.2.tar.gz # wget命令和下载参数 可以从网络下载安装包

tar xvf node-v14.18.2.tar.gz    # 使用tar命令进行解压
cd node-v*                     # 使用cd命令进入解压后的目录
yum install gcc gcc-c++        # 安装相关的依赖库(gcc 和 gcc-
c++)

# 完成基本配置文件生成, 指定配置目录 /usr/local/node
# 使用configure命令, 完成 MakeFile 配置文件的生成
./configure --prefix=/usr/local/node
# 使用 make 命令, 生成编译好的库文件
make
# 最后使用 make install 将软件安装在linux中
make install
```
 - 两者差别:
 - 库安装的速度快, 但源代码安装 运行的速度更快。

进程管理

感知、控制自身进程的运行环境和状态，可以创建子进程并与其协同工作，可以把多个程序组合在一起共同完成某项工作，并在其中充当胶水和调度器的作用。

child_process模块

可以创建和控制子进程。该模块提供的API中最核心的是 `.spawn`，其余API都是针对特定使用场景对它的进一步封装，算是一种语法糖。