Nodejs

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

Node.js 是一个开源与跨平台的 JavaScript 运行时环境。 它是一个可用于几乎任何项目的流行工具！

Node.js 在浏览器外运行 V8 JavaScript 引擎（Google Chrome 的内核）。 这使 Node.js 表现得非常出色。

Node.js 应用程序运行于单个进程中，无需为每个请求创建新的线程。 Node.js 在其标准库中提供了一组异步的 I/O 原生功能（用以防止 JavaScript 代码被阻塞），并且 Node.js 中的库通常是使用非阻塞的范式编写的（从而使阻塞行为成为例外而不是规范）。

当 Node.js 执行 I/O 操作时（例如从网络读取、访问数据库或文件系统），Node.js 会在响应返回时恢复操作，而不是阻塞线程并浪费 CPU 循环等待。

这使 Node.js 可以在一台服务器上处理数千个并发连接，而无需引入管理线程并发的负担（这可能是重大 bug 的来源）。

Node.js 具有独特的优势，因为为浏览器编写 JavaScript 的数百万前端开发者现在除了客户端代码之外还可以编写服务器端代码，而无需学习完全不同的语言。

在 Node.js 中，可以毫无问题地使用新的 ECMAScript 标准，因为不必等待所有用户更新其浏览器，你可以通过更改 Node.js 版本来决定要使用的 ECMAScript 版本，并且还可以通过运行带有标志的 Node.js 来启用特定的实验中的特性。

大量的库

npm 的简单结构有助于 Node.js 生态系统的激增，现在 npm 仓库托管了超过 1,000,000 个可以自由使用的开源库包。

13.1.1 Nodejs简介与环境搭建

Nodejs简史

Nodejs诞生与2009年，

JavaScript 是一门被创建于 Netscape（作为用于在其浏览器 Netscape Navigator 中操纵网页的脚本工具）中的编程语言。

Netscape 的商业模式的其中一部分是出售 Web 服务器，其中包括一个被称为 Netscape LiveWire 的环境，该环境可以使用服务器端 JavaScript 创建动态页面。 不幸的是，Netscape LiveWire 并不十分成功，并且服务器端 JavaScript 也没有普及，直到引入了 Node.js。

引领 Node.js 兴起的一个关键因素是时机。 仅仅几年前，多亏 "Web 2.0" 应用程序（例如 Flickr、Gmail 等）向世界展示了 Web 上的现代体验，JavaScript 开始被视为一种更为严肃的语言。

随着许多浏览器竞相为用户提供最佳的性能，JavaScript 引擎也变得更好。 主流浏览器背后的开发团队都在努力为 JavaScript 提供更好的支持，并找出使 JavaScript 运行更快的方法。 多亏这场竞争，Node.js 使用的 V8 引擎（也称为 Chrome V8，是 Chromium 项目开源的 JavaScript 引擎）获得了显着的改进。

Node.js 恰巧构建于正确的地点和时间，但是运气并不是其今天流行的唯一原因。 它为 JavaScript 服务器端开发引入了许多创新思维和方法，这已经对许多开发者带来了帮助。

Nodejs优点

1. 运行在V8JavaScript引擎上（高性能）
2. 事件驱动
3. 非阻塞的IO模型
4. 丰富的生态圈（npm下载资源）

Nodejs安装

首先我们可以在官网下载最新版本的Nodejs

安装的流程非常简单，下载软件，根据提示下一步进而完成安装

我们需要熟悉命令行的相关操作，此处不做赘述，可以参考ES6部分课程中的命令行操作

创建app.js文件，运行Node代码

```
console.log("Hello Nodejs")

// node app.js
```

13.1.2 全局对象

Nodejs提供了很多全局对象，这些对象可以直接使用，不需要做单独的引入例如：

在js中的日志信息打印就是其中之一，文档

```
console.log("测试");
```

定时器也是其中之一,文档

```
var time = 0;

var timer = setInterval(function() {
    time += 2;
    console.log(time + " seconds have passed");
    if (time > 5) {
        clearInterval(timer);
    }
}, 2000);
```

还用一些常用的全局对象例如:

```
console.log(__dirname);  // 获得当前路径
console.log(__filename); // 获得当前路径及文件名字
```

13.1.3 回调函数

我们需要熟悉一下回调函数，因为在整个Nodejs中，慢慢的都是回调函数

我们先看下面的基础代码

```
function sayHi(){
    console.log("hi");
}

var sayBye = function(){
    console.log("bye");
}

sayHi();
sayBye();
```

以上是两种函数的声明方式

然后我们增加回调函数

```
function sayHi(){
    console.log("hi");
}

var sayBye = function(){
    console.log("bye");
}

sayHi();
sayBye();

function callFunction(callback){
    callback()
}

callFunction(sayBye);
```

上述代码其中`callFunction`是一个回调函数，调用了`sayBye`函数的执行

我们同样在回调函数中是可以传递参数的

```

```

```
var sayBye = function(name){
    console.log(name+":bye");
}

function callFunction(callback,name){
    callback(name)
}

callFunction(sayBye,'iwen');
```

我们还可以变换书写方式

```
function callFunction(callback,name){
    callback(name)
}

callFunction(function(name){
    console.log(name+":bye");
},'iwen');
```

13.1.4 模块(Commonjs规范)

在JavaScript的ES5版本中，最大的问题就是没有模块的概念，但是nodejs中，增加了commonjs规范来处理这一问题。

我们看如下代码

```
// utils.js

var adder = function(a,b) {
    return `the sum of the 2 numbers is ${a+b}`;
}

module.exports = adder
```

```
// app.js

var adder = require('./utils.js');

adder(10,20) // 30
```

如果我们要导出多个对象可以如下代码

```

```

```
// utils.js

var adder = function(a,b) {
    return `the sum of the 2 numbers is ${a+b}`;
}

var counter = function(arr) {
    return "There are " + arr.length + " elements in the array";
}

module.exports.adder = adder
module.exports.counter = counter
```

```
// app.js

var utils = require('./utils.js');

utils.adder(10,20) // 30
utils.counter([10,20,30]) // 3
```

如上我们导出了多个对象，但是写法过于麻烦，还可以改成如下

```
// utils

var adder = function(a,b) {
    return `the sum of the 2 numbers is ${a+b}`;
}

var counter = function(arr) {
    return "There are " + arr.length + " elements in the array";
}

module.exports = {
    adder:adder,
    counter:counter
}
```

一会可能出现导出的时候，直接附带函数

```
var adder = function(a,b) {
    return `the sum of the 2 numbers is ${a+b}`;
}

module.exports = {
    adder:adder,
    counter:function(arr) {
```

```
        return "There are " + arr.length + " elements in the array";
    }
}
```

13.1.5 事件

大多数 Node.js 核心 API 构建于惯用的异步事件驱动架构，其中某些类型的对象（又称触发器，Emitter）会触发命名事件来调用函数（又称监听器，Listener）。

例如，net.Server 会在每次有新连接时触发事件，fs.ReadStream 会在打开文件时触发事件，stream会在数据可读时触发事件。

所有能触发事件的对象都是 EventEmitter 类的实例。 这些对象有一个 eventEmitter.on() 函数，用于将一个或多个函数绑定到命名事件上。 事件的命名通常是驼峰式的字符串，但也可以使用任何有效的 JavaScript 属性键。。

当 EventEmitter 对象触发一个事件时，所有绑定在该事件上的函数都会被同步地调用。 被调用的监听器返回的任何值都将会被忽略并丢弃。

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

var myEmitter = new events.EventEmitter();

myEmitter.on('someEvent', function(message) {
    console.log(message);
})

myEmitter.emit('someEvent', 'the event was emitted');
```

出了直接使用事件，我们还可以让对象继承事件，来给对象添加事件

```
var events = require('events');
var util = require('util');

var Person = function(name) {
    this.name = name
}

// inherits继承
util.inherits(Person, events.EventEmitter);

var xiaoming = new Person('xiaoming');
var lili = new Person('lili');
var lucy = new Person('lucy');

var person = [xiaoming, lili, lucy];

person.forEach(function(person) {
    person.on('speak', function(message) {
```

```
        console.log(person.name + " said: " + message);
    })
})

xiaoming.emit('speak', 'hi');
lucy.emit('speak', 'I want a curry');
```

13.1.6 文件读写

在Nodejs中有文件系统，是对本地文件进行读写操作，当然，如果要使用我们需要引入fs对象

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

var readMe = fs.readFileSync("./readme.txt",'utf8');
console.log(readMe);
```

我们同样可以将数据写入到文件

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

var readMe = fs.readFileSync("./readme.txt",'utf8');

fs.writeFileSync("writeMe.txt",readMe)
```

接下来，我们需要分析一下异步与同步的问题，上述代码是同步的效果，测试如下

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

var readMe = fs.readFileSync("./readme.txt",'utf8');

console.log(readMe);
console.log("finished");
```

上述代码先打印readMe的结果，在打印finished，我们知道的时候，文件的读取是一个比较耗时的操作，但是结果依然是上述效果，那就说说明上述是同步代码，接下来我们看看异步代码效果

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

var readMe = fs.readFile("./readme.txt",'utf8',function(err,data){
    console.log(data);
});

console.log("finished");
```

上述代码的打印与之前完全相反，先打印finished,然后输出data数据，此时就是异步效果

我们对耗时的代码一定要进行处理，否则我们的主线程则处于卡顿的等待过程，如下代码

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

var readMe = fs.readFile("./readme.txt",'utf8',function(err,data){
    console.log(data);
});

var waitTill = new Date(new Date().getTime() + 4 * 1000);
while (waitTill > new Date()) {}

console.log("finished");
```

我们做了一个4秒的等待，所以，打印需要在4秒之后才会出现。这就是卡顿的效果，用户体验一定是极差的

接下里我们对读写都使用异步的方案处理

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

var readMe = fs.readFile("readMe.txt", "utf8", function(err, data) {
    fs.writeFile('writeMe.txt', data, function() {
        console.log('writeMe has finished');
    })
});

console.log("finished");
```

13.1.7 流和管道

流的概念并不难理解，例如：我们平时前后端交互其实就是转换成流来进行交互的，我们之前也讲过文件的读写，文件的读写也属于流的操作的体现。这是如果文件特别大的时候，我们还是要采取buffer处理

我们现在命令行中操作一个基础操作

```
ls
ls | grep app
```

ls | grep app 这个命令在 linux 或 mac 才适合，或者 windows 的 git bash 也可以的。

如果是 windows 的命令提示符，对应的查找文件的命令应该是： dir | findstr app

下面我们看一下流的具体操作

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

var myReadStream = fs.createReadStream(__dirname + '/readMe.txt');

myReadStream.on('data', function(chunk) {
    console.log(chunk);
})
```

上述代码读到的是buffer对象，这也是他性能提升的主要原因。如果文件过大，会处理成多个buffer对象

如果想直接读取数据如下

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

var myReadStream = fs.createReadStream(__dirname + '/readMe.txt',"utf8");

myReadStream.on('data', function(chunk) {
    console.log(chunk);
})
```

但是一般我们直接在data事件中使用数据，因为可能数据还没有读取完成，所以我们可以监听end

```
var fs = require('fs');
var myReadStream = fs.createReadStream(__dirname + '/readMe.txt');
myReadStream.setEncoding('utf8');
var data = ""
myReadStream.on('data', function(chunk) {
    data += chunk;
})
myReadStream.on('end', function() {
    console.log(data);
})
```

上述是一个读取流的操作，那么写入流的操作如何使用呢，如下

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

var myReadStream = fs.createReadStream(__dirname + '/readMe.txt',"utf8");
var myWriteStream = fs.createWriteStream(__dirname + '/writeMe.txt');

var data = ""

myReadStream.on('data', function(chunk) {
    myWriteStream.write(chunk);
})
```

```
myReadStream.on('end', function() {
    console.log("end");
})
```

当然。我们也可以单独实现写入流的操作

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

var myWriteStream = fs.createWriteStream(__dirname + '/writeMe.txt');

var writeData = "hello world";
myWriteStream.write(writeData);
myWriteStream.end();
myWriteStream.on('finish', function() {
    console.log('finished');
})
```

接下里我们使用管道操作，如果使用管道操作，代码量更少

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

var myReadStream = fs.createReadStream(__dirname + '/readMe.txt');
var myWriteStream = fs.createWriteStream(__dirname + '/writeMe.txt');

myReadStream.pipe(myWriteStream);
```

下面我们来看一个示例，我们可以使用管道及一些第三方，来进行一个压缩操作

```
// 压缩
var crypto = require('crypto');
var fs = require('fs');
var zlib = require('zlib');

var password = new Buffer(process.env.PASS || 'password');
var encryptStream = crypto.createCipher('aes-256-cbc', password);

var gzip = zlib.createGzip();
var readStream = fs.createReadStream(__dirname + "/readMe.txt"); //
current file
var writeStream = fs.createWriteStream(__dirname + '/out.gz');

readStream // reads current file
    .pipe(encryptStream) // encrypts
    .pipe(gzip) // compresses
    .pipe(writeStream) // writes to out file
    .on('finish', function() { // all done
```

```
        console.log('done');
    });
```

上述代码实现了压缩，我们也可以进行解压操作

```
// 解压
var crypto = require('crypto');
var fs = require('fs');
var zlib = require('zlib');

var password = new Buffer(process.env.PASS || 'password');
var decryptStream = crypto.createDecipher('aes-256-cbc', password);

var gzip = zlib.createGunzip();
var readStream = fs.createReadStream(__dirname + '/out.gz');

readStream // reads current file
    .pipe(gzip) // uncompresses
    .pipe(decryptStream) // decrypts
    .pipe(process.stdout) // writes to terminal
    .on('finish', function() { // finished
        console.log('done');
    });
```

13.1.8 Web服务器输出内容

对于我们本套课程来说，我们用的最多的就是用Node构建一个服务器，并且输出内容，内容输出绝大多数为JSON数据

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

var onRequest = function(request, response) {
    console.log('Request received');
    response.writeHead(200, { 'Content-Type': 'text/plain' });
    // response.write('Hello from out application');
    response.end('Hello from out application');
}

var server = http.createServer(onRequest);

server.listen(3000, '127.0.0.1');
console.log('Server started on localhost port 3000');
```

上述代码我们完成了一个文本信息的输出，访问如下

```
http://localhost:3000/
```

我们还可以输出JSON格式的数据

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

var onRequest = function(request, response) {
    console.log('Request received');
    response.writeHead(200, { 'Content-Type': 'application/json' });
    // response.write('Hello from out application');
    var myObj = {
        name: "itbaizhan",
        job: "learn",
        age: 27
    };
    response.end(JSON.stringify(myObj));
}

var server = http.createServer(onRequest);

server.listen(3000, '127.0.0.1');
console.log('Server started on localhost port 3000');
```

当然，个别时候，我们可能需要接受一个服务器返回的页面，例如，支付相关的返回一般都是返回一个页面直接渲染

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

var onRequest = function(request, response) {
    console.log('Request received');
    response.writeHead(200, { 'Content-Type': 'text/html' });
    var myReadStream = fs.createReadStream(__dirname + '/index.html',
'utf8');
    // response.write('Hello from out application');
    myReadStream.pipe(response);
}

var server = http.createServer(onRequest);

server.listen(3000, '127.0.0.1');
console.log('Server started on localhost port 3000');
```

13.1.9 模块化组织代码

代码的组织能力也很重要，所以我们需要利用之前所学模块进行重新组织代码

```
// server.js

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

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

function startServer() {
    var onRequest = function(request, response) {
        console.log('Request received');
        response.writeHead(200, { 'Content-Type': 'text/html' });
        var myReadStream = fs.createReadStream(__dirname + '/index.html',
'utf8');
        // response.write('Hello from out application');
        myReadStream.pipe(response);
    }

    var server = http.createServer(onRequest);

    server.listen(3000, '127.0.0.1');
    console.log('Server started on localhost port 3000');
}

exports.startServer = startServer;
```

```
// app.js

var server = require('./server');

server.startServer();
```

13.1.10 路由

在实际的开发场景中，我们需要根据不同的地址返回不同的数据，也就是我们日常所说的路由效果

在上一小节中，如果我们直接访问http://localhost:3000/home和之前的访问是没有区别的，也就是我们无法根据不同地址返回不同的数据，接下来我们处理一下，形成路由

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

function startServer() {
    var onRequest = function(request, response) {
        console.log('Request received ' + request.url);
        if (request.url === '/' || request.url === '/home') {
            response.writeHead(200, { 'Content-Type': 'text/html' });
            fs.createReadStream(__dirname + '/index.html',
'utf8').pipe(response);
        } else if (request.url === '/review') {
            response.writeHead(200, { 'Content-Type': 'text/html' });
            fs.createReadStream(__dirname + '/review.html',
'utf8').pipe(response);
        } else if (request.url === '/api/v1/records') {
            response.writeHead(200, { 'Content-Type': 'application/json'
```

```
});
            var jsonObj = {
                name: "itbaizhan"
            };
            response.end(JSON.stringify(jsonObj));
        } else {
            response.writeHead(200, { 'Content-Type': 'text/html' });
            fs.createReadStream(__dirname + '/404.html',
'utf8').pipe(response);
        }
    }

    var server = http.createServer(onRequest);

    server.listen(3000, '127.0.0.1');
    console.log('Server started on localhost port 3000');
}

exports.startServer = startServer;
```

13.1.11 重构路由代码

上一小节已经完成了路由的效果，但是代码是在难看。所以我们需要重新整理代码结构

```
// app.js

var server = require('./server');
var router = require('./router');
var handler = require('./handler');

var handle = {};
handle["/"] = handler.home;
handle['/home'] = handler.home;
handle['/review'] = handler.review;
handle['/api/v1/records'] = handler.api_records;

server.startServer(router.route, handle);
```

```
// server.js

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

function startServer(route, handle) {
    var onRequest = function(request, response) {
        console.log('Request received ' + request.url);
        route(handle, request.url, response);
    }
```

```
    var server = http.createServer(onRequest);

    server.listen(3000, '127.0.0.1');
    console.log('Server started on localhost port 3000');
}

module.exports.startServer = startServer;
```

```
// router.js

var fs = require('fs');

function route(handle, pathname, response) {
    console.log('Routing a request for ' + pathname);
    if (typeof handle[pathname] === 'function') {
        handle[pathname](response);
    } else {
        response.writeHead(200, { 'Content-Type': 'text/html' });
        fs.createReadStream(__dirname + '/404.html',
'utf8').pipe(response);
    }
}

module.exports.route = route;
```

```
// handler.js

var fs = require('fs');

function home(response) {
    response.writeHead(200, { 'Content-Type': 'text/html' });
    fs.createReadStream(__dirname + '/index.html', 'utf8').pipe(response);
}

function review(response) {
    response.writeHead(200, { 'Content-Type': 'text/html' });
    fs.createReadStream(__dirname + '/review.html',
'utf8').pipe(response);
}

function api_records(response) {
    response.writeHead(200, { 'Content-Type': 'application/json' });
    var jsonObj = {
        name: "itbaizhan"
    };
    response.end(JSON.stringify(jsonObj));
}

module.exports = {
```

```
    home: home,
    review: review,
    api_records: api_records
}
```

13.1.12 使用Get或POST发送数据

我们常用的请求方式有很多，但是其中get和post是最常用的，那么如何区分是get请求还是post请求呢?

```
// server.js

var http = require('http');
var url = require('url');
var querystring = require('querystring');

function startServer(route, handle) {
    var onRequest = function(request, response) {
        var pathname = url.parse(request.url).pathname;
        console.log('Request received ' + pathname);
        var data = [];
        request.on("error", function(err) {
            console.error(err);
        }).on("data", function(chunk) {
            data.push(chunk);
        }).on('end', function() {
            if (request.method === "POST") {
                data = Buffer.concat(data).toString();
                route(handle, pathname, response,
querystring.parse(data));
            } else {
                var params = url.parse(request.url, true).query;
                route(handle, pathname, response, params);
            }
        });
    }

    var server = http.createServer(onRequest);

    server.listen(3000, '127.0.0.1');
    console.log('Server started on localhost port 3000');
}

module.exports.startServer = startServer;
```

```
// router.js

var fs = require('fs');

function route(handle, pathname, response,params) {
```

```
    console.log('Routing a request for ' + pathname);
    if (typeof handle[pathname] === 'function') {
        handle[pathname](response,params);
    } else {
        response.writeHead(200, { 'Content-Type': 'text/html' });
        fs.createReadStream(__dirname + '/404.html',
'utf8').pipe(response);
    }
}

module.exports.route = route;
```

```
// handler.js

var fs = require('fs');

function home(response) {
    response.writeHead(200, { 'Content-Type': 'text/html' });
    fs.createReadStream(__dirname + '/index.html', 'utf8').pipe(response);
}

function review(response) {
    response.writeHead(200, { 'Content-Type': 'text/html' });
    fs.createReadStream(__dirname + '/review.html',
'utf8').pipe(response);
}

function api_records(response, params) {
    response.writeHead(200, { 'Content-Type': 'application/json' });
    response.end(JSON.stringify(params));
}

module.exports = {
    home: home,
    review: review,
    api_records: api_records
}
```

13.1.13 npm命令

npm 为你和你的团队打开了连接整个 JavaScript 天才世界的一扇大门。它是世界上最大的软件注册表，每星期大约有 30 亿次的下载量，包含超过 600000 个 包（package） （即，代码模块）。来自各大洲的开源软件开发者使用 npm 互相分享和借鉴。包的结构使您能够轻松跟踪依赖项和版本。

我们使用npm也避免了重复造轮子的问题

安装依赖

我们需要第三方依赖，可以直接通过npm进行下载

```
npm install express
```

cnpm镜像

npm是在远程仓库下载，我们知道他的仓库并不在国内，所以，我们需要找一个国内的仓库镜像

```
npm install -g cnpm --registry=https://registry.npm.taobao.org
```

package.json

因为node_modules文件依赖文件太多，而且他也不属于我们的源代码，所以我们在上传源代码的时候并不会上传这个文件夹，那么别人如何知道我们安装过了哪些包呢?

```
npm init
```

初始化一个package.json文件

```
{
  "name": "1",
  "version": "1.0.0",
  "description": "",
  "main": "app.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "node server.js"
  },
  "author": "",
  "license": "ISC"
}
```

然后在安装依赖的时候。我们需要在命令行上加入--save或者--sage-dev

```
npm install --save express
npm install --save-dev gulp
```

上述两种方式，区别在于--save是生产环境，--save-dev是开发环境

```
"dependencies": {
    "express": "^4.16.2"
},
"devDependencies": {
```

```
    "gulp": "^3.9.1"
},
```

有了上述的描述，我们删除node_modules别人就可以根据描述来进行安装了

```
npm install
```

scripts脚本

我们还有一个scripts脚本可以使用，一个项目的入口文件不是固定的，所以如果别人拿到你的代码，不知道你的入口文件，则无法运行你的项目，下面的脚本可以有效的解决这个问题

```
"scripts": {
    "start": "node app.js"
},
```

13.1.14 nodemon

nodemon是一种工具，可以自动检测到目录中的文件更改时通过重新启动应用程序来调试基于node.js的应用程序。

```
npm install -g nodemon
```

有了这个命令，我们就不再需要每次修改完毕代码进行重启了，他可以检测文件的改变而自动重启

```
nodemon app.js
```

当然。nodemon不仅有以上功能，他还有很多配置，但是对于目前我们的需求来说不太需要，在这里不做叙述