

## Node.js（最全）基础+全栈项目

### 一、Node.js基础

#### 1. 认识Node.js

01 nodejs的特性

02 使用 Node.js 需要了解多少 JavaScript

03 浏览器环境vs node环境

#### 2. 开发环境搭建

#### 3. 模块、包、commonJS

02 CommonJS规范

03 modules模块化规范写法

#### 4. Npm&Yarn

01 npm的使用

02 全局安装 nrm

03 yarn使用

#### 5. 内置模块

01 http模块

02 url模块

03 querystring模块

04 http模块补充

05 event模块

06 fs文件操作模块

07 stream流模块

08 zlib

09 crypto

#### 6. 路由

01 基础

02 获取参数

03 静态资源处理

### 二、Express

#### 1.特色

#### 2.安装

#### 3.路由

#### 4.中间件

(1) 应用级中间件

(2) 路由级中间件

(3) 错误处理中间件

(4) 内置的中间件

(5) 第三方中间件

#### 5. 获取请求参数

#### 6.利用 Express 托管静态文件

#### 7.服务端渲染（模板引擎）

### 三、MongoDB

#### 1.关系型与非关系型数据库

#### 2.安装数据库

#### 3.启动数据库

(1) windows

(2) mac

#### 4.在命令行中操作数据库

#### 5.可视化工具进行增删改查

#### 6.nodejs连接操作数据库

### 四、接口规范与业务分层

#### 1.接口规范

#### 2.业务分层

### 五、登录鉴权

#### 1. Cookie&Session

## 2. JSON Web Token (JWT)

(1) 介绍

(2) 实现

## 六、文件上传管理

## 七、APIDOC - API 文档生成工具

## 八、Koa2

1.简介

2. 快速开始

    2.1 安装koa2

    2.2 hello world 代码

    2.3 启动demo

3. koa vs express

    3.1更轻量

    3.2 Context对象

    3.3 异步流程控制

    3.4 中间件模型

4. 路由

    4.1基本用发

    4.2 router.allowedMethods作用

    4.3 请求方式

    4.4 拆分路由

    4.5 路由前缀

    4.6 路由重定向

5. 静态资源

6. 获取请求参数

    6.1get参数

    6.2post参数

7. ejs模板

    7.1 安装模块

    7.2 使用模板引擎

8. cookie&session

    8.1 cookie

    8.2 session

9. JWT

10.上传文件

11.操作MongoDB

## 九、MySQL

1.介绍

2.与非关系数据库区别

3.sql语句

4.nodejs 操作数据库

## 十、Socket编程

1.websocket介绍

2.ws模块

3.socket.io模块

## 十一、mocha

1.编写测试

2.chai断言库

3.异步测试

4.http测试

5.钩子函数

# Node.js (最全) 基础+全栈项目

作者: kerwin

版本: QF1.0

版权: 千锋HTML5大前端教研院

公众号: 大前端私房菜

## 一、Node.js基础

### 1. 认识Node.js

Node.js是一个javascript运行环境。它让javascript可以开发后端程序，实现几乎其他后端语言实现的所有功能，可以与PHP、Java、Python、.NET、Ruby等后端语言平起平坐。

Nodejs是基于V8引擎，V8是Google发布的开源JavaScript引擎，本身就是用于Chrome浏览器的js解释部分，但是Ryan Dahl这哥们，鬼才般的，把这个V8搬到了服务器上，用于做服务器的软件。

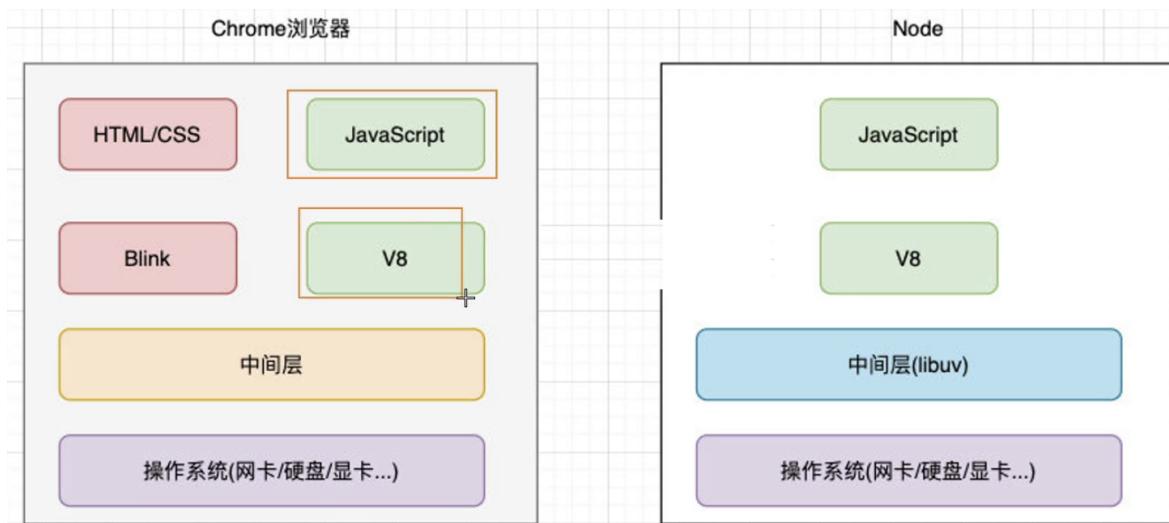
#### 01 nodejs的特性

- Nodejs语法完全是js语法，只要你懂js基础就可以学会Nodejs后端开发
- Nodejs超强的高并发能力,实现高性能服务器
- 开发周期短、开发成本低、学习成本低

#### 02 使用 Node.js 需要了解多少 JavaScript

<http://nodejs.cn/learn/how-much-javascript-do-you-need-to-know-to-use-nodejs>

#### 03 浏览器环境vs node环境



Node.js 可以解析JS代码（没有浏览器安全级别的限制）提供很多系统级别的API，如：

- 文件的读写 (File System)

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

fs.readFile('./ajax.png', 'utf-8', (err, content) => {
  console.log(content)
})
```

- 进程的管理 (Process)

```
function main(argv) {
  console.log(argv)
}

main(process.argv.slice(2))
```

- 网络通信 (HTTP/HTTPS)

```
const http = require("http")

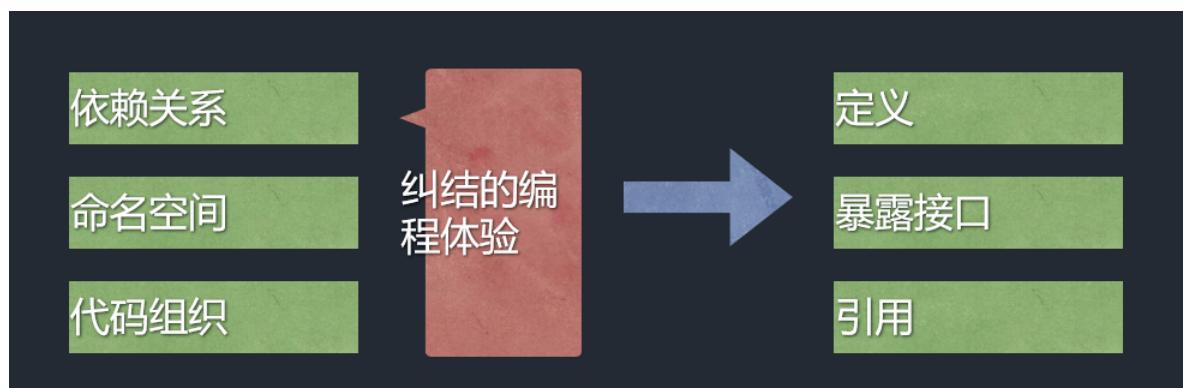
http.createServer((req,res) => {
  res.writeHead(200, {
    "content-type": "text/plain"
  })
  res.write("hello nodejs")
  res.end()
}).listen(3000)
```

## 2. 开发环境搭建

<http://nodejs.cn/download/>



## 3. 模块、包、commonJS



### 02 CommonJS规范

#### 03 modules模块化规范写法

我们可以把公共的功能抽离成为一个单独的 js 文件 作为一个模块，默认情况下这个模块里面的方法或者属性，外面是没法访问的。如果要让外部可以访问模块里面的方法或者属性，就必须在模块里面通过 exports 或者 module.exports 暴露属性或者方法。

# CommonJS 规范

m1.js:

```
const name = 'gp19'

const sayName = () => {
    console.log(name)
}

console.log('module 1')

// 接口暴露方法一:
module.exports = {
    say: sayName
}

// 接口暴露方法二:
exports.say = sayName

// 错误!
exports = {
    say: sayName
}
```

main.js:

```
const m1 = require('./m1')
m1.say()
```

## 4. Npm&Yarn

### 01 npm的使用

```
npm init
npm install 包名 -g (uninstall,update)
npm install 包名 --save-dev (uninstall,update)
npm list -g (不加-g, 列举当前目录下的安装包)
npm info 包名 (详细信息)  npm info 包名 version(获取最新版本)
npm install md5@1 (安装指定版本)
npm outdated( 检查包是否已经过时)
```

"dependencies": { "md5": "^2.1.0" } ^ 表示 如果 直接npm install 将会 安装  
2.\*.\* 最新版本

"dependencies": { "md5": "~2.1.0" } ~ 表示 如果 直接npm install 将会 安装  
md5 2.1.\* 最新版本

```
"dependencies": { "md5": "*" } * 表示如果直接npm install 将会安装 md5  
最新版本
```

## 02 全局安装 nrm

NRM (npm registry manager)是npm的镜像源管理工具，有时候国外资源太慢，使用这个就可以快速地在 npm 源间切换。

```
手动切换方法: npm config set registry https://registry.npm.taobao.org
```

### 安装 nrm

在命令行执行命令，npm install -g nrm，全局安装nrm。

### 使用 nrm

执行命令 nrm ls 查看可选的源。其中，带\*的是当前使用的源，上面的输出表明当前源是官方源。

### 切换 nrm

如果要切换到taobao源，执行命令nrm use taobao。

### 测试速度

你还可以通过 nrm test 测试相应源的响应时间。

```
nrm test
```

扩展：

### 中国 NPM 镜像

这是一个完整 [npmjs.org](#) 镜像，你可以用此代替官方版本(只读)，同步频率目前为 10分钟 一次以保证尽量与官方服务同步。

```
npm install -g cnpm --registry=https://registry.npmmirror.com
```

## 03 yarn使用

```
npm install -g yarn
```

对比npm：

速度超快：Yarn 缓存了每个下载过的包，所以再次使用时无需重复下载。同时利用并行下载以最大化资源利用率，因此安装速度更快。

超级安全：在执行代码之前，Yarn 会通过算法校验每个安装包的完整性。

开始新项目

```
yarn init
```

添加依赖包

```
yarn add [package]
```

```
yarn add [package]@[version]
```

```
yarn add [package] --dev
```

升级依赖包

```
yarn upgrade [package]@[version]
```

移除依赖包

```
yarn remove [package]
```

安装项目的全部依赖

```
yarn install
```

## 5. 内置模块

### 01 http模块

要使用 HTTP 服务器和客户端，则必须 `require('http')`。

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

// 创建本地服务器来从其接收数据
const server = http.createServer((req, res) => {
  res.writeHead(200, { 'Content-Type': 'application/json' });
  res.end(JSON.stringify({
    data: 'Hello world!'
  }));
});

server.listen(8000);
```

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

// 创建本地服务器来从其接收数据
const server = http.createServer();

// 监听请求事件
server.on('request', (request, res) => {
  res.writeHead(200, { 'Content-Type': 'application/json' });
  res.end(JSON.stringify({
    data: 'Hello world!'
  }));
});

server.listen(8000);
```

### 02 url模块

#### 02.1 parse

```
const url = require('url')
const urlString = 'https://www.baidu.com:443/ad/index.html?
id=8&name=mouse#tag=110'
const parsedStr = url.parse(urlString)
console.log(parsedStr)
```

#### 02.2 format

```
const url = require('url')
const urlObject = {
  protocol: 'https:',
  slashes: true,
```

```
auth: null,
host: 'www.baidu.com:443',
port: '443',
hostname: 'www.baidu.com',
hash: '#tag=110',
search: '?id=8&name=mouse',
query: { id: '8', name: 'mouse' },
pathname: '/ad/index.html',
path: '/ad/index.html?id=8&name=mouse'
}
const parsedObj = url.format(urlObject)
console.log(parsedObj)
```

## 02.3 resolve

```
const url = require('url')
var a = url.resolve('/one/two/three', 'four')  ( 注意最后加 / , 不加 / 的区别 )
var b = url.resolve('http://example.com/', '/one')
var c = url.resolve('http://example.com/one', '/two')
console.log(a + "," + b + "," + c)
```

## 03 querystring模块

### 03.1 parse

```
const querystring = require('querystring')
var qs = 'x=3&y=4'
var parsed = querystring.parse(qs)
console.log(parsed)
```

### 03.2 stringify

```
const querystring = require('querystring')
var qo = {
  x: 3,
  y: 4
}
var parsed = querystring.stringify(qo)
console.log(parsed)
```

### 03.3 escape/unescape

```

1  'use strict';
2
3  const mysql = require('mysql');
4
5  let param = 'ns';
6  let pool = mysql.createPool({
7    user: 'root',
8    password: 'root',
9    database: 'nlp_dict'
10 });
11
12 pool.getConnection(function (err, conn) {
13   let sql = 'select * from tb_nature where nature = "' + param + '" and del_status=1';
14   conn.query(sql, function (err, result) {
15     console.log(result);
16   })
17 });

```

这时正常情况下能查询到一条数据，如果将param修改成

```
let param = 'ns"-- ';
```

sql语句就会变成

```
select * from tb_nature where nature = "ns"-- " and del_status=1
```

后面的del\_status就会被参数中的 -- 注释掉，失去作用，能查询到多条数据。

如果对param使用escape包装下，就能将参数中的特殊字符进行转义，防止sql的注入。

```
let sql = 'select * from tb_nature where nature = ' + mysql.escape(param) + ' and
del_status=1';
```

```

const querystring = require('querystring')
var str = 'id=3&city=北京&url=https://www.baidu.com'
var escaped = querystring.escape(str)
console.log(escaped)

```

```

const querystring = require('querystring')
var str =
'id%3D%26city%3D%E5%8C%97%E4%BA%AC%26url%3Dhttps%3A%2F%2Fwww.baidu.com'
var unescaped = querystring.unescape(str)
console.log(unescaped)

```

## 04 http模块补充

### 04.1 接口: jsonp

```

const http = require('http')
const url = require('url')

const app = http.createServer((req, res) => {
  let urlObj = url.parse(req.url, true)

  switch (urlObj.pathname) {
    case '/api/user':
      res.end(` ${urlObj.query.cb}({ "name": "gp145" }) `)
  }
})

```

```

        break
    default:
        res.end('404.')
        break
    }
})

app.listen(8080, () => {
    console.log('localhost:8080')
})

```

## 04.2 跨域: CORS

```

const http = require('http')
const url = require('url')
const querystring = require('querystring')

const app = http.createServer((req, res) => {
    let data = ''
    let urlObj = url.parse(req.url, true)

    res.writeHead(200, {
        'content-type': 'application/json;charset=utf-8',
        'Access-Control-Allow-Origin': '*'
    })

    req.on('data', (chunk) => {
        data += chunk
    })

    req.on('end', () => {
        responseResult(querystring.parse(data))
    })
}

function responseResult(data) {
    switch (urlObj.pathname) {
        case '/api/login':
            res.end(JSON.stringify({
                message: data
            }))
            break
        default:
            res.end('404.')
            break
    }
}
})

app.listen(8080, () => {
    console.log('localhost:8080')
})

```

## 04.3 模拟get

```

var http = require('http')
var https = require('https')

```

```

// 1、接口 2、跨域
const server = http.createServer((request, response) => {
  var url = request.url.substr(1)

  var data = ''

  response.writeHead(200, {
    'content-type': 'application/json;charset=utf-8',
    'Access-Control-Allow-Origin': '*'
  })

  https.get(`https://m.lagou.com/listmore.json?url`, (res) => {

    res.on('data', (chunk) => {
      data += chunk
    })

    res.on('end', () => {
      response.end(JSON.stringify({
        ret: true,
        data
      }))
    })
  })
}

server.listen(8080, () => {
  console.log('localhost:8080')
})

```

#### 04.4 模拟post：服务器提交（攻击）

```

const https = require('https')
const querystring = require('querystring')

const postData = querystring.stringify({
  province: '上海',
  city: '上海',
  district: '宝山区',
  address: '同济支路199号智慧七立方3号楼2-4层',
  latitude: 43.0,
  longitude: 160.0,
  message: '求购一条小鱼',
  contact: '13666666',
  type: 'sell',
  time: 1571217561
})

const options = {
  protocol: 'https:',
  hostname: 'ik9hkddr.qcloud.la',
  method: 'POST',
  port: 443,
  path: '/index.php/trade/add_item',
  headers: {

```

```

    'Content-Type': 'application/x-www-form-urlencoded',
    'Content-Length': Buffer.byteLength(postData)
  }
}

function doPost() {
  let data

  let req = https.request(options, (res) => {
    res.on('data', chunk => data += chunk)
    res.on('end', () => {
      console.log(data)
    })
  })

  req.write(postData)
  req.end()
}

// setInterval(() => {
//   doPost()
// }, 1000)

```

## 04.5 爬虫

```

const https = require('https')
const http = require('http')
const cheerio = require('cheerio')

http.createServer((request, response) => {
  response.writeHead(200, {
    'content-type': 'application/json;charset=utf-8'
  })

  const options = {
    // protocol: 'https:',
    hostname: 'i.maoyan.com',
    port: 443,
    path: '/',
    method: 'GET'
  }

  const req = https.request(options, (res) => {
    let data = ''
    res.on('data', (chunk) => {
      data += chunk
    })

    res.on('end', () => {
      filterData(data)
    })
  })

  function filterData(data) {
    // console.log(data)
  }
})

```

```

let $ = cheerio.load(data)
let $movieList = $('.column.content')
console.log($movieList)
let movies = []
$movieList.each((index, value) => {
  movies.push({
    title: $(value).find('.movie-title .title').text(),
    detail: $(value).find('.detail .actor').text(),
  })
})

response.end(JSON.stringify(movies))
}

req.end()
}).listen(3000)

```

## 05 event模块

```

const EventEmitter = require('events')

class MyEventEmitter extends EventEmitter {}

const event = new MyEventEmitter()

event.on('play', (movie) => {
  console.log(movie)
})

event.emit('play', '我和我的祖国')
event.emit('play', '中国机长')

```

## 06 fs文件操作模块

```

const fs = require('fs')

// 创建文件夹
fs.mkdir('./logs', (err) => {
  console.log('done.')
})

// 文件夹改名
fs.rename('./logs', './log', () => {
  console.log('done')
})

// 删除文件夹
fs.rmdir('./log', () => {
  console.log('done.')
})

// 写内容到文件里
fs.writeFile(
  './logs/log1.txt',
  'hello',
  // 错误优先的回调函数
  (err) => {

```

```
        if (err) {
            console.log(err.message)
        } else {
            console.log('文件创建成功')
        }
    }

// 给文件追加内容
fs.appendFile('./logs/log1.txt', '\nworld', () => {
    console.log('done.')
})

// 读取文件内容
fs.readFile('./logs/log1.txt', 'utf-8', (err, data) => {
    console.log(data)
})

// 删除文件
fs.unlink('./logs/log1.txt', (err) => {
    console.log('done.')
})

// 批量写文件
for (var i = 0; i < 10; i++) {
    fs.writeFile(`./logs/log-${i}.txt`, `log-${i}`, (err) => {
        console.log('done.')
    })
}

// 读取文件/目录信息
fs.readdir('./', (err, data) => {
    data.forEach((value, index) => {
        fs.stat(`./${value}`, (err, stats) => {
            // console.log(value + ':' + stats.size)
            console.log(value + ' is ' + (stats.isDirectory() ? 'directory' : 'file'))
        })
    })
})

// 同步读取文件
try {
    const content = fs.readFileSync('./logs/log-1.txt', 'utf-8')
    console.log(content)
    console.log(0)
} catch (e) {
    console.log(e.message)
}

// 异步读取文件: 方法一
fs.readFile('./logs/log-0.txt', 'utf-8', (err, content) => {
    console.log(content)
    console.log(0)
})
console.log(1)

// 异步读取文件: 方法二
const fs = require("fs").promises
```

```
fs.readFile('./logs/log-0.txt', 'utf-8').then(result => {
  console.log(result)
})
```

在 `fs` 模块中，提供同步方法是为了方便使用。那我们到底是应该用异步方法还是同步方法呢？

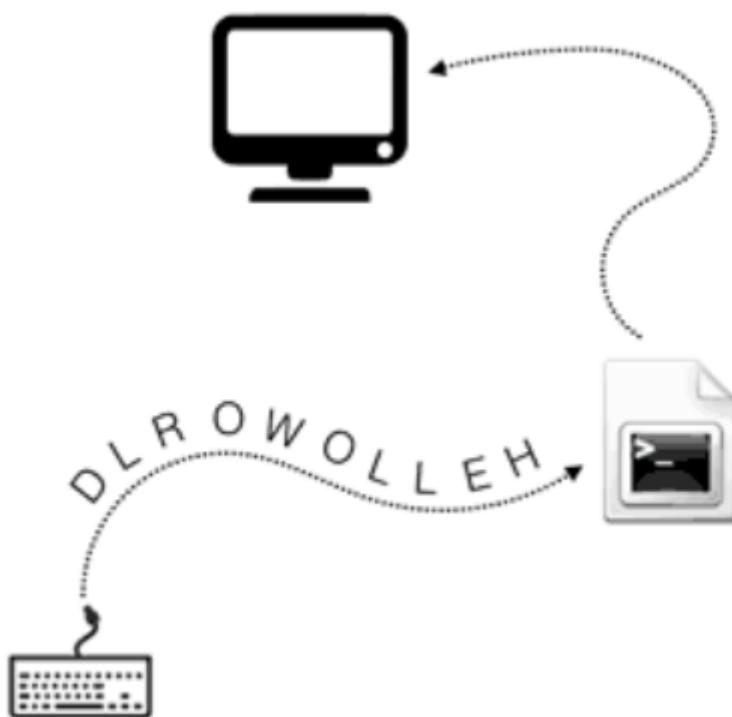
由于Node环境执行的JavaScript代码是服务器端代码，所以，绝大部分需要在服务器运行期反复执行行业务逻辑的代码，必须使用异步代码，否则，同步代码在执行时期，服务器将停止响应，因为JavaScript只有一个执行线程。

服务器启动时如果需要读取配置文件，或者结束时需要写入到状态文件时，可以使用同步代码，因为这些代码只在启动和结束时执行一次，不影响服务器正常运行时的异步执行。

## 07 stream流模块

`stream` 是Node.js提供的又一个仅在服务区端可用的模块，目的是支持“流”这种数据结构。

什么是流？流是一种抽象的数据结构。想象水流，当在水管中流动时，就可以从某个地方（例如自来水厂）源源不断地到达另一个地方（比如你家的洗手池）。我们也可以把数据看成是数据流，比如你敲键盘的时候，就可以把每个字符依次连起来，看成字符流。这个流是从键盘输入到应用程序，实际上它还对应着一个名字：标准输入流 (`stdin`) 。



如果应用程序把字符一个一个输出到显示器上，这也可以看成是一个流，这个流也有名字：标准输出流 (`stdout`) 。流的特点是数据是有序的，而且必须依次读取，或者依次写入，不能像Array那样随机定位。

有些流用来读取数据，比如从文件读取数据时，可以打开一个文件流，然后从文件流中不断地读取数据。有些流用来写入数据，比如向文件写入数据时，只需要把数据不断地往文件流中写进去就可以了。

在Node.js中，流也是一个对象，我们只需要响应流的事件就可以了：`data` 事件表示流的数据已经可以读取了，`end` 事件表示这个流已经到末尾了，没有数据可以读取了，`error` 事件表示出错了。

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

// 打开一个流：
```

```
var rs = fs.createReadStream('sample.txt', 'utf-8');

rs.on('data', function (chunk) {
    console.log('DATA:')
    console.log(chunk);
});

rs.on('end', function () {
    console.log('END');
});

rs.on('error', function (err) {
    console.log('ERROR: ' + err);
});
```

要注意，`data` 事件可能会有多次，每次传递的 `chunk` 是流的一部分数据。

要以流的形式写入文件，只需要不断调用 `write()` 方法，最后以 `end()` 结束：

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

var ws1 = fs.createWriteStream('output1.txt', 'utf-8');
ws1.write('使用Stream写入文本数据...\n');
ws1.write('END.');
ws1.end();
```

`pipe` 就像可以把两个水管串成一个更长的水管一样，两个流也可以串起来。一个 `Readable` 流和一个 `Writable` 流串起来后，所有的数据自动从 `Readable` 流进入 `Writable` 流，这种操作叫 `pipe`。

在Node.js中，`Readable` 流有一个 `pipe()` 方法，就是用来干这件事的。

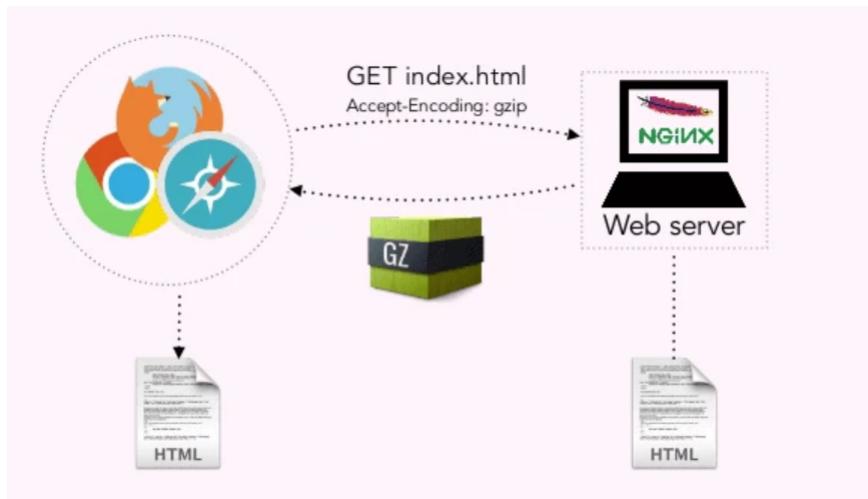
让我们用 `pipe()` 把一个文件流和另一个文件流串起来，这样源文件的所有数据就自动写入到目标文件里了，所以，这实际上是一个复制文件的程序：

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

const readstream = fs.createReadStream('./1.txt')
const writestream = fs.createWriteStream('./2.txt')

readstream.pipe(writestream)
```

08 zlib



```

const fs = require('fs')
const zlib = require('zlib')

const gzip = zlib.createGzip()

const readstream = fs.createReadStream('./note.txt')
const writestream = fs.createWriteStream('./note2.txt')

readstream
  .pipe(gzip)
  .pipe(writestream)
  
```

## 09 crypto

crypto模块的目的是为了提供通用的加密和哈希算法。用纯JavaScript代码实现这些功能不是不可能，但速度会非常慢。Node.js用C/C++实现这些算法后，通过crypto这个模块暴露为JavaScript接口，这样用起来方便，运行速度也快。

MD5是一种常用的哈希算法，用于给任意数据一个“签名”。这个签名通常用一个十六进制的字符串表示：

```

const crypto = require('crypto');

const hash = crypto.createHash('md5');

// 可任意多次调用update():
hash.update('Hello, world!');
hash.update('Hello, nodejs!');

console.log(hash.digest('hex'));
  
```

`update()`方法默认字符串编码为UTF-8，也可以传入Buffer。

如果要计算SHA1，只需要把`'md5'`改成`'sha1'`，就可以得到SHA1的结果  
`1f32b9c9932c02227819a4151feed43e131aca40`。

Hmac算法也是一种哈希算法，它可以利用MD5或SHA1等哈希算法。不同的是，Hmac还需要一个密钥：

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

const hmac = crypto.createHmac('sha256', 'secret-key');

hmac.update('Hello, world!');
hmac.update('Hello, nodejs!');

console.log(hmac.digest('hex')) // 80f7e22570...
```

只要密钥发生了变化，那么同样的输入数据也会得到不同的签名，因此，可以把Hmac理解为用随机数“增强”的哈希算法。

AES是一种常用的对称加密算法，加解密都用同一个密钥。crypto模块提供了AES支持，但是需要自己封装好函数，便于使用：

```
const crypto = require("crypto");

function encrypt(key, iv, data) {
    let decipher = crypto.createCipheriv('aes-128-cbc', key, iv);
    // decipher.setAutoPadding(true);
    return decipher.update(data, 'binary', 'hex') + decipher.final('hex');
}

function decrypt(key, iv, crypted) {
    crypted = Buffer.from(crypted, 'hex').toString('binary');
    let decipher = crypto.createDecipheriv('aes-128-cbc', key, iv);
    return decipher.update(crypted, 'binary', 'utf8') + decipher.final('utf8');
}
key,iv必须是16个字节
```

可以看出，加密后的字符串通过解密又得到了原始内容。

## 6. 路由

### 01 基础

```
/*
 * @作者: kerwin
 * @公众号: 大前端私房菜
 */
var fs = require("fs")
var path = require("path")

function render(res, path) {
    res.writeHead(200, { "Content-Type": "text/html; charset=utf8" })
    res.write(fs.readFileSync(path, "utf8"))
    res.end()
}

const route = {
    "/login": (req, res) => {
        render(res, "./static/login.html")
```

```

},
"/home": (req, res) => {
  render(res, "./static/home.html")
},
"/404": (req, res) => {
  res.writeHead(404, { "Content-Type": "text/html; charset=utf8" })
  res.write(fs.readFileSync("./static/404.html", "utf8"))
}
}

```

## 02 获取参数

get请求

```

"/api/login":(req,res)=>{
  const myURL = new URL(req.url, 'http://127.0.0.1:3000');
  console.log(myURL.searchParams.get("username"))
  render(res,`{ok:1}`)
}

```

post请求

```

"/api/login": (req, res) => {
  var post = '';
  // 通过req的data事件监听函数，每当接受到请求体的数据，就累加到post变量中
  req.on('data', function (chunk) {
    post += chunk;
  });

  // 在end事件触发后，通过querystring.parse将post解析为真正的POST请求格式，然后向
  // 客户端返回。
  req.on('end', function () {
    post = JSON.parse(post);
    render(res, `ok:1`)
  });
}

```

## 03 静态资源处理

```

function readStaticFile(req, res) {
  const myURL = new URL(req.url, 'http://127.0.0.1:3000')
  var filePathname = path.join(__dirname, "/static", myURL.pathname);

  if (fs.existsSync(filePathname)) {
    // console.log(1111)
    res.writeHead(200, { "Content-Type":
      `${mime.getType(myURL.pathname.split(".")[1])}; charset=utf8` })
    res.write(fs.readFileSync(filePathname, "utf8"))
    res.end()
    return true
  } else {
}

```

```
        return false
    }
}
```

## 二、Express

<https://www.expressjs.com.cn/>

基于 Node.js 平台，快速、开放、极简的 web 开发框架。

### 1. 特色

#### 1、Web 应用

Express 是一个基于 Node.js 平台的极简、灵活的 web 应用开发框架，它提供一系列强大的特性，帮助你创建各种 Web 和移动设备应用。

#### 2、API

丰富的 HTTP 快捷方法和任意排列组合的 Connect 中间件，让你创建健壮、友好的 API 变得既快速又简单。

#### 3、性能

Express 不对 Node.js 已有的特性进行二次抽象，我们只是在它之上扩展了 Web 应用所需的基本功能。

### 2. 安装

```
$ npm install express --save
```

### 3. 路由

路由是指如何定义应用的端点 (URLs) 以及如何响应客户端的请求。

路由是由一个 URI、HTTP 请求 (GET、POST 等) 和若干个句柄组成，它的结构如下：

app.METHOD(path, [callback...], callback)， app 是 express 对象的一个实例，METHOD 是一个 HTTP 请求方法，path 是服务器上的路径，callback 是当路由匹配时要执行的函数。

下面是一个基本的路由示例：

```
var express = require('express');
var app = express();

// respond with "hello world" when a GET request is made to the homepage
app.get('/', function(req, res) {
  res.send('hello world');
});
```

路由路径和请求方法一起定义了请求的端点，它可以是字符串、字符串模式或者正则表达式。

```
// 匹配根路径的请求
app.get('/', function (req, res) {
  res.send('root');
});

// 匹配 /about 路径的请求
app.get('/about', function (req, res) {
```

```

    res.send('about');

});

// 匹配 /random.text 路径的请求
app.get('/random.text', function (req, res) {
  res.send('random.text');
});

```

使用字符串模式的路由路径示例：

```

// 匹配 acd 和 abcd
app.get('/ab?cd', function(req, res) {
  res.send('ab?cd');
});

// 匹配 /ab/********
app.get('/ab/:id', function(req, res) {
  res.send('aaaaaaaa');
});

// 匹配 abcd、abcd、abbcd等
app.get('/ab+cd', function(req, res) {
  res.send('ab+cd');
});

// 匹配 abcd、abxcd、abRABDOMcd、ab123cd等
app.get('/ab*cd', function(req, res) {
  res.send('ab*cd');
});

// 匹配 /abe 和 /abcde
app.get('/ab(cd)?e', function(req, res) {
  res.send('ab(cd)?e');
});

```

使用正则表达式的路由路径示例：

```

// 匹配任何路径中含有 a 的路径:
app.get(/a/, function(req, res) {
  res.send('/a/');
});

// 匹配 butterfly、dragonfly，不匹配 butterflyman、dragonfly man等
app.get('.*fly$', function(req, res) {
  res.send('/.*fly$/');
});

```

可以为请求处理提供多个回调函数，其行为类似 中间件。唯一的区别是这些回调函数有可能调用 next('route') 方法而略过其他路由回调函数。可以利用该机制为路由定义前提条件，如果在现有路径上继续执行没有意义，则可将控制权交给剩下的路径。

```

app.get('/example/a', function (req, res) {
  res.send('Hello from A!');
});

```

使用多个回调函数处理路由（记得指定 next 对象）：

```
app.get('/example/b', function (req, res, next) {
  console.log('response will be sent by the next function ...');
  next();
}, function (req, res) {
  res.send('Hello from B!');
});
```

使用回调函数数组处理路由：

```
var cb0 = function (req, res, next) {
  console.log('CB0')
  next()
}

var cb1 = function (req, res, next) {
  console.log('CB1')
  next()
}

var cb2 = function (req, res) {
  res.send('Hello from C!')
}

app.get('/example/c', [cb0, cb1, cb2])
```

混合使用函数和函数数组处理路由：

```
var cb0 = function (req, res, next) {
  console.log('CB0')
  next()
}

var cb1 = function (req, res, next) {
  console.log('CB1')
  next()
}

app.get('/example/d', [cb0, cb1], function (req, res, next) {
  console.log('response will be sent by the next function ...')
  next()
}, function (req, res) {
  res.send('Hello from D!')
})
```

## 4.中间件

Express 是一个自身功能极简，完全是由路由和中间件构成一个的 web 开发框架：从本质上来说，一个 Express 应用就是在调用各种中间件。

中间件（Middleware）是一个函数，它可以访问请求对象（request object (req)），响应对象（response object (res)），和 web 应用中处于请求-响应循环流程中的中间件，一般被命名为 next 的变量。

中间件的功能包括：

- 执行任何代码。
- 修改请求和响应对象。
- 终结请求-响应循环。
- 调用堆栈中的下一个中间件。

如果当前中间件没有终结请求-响应循环，则必须调用 `next()` 方法将控制权交给下一个中间件，否则请求就会挂起。

Express 应用可使用如下几种中间件：

- 应用级中间件
- 路由级中间件
- 错误处理中间件
- 内置中间件
- 第三方中间件

使用可选则挂载路径，可在应用级别或路由级别装载中间件。另外，你还可以同时装在一系列中间件函数，从而在一个挂载点上创建一个子中间件栈。

### (1) 应用级中间件

应用级中间件绑定到 `app` 对象 使用 `app.use()` 和 `app.METHOD()`，其中，`METHOD` 是需要处理的 HTTP 请求的方法，例如 `GET`, `PUT`, `POST` 等等，全部小写。例如：

```
var app = express()

// 没有挂载路径的中间件，应用的每个请求都会执行该中间件
app.use(function (req, res, next) {
  console.log('Time:', Date.now())
  next()
})
```

### (2) 路由级中间件

路由级中间件和应用级中间件一样，只是它绑定的对象为 `express.Router()`。

```
var router = express.Router()

var app = express()
var router = express.Router()

// 没有挂载路径的中间件，通过该路由的每个请求都会执行该中间件
router.use(function (req, res, next) {
  console.log('Time:', Date.now())
  next()
})

// 一个中间件栈，显示任何指向 /user/:id 的 HTTP 请求的信息
router.use('/user/:id', function(req, res, next) {
  console.log('Request URL:', req.originalUrl)
  next()
}, function (req, res, next) {
  console.log('Request Type:', req.method)
  next()
})
```

```
})

// 一个中间件栈，处理指向 /user/:id 的 GET 请求
router.get('/user/:id', function (req, res, next) {
  // 如果 user id 为 0，跳到下一个路由
  if (req.params.id == 0) next('route')
  // 负责将控制权交给栈中下一个中间件
  else next() //
}, function (req, res, next) {
  // 渲染常规页面
  res.render('regular')
})

// 处理 /user/:id，渲染一个特殊页面
router.get('/user/:id', function (req, res, next) {
  console.log(req.params.id)
  res.render('special')
})

// 将路由挂载至应用
app.use('/', router)
```

### (3) 错误处理中间件

错误处理中间件和其他中间件定义类似，只是要使用 4 个参数，而不是 3 个，其签名如下：(err, req, res, next)。

```
app.use(function(err, req, res, next) {
  console.error(err.stack)
  res.status(500).send('Something broke!')
})
```

### (4) 内置的中间件

express.static 是 Express 唯一内置的中间件。它基于 serve-static，负责在 Express 应用中托管静态资源。每个应用可有多个静态目录。

```
app.use(express.static('public'))
app.use(express.static('uploads'))
app.use(express.static('files'))
```

### (5) 第三方中间件

安装所需功能的 node 模块，并在应用中加载，可以在应用级加载，也可以在路由级加载。

下面的例子安装并加载了一个解析 cookie 的中间件：cookie-parser

```
$ npm install cookie-parser
```

```
var express = require('express')
var app = express()
var cookieParser = require('cookie-parser')

// 加载用于解析 cookie 的中间件
app.use(cookieParser())
```

## 5. 获取请求参数

get

```
req.query
```

post

```
app.use(express.urlencoded({extended:false}))  
app.use(express.json())  
req.body
```

## 6. 利用 Express 托管静态文件

通过 Express 内置的 express.static 可以方便地托管静态文件，例如图片、CSS、JavaScript 文件等。

将静态资源文件所在的目录作为参数传递给 express.static 中间件就可以提供静态资源文件的访问了。例如，假设在 public 目录放置了图片、CSS 和 JavaScript 文件，你就可以：

```
app.use(express.static('public'))
```

现在，public 目录下面的文件就可以访问了。

```
http://localhost:3000/images/kitten.jpg  
http://localhost:3000/css/style.css  
http://localhost:3000/js/app.js  
http://localhost:3000/images/bg.png  
http://localhost:3000/hello.html
```

所有文件的路径都是相对于存放目录的，因此，存放静态文件的目录名不会出现在 URL 中。

如果你的静态资源存放在多个目录下面，你可以多次调用 express.static 中间件：

```
app.use(express.static('public'))  
app.use(express.static('files'))
```

访问静态资源文件时，express.static 中间件会根据目录添加的顺序查找所需的文件。

如果你希望所有通过 express.static 访问的文件都存放在一个“虚拟（virtual）”目录（即目录根本不存在）下面，可以通过为静态资源目录指定一个挂载路径的方式来实现，如下所示：

```
app.use('/static', express.static('public'))
```

现在，你就可以通过带有 “/static” 前缀的地址来访问 public 目录下面的文件了。

```
http://localhost:3000/static/images/kitten.jpg  
http://localhost:3000/static/css/style.css  
http://localhost:3000/static/js/app.js  
http://localhost:3000/static/images/bg.png  
http://localhost:3000/static/hello.html
```

## 7.服务端渲染（模板引擎）

```
npm i ejs
```

- b. 把前端代码提供给后端，后端要把静态html以及里面的假数据给删掉，通过模板进行动态生成html的内容
- 2. 前后端分离，BSR（前端中组装页面）
  - a. 做好静态页面，动态效果。
  - b. json 模拟，ajax,动态创建页面。
  - c. 真实接口数据，前后联调。
  - d. 把前端提供给后端静态资源文件夹。
- view engine, 模板引擎，比如：app.set('view engine', 'ejs')

```
<% %>流程控制标签(写的是if else, for)  
<%= %>输出标签 (原文输出HTML标签)  
<%- %>输出标签 (HTML会被浏览器解析)  
<%# %>注释标签  
<%- include('user/show', {user: user}) %> 导入公共的模板内容
```

## 三、MongoDB

### 1.关系型与非关系型数据库

#### 关系型数据库与非关系型数据库的区别

##### 1. 关系型数据库

特点：

- (1) sql语句增删改查操作
  - (2) 保持事务的一致性,事物机制（回滚）
- mysql,sqlserver,db2,oracle

##### 2. 非关系型数据库

特点：

- (1) no sql:not only sql;
  - (2) 轻量，高效,自由。
- mongodb ,Hbase,Redis

##### 3. 为啥喜欢mongodb?

由于MongoDB独特的数据处理方式，可以将热点数据加载到内存，故而对查询来讲，会非常快（当然也会非常消耗内存）；同时由于采用了BSON的方式存储数据，故而对JSON格式数据具有非常好的支持性以及友好的表结构修改性，文档式的存储方式，数据友好可见；数据库的分片集群负载具有非常好的扩展性以及非常不错的自动故障转移。

<input type="checkbox"/> id	<input type="checkbox"/> im_id	<input type="checkbox"/> room_name	<input type="checkbox"/> room_desc
<input type="checkbox"/> 13	aaaa	铁锤	aaaa
<input type="checkbox"/> 14	bbbb	钢蛋	bbbb
<input type="checkbox"/> 15	cccc	诸葛山珍	cccc
<input type="checkbox"/> 16	dddd	轩辕翠花	dddd

SQL术语/概念	MongoDB术语/概念	解释/说明
database	database	数据库
table	collection	数据库表/集合
row	document	数据记录行/文档
column	field	数据字段/域
index	index	索引
table joins		表连接,MongoDB不支持
primary key	primary key	主键,MongoDB自动将_id字段设置为主键
		<pre>(1) ObjectId("62469603a4aa5b5b894055cb") { 8 fields } (2) ObjectId("6246a08e94cf3bd86e29a58d") { 8 fields }  {   "_id" : ObjectId("6246a08e94cf3bd86e29a58d"),   "title" : "aaa111",   "content" : "&lt;p&gt;aaaaaaaa111&lt;/p&gt;",   "category" : 2,   "cover" : "/newsuploads/c179792b6a64c1cb156d8946b72a2b2f",   "isPublish" : 0,   "editTime" : ISODate("2022-04-01T07:06:43.877Z"),   "__v" : 0 }</pre>

## 2.安装数据库

<https://docs.mongodb.com/manual/administration/install-community/>

## 3.启动数据库

### (1) windows

```
mongod --dbpath d:/data/db
mongo
```

### (2) mac

```
mongod --config /usr/local/etc/mongod.conf
mongo
```

## 4.在命令行中操作数据库

(1) Help查看命令提示  
 help  
 db.help()  
 db.test.help()  
 db.test.find().help()

(2) 创建/切换数据库  
 use music

(3) 查询数据库  
 show dbs

(4) 查看当前使用的数据库  
 db/db.getName()

(5) 显示当前DB状态  
 db.stats()

(6) 查看当前DB版本  
 db.version()

(7) 查看当前DB的链接机器地址  
 db.getMongo()

(8) 删除数据库  
 db.dropDatabase()

## 5.可视化工具进行增删改查

Robomongo Robo3T  
 adminMongo

## (1) 创建一个聚集集合

```
db.createCollection("collName", {size: 5242880, capped: true, max: 5000}); 最大存储空间为 5m, 最多 5000 个文档的集合
```

## (2) 得到指定名称的聚集集合

```
db.getCollection("account");
```

## (3) 得到当前db的所有聚集集合

```
db.getCollectionNames();
```

## (4) 显示当前db所有聚集的状态

```
db.printCollectionStats();
```

## (5) 删除

```
db.users.drop();
```

### (1) 添加

```
db.users.save({name: 'zhangsan', age: 25, sex: true});  
db.users.save([{name: 'zhangsan', age: 25, sex: true}, {name: "kerin", age: 100}]);
```

### (2) 修改

```
db.users.update({age: 25}, {$set: {name: 'changeName'}}, false, true);  
相当于: update users set name = 'changeName' where age = 25;  
db.users.update({name: 'Lisi'}, {$inc: {age: 50}}, false, true);  
相当于: update users set age = age + 50 where name = 'Lisi';  
db.users.update({name: 'Lisi'}, {$inc: {age: 50}, $set: {name: 'kerwin'}}, false, true);  
相当于: update users set age = age + 50, name = 'hoho' where name = 'kerwin';
```

### (3) 删除

```
db.users.remove({age: 132}); 删除所有 db.users.remove({})
```

## (1) 查询所有记录

```
db.userInfo.find();  
相当于: select * from userInfo;
```

## (2) 查询某字段去重后数据

```
db.userInfo.distinct("name");  
相当于: select distinct name from userInfo;
```

## (3) 查询age = 22的记录

```
db.userInfo.find({"age": 22});  
相当于: select * from userInfo where age = 22;
```

## (4) 查询age > 22的记录

```
db.userInfo.find({age: {$gt: 22}});  
相当于: select * from userInfo where age > 22;
```

## (5) 查询age < 22的记录

```
db.userInfo.find({age: {$lt: 22}});  
相当于: select * from userInfo where age < 22;
```

## (6) 查询age >= 25的记录

```
db.userInfo.find({age: {$gte: 25}});  
相当于: select * from userInfo where age >= 25;
```

## (7) 查询age <= 25的记录

```
db.userInfo.find({age: {$lte: 25}});  
(8) 查询age >= 23 并且 age <= 26
```

```
db.userInfo.find({age: {$gte: 23, $lte: 26}});
```

## (9) 查询name中包含 mongo的数据

```
db.userInfo.find({name: /mongo/});  
//相当于%  
select * from userInfo where name like '%mongo%';
```

## (10) 查询name中以 mongo开头的

```
db.userInfo.find({name: /^mongo/});  
select * from userInfo where name like 'mongo%';
```

## (11) 查询指定列name、age数据(想显示哪列, 就将字段设置为1, 不想显示就设置成0)

```
db.userInfo.find({}, {name: 1, age: 1});
```

相当于: select name, age from userInfo;

## (12) 查询指定列name、age数据, age > 25

```
db.userInfo.find({age: {$gt: 25}}, {name: 1, age: 1});
```

相当于: select name, age from userInfo where age > 25;

## (13) 按照年龄排序 (写成数组就是多列查询)

升序: db.userInfo.find().sort({age: 1});

降序: db.userInfo.find().sort({age: -1});

## (14) 查询name = zhangsan, age = 22的数据

```
db.userInfo.find({name: 'zhangsan', age: 22});
```

相当于: select \* from userInfo where name = 'zhangsan' and age = 22;

## (15) 查询前5条数据

```
db.userInfo.find().limit(5);
```

相当于: select top 5 \* from userInfo;

The screenshot shows the Robomongo interface with two connections. The left connection, 'company-system', has a collection named 'news' selected. The right connection shows the results of the query 'db.getCollection('news').find()' with two documents returned. The first document is ObjectId('62469603a4aa5b5b894055cb') and the second is ObjectId('6246a08e94cf3bd86e29a58d'). Both documents have 8 fields.

(16)查询10条以后的数据

```
db.userInfo.find().skip(10);  
相当于: select * from userInfo where id not in (  
    select top 10 * from userInfo)
```

## 6.nodejs连接操作数据库

连接数据库

```
const mongoose =  
require("mongoose")  
  
mongoose.connect("mongodb://127.0.  
0.1:27017/company-system")
```

db.userInfo.findOne();

相当于: select top 1 \* from userInfo;

db.userInfo.find().limit(1);

(20)查询某个结果集的记录条数

创建模型

```
const mongoose =  
require("mongoose")  
  
const Schema = mongoose.Schema  
  
const UserType = {  
    username:String,  
    password:String,  
    gender:Number,  
    introduction:String,  
    avatar:String,  
    role:Number  
}  
const UserModel =  
mongoose.model("user",new  
Schema(UserType))  
module.exports = UserModel
```

增加数据

```
UserModel.create({  
    introduction,username,gender,avatar,password,role  
})
```

查询数据

```
UserModel.find({username:"kerwin"},  
["username","role","introduction","password"]).sort({createTime:-1}).skip(10).li  
mit(10)
```

更新数据

```
UserModel.updateOne({  
    _id  
}, {  
    introduction,username,gender,avatar  
})
```

删除数据

```
UserModel.deleteOne({_id})
```

## 四、接口规范与业务分层

### 1. 接口规范

#### RESTful架构

服务器上每一种资源，比如一个文件，一张图片，一部电影，都有对应的url地址，如果我们的客户端需要对服务器上的这个资源进行操作，就需要通过http协议执行相应的动作来操作它，比如进行获取，更新，删除。

简单来说就是**url地址中只包含名词表示资源，使用http动词表示动作进行操作资源**  
举个例子：左边是错误的设计，而右边是正确的

```
GET /blog/getArticles --> GET /blog/Articles    获取所有文章  
GET /blog/addArticles --> POST /blog/Articles   添加一篇文章  
GET /blog/editArticles --> PUT /blog/Articles   修改一篇文章  
GET /rest/api/deleteArticles?id=1 --> DELETE /blog/Articles/1  删除一篇文章
```

#### 使用方式

#### 2. 业务分层

GET http://www.birjemin.com/api/user # 获取列表

POST http://www.birjemin.com/api/user # 创建用户

PUT http://www.birjemin.com/api/user/{id} # 修改用户信息

DELETE http://www.birjemin.com/api/user/{id} # 删除用户信息

#### 过滤信息

用于补充规范一些通用字段

?limit=10 # 指定返回记录的数量

?offset=10 # 指定返回记录的开始位置

?page=2&per\_page=100 # 指定第几页，以及每页的记录数

?sortby=name&order=asc # 指定返回结果按照哪个属性排序，以及排序顺序

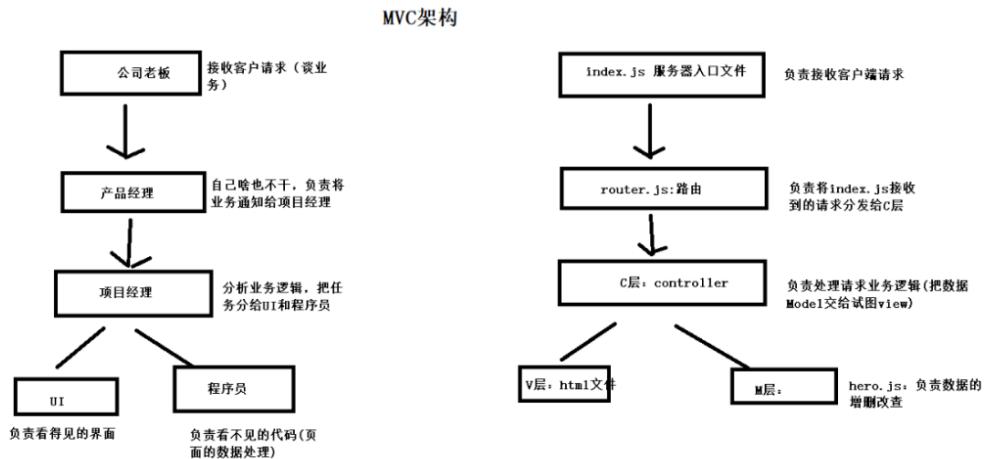
?state=close # 指定筛选条件

router.js: 负责将请求分发给C层

controller.js: C层负责处理业务逻辑 (V与M之间的沟通)

views: V层: 负责展示页面

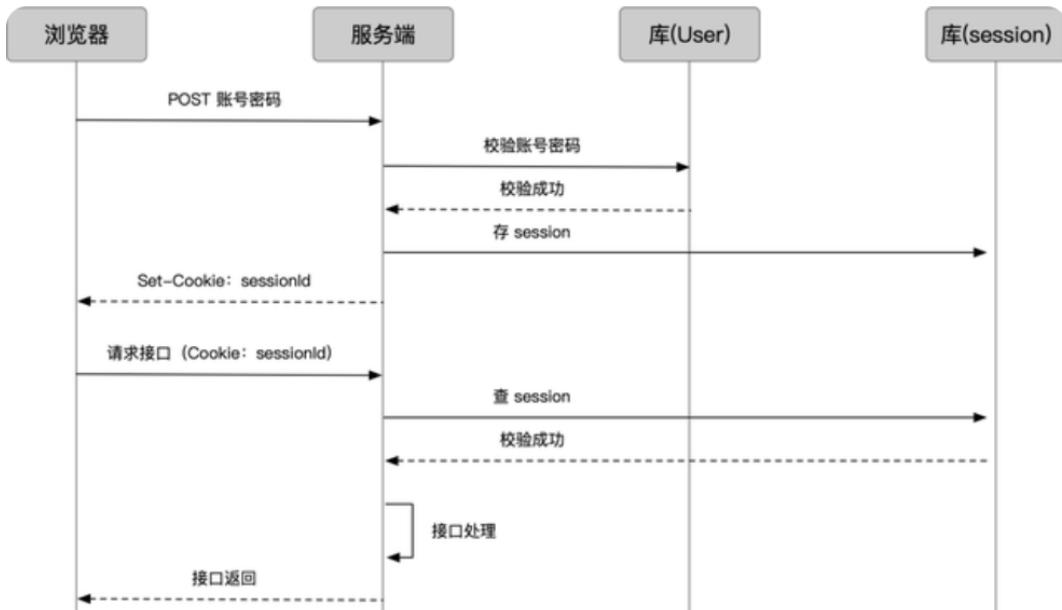
model: M层: 负责处理数据 (增删改查)



## 五、登录鉴权

### 1. Cookie&Session

「HTTP 无状态」我们知道，HTTP 是无状态的。也就是说，HTTP 请求方和响应方间无法维护状态，都是一次性的，它不知道前后的请求都发生了什么。但有的场景下，我们需要维护状态。最典型的，一个用户登陆微博，发布、关注、评论，都应是在登录后的用户状态下的。「标记」那解决办法是什么呢？



```
const express = require("express");
const session = require("express-session");
const MongoStore = require("connect-mongo");
const app = express();

app.use(
  session({
```

```

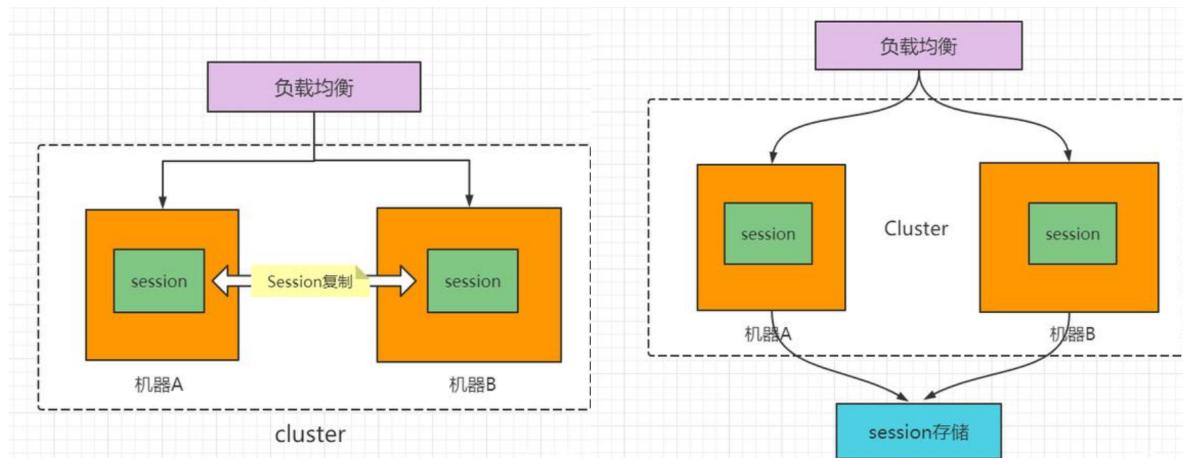
secret: "this is session", // 服务器生成 session 的签名
resave: true,
saveUninitialized: true, // 强制将为初始化的 session 存储
cookie: {
  maxAge: 1000 * 60 * 10, // 过期时间
  secure: false, // 为 true 时候表示只有 https 协议才能访问 cookie
},
rolling: true, // 为 true 表示 超时前刷新, cookie 会重新计时; 为 false 表示在超时前刷新多少次, 都是按照第一次刷新开始计时。
store: MongoStore.create({
  mongoUrl: 'mongodb://127.0.0.1:27017/kerwin_session',
  ttl: 1000 * 60 * 10 // 过期时间
}),
),
);

app.use((req, res, next) =>{
  if(req.url === "/login"){
    next()
    return;
  }
  if(req.session.user){
    req.session.garbage = Date();
    next();
  }else{
    res.redirect("/login")
  }
})

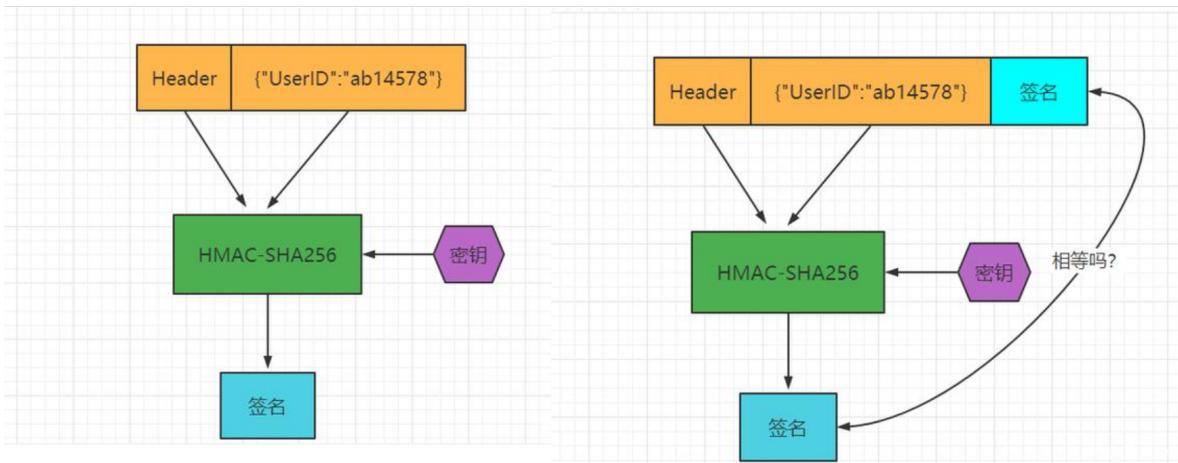
```

## 2. JSON Web Token (JWT)

### (1) 介绍



我为什么要保存这可恶的session呢， 只让每个客户端去保存该多好？



当然，如果一个人的token 被别人偷走了，那我也没办法，我也会认为小偷就是合法用户，这其实和一个人的session id 被别人偷走是一样的。

这样一来，我就不保存session id 了，我只是生成token，然后验证token，我用我的CPU计算时间获取了我的session 存储空间！

解除了session id这个负担，可以说是无事一身轻，我的机器集群现在可以轻松地做水平扩展，用户访问量增大，直接加机器就行。这种无状态的感觉实在是太好了！

缺点：

1. 占带宽，正常情况下要比 session\_id 更大，需要消耗更多流量，挤占更多带宽，假如你的网站每月有 10 万次的浏览器，就意味着要多开销几十兆的流量。听起来并不多，但日积月累也是不小一笔开销。实际上，许多人在 JWT 中存储的信息会更多；
2. 无法在服务端注销，那么久很难解决劫持问题；
3. 性能问题，JWT 的卖点之一就是加密签名，由于这个特性，接收方得以验证 JWT 是否有效且被信任。对于有着严格性能要求的 Web 应用，这并不理想，尤其对于单线程环境。

注意：

CSRF攻击的原因是浏览器会自动带上cookie，而不会带上token；

以CSRF攻击为例：

cookie：用户点击了链接，cookie未失效，导致发起请求后后端以为是用户正常操作，于是进行扣款操作；

token：用户点击链接，由于浏览器不会自动带上token，所以即使发了请求，后端的token验证不会通过，所以不会进行扣款操作；

## (2) 实现

```
//jsonwebtoken 封装
const jsonwebtoken = require("jsonwebtoken")
const secret = "kerwin"
const JWT = {
  generate(value, expires) {
    return jsonwebtoken.sign(value, secret, { expiresIn: expires })
  },
  verify(token) {
    try {
      return jsonwebtoken.verify(token, secret)
    } catch(e) {
      return false
    }
  }
}
```

```
}

module.exports = JWT
```

```
//node中间件校验
app.use((req,res,next)=>{
    // 如果token有效 ,next()
    // 如果token过期了， 返回401错误
    if(req.url === "/login"){
        next()
        return;
    }

    const token = req.headers["authorization"].split(" ")[1]
    if(token){
        var payload = JWT.verify(token)
        // console.log(payload)
        if(payload){
            const newToken = JWT.generate({
                _id:payload._id,
                username:payload.username
            }, "1d")
            res.header("Authorization",newToken)
            next()
        }else{
            res.status(401).send({errCode:"-1",errorInfo:"token过期"})
        }
    }
})
```

```
//生成token
const token = JWT.generate({
    _id: result[0]._id,
    username: result[0].username
}, "1d")

res.header("Authorization", token)
```

```
//前端拦截
/*
 * @作者: kerwin
 * @公众号: 大前端私房菜
 */
import axios from 'axios'
// Add a request interceptor
axios.interceptors.request.use(function (config) {
    const token = localStorage.getItem("token")
    config.headers.Authorization = `Bearer ${token}`

    return config;
}, function (error) {
    return Promise.reject(error);
});
```

```
// Add a response interceptor
axios.interceptors.response.use(function (response) {

    const {authorization} = response.headers
    authorization && localStorage.setItem("token", authorization)
    return response;
}, function (error) {

    const {status} = error.response
    if(status === 401){
        localStorage.removeItem("token")
        window.location.href = "/login"
    }
    return Promise.reject(error);
});
```

## 六、文件上传管理

Multer 是一个 node.js 中间件，用于处理 `multipart/form-data` 类型的表单数据，它主要用于上传文件。

**注意：**Multer 不会处理任何非 `multipart/form-data` 类型的表单数据。

```
npm install --save multer
```

```
//前端分离-前端

const params = new FormData()
params.append('kerwinfile', file.file)
params.append('username', this.username)
const config = {
  headers: {
    "Content-Type": "multipart/form-data"
  }
}
http.post('/api/upload', params, config).then(res => {
  this.imgpath = 'http://localhost:3000' + res.data
})
```

Multer 会添加一个 `body` 对象以及 `file` 或 `files` 对象到 express 的 `request` 对象中。`body` 对象包含表单的文本域信息，`file` 或 `files` 对象包含对象表单上传的文件信息。

```
//前端分离-后端
router.post('/upload', upload.single('kerwinfile'), function(req, res, next) {
  console.log(req.file)
})
```

## 七、APIDOC - API 文档生成工具

apidoc 是一个简单的 RESTful API 文档生成工具，它从代码注释中提取特定格式的内容生成文档。支持诸如 Go、Java、C++、Rust 等大部分开发语言，具体可使用 `apidoc lang` 命令行查看所有的支持列表。

apidoc 拥有以下特点：

1. 跨平台，linux、windows、macOS 等都支持；
2. 支持语言广泛，即使是不支持，也很方便扩展；
3. 支持多个不同语言的多个项目生成一份文档；
4. 输出模板可自定义；
5. 根据文档生成 mock 数据；

```
npm install -g apidoc
```

Add some apidoc comments anywhere in your source code:

```
/**  
 * @api {get} /user/:id Request User information  
 * @apiName GetUser  
 * @apiGroup User  
 *  
 * @apiParam {Number} id User's unique ID.  
 *  
 * @apiSuccess {String} firstname Firstname of the User.  
 * @apiSuccess {String} lastname Lastname of the User.  
 */
```

Now generate the documentation from `src/` into `doc/`.

```
$ apidoc -i src/ -o doc/
```

---

注意：

(1) 在当前文件夹下 apidoc.json

```
{  
  "name": "****接口文档",  
  "version": "1.0.0",  
  "description": "关于****的接口文档描述",  
  "title": "****"  
}
```

(2) 可以利用vscode apidoc snippets 插件创建api

## 八、Koa2



基于 Node.js 平台的下一代 web 开发框架

### 1.简介

koa 是由 Express 原班人马打造的，致力于成为一个更小、更富有表现力、更健壮的 Web 框架。使用 koa 编写 web 应用，通过组合不同的 generator，可以免除重复繁琐的回调函数嵌套，并极大地提升错误处理的效率。koa 不在内核方法中绑定任何中间件，它仅仅提供了一个轻量优雅的函数库，使得编写 Web 应用变得得心应手。

## 2. 快速开始

### 2.1 安装koa2

```
# 初始化package.json
npm init

# 安装koa2
npm install koa
```

### 2.2 hello world 代码

```
const Koa = require('koa')
const app = new Koa()

app.use(async (ctx) => {
  ctx.body = 'hello koa2' //json数据
})

app.listen(3000)
```

**ctx.req**  
Node 的 request 对象.

**ctx.res**

Node 的 response 对象.

绕过 Koa 的 response 处理是 **不被支持的**. 应避免使用以下 node 属性:

- res.statusCode
- res.writeHead()
- res.write()
- res.end()

**ctx.request**

koa 的 Request 对象.

**ctx.response**

koa 的 Response 对象.

以下访问器和 [Request](#) 别名等效: 以下访问器和 [Response](#) 别名等效:

- |   |   |
|---|---|
| <ul style="list-style-type: none"><li>• ctx.header</li><li>• ctx.headers</li><li>• ctx.method</li><li>• ctx.method=</li><li>• ctx.url</li><li>• ctx.url=</li><li>• ctx.originalUrl</li><li>• ctx.origin</li><li>• ctx.href</li><li>• ctx.path</li><li>• ctx.path=</li><li>• ctx.query</li><li>• ctx.query=</li><li>• ctx.querystring</li><li>• ctx.querystring=</li><li>• ctx.host</li><li>• ctx.hostname</li></ul> | <ul style="list-style-type: none"><li>• ctx.body</li><li>• ctx.body=</li><li>• ctx.status</li><li>• ctx.status=</li><li>• ctx.message</li><li>• ctx.message=</li><li>• ctx.length</li><li>• ctx.length=</li><li>• ctx.type</li><li>• ctx.type=</li><li>• ctx.headerSent</li><li>• ctx.redirect()</li><li>• ctx.attachment()</li><li>• ctx.set()</li><li>• ctx.append()</li><li>• ctx.remove()</li><li>• ctx.lastModified=</li><li>• ctx.etag=</li></ul> |
|---|---|

### 2.3 启动demo

```
node index.js
```

## 3. koa vs express

通常都会说 Koa 是洋葱模型，这重点在于中间件的设计。但是按照上面的分析，会发现 Express 也是类似的，不同的是 Express 中间件机制使用了 Callback 实现，这样如果出现异步则可能会使你在执行顺序上感到困惑，因此如果我们想做接口耗时统计、错误处理 Koa 的这种中间件模式处理起来更方便些。最后一点响应机制也很重要，Koa 不是立即响应，是整个中间件处理完成在最外层进行了响应，而 Express 则是立即响应。

### 3.1 更轻量

- koa 不提供内置的中间件;
- koa 不提供路由，而是把路由这个库分离出来了 (koa/router)

### 3.2 Context对象

koa增加了一个Context的对象，作为这次请求的上下文对象（在koa2中作为中间件的第一个参数传入）。同时Context上也挂载了Request和Response两个对象。和Express类似，这两个对象都提供了大量的便捷方法辅助开发，这样的话对于在保存一些公有的参数的话变得更加合情合理

### 3.3 异步流程控制

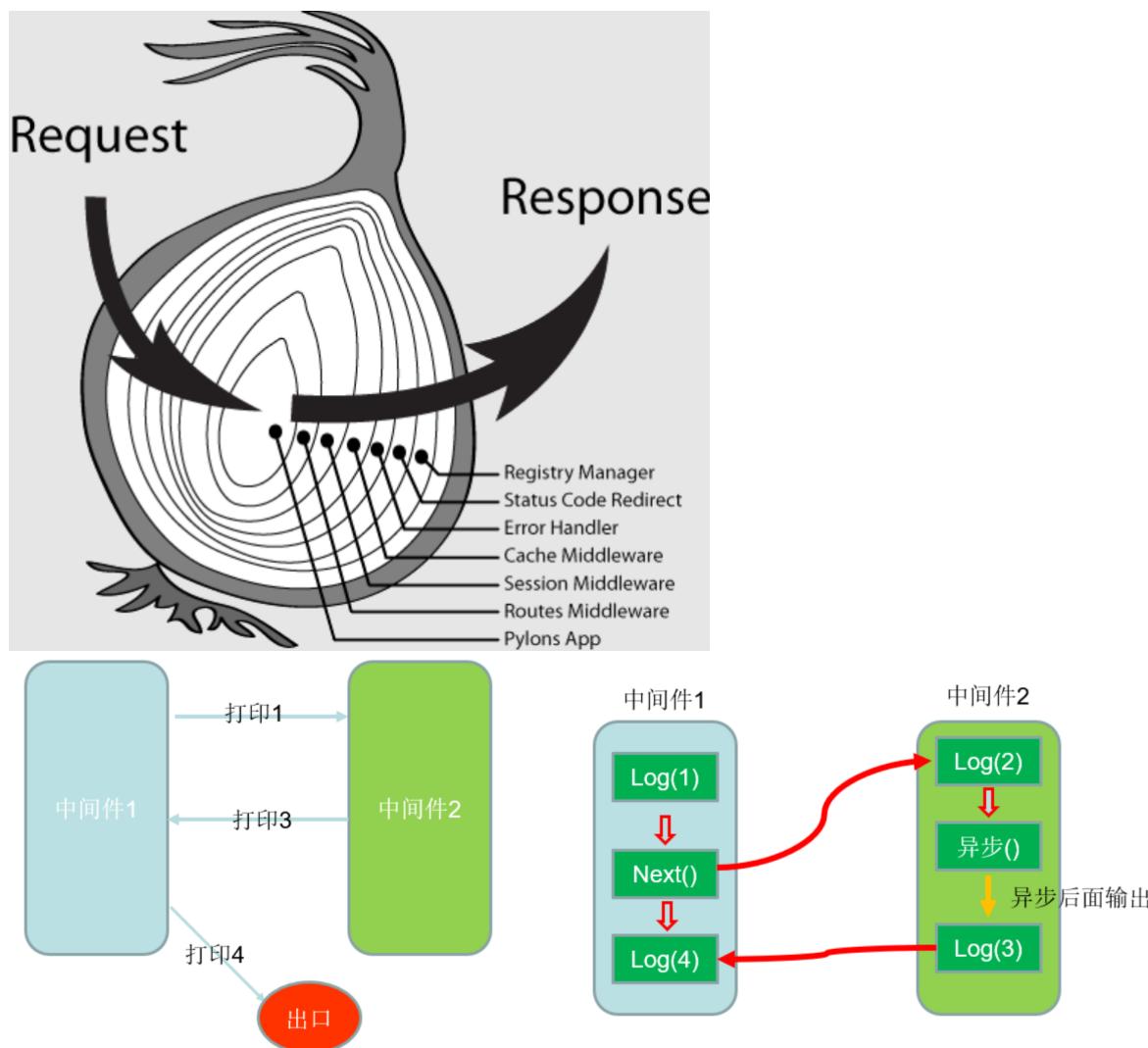
express采用callback来处理异步，koa v1采用generator，koa v2 采用async/await。

generator和async/await使用同步的写法来处理异步，明显好于callback和promise，

### 3.4 中间件模型

express基于connect中间件，线性模型；

koa中间件采用洋葱模型（对于每个中间件，在完成了一些事情后，可以非常优雅的将控制权传递给下一个中间件，并能够等待它完成，当后续的中间件完成处理后，控制权又回到了自己）



```
//同步
var express = require("express")
var app = express()

app.use((req,res,next)=>{
  console.log(1)
  // ...
  next()
})
```

```
next()
console.log(4)
res.send("hello")
})

app.use(()=>{
    console.log(3)
})

app.listen(3000)
//异步
var express = require("express")
var app = express()

app.use(async (req,res,next)=>{
    console.log(1)
    await next()
    console.log(4)
    res.send("hello")
})

app.use(async ()=>{
    console.log(2)
    await delay(1)
    console.log(3)
})

function delay(time){
    return new Promise((resolve,reject)=>{
        setTimeout(resolve,1000)
    })
}
```

```
//同步
var koa = require("koa")
var app = new koa()

app.use((ctx,next)=>{
    console.log(1)
    next()
    console.log(4)
    ctx.body="hello"
})

app.use(()=>{
    console.log(3)
})

app.listen(3000)

//异步
var koa = require("koa")
var app = new koa()

app.use(async (ctx,next)=>{
    console.log(1)
    await next()
    console.log(4)
    ctx.body="hello"
})
```

```

app.use(async ()=>{
  console.log(2)
  await delay(1)
  console.log(3)
})

function delay(time){
  return new Promise((resolve,reject)=>{
    setTimeout(resolve,1000)
  })
}

app.listen(3000)

```

## 4. 路由

### 4.1 基本用法

```

var Koa = require("koa")
var Router = require("koa-router")

var app = new Koa()
var router = new Router()

router.post("/list", (ctx)=>{
  ctx.body=["111", "222", "333"]
})
app.use(router.routes()).use(router.allowedMethods())
app.listen(3000)

```

### 4.2 router.allowedMethods作用

The screenshot shows a browser's developer tools Network tab. A request to `/list` is selected. In the Headers section, the `Allow` header is highlighted with a red box and contains the value `POST`. The Response section shows the status code as `405 Method Not Allowed`.

Name	Headers	Preview	Response	Initiator	Timing
list	<b>General</b> Request URL: http://localhost:3000/list Request Method: GET Status Code: 405 Method Not Allowed Remote Address: [::1]:3000 Referrer Policy: strict-origin-when-cross-origin				
	<b>Response Headers</b> Allow: POST Connection: keep-alive				

### 4.3 请求方式

Koa-router 请求方式: `get`、`put`、`post`、`patch`、`delete`、`del`，而使用方法就是 `router.方法()`，比如 `router.get()` 和 `router.post()`。而 `router.all()` 会匹配所有的请求方法。

```

var Koa = require("koa")
var Router = require("koa-router")

var app = new Koa()
var router = new Router()

```

```

router.get("/user", (ctx)=>{
  ctx.body=["aaa", "bbb", "ccc"]
})
.put("/user/:id", (ctx)=>{
  ctx.body={ok:1,info:"user update"}
})
.post("/user", (ctx)=>{
  ctx.body={ok:1,info:"user post"}
})
.del("/user/:id", (ctx)=>{
  ctx.body={ok:1,info:"user del"}
})

app.use(router.routes()).use(router.allowedMethods())
app.listen(3000)

```

#### 4.4 拆分路由

list.js

```

var Router = require("koa-router")
var router = new Router()
router.get("/", (ctx)=>{
  ctx.body=["111", "222", "333"]
})
.put("/:id", (ctx)=>{
  ctx.body={ok:1,info:"list update"}
})
.post("/", (ctx)=>{
  ctx.body={ok:1,info:"list post"}
})
.del("/:id", (ctx)=>{
  ctx.body={ok:1,info:"list del"}
})
module.exports = router

```

index.js

```

var Router = require("koa-router")
var router = new Router()
var user = require("./user")
var list = require("./list")
router.use('/user', user.routes(), user.allowedMethods())
router.use('/list', list.routes(), list.allowedMethods())

module.exports = router

```

entry入口

```
var Koa = require("koa")
var router = require("./router/index")

var app = new Koa()
app.use(router.routes()).use(router.allowedMethods())
app.listen(3000)
```

## 4.5 路由前缀

```
router.prefix('/api')
```

## 4.6 路由重定向

```
router.get("/home", (ctx)=>{
    ctx.body="home页面"
})
//写法1
router.redirect('/', '/home');
//写法2
router.get("/", (ctx)=>{
    ctx.redirect("/home")
})
```

## 5. 静态资源

```
const Koa = require('koa')
const path = require('path')
const static = require('koa-static')

const app = new Koa()

app.use(static(
    path.join(__dirname, "public")
))

app.use(async (ctx) => {
    ctx.body = 'hello world'
})

app.listen(3000, () => {
    console.log('[demo] static-use-middleware is starting at port 3000')
})
```

## 6. 获取请求参数

### 6.1 get参数

在koa中，获取GET请求数据源头是koa中request对象中的query方法或querystring方法，query返回是格式化好的参数对象，querystring返回的是请求字符串，由于ctx对request的API有直接引用的方式，所以获取GET请求数据有两个途径。

- 是从上下文中直接获取 请求对象ctx.query，返回如 { a:1, b:2 } 请求字符串 ctx.querystring，返回如 a=1&b=2
- 是从上下文的request对象中获取 请求对象ctx.request.query，返回如 { a:1, b:2 } 请求字符串 ctx.request.querystring，返回如 a=1&b=2

## 6.2 post参数

对于POST请求的处理，koa-bodyparser中间件可以把koa2上下文的formData数据解析到ctx.request.body中

```
const bodyParser = require('koa-bodyparser')

// 使用ctx.body解析中间件
app.use(bodyParser())
```

## 7. ejs模板

### 7.1 安装模块

```
# 安装koa模板使用中间件
npm install --save koa-views

# 安装ejs模板引擎
npm install --save ejs
```

### 7.2 使用模板引擎

#### 文件目录

```
├── package.json
├── index.js
└── view
    └── index.ejs
```

#### ./index.js文件

```
const Koa = require('koa')
const views = require('koa-views')
const path = require('path')
const app = new Koa()

// 加载模板引擎
app.use(views(path.join(__dirname, './view'), {
  extension: 'ejs'
}))

app.use(async (ctx) => {
  let title = 'hello koa2'
```

```
    await ctx.render('index', {
      title,
    })
  })

app.listen(3000)
```

## ./view/index.ejs 模板

```
<!DOCTYPE html>
<html>
<head>
  <title><%= title %></title>
</head>
<body>
  <h1><%= title %></h1>
  <p>EJS Welcome to <%= title %></p>
</body>
</html>
```

## 8. cookie&session

### 8.1 cookie

koa提供了从上下文直接读取、写入cookie的方法

- ctx.cookies.get(name, [options]) 读取上下文请求中的cookie
- ctx.cookies.set(name, value, [options]) 在上下文中写入cookie

### 8.2 session

- koa-session-minimal 适用于koa2 的session中间件，提供存储介质的读写接口。

```
const session = require('koa-session-minimal')
app.use(session({
  key: 'SESSION_ID',
  cookie: {
    maxAge: 1000 * 60
  }
}))
```

```
app.use(async (ctx, next) => {
  //排除login相关的路由和接口
  if (ctx.url.includes("login")) {
    await next()
    return
  }

  if (ctx.session.user) {
    //重新设置以下session
    ctx.session.mydate = Date.now()
    await next()
  } else {
```

```
        ctx.redirect("/login")
    }
})
```

## 9. JWT

```
app.use(async(ctx, next) => {
    //排除login相关的路由和接口
    if (ctx.url.includes("login")) {
        await next()
        return
    }
    const token = ctx.headers["authorization"]?.split(" ")[1]
    // console.log(req.headers["authorization"])
    if(token){
        const payload= JWT.verify(token)
        if(payload){
            //重新计算token过期时间
            const newToken = JWT.generate({
                _id:payload._id,
                username:payload.username
            }, "10s")

            ctx.set("Authorization",newToken)
            await next()
        }else{
            ctx.status = 401
            ctx.body = {errCode:-1,errInfo:"token过期"}
        }
    }else{
        await next()
    }
})
```

## 10.上传文件

<https://www.npmjs.com/package/@koa/multer>

```
npm install --save @koa/multer multer
```

```
const multer = require('@koa/multer');
const upload = multer({ dest: 'public/uploads/' })

router.post("/",upload.single('avatar'),
(ctx,next)=>{
    console.log(ctx.request.body,ctx.file)
    ctx.body={
        ok:1,
        info:"add user success"
    }
})
```

## 11.操作MongoDB

```

const mongoose = require("mongoose")

mongoose.connect("mongodb://127.0.0.1:27017/kerwin_project")
//插入集合和数据，数据库kerwin_project会自动创建

```

```

const mongoose = require("mongoose")
const Schema = mongoose.Schema
const UserType = {
    username:String,
    password:String,
    age:Number,
    avatar:String
}

const UserModel = mongoose.model("user", new Schema(UserType))
// 模型user 将会对应 users 集合，
module.exports = UserModel

```

## 九、MySQL

### 1.介绍

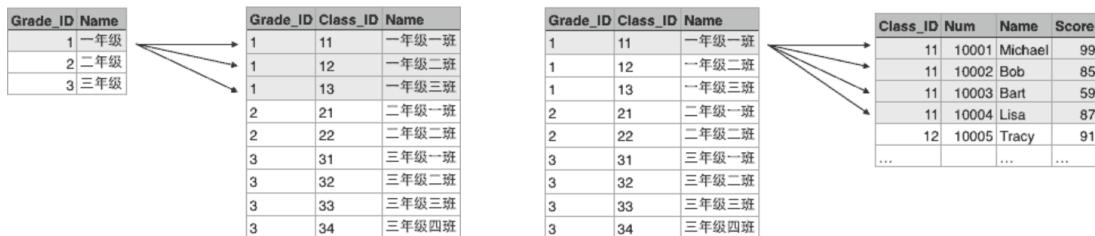
付费的商用数据库：

- Oracle，典型的高富帅；
- SQL Server，微软自家产品，Windows定制专款；
- DB2，IBM的产品，听起来挺高端；
- Sybase，曾经跟微软是好基友，后来关系破裂，现在家境惨淡。

这些数据库都是不开源而且付费的，最大的好处是花了钱出了问题可以找厂家解决，不过在Web的世界里，常常需要部署成千上万的数据库服务器，当然不能把大把大把的银子扔给厂家，所以，无论是Google、Facebook，还是国内的BAT，无一例外都选择了免费的开源数据库：

- MySQL，大家都在用，一般错不了；
- PostgreSQL，学术气息有点重，其实挺不错，但知名度没有MySQL高；
- sqlite，嵌入式数据库，适合桌面和移动应用。

作为一个JavaScript全栈工程师，选择哪个免费数据库呢？当然是MySQL。因为MySQL普及率最高，出了错，可以很容易找到解决方法。而且，围绕MySQL有一大堆监控和运维的工具，安装和使用很方便。



### 2.与非关系数据库区别

关系型和非关系型数据库的主要差异是数据存储的方式。关系型数据天然就是表格式的，因此存储在数据表的行和列中。数据表可以彼此关联协作存储，也很容易提取数据。

与其相反，非关系型数据不适合存储在数据表的行和列中，而是大块组合在一起。非关系型数据通常存储在数据集中，就像文档、键值对或者图结构。你的数据及其特性是选择数据存储和提取方式的首要影响因素。

### 关系型数据库最典型的数据结构是表，由二维表及其之间的联系所组成的一个数据组织

优点：

- 1、易于维护：都是使用表结构，格式一致；
- 2、使用方便：SQL语言通用，可用于复杂查询；
- 3、复杂操作：支持SQL，可用于一个表以及多个表之间非常复杂的查询。

缺点：

- 1、读写性能比较差，尤其是海量数据的高效率读写；
- 2、固定的表结构，灵活度稍欠；
- 3、高并发读写需求，传统关系型数据库来说，硬盘I/O是一个很大的瓶颈。

非关系型数据库严格上不是一种数据库，应该是一种数据结构化存储方法的集合，可以是文档或者键值对等。

优点：

- 1、格式灵活：存储数据的格式可以是key,value形式、文档形式、图片形式等等，文档形式、图片形式等等，使用灵活，应用场景广泛，而关系型数据库则只支持基础类型。
- 2、速度快：nosql可以使用硬盘或者随机存储器作为载体，而关系型数据库只能使用硬盘；
- 3、高扩展性；
- 4、成本低：nosql数据库部署简单，基本都是开源软件。

缺点：

- 1、不提供sql支持；
- 2、无事务处理；
- 3、数据结构相对复杂，复杂查询方面稍欠。

## 3.sql语句

#	名字	类型	排序规则	属性	空	默认	注释	额外	操作
□	1  id	int(11)			否	无	AUTO_INCREMENT	 修改  删除 ▾ 更多	
□	2 name	varchar(100)	utf8_general_ci		否	无		 修改  删除 ▾ 更多	
□	3 score	int(11)			否	无		 修改  删除 ▾ 更多	
□	4 gender	int(11)			否	无		 修改  删除 ▾ 更多	
□	5 create_time	timestamp			否	CURRENT_TIMESTAMP		 修改  删除 ▾ 更多	

插入：

```
INSERT INTO `students`(`id`, `name`, `score`, `gender`) VALUES  
(null, 'kerwin', 100, 1)  
//可以不设置id,create_time
```

更新：

```
UPDATE `students` SET `name`='tiechui', `score`=20, `gender`=0 WHERE id=2;
```

删除：

```
DELETE FROM `students` WHERE id=2;
```

查询：

查所有的数据所有的字段

```
SELECT * FROM `students` WHERE 1;
```

查所有的数据某个字段

```
SELECT `id`, `name`, `score`, `gender` FROM `students` WHERE 1;
```

条件查询

```
SELECT * FROM `students` WHERE score>=80;
```

```
SELECT * FROM `students` where score>=80 AND gender=1
```

模糊查询

```
SELECT * FROM `students` where name like '%k%'
```

排序

```
SELECT id, name, gender, score FROM students ORDER BY score;
```

```
SELECT id, name, gender, score FROM students ORDER BY score DESC;
```

分页查询

```
SELECT id, name, gender, score FROM students LIMIT 50 OFFSET 0
```

记录条数

```
SELECT COUNT(*) FROM students;
```

```
SELECT COUNT(*) kerwinnum FROM students;
```

多表查询

`SELECT * FROM students, classes;` (这种多表查询又称笛卡尔查询，使用笛卡尔查询时要非常小心，由于结果集是目标表的行数乘积，对两个各自有100行记录的表进行笛卡尔查询将返回1万条记录，对两个各自有1万行记录的表进行笛卡尔查询将返回1亿条记录)

`SELECT`

```
    students.id sid,  
    students.name,  
    students.gender,  
    students.score,  
    classes.id cid,  
    classes.name cname
```

`FROM students, classes;` (要使用表名.列名这样的方式来引用列和设置别名，这样就避免了结果集的列名重复问题。)

`SELECT`

```
    s.id sid,  
    s.name,  
    s.gender,  
    s.score,  
    c.id cid,  
    c.name cname
```

`FROM students s, classes c;` (SQL还允许给表设置一个别名)

联表查询

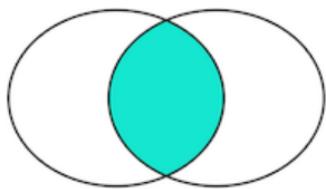
```
SELECT s.id, s.name, s.class_id, c.name class_name, s.gender, s.score
```

```
FROM students s
```

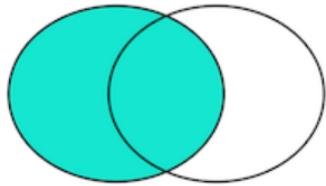
```
INNER JOIN classes c
```

`ON s.class_id = c.id;` (连接查询对多个表进行JOIN运算，简单地说，就是先确定一个主表作为结果集，然后，把其他表的行有选择性地“连接”在主表结果集上。)

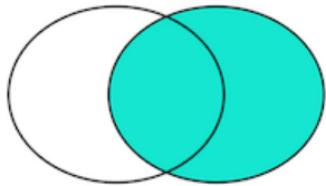
我们把tableA看作左表，把tableB看成右表，那么INNER JOIN是选出两张表都存在的记录：



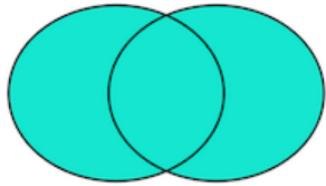
LEFT OUTER JOIN是选出左表存在的记录：



RIGHT OUTER JOIN是选出右表存在的记录：



FULL OUTER JOIN则是选出左右表都存在的记录：



注意：

1. InnoDB 支持事务，MyISAM 不支持事务。这是 MySQL 将默认存储引擎从 MyISAM 变成 InnoDB 的重要原因之一；
2. InnoDB 支持外键，而 MyISAM 不支持。对一个包含外键的 InnoDB 表转为 MYISAM 会失败；

## 外键约束

CASCADE

在父表上update/delete记录时，同步update/delete掉子表的匹配记录

SET NULL

在父表上update/delete记录时，将子表上匹配记录的列设为null (要注意子表的外键列不能为not null)

NO ACTION

如果子表中有匹配的记录，则不允许对父表对应候选键进行update/delete操作

RESTRICT

同no action, 都是立即检查外键约束

## 4.nodejs 操作数据库

```
const express = require('express')
```

```

const app = express()
const mysql2 = require('mysql2')
const port = 9000

app.get('/',async (req, res) => {
  const config = getDBConfig()
  const promisePool = mysql2.createPool(config).promise();
  // console.log(promisePool)
  let user = await promisePool.query('select * from students');

  console.log(user)
  if (user[0].length) {
    //存在用户
    res.send(user[0])
  } else {
    //不存在
    res.send( {
      code: -2,
      msg: 'user not exsit',
    })
  }
})

app.listen(port, () => {
  console.log(`Example app listening at http://localhost:${port}`)
})

function getDBConfig() {
  return {
    host: '127.0.0.1',
    user: 'root',
    port: 3306,
    password: '',
    database: 'kerwin_test',
    connectionLimit: 1 //创建一个连接池
  }
}

```

查询:

```
promisePool.query('select * from users');
```

插入:

```
promisePool.query('INSERT INTO `users`(`id`, `name`, `age`, `password`) VALUES (?, ?, ?, ?)', [null, "kerwin", 100, "123456"]);
```

更新:

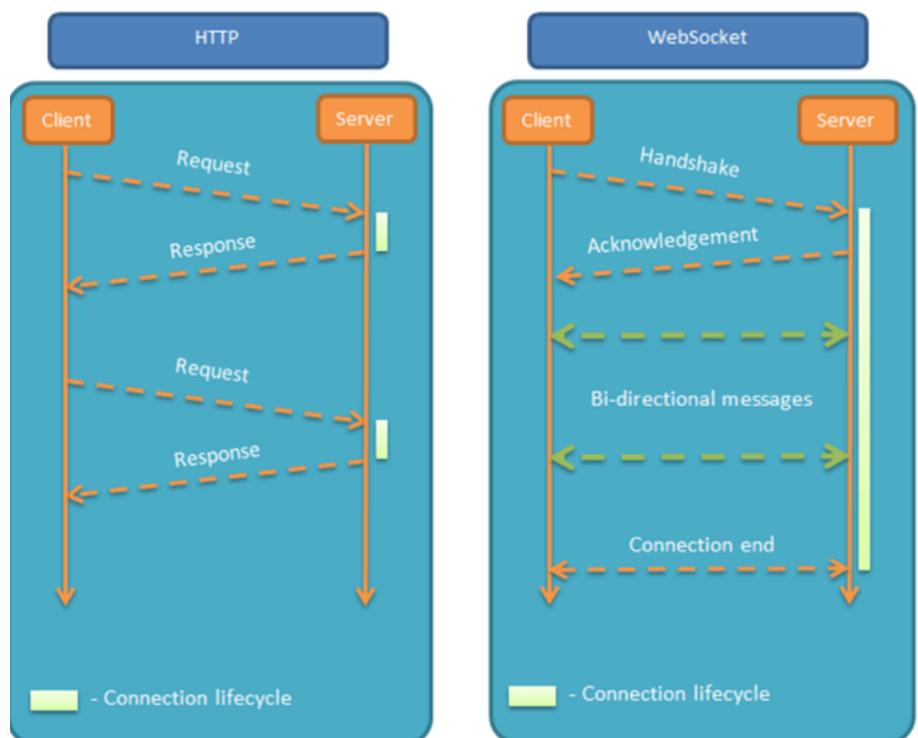
```
promisePool.query(`UPDATE users SET name = ? ,age=? WHERE id = ?`, ["xiaoming2", 20, 1])
```

删除:

```
promisePool.query(`delete from users where id=?`, [1])
```

## 十、Socket编程

## 1.websocket介绍



### 应用场景：

- 弹幕
- 媒体聊天
- 协同编辑
- 基于位置的应用
- 体育实况更新
- 股票基金报价实时更新

WebSocket并不是全新的协议，而是利用了HTTP协议来建立连接。我们来看看WebSocket连接是如何创建的。

首先，WebSocket连接必须由浏览器发起，因为请求协议是一个标准的HTTP请求，格式如下：

```
GET ws://localhost:3000/ws/chat HTTP/1.1
Host: localhost
Upgrade: websocket
Connection: Upgrade
Origin: http://localhost:3000
Sec-WebSocket-Key: client-random-string
Sec-WebSocket-Version: 13
```

该请求和普通的HTTP请求有几点不同：

1. GET请求的地址不是类似 /path/，而是以 ws:// 开头的地址；
2. 请求头 `Upgrade: websocket` 和 `Connection: Upgrade` 表示这个连接将要被转换为WebSocket连接；
3. `Sec-WebSocket-Key` 是用于标识这个连接，并非用于加密数据；
4. `Sec-WebSocket-Version` 指定了WebSocket的协议版本。

随后，服务器如果接受该请求，就会返回如下响应：

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: server-random-string
```

该响应代码 101 表示本次连接的HTTP协议即将被更改，更改后的协议就是 `upgrade: websocket` 指定的WebSocket协议。

版本号和子协议规定了双方能理解的数据格式，以及是否支持压缩等等。如果仅使用WebSocket的 API，就不需要关心这些。

现在，一个WebSocket连接就建立成功，浏览器和服务器就可以随时主动发送消息给对方。消息有两种，一种是文本，一种是二进制数据。通常，我们可以发送JSON格式的文本，这样，在浏览器处理起来就十分容易。

为什么WebSocket连接可以实现全双工通信而HTTP连接不行呢？实际上HTTP协议是建立在TCP协议之上的，TCP协议本身就实现了全双工通信，但是HTTP协议的请求 - 应答机制限制了全双工通信。

WebSocket连接建立以后，其实只是简单规定了一下：接下来，咱们通信就不使用HTTP协议了，直接互相发数据吧。

安全的WebSocket连接机制和HTTPS类似。首先，浏览器用 `wss://xxx` 创建WebSocket连接时，会先通过HTTPS创建安全的连接，然后，该HTTPS连接升级为WebSocket连接，底层通信走的仍然是安全的SSL/TLS协议。

## 浏览器支持

很显然，要支持WebSocket通信，浏览器得支持这个协议，这样才能发出 `ws://xxx` 的请求。目前，支持WebSocket的主流浏览器如下：

- Chrome
- Firefox
- IE >= 10
- Safari >= 6
- Android >= 4.4
- iOS >= 8

## 服务器支持

由于WebSocket是一个协议，服务器具体怎么实现，取决于所用编程语言和框架本身。Node.js本身支持的协议包括TCP协议和HTTP协议，要支持WebSocket协议，需要对Node.js提供的HTTPServer做额外的开发。已经有若干基于Node.js的稳定可靠的WebSocket实现，我们直接用npm安装使用即可。

## 2.ws模块

服务器：

```
const WebSocket = require("ws")
WebSocketServer = WebSocket.WebSocketServer
const wss = new WebSocketServer({ port: 8080 });
wss.on('connection', function connection(ws) {
    ws.on('message', function message(data, isBinary) {
        wss.clients.forEach(function each(client) {
            if (client !== ws && client.readyState === WebSocket.OPEN) {
                client.send(data, { binary: isBinary });
            }
        });
    });
});
```

```
    ws.send('欢迎加入聊天室');
});
```

客户端：

```
var ws = new WebSocket("ws://localhost:8080")
ws.onopen = ()=>{
    console.log("open")
}
ws.onmessage = (evt)=>{
    console.log(evt.data)
}
```

授权验证：

```
//前端
var ws = new WebSocket(`ws://localhost:8080?
token=${localStorage.getItem("token")}`)
ws.onopen = () => {
    console.log("open")
    ws.send(JSON.stringify({
        type: WebSocketType.GroupList
    }))
}
ws.onmessage = (evt) => {
    console.log(evt.data)
}
//后端
const WebSocket = require("ws");
const JWT = require('../util/JWT');
WebSocketServer = WebSocket.WebSocketServer
const wss = new WebSocketServer({ port: 8080 });
wss.on('connection', function connection(ws, req) {
    const myURL = new URL(req.url, 'http://127.0.0.1:3000');
    const payload = JWT.verify(myURL.searchParams.get("token"))
    if (payload) {
        ws.user = payload
        ws.send(createMessage(WebSocketType.GroupChat, ws.user, "欢迎来到聊天室"))

        sendBroadList() //发送好友列表
    } else {
        ws.send(createMessage(WebSocketType.Error, null, "token过期"))
    }
    // console.log(3333,url)
    ws.on('message', function message(data, isBinary) {
        const messageObj = JSON.parse(data)
        switch (messageObj.type) {
            case WebSocketType.GroupList:
                ws.send(createMessage(WebSocketType.GroupList, ws.user,
JSON.stringify(Array.from(wss.clients).map(item => item.user))))
                break;
            case WebSocketType.GroupChat:
                wss.clients.forEach(function each(client) {
                    if (client !== ws && client.readyState === WebSocket.OPEN) {
```

```

        client.send(createMessage(WebSocketType.GroupChat, ws.user,
messageObj.data));
    }
});
break;
case WebSocketType.Singlechat:
    wss.clients.forEach(function each(client) {
        if (client.user.username === messageObj.to && client.readyState ===
WebSocket.OPEN) {
            client.send(createMessage(WebSocketType.Singlechat, ws.user,
messageObj.data));
        }
    });
break;
}

ws.on("close",function(){
//删除当前用户
wss.clients.delete(ws.user)
sendBroadList() //发送好用列表
})
});

const WebSocketType = {
    Error: 0, //错误
    GroupList: 1,//群列表
    GroupChat: 2,//群聊
    SingleChat: 3//私聊
}
function createMessage(type, user, data) {
    return JSON.stringify({
        type: type,
        user: user,
        data: data
    });
}

function sendBroadList(){
    wss.clients.forEach(function each(client) {
        if (client.readyState === WebSocket.OPEN) {
            client.send(createMessage(WebSocketType.GroupList, client.user,
JSON.stringify(Array.from(wss.clients).map(item => item.user))))
        }
    });
}
}

```

### 3.socket.io模块

服务端：

```

const io = require('socket.io')(server);
io.on('connection', (socket) => {

    const payload = JWT.verify(socket.handshake.query.token)

```

```

if (payload) {
    socket.user = payload
    socket.emit(webSocketType.GroupChat, createMessage(socket.user, "欢迎来到聊天室"))
    sendBroadList() //发送好友列表
} else {
    socket.emit(webSocketType.Error, createMessage(null, "token过期"))
}

socket.on(webSocketType.GroupList, () => {
    socket.emit(webSocketType.GroupList, createMessage(null,
Array.from(io.sockets.sockets).map(item => item[1].user).filter(item=>item)));
})

socket.on(webSocketType.Groupchat, (messageObj) => {
    socket.broadcast.emit(webSocketType.GroupChat, createMessage(socket.user,
messageObj.data));
})

socket.on(webSocketType.Singlechat, (messageObj) => {
    Array.from(io.sockets.sockets).forEach(function (socket) {
        if (socket[1].user.username === messageObj.to) {
            socket[1].emit(webSocketType.SingleChat, createMessage(socket[1].user,
messageObj.data));
        }
    })
})

socket.on('disconnect', reason => {

    sendBroadList() //发送好友列表
});

function sendBroadList() {
    io.sockets.emit(webSocketType.GroupList, createMessage(null,
Array.from(io.sockets.sockets).map(item => item[1].user).filter(item=>item)))
}
//最后filter，是因为 有可能存在null的值

```

客户端：

```

const WebSocketType = {
    Error: 0, //错误
    GroupList: 1, //群列表
    GroupChat: 2, //群聊
    SingleChat: 3 //私聊
}

const socket = io(`ws://localhost:3000?token=${localStorage.getItem("token")}`);
socket.on("connect", ()=>{
    socket.emit(webSocketType.GroupList)
})

```

```

socket.on(webSocketType.GroupList, (messageObj) => {
    select.innerHTML = ""
    select.innerHTML = `<option value="all">all</option>` +
    messageObj.data.map(item =>
        `<option value="${item.username}">${item.username}</option>`).join("")
})

socket.on(webSocketType.GroupChat, (msg) => {
    console.log(msg)
})

socket.on(webSocketType.SingleChat, (msg) => {
    console.log(msg)
})

socket.on(webSocketType.Error, (msg) => {
    localStorage.removeItem("token")
    location.href = "/login"
})

send.onclick = () => {
    if (select.value === "all") {
        socket.emit(webSocketType.GroupChat, {
            data: text.value
        })
    } else {
        socket.emit(webSocketType.SingleChat, {
            data: text.value,
            to: select.value
        })
    }
}
}

```

## 十一、mocha

单元测试是用来对一个模块、一个函数或者一个类来进行正确性检验的测试工作。

比如对函数abs()，我们可以编写出以下几个测试用例：

输入正数，比如1、1.2、0.99，期待返回值与输入相同；

输入负数，比如-1、-1.2、-0.99，期待返回值与输入相反；

输入0，期待返回0；

输入非数值类型，比如null、[]、{}，期待抛出Error。

把上面的测试用例放到一个测试模块里，就是一个完整的单元测试。

如果单元测试通过，说明我们测试的这个函数能够正常工作。如果单元测试不通过，要么函数有bug，要么测试条件输入不正确，总之，需要修复使单元测试能够通过。

单元测试通过后有什么意义呢？如果我们对abs()函数代码做了修改，只需要再跑一遍单元测试，如果通过，说明我们的修改不会对abs()函数原有的行为造成影响，如果测试不通过，说明我们的修改与原有行为不一致，要么修改代码，要么修改测试。

这种以测试为驱动的开发模式最大的好处就是确保一个程序模块的行为符合我们设计的测试用例。在未来修改的时候，可以极大程度地保证该模块行为仍然是正确的。

mocha是JavaScript的一种单元测试框架，既可以在浏览器环境下运行，也可以在Node.js环境下运行。

使用mocha，我们就只需要专注于编写单元测试本身，然后，让mocha去自动运行所有的测试，并给出测试结果。

mocha的特点主要有：

1. 既可以测试简单的JavaScript函数，又可以测试异步代码，因为异步是JavaScript的特性之一；
2. 可以自动运行所有测试，也可以只运行特定的测试；
3. 可以支持before、after、beforeEach和afterEach来编写初始化代码。

## 1. 编写测试

```
const assert = require('assert');
const sum = require('../test');
describe('#hello.js', () => {

  describe('#sum()', () => {
    it('sum() should return 0', () => {
      assert.strictEqual(sum(), 0);
    });

    it('sum(1) should return 1', () => {
      assert.strictEqual(sum(1), 1);
    });

    it('sum(1, 2) should return 3', () => {
      assert.strictEqual(sum(1, 2), 3);
    });

    it('sum(1, 2, 3) should return 6', () => {
      assert.strictEqual(sum(1, 2, 3), 6);
    });
  });
});
```

## 2. chai断言库

### 断言

mocha允许你使用任意你喜欢的断言库，在上面的例子中，我们使用了Node.js的内置的断言模块作为断言。如果能够抛出一个错误，它就能够运行。这意味着你能使用下面的这些仓库，比如：

[should.js - BDD风格贯穿始终](#)

[chai - expect\(\), assert\(\)和should风格的断言](#)

[expect.js - expect\(\)样式断言](#)

[better-assert - C风格的自文档化的assert\(\)](#)

[unexpected - “可扩展的BDD断言工具”](#)

```
var chai = require('chai')
var assert = chai.assert;

describe('assert Demo', function () {
  it('use assert lib', function () {
    var value = "hello";
    assert.typeOf(value, 'string')
    assert.equal(value, 'hello')
    assert.lengthOf(value, 5)
  })
})
```

```
var chai = require('chai');
chai.should();

describe('should Demo', function(){
  it('use should lib', function () {
    var value = 'hello'

    value.should.exist.and.equal('hello').and.have.length(5).and.be.a('string')
    // value.should.be.a('string')
    // value.should.equal('hello')
    // value.should.not.equal('hello2')
    // value.should.have.length(5);
  })
});
```

```
var chai = require('chai');
var expect = chai.expect;

describe('expect Demo', function() {
  it('use expect lib', function () {
    var value = 'hello'
    var number = 3

    expect(number).to.be.at.most(5)
    expect(number).to.be.at.least(3)
    expect(number).to.be.within(1, 4)

    expect(value).to.exist
    expect(value).to.be.a('string')
    expect(value).to.equal('hello')
    expect(value).to.not.equal('您好')
    expect(value).to.have.length(5)
  })
});
```

### 3.异步测试

```
var fs = require("fs").promises
var chai = require('chai');
var expect = chai.expect;
it('test async function',async function () {
    const data =await fs.readFile('./1.txt',"utf8");
    expect(data).to.equal('hello')
});
```

## 4.http测试

```
const request = require('supertest')
const app = require('../app');

describe('#test koa app', () => {
  let server = app.listen(3000);
  describe('#test server', () => {
    it('#test GET /', async () => {
      await request(server)
        .get('/')
        .expect('Content-Type', /text\/html/)
        .expect(200, '<h1>hello world</h1>');
    });

    after(function () {
      server.close()
    });
  });
});
```

## 5.钩子函数

```
describe('#hello.js', () => {
  describe('#sum()', () => {
    before(function () {
      console.log('before:');
    });

    after(function () {
      console.log('after.');
    });

    beforeEach(function () {
      console.log('  beforeEach:');
    });

    afterEach(function () {
      console.log('  afterEach:');
    });
  });
});
```

