

常用第三方的模块

request

可用于发起 http 或 https 请求，可理解成服务端的 ajax 请求。可用于简单的服务器代理，用法和 ajax 类似。

- 安装: `npm install request --save`
- GET 请求

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

request.get('https://cnodejs.org/api/v1/topics?page=1&limit=10', (error, response, body) => {
  // error: 错误信息, 默认null
  // response: 相应对象
  // body: 请求响应内容
  console.log(body)
});
//or
request('https://cnodejs.org/api/v1/topics?page=1&limit=10', (error, response, body) => {
  console.log(body)
});
```

- 参数设置

```
const request = require('request');
request.get({
  url: url,
  method: 'get',
  timeout: 10000,
  headers: {},
  proxy: {} ,
  agentOptions: {},
  params: {}
}, function(err, res, body) {
```

```
});
```

- POST 请求

request支持application/x-www-form-urlencoded和multipart/form-data实现表单上传。

- application/x-www-form-urlencoded (URL-Encoded Forms)

```
request.post('http://service.com/upload', {form:{key:'value'}})
// or
request.post('http://service.com/upload').form({key:'value'})
// or
request.post({url:'http://service.com/upload', form: {key:'value'}}, function(err,httpResponse,body){ /* ... */ })
```

- multipart/form-data (Multipart Form Uploads)

```
var formData = {
  // 普通数据
  my_field: 'my_value',

  // Buffers
  my_buffer: new Buffer([1, 2, 3]),

  // 文件流
  my_file: fs.createReadStream(__dirname + '/unicycle.jpg'),

  // 数组
  attachments: [
    fs.createReadStream(__dirname + '/attachment1.jpg'),
    fs.createReadStream(__dirname + '/attachment2.jpg')
  ],

  // 对象
  custom_file: {
    value: fs.createReadStream('/dev/urandom'),
    options: {
      filename: 'topsecret.jpg',
      contentType: 'image/jpeg'
    }
  }
};
```

```
request.post({
  url: 'http://service.com/upload',
  formData
}, (err, httpResponse, body) => {
  if (err) {
    return console.error('upload failed:', err);
  }
  console.log('Upload successful! Server responded with:', body);
});
```

- 流

```
request('http://img.zcool.cn/community/018d4e554967920000019ae9df1533.jpg@900w_1l_2o_100sh.jpg').pipe(fs.createWriteStream('test.png'))
request('https://cnodejs.org/api/v1/topics?page=1&limit=10').pipe(fs.createWriteStream('cnodejs.json'));
```

cheerio

一个专门为服务器设计包含jquery核心库的第三方模块

- cheerio.load(html)

用于在html代码以便进一步的精确选取，它不会执行html代码中的css和js

- 爬虫应用(爬取图片到本地)

又被称为网页蜘蛛，网络机器人，主要是在服务端去请求外部的 url 拿到对方的资源，然后进行分析并抓取有效数据。

```
//这里用 request 实现一个简单的图片抓取的小爬虫
const request = require('request');
const fs = require('fs');
const cheerio = require('cheerio');// cheerio为包含jQuery 核心的子集
const path = require('path');

request('http://www.lanrentuku.com/', (error, response, body) => {
  // 执行load方法载入获取到的html结构
  let $ = cheerio.load(body);

  // 利用jquery的核心方法获取html代码中的具体元素
  $('img', '.in-ne').each((i, e) => {
    let src = $(e).attr('src');
```

```
    let filename = path.basename(src);

    // 利用request与stream数据流保存爬取到的图片到本地硬盘
    request(src).pipe(fs.createWriteStream(filename))
  })
});
```

Async异步

Node.js 是一个异步机制的服务端语言，在大量异步的场景下需要按顺序执行，那正常做法就是回调嵌套回调，回调嵌套太多的问题被称之为回调地狱。

为解决这一问题推出了异步控制流 —— async

async/await

async/await 就 ES7 的方案，结合 ES6 的 Promise 对象，使用前请确定 Node.js 的版本是 7.6 以上。主要益处是可以避免回调地狱（callback hell），且以最接近同步代码的方式编写异步代码。

- 基本规则
 - async 表示这是一个async函数，await只能用在这个函数里面。
 - await 表示在这里等待promise返回结果了，再继续执行，后面跟着的应该是一个promise对象

对比使用

场景：3秒后返回一个值

- 原始时代

```
let sleep = (time, cb) => {
  setTimeout(() => {
    cb('ok');
  }, time);
}

let start = () => {
  sleep(3000, (result) => {
    console.log(result)
  })
}
```

```
start();
```

- Promise 时代

```
let sleep = (time) => {  
  return new Promise((resolve, reject) => {  
    setTimeout(() => {  
      resolve('ok') ;  
    }, time);  
  })  
}  
  
let start = () => {  
  sleep(3000).then((result) => {  
    console.log(result)  
  })  
}  
  
start()
```

- async/await 时代

```
let sleep = (time) => {  
  return new Promise((resolve, reject) => {  
    setTimeout(() => {  
      resolve('ok') ;  
    }, time);  
  })  
}  
  
let start = async () => {  
  let result = await sleep(3000);  
  console.log(result)  
}  
  
start();
```

捕捉错误try...catch

```

let sleep = (time) => {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      reject('error') ;
    }, time);
  })
}

let start = async () => {
  try{
    let result = await sleep(3000);
    console.log(result)
  } catch(err) {
    console.log('error')
  }
}

start();

```

在循环中使用

```

let start = async () => {
  for (var i = 1; i <= 3; i++) {
    console.log(`当前是第${i}次等待..`);
    await sleep(1000);
  }
}

start();

```

爬虫中使用

使用的模块：request、fs、cheerio

```

const request = require('request');
const fs = require('fs');
const cheerio = require('cheerio');
const path = require('path');

let spider = (url) => {
  return new Promise((resolve, reject) => {

```

```
        request(url, (error, response, body) => {
            resolve(body);
        })
    })
}

let start = async () => {
    let dom = await spider('http://www.lanrentuku.com/');
    let $ = cheerio.load(dom);
    $('img', '.in-ne').each((i, e) => {
        let src = $(e).attr('src');
        let filename = path.basename(src);
        request(src).pipe(fs.createWriteStream(filename))
    })
}

start();
```

Express

Express 是一个第三方模块，对原生模块封装了一套更灵活、更简洁的应用框架，其在 Node.js 环境的地位和作用好比 jQuery 在前端的地位和作用。使用 Express 可以快速地搭建一个完整功能的网站

安装

```
npm install express
```

使用

```
//引入模块
var express = require('express');
var app = express();

//开启服务器，定义端口8080:
app.listen(8080, function(){
    console.log('Server running on http://localhost:8080');
});
```

Express中间件(middleware)

中间件是一个封装了某些处理数据功能的函数，在request或response调用之前执行，从本质上来说，一个Express应用其实就是在调用各种中间件

使用中间件

格式： `app.use([path],...middlewares)`

内置中间件：

- `express.static(root, [options])` 基于 server-static 开发的中间件，负责托管 Express 应用内的静态资源，如：图片，CSS, JavaScript 等，一般用于实现静态资源服务器
 - root 参数指的是静态资源文件所在的根目录。
 - options 对象是可选的，支持以下属性：

- maxAge

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

自定义中间件

格式为： `function(request,response,next){}`

- next():next是一个方法，因为一个应用中可以使用多个中间件，而要想运行下一个中间件，那么上一个中间件必须运行

```
app.use((req,res,next)=>{
  // 任何请求都进入此中间件
});

app.use('/goods',(req,res,next)=>{
  //只有请求地址为/goods时才进入此中间件
})
```

第三方中间件

- body-parser：解析body中的数据，并将其保存为Request对象的body属性。
- multer：node.js 中间件，用于处理 enctype="multipart/form-data"（设置表单的MIME编码）的表单数据。
- cookie-parser：解析客户端cookie中的数据，并将其保存为Request对象的cookie属性

- express-session：解析服务端生成的sessionid对应的session数据，并将其保存为Request对象的session属性
- http-proxy-middleware

定义路由

GET

- 定义主页路由,当我们访问: `http://localhost:8080/` 时触发

```
app.get('/', function(request, response){  
  response.send('首页');  
});
```

- 定义任意路由(如: cart), 当我们访问: `http://localhost:8080/cart` 时触发

```
app.get('/cart', function(request, response){  
  response.send('购物车页面');  
})
```

- 定义带参数路由

- 查询参数: 通过 `request.query` 来获取参数

```
//访问地址: http://localhost:8080/search?keyword=iphonX  
app.get('/search', function(request, response){  
  var params = {  
    username: request.query.keyword  
  }  
  response.send(params);  
});
```

- 动态路由: 通过 `request.params` 来获取参数

```
//访问地址: http://localhost:8080/goods/10086,  
app.get('/goods/:id', function(request, response){  
  var params = {  
    username: request.params.id  
  }  
}
```

```
    response.send(params);
  });
```

POST

post 参数接收，可依赖第三方模块 `body-parser` 作为**express中间件**进行转换会更方便、更简单

```
//参数接受和 GET 基本一样，不同的在于 GET 是 request.query 而 POST 的是 request.body
var bodyParser = require('body-parser');

// 创建urlencoded 编码解析（把请求头content-type值为application/x-www-form-urlencoded时的数据格式化到request.body中）
var urlencodedParser = bodyParser.urlencoded({ extended: false });

// 创建json编码解析（把请求头content-type值为application/json时的数据格式化到request.body中）
const jsonParser = bodyParser.json();

app.post('/getUsers', urlencodedParser, function (request, response) {
  var params = {
    username: request.body.username,
    age: request.body.age
  }
  response.send(params);
});
```

跨域支持

把这个路由配置放在所有路由的前面，方便调用next操作

```
app.all('*', function(req, res, next) {
  res.header("Access-Control-Allow-Origin", "*");
  res.header("Access-Control-Allow-Headers", "Content-Type,Content-Length, Authorization, Accept,X-Requested-With");
  res.header("Access-Control-Allow-Methods", "PUT,POST,GET,DELETE,OPTIONS");

  // 跨域请求CORS中的预请求
  if(req.method=="OPTIONS") {
    res.sendStatus(200);/*让options请求快速返回*/
  } else{
    next();
  }
});
```

代理服务器

代理服务器最关键和主要的作用就是请求转发，即代理服务器将实际的浏览器请求转发至目标服务器，

- 实现这个功能，关键就在下面两点：
 - 请求转发至目标服务器。
 - 响应转发至浏览器。
- 无论请求还是响应，转发有需要注意两个点：
 - 请求或响应头。
 - | http请求是无状态的，我们使用session的方式验证用户权限，session等经常保存在cookie中，所以，头的转发是必须的。
 - 请求或响应实体数据。
- 利用http-proxy-middleware实现代理 webpack代理服务器使用的中间件

Socket

| ws模块

协议

HTTP/HTTPS 协议

http/https协议可以总结几个特点：

- 一次性的、无状态的短连接：客户端发起请求、服务端响应、结束。
- 被动性响应：只有当客户端请求时才被执行，给予响应，不能主动向客户端发起响应。
- 信息安全性：得在服务器添加 SSL 证书，访问时用 HTTPS。
- 跨域：服务器默认不支持跨域，可在服务端设置支持跨域的代码或对应的配置。

TCP 协议

TCP 协议可以总结几个特点：

- 有状态的长连接：客户端发起连接请求，服务端响应并建立连接，连接会一直保持直到一方主动断开。
- 主动性：建立起与客户端的连接后，服务端可主动向客户端发起调用。

- 信息安全性：同样可以使用 SSL 证书进行信息加密，访问时用 WSS。
- 跨域：默认支持跨域。

WebSocket 协议

WebSocket 目前由 W3C 进行标准化。WebSocket 已经受到 Firefox 4、Chrome 4、Opera 10.70 以及 Safari 5 等浏览器的支持。如果在前端我们可以把 AJAX 请求当作一个 HTTP 协议的实现，那么，WebSocket 就是 TCP 协议的一种实现。

使用Socket

服务端

- 安装ws模块

```
npm install ws
```

- 开启WebSocket服务器

```
let socketServer = require('ws').Server;  
let wss = new socketServer({  
  port: 1001  
});
```

- 配合express开启服务器

```
let express = require('express');  
let http = require('http');  
let ws = require('ws');  
  
let app = express();  
let server = http.Server(app);  
let SocketServer = ws.Server;  
let wss = new SocketServer({  
  server,  
  port: 1001  
});
```

- 用 on 来进行事件监听

- connection: 连接监听, 当客户端连接到服务端时触发该事件, 返回连接客户端对象
 - close: 连接断开监听, 当客户端断开与服务器的连接时触发
 - message: 消息接受监听, 当客户端向服务端发送信息时触发该事件
 - send: 向客户端推送信息

客户端

WebSocket是HTML5开始提供的一种基于 TCP 的协议, 连接建立以后, 客户端和服务端就可以通过 TCP 连接直接交换数据

- 实例化 WebSocket , 参数为 WebSocket 服务器地址, 建立与服务器的连接
- 事件
 - onopen: 当网络连接建立时触发该事件
 - onclose: 当服务端关闭时触发该事件
 - onerror: 当网络发生错误时触发该事件
 - onmessage: 当接收到服务器发来的消息的时触发的事件, 也是通信中最重要的一個监听事件
- 方法
 - close(): 在客户端断开与服务端的连接 socket.close();
 - send(): 向服务端推送消息

```
//连接 socket 服务器
var socket = new WebSocket('ws://localhost:1001');

//监听 socket 的连接
socket.onopen = function(){
    document.write("服务已连接 ws://localhost:1001");
}

//监听服务端断开
socket.onclose = function(){
    document.write("服务已断开");
    socket = null;
}

//监听服务端异常
socket.onerror = function(){
    document.write("服务错误");
    socket = null;
}

//监听服务端广播过来的消息
```

```

socket.onmessage = function(msg){
    var msgObj = JSON.parse(msg.data);
    if(msgObj.status == 0){
        $('<p>' + msgObj.nickname + '[' + msgObj.time + ']退出聊天</p>').appendTo('.msgList');
    } else{
        $('<p>' + msgObj.nickname + '[' + msgObj.time + ']: ' + msgObj.message + '</p>').appendTo('.msgList');
    }
}

var sendMessage = function(_mess){
    if(socket){
        var myDate = new Date();
        var now = myDate.getMonth() + '-' + myDate.getDate() + ' ' + myDate.getHours() + ':' + myDate.getMinutes() + ':' + myDate.getSeconds();

        var mesObj = {
            nickname: $('#nickName').val(),
            message: _mess || $('#mesBox').val(),
            time: now
        }
        //向服务端发送消息
        socket.send(JSON.stringify(mesObj));
    }
}

```

项目应用

- 服务端开器一个服务
- 客户端建立和服务端的连接;
- 建立连接的同时发送上线信息给服务端：'xxx加入聊天';
- 服务端接受到客户端的消息触发 message 方法，然后将该消息广播给所有在线的用户
- 所有客户端收到来自服务端广播的消息，然后将该消息显示在聊天列表。
- 聊天和退出聊天都是重复着客户端发送消息，服务端接受消息然后向客户端广播消息，客户端显示广播消息。

##常用第三方模块工具

- cnpm
- gulp
- supervisor
- http-server
- yarn

• ...