常用模块

1. http 模块的简单介绍

node.js 当中的 http 内置模块可以用于创建 http 服务器与 http 客户端。

const http = require('http');

1. 创建 http 服务器

虽然 Node.js 中的 http 核心模块可以用来实现 web 服务器，但要做很多工作，例如要检测每个文件的 content type

因此我们决定用 Connect 模块，非常流行的 Express web 框架就是建立在 Connect 之上的.

```
var server = http.createServer((req,res)=>{
});
```

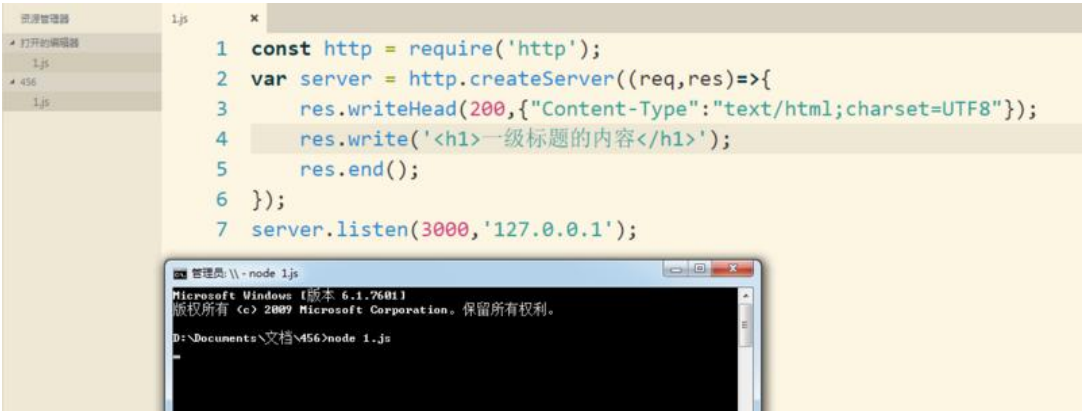
使用 http 的.createServer()方法可以用于返回一个 http 服务器实例，用自定义的 server 变量来接收。当该服务器每次接收到客户端的请求时触发调用其内部的回调函数，客户端每访问一次，都会触发调用一次。该回调函数有两个参数，req 和 res,顺序不可颠倒，req 表示请求 request,res 表示响应 response。

该回调函数内部语句的一定要有 res.end();，因为如果没有，浏览器会认为一直没有得到服务器的响应，则浏览器一直会处于被挂起的状态，此时浏览器内部有一个超时机制，一旦超时，则会报告错误。

该回调函数当中的常用代码语句有：
设置响应头，res.writeHead(状态码,{});
其中 HTTP 状态码常用的有 200（成功返回）、404（找不到该页面，返回错误）等。
第二个参数传入一个对象，用于设置响应文本的渲染解析类型。
如常用的有
对于 html 代码设置为，res.writeHead(200,{"Content-Type":"text/html;charset=UTF8"});
对于 css 文件的设置为 res.writeHead(200,{"Content-Type":"text/css"});
对于图片的设置为 res.writeHead(200,{"Content-Type":"image/jpg"});
对于纯文本的设置为 res.writeHead(200,{"Content-Type":"text/plain"});
设置返回的内容，res.write('');

2. 让该服务器监听特定的端口号

用 server 这个自定义的变量来表示创建的服务器来监听某个指定的端口号。
server.listen(3000,'192.168.155.1');外界客户端可以通过这个 ip 地址和端口号来访问这个服务器。



此时表示该服务器处于挂起的状态，此时在浏览器当中输入对应的 ip 地址与端口号即可得到服务端响应的内容。

3. 完整的示例代码：

```
const http = require('http');
const url = require('url');
const qs = require('qs');
var server = http.createServer((req,res)=>{
   res.writeHead(200,{"Content-Type":"text/html;charset=UTF8"});
   res.write('hello world');
   const urlJson = url.parse(req.url);
      var query = qs.parse(urlJson.query);
      console.log(query);
      res.end();
});
server.listen(3001);
console.log('listen 3001');
```

Connect 介绍

Connect 是一个 node 中间件（middleware）框架。如果把一个 http 处理过程比作是污水处理，中间件就像是一层层的过滤网。每个中间件在 http 处理过程中通过改写 request

或（和）response 的数据、状态，实现了特定的功能。

https://www.npmjs.com/package/connect

尝试做一个最简单的 web 服务器

```
var connect = require('connect');
var app = connect()
	.use(connect.logger('dev'))
	.use(function (req, res) {
		res.end('hello world\n');
	})
	.listen(3000);
```

url & qs

安装： npm install url qs --save-dev

url 顾名思义，就是为客户端请求地址的一个解析。而 qs 则是在 url 处理结果的基础上再进行字段解析。

```
const url = require('url');
const app= require('connect');
const qs = require('qs');
app()
	.use(function(req, res) {
		const urlJson = url.parse(req.url);
		console.log(urlJson);
		/*  打印可得
			{
				protocol: null,
				slashes: null,
				auth: null,
				host: null,
				port: null,
				hostname: null,
				hash: null,
				search: '?title=test&name=laney',
				query: 'title=test&name=laney',
				pathname: '/',
```

```
                path: '/?title=test&name=laney',
                href: '/?title=test&name=laney'
            }
        */

        const query = qs.parse(urlJson.query);
    })
    .listen(3000);
```

Connect 工作机制

1. 自定义中间件
2. use() 函数支持链式调用
3. 中间件的顺序问题

在 Connect 中，中间件组件是一个函数，它**拦截 HTTP 服务器提供的请求和响应对象**，执行逻辑，然后或者结束响应，或者把它传递给下一个中间件组件。

Connect 分配器会依次调用所有附着的中间件组件，直至其中一个组件决定响应该请求，如果知道中间件组件列表末尾还是没有组件决定响应，程序会返回 404 作为响应。（ex1 这个程序没有中间件组件，故会返回 404 Not Found 状态码响应他接收到的所有 HTTP 请求。）

Connect 中，中间件组件是一个 JS 函数，接收 3 个参数

a. 请求对象**（req）**

b. 响应对象**（res）**

c. 通常会命名为 **next** 的参数:一个回调函数，表明当前中间件已经执行完毕，可以执行下一个中间件组件

简单的 hello world + 日志 log

```
var connect = require("connect");
var app = connect()
            .use(logger)
            .use(hello)
            .listen(3000);
function logger(req, res, next) {
    console.log(req.method + ' ' + req.url);
    next(); //执行下一个中间件
}
function hello(req, res) {
    res.setHeader('Content-type', 'text/plain');
    res.end("hello world");
}
```

中间件的顺序问题

```
var app = connect()
            .use(hello)
            .use(logger)
            .listen(3000);
```

先执行了 hello 的中间件，响应了 HTTP 的请求，同时没有调用 next()，控制权不会回到分配器去调用下一个中间件，所以 logger 中间件不会调用。

注意：当一个中间件组件不调用 next()时，命令链中的其他中间件都不会被调用。

connect 常用中间件

connect 是一个非常轻量级 Node 中间件框架，对于一个中小型项目，connect 需要配合很多三方模块才能完善服务器逻辑。所以现在的工程中，为了简化开发，往往会采用它的升级版 express ，或是 koa 作为基础框架进行扩展。当然，在使用这些框架之前，对于最原始的细节，同样也是需要去仔细钻研的。

body-parser

安装： yarn add 'body-parser
用来解析请求主体

Node 或是使用 Connect 在处理请求数据时，是需要解析 req 的，body-parser 中间件恰好提供了这一操作。

- bodyParser()中间件的作用是给 req 添加 body 属性，可以用来解析 JSON、x-www-form-urlencoded 和 multipart/form-data 请求
- body-parser 只处理 POST 请求
- body-parser 模块导出一个对象，上面有两个方法 urlencoded 和 json，分别处理表单提交和 json 格式的请求体参数
- 如果是 multipart/form-data 请求，比如文件上传，则还有 req.files 对象
- 这是个非常有用的组件，实际上它整合了其他三个更小的组件：json(), urlencoded(), 和 multipart()
- 旧版本的 body-parser 还可以处理文件上传(现在需要使用 formidable 一类的文件上传模块)。

当请求体解析之后，解析值会被放到 req.body 属性中，当内容为空时候，为一个空对象{}
---bodyParser.json()--解析 JSON 格式
---bodyParser.raw()--解析二进制格式
---bodyParser.text()--解析文本格式
---bodyParser.urlencoded()--解析文本格式

```
const app= require('connect')();
const bodyParser = require('body-parser');
app
    .use(function(req, res, next) {
        console.log(req.body); // undefined
        next();
    })
    .use(bodyParser())
    .use(function(req, res) {
        console.log(req.body); // post 请求数据
    })
    .listen(3000);
```

不仅仅是 connect ，在 express 和 koa 中，同样也有 body-parser 中间件。

cookie-parser & jsonwebtoken

和 body-parser 类似，cookie-parser 将来自浏览器的 cookie 请求体转化到 req.cookies 上。它可以处理常规 cookie 和 签名 cookie。

在现在的前后端通信，特别是第三方认证盛行的模式下，cookie 和 sesssion 已不在是主流，再加之为了遵守 http 无状态的本质，往往会选择 token 方式进行认证。而 jsonwebtoken 恰好能让我们的 Node 程序实现这一认证。

```
const jwt = require('jsonwebtoken');
/**
 * 对验证成功地请求体进行签名
 * obj 被签名的主体
 * signedText 密钥
 * options 签名设置，如过期日期
*/
const token = jwt.sign(obj, signedText, options);
/**
 * 进行验证
 * client_token 客户端传来的 token 值
 * signedText 密钥
*/
jwt.verify(client_token, signedText, function(err, decode) {
    if (err) return;
    console.log(decode);
    // 验证成功后 token 信息
});
```

serve-static

安装 ： yarn add serve-static

首先，由于 Node.js 的事件驱动机制类似 Nginx，虽然，Node.js 主要是为动态页面服务的服务器，但是我们也可以用作像 Nginx Apache 那样静态 Html 页面服务器，主要是没有 Nginx 那么多配置优化，直接简单启动就可以了，步骤如下：

1. 使用 NPM 安装 connect 和 serve-static

```
npm install connect serve-static
```

2. 在当前 bin 目录创建一个文件名 server.js 的文件，内容如下

```
var connect = require('connect');
var serveStatic = require('serve-static');
connect().use(serveStatic(__dirname)).listen(8080);
```

3. 运行 NodeJS

node server.js

注意，在 bin 目录下创建 index.html 和其他静态资源如图片以后，就可以通过浏览器访问你的文件：http://localhost:8080/index.html

配置

Index.js 配置如下：

```
var connect = require('connect');  //创建连接
var bodyParser = require('body-parser');   //body 解析
var cookieParser = require('cookie-parser');

var app = connect()
   .use(bodyParser.json())   //JSON 解析
   .use(bodyParser.urlencoded({extended: true}))
   .use(cookieParser())
```

增加跨域的处理

```
.use(function (req, res, next) {
      //跨域处理
      // Website you wish to allow to connect
      res.setHeader('Access-Control-Allow-Origin', '*');  //允许任何源
      // Request methods you wish to allow
       res.setHeader('Access-Control-Allow-Methods', 'GET, POST, OPTIONS, PUT, PATCH, DELETE');
//允许任何方法
      // Request headers you wish to allow
      res.setHeader('Access-Control-Allow-Headers', '*');   //允许任何类型
      res.writeHead(200, {"Content-Type": "text/plain;charset=utf-8"});    //utf-8 转码
      next();  //next 方法就是一个递归调用
   })

   添加一个  简单的接口
.use('/info', function(req, res, next) {
      //response 响应   request 请求
      // 中间件
      console.log(req.method + ' ' + req.url);
      // console.log(req.body);
      // console.log(req.originalUrl, req.url);

       // Cookies that have not been signed
```

```
        console.dir('Cookies: ', req.cookies)

        // Cookies that have been signed
        console.log('Signed Cookies: ', req.signedCookies)

       var data={
            "code": "200",
            "msg": "success",
            "result": [{
               "id":1,
               "name": "sonia",
               "content": "广告投放 1"
            },
            {
               "id":2,
               "name": "ben",
               "content": "广告投放 2"
            },
            {
               "id":3,
               "name": "lili",
               "content": "广告投放 3"
            }]
          }
          res.end(JSON.stringify(data));
          next();
    });

app.listen(3000);
```

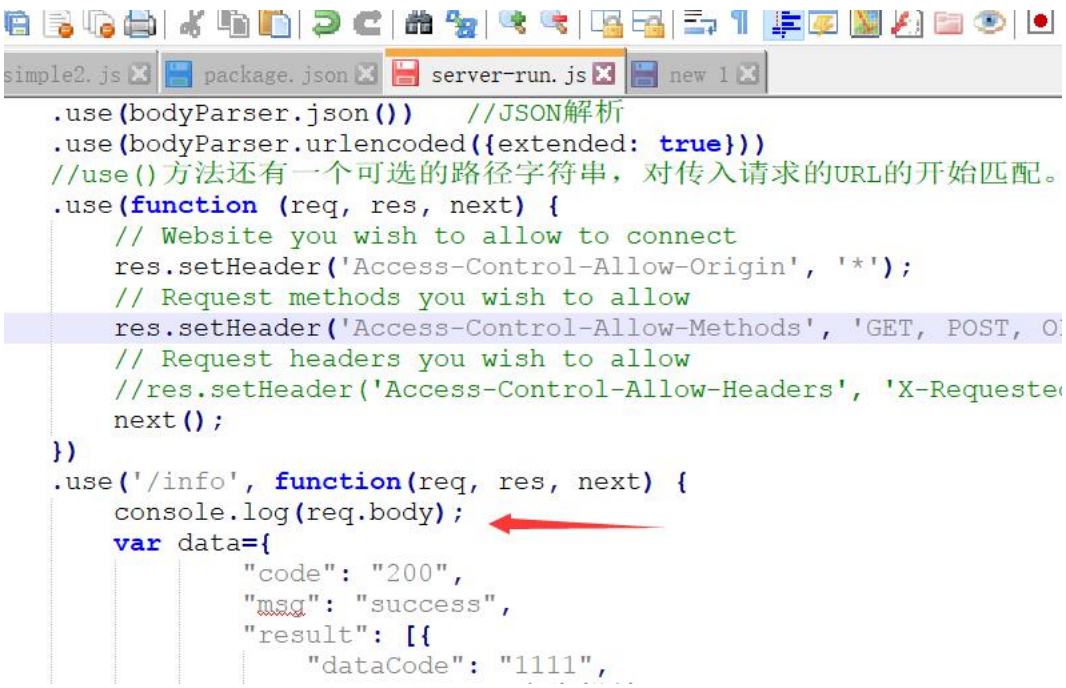
运行

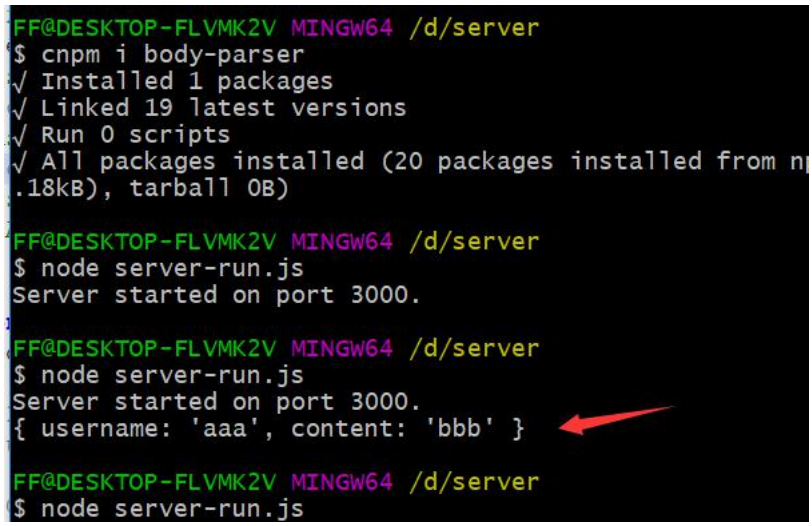
依赖安装完成后输入 node server-run.js

接口访问可通过 http://localhost:3000/info

如何查看接受参数

POST 请求





GET 请求

安装 url , qs

```
const url = require('url');
const qs = require('qs');

const urlJson = url.parse(req.url);
const query = qs.parse(urlJson.query);
```

所以最后：

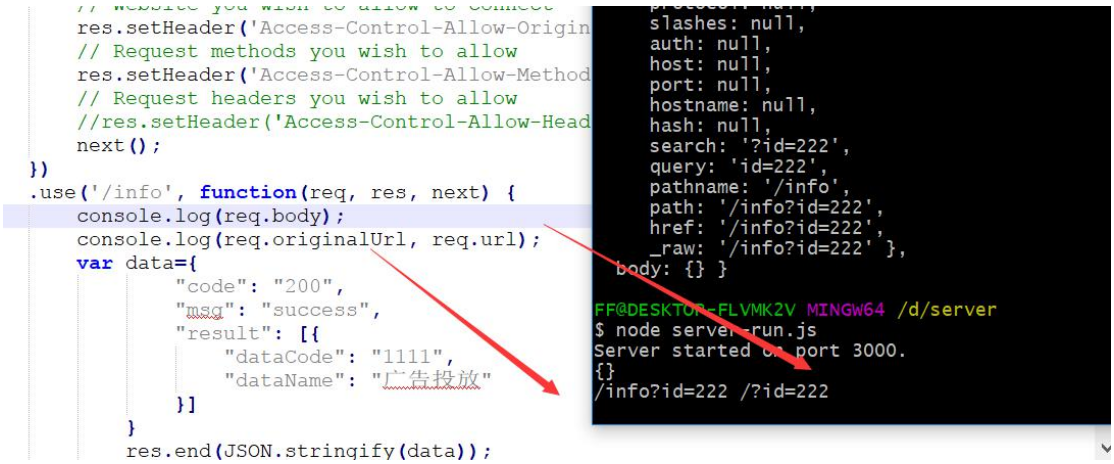
获取参数的最后可以封装写成

```
function getParams(req){
```

```
    //判断是什么请求
    var query={};
    if(req.method.toLocaleLowerCase()=='post') {
        console.log(req.body);
        query = req.body;
    } else {
        const urlJson = url.parse(req.url);
        query = qs.parse(urlJson.query);
        console.log(query);
    }
    return query;
}
```

然后在接口中调用

```
var queryres =  getParams(req);
 console.log(queryres);
```



测试：

封装的 ajax

```
function xmlAjax(opt){
    var optInit = {
        method:'post',
        url:'',
        data:null,
        async:true,
```

```
    contentType:'application/json;charset=UTF-8',
    done:function(){},
    fail:function(){}
  }
  var endObj = Object.assign({},optInit,opt);
  var xml = new XMLHttpRequest();

  xml.open(endObj.method,endObj.url,optInit.async);
  xml.setRequestHeader('Authorization','99a9df06-c224-436b-a48a-65884dea37c9');
  xml.setRequestHeader('Content-Type',endObj.contentType);

  var isGet = endObj.method.toLowerCase()=='get';
  if(isGet && endObj.data) {
    var keys = Object.keys(endObj.data);
    var str = '';
    for(var key in endObj.data) {
      str+= key+'='+endObj.data[key]+'&';
    }
    str = str.substring(0,str.length-1);
    endObj.url = endObj.url+'?'+str;
  }
  if(isGet){
    xml.send();
  } else {
    if(endObj.data) {
      var dataParam = JSON.stringify(endObj.data)
      xml.send(dataParam);
    } else {
      xml.send();
    }
  }

  xml.onreadystatechange = function(){
    if(xml.readyState==4) {
      if(xml.status==200){
        let response = JSON.parse(xml.responseText);
        endObj.done(response.data);
      } else {
        endObj.fail(response);
      }
    }
  }
}
```

HTML 编码

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<script src="global.js" ></script>

</head>
<body>

<div id="myDiv"><h2>使用 AJAX 修改该文本内容</h2></div>
<button type="button" onclick="loadXMLDoc()">修改内容</button>

<script>
   var baseurl = 'http://localhost:3000/info';
function loadXMLDoc(){

   xmlAjax({
      method:'post',
      data:{
         title:'测试 1',
         name:'laney'
      },
      url:baseurl,
      done:function(data){
         console.log(data);
      }

   })
}
</script>
</body>
</html>
```

Nodejs 热加载

方式一、supervisor

```
sudo npm install -g supervisor    #安装
supervisor app.js    #启动
```

方式二、hotnode

sudo npm -g install hotcode　#安装
hotnode app.js　#启动

方式三：
yarn global add nodemon
nodemon app.js

使用 http-server 开启一个本地服务器

下载安装

npm install http-server -g

开启　http-server 服务

终端进入目标文件夹，然后在终端输入：

http-server -c-1　（⚠　只输入 http-server 的话，更新了代码后，页面不会同步更新）

关闭　http-server 服务

按快捷键 CTRL-C

终端显示^Chttp-server stopped.即关闭服务成功。

formidable 处理 POST 方式上传的文件或图片

`form.html` 文件

```
<form action="http://192.168.155.1:3000/dopost" method="POST"
enctype="multipart/form-data">
    <p><input type="file" name="uploadImg"></p>
    <p><input type="submit" value="提交"></p>
</form>
```

当表单提交的过程中涉及文件或图片上传，则一定要设置表单头，即在 form 标签上加上固定写法的属性为 enctype="multipart/form-data"，否则文件或图片会上传失败。其中<input type="file" name="uploadImg">，当中的 name 属性一定要赋值。

当要使用 formidable 来处理上传的图片时，常用的代码段为：

```
const formidable = require('formidable');
var connect = require('connect');
var app = connect();

app.use('/upload/img',function(req, res, next){
   var form = new formidable.IncomingForm();
   form.encoding='utf-8';
   form.uploadDir = path.join(__dirname,'./static');
   form.keepExtensions=true;
   form.parse(req,function(err,fields,files){
      console.log(files)
      if (err){
         return;
      };
      var size=parseInt(files.uploadImg.size);
      if (size>1024*1024){
         res.send("图片过大！")
         fs.unlink(files.uploadImg.path);
         return;
      };
      res.end('upload success');
      next();
   });

})
```

Formidable

下载并引包

npm install formidable
再通过 const formidable = require('formidable');来引包。

要正确处理上传的文件，并接收到文件的内容，需要把表单的 enctype 属性设为 multipart/form-data。

Node 社区有几个可以完成这个任务的模块，formidable 就是之一。简单使用如下:

```
const formidable = require('formidable');
const app= require('connect')();
app
    .use(function(req, res) {
        if (req.method == 'POST') {
            const form = new formidable.IncomingForm();
            // 上传路径
            form.uploadDir = __dirname + '/public';
            form.parse(req, function(err, fields, files) {
                res.end('upload complete!');
            });
        }
    })
    .listen(3000);
```