[TOC]

Nodejs

了解Nodejs

Nodejs是2009由Ryan Dahl推出的运行在服务端的 JavaScript（类似于java,php,python,.net等服务端语言），基于Google的V8引擎，V8引擎执行Javascript的速度非常快，性能非常好。

版本

> 在命令窗口中输入 `node -v` 可以查看版本

- 0.x 完全不技术 ES6
- 4.x 部分支持 ES6 特性
- 5.x 部分支持ES6特性（比4.x多些），属于过渡产品，现在来说应该没有什么理由去用这个了
- 6.x 支持98%的 ES6 特性
- 8.x 支持 ES6 特性
- 10.x 最新稳定版本

安装与配置

1. 下载安装文件
2. 下载完后进行安装，建议安装到默认路径（注意不要有中文路径）

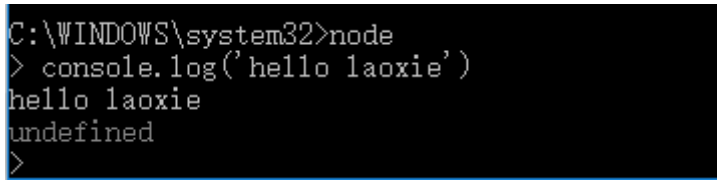
3. 配置环境变量（安装时勾选）
4. 在命令窗口中输入 `node -v`，如果正常显示版本号则表示安装成功

Nodejs的使用

REPL(Read Eval Print Loop 交互式解释器)

> 类似于在浏览器的开发人员工具的控制台，一般用于在服务端执行一些简单的javascript语句

- 进入REPL：node并回车
- 退出：两次 Ctrl + C



执行js文件代码

REPL 只适用于一些简单的 Javascript 语法，对于稍复杂的程序，可以直接写到 js 文件当中，并执行以下命令

```
node hello.js
```

不同于前端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
- 进程的管理
- 网络通信
- ……

NPM(Node Package Manager包管理)

Node.js 的包管理器 npm，是全球最大的开源库生态系统，随着nodejs一起安装（ `npm -v` 查看安装版本）

模块化开发

为了让Nodejs的文件可以相互调用，Nodejs提供了一个简单的模块系统，一个文件即一个模块，采用同步的commonJS规范

模块分类

> 模块系统是 Nodejs 最基本也是最常用的。一般情况模块可分为四类：

- 原生模块（Nodejs内置模块）
- 文件模块（json文件等）
- 第三方模块
- 自定义模块

创建自定义模块

```
//hello.js

function hello(){
    return 'hello laoxie';
}

//对外暴露接口
module.exports = hello;
```

导出模块

> 如果没有这句话，引入模块时 就会得到 空对象

- module.exports

  > 对外暴露单个接口，一个模块中只能有一个 module.exports，多个会被覆盖。

- exports

  > 在一个模块中暴露多个exports接口

```
//person.js
function setName(){
    return 'laoxie'
}

function setAge(){
    return 18
}

// 对外博陆接口
exports.setName = setName;
exports.setAge = setAge;
```

引入模块require()

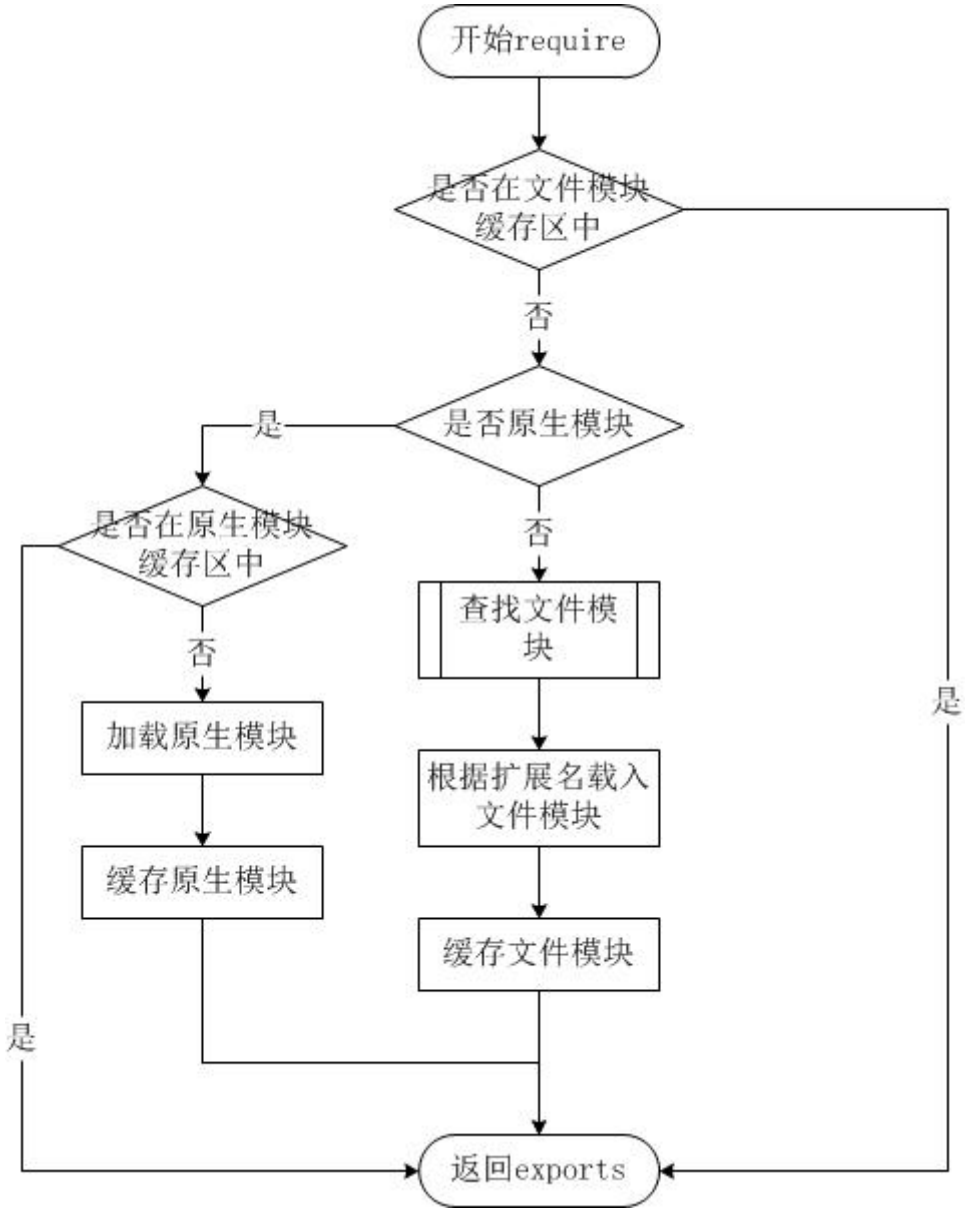
> 引入模块，使用nodejs内置的require()方法（同步的commonJS规范）

```
//page.js

//得到一个对象，包含暴露的setName,setAge方法
let person = require('./person.js');

// 既然是得到对象，也可以直接解构
let {setName,setAge} = require('./person.js');
```

- require 方法中的文件查找策略



原生模块

> 原生模块不需要安装，直接引入使用

http 模块

> 利用nodejs创建http服务器

```
//引入原生模块
var http = require('http');
http.createServer(function(request, response){
    response.end('服务器启动成功，端口为8080');
}).listen(8080);
```

- 常用属性/方法
  - request.url
  - request.method
  - request.on()
  - response.write()
  - response.writeHead()
  - response.end()
- 参数处理之GET请求

> get请求的参数在url地址中，所以需要对url进行处理，需要用到url模块（详情请查看url模块 和 querystring模块）

```
var http = require('http');
var url = require('url');
http.createServer(function(req, res){
    //请求地址通过req.url获取
    //获取url中?后的参数：query
```

```
    var params = url.parse(req.url, true).query;
    res.end(params);
}).listen(3000);
```

- 参数处理之POST请求

> 不同于 GET 请求，POST 请求不能通过 url 进行获取，需要用到请求体（request body）的事件进行监听获取，需要用到querystring模块（querystring模块）

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

http.createServer(function(req, res){
    // 定义了一个post变量，用于暂存请求体的信息
    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 = querystring.parse(post);
    });
}).listen(3000);
```

url 模块

> 向服务器发起请求的 url 地址都是字符串类型，url 所包含的信息也比较多，如：协议、主机名、端口、路径、参数、锚点等，直接操作字符串会相对麻烦，使用Nodejs 的 url 原生模块可轻松解决这一问题

- url常用属性
  - href： 解析前的完整原始 URL，协议名和主机名已转为小写

    - protocol： 请求协议，小写
    - host： url 主机名，包括端口信息，小写
    - hostname: 主机名，小写
    - port: 主机的端口号
    - pathname: URL中路径，下面例子的 /one
    - search: 查询对象，即：queryString，包括之前的问号"?"
    - path: pathname 和 search的合集
    - query: 查询字符串中的参数部分（问号后面部分字符串），或者使用 querystring.parse() 解析后返回的对象
    - hash: 锚点部分（即："#"及其后的部分）

- url.parse(url, boolean)：把url信息转成对象
    - url：字符串格式的 url
    - boolean：是否把url参数转为对象，
        - false(默认)：参数为字符串
        - true：将参数转转对象

```
var url = require('url');
//第二个参数为 true => {a: 'index', t: 'article', m: 'default'}
var params = url.parse('http://www.laoxie.com/one?a=index&t=article&m=default#laoxie', true);
//params.query 为一个对象
console.log(params.query);
//第二个参数为 false
params = url.parse('http://www.laoxie.com/one?a=index&t=article&m=default#laoxie', false);
//params.query 为一个字符串 => ?a=index&t=article&m=default
console.log(params.query);
```

- url.format(params)：对象转字符串，url.parse的反过程
- url.resolve(): 以一种 Web 浏览器解析超链接的方式把一个目标 URL 解析成相对于一个基础 URL

```
var url = require('url');
url.resolve('http://laoxie.com/', '/one')// 'http://laoxie.com/one'
```

querystring模块： 查询字符串处理模块

> 用于处理url地址中的参数信息

- parse()：字符串转对象

```
let qs = require('querystring');
let str = 'firstname=laoxie&url=http%3A%2F%2Flaoxie.com&lastname=xie&passowrd=123456';
let param = qs.parse(param);//{firstname:"laoxie", url:"http://laoxie.com", lastname: 'xie', passowrd: 123456};
```

- stringify()：对象转字符串

```
let qs = require('querystring');

let obj = {firstname:"laoxie", url:"http://laoxie.com", lastname: 'xie', passowrd: 123456};
//将对象转换成字符串
let param = qs.stringify(obj);//firstname=laoxie&url=http%3A%2F%2Flaoxie.com&lastname=xie&passowrd=123456
```

path模块

> 用于处理文件与目录的路径

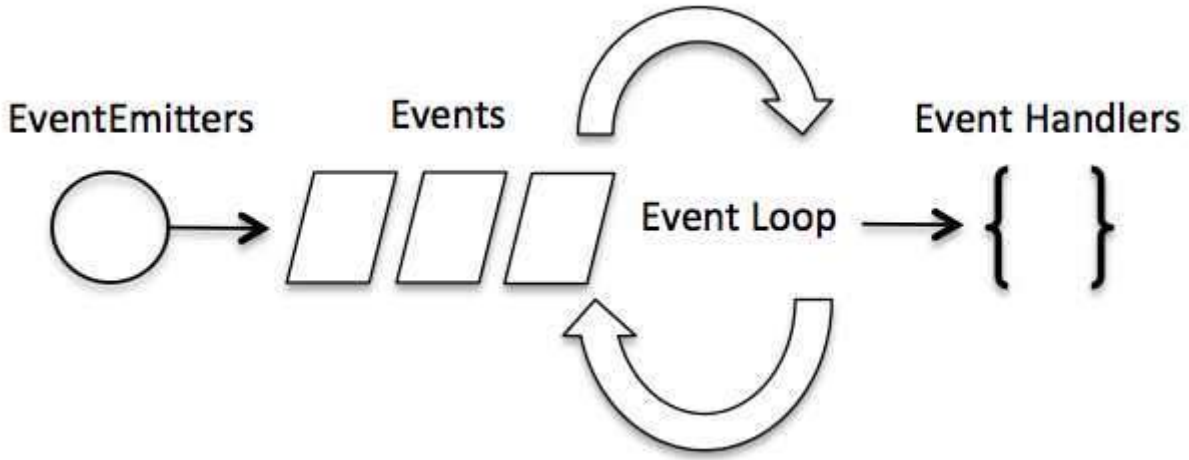
- path.parse(path) 返回一个对象，其中对象的属性表示 path 的元素
  - dir
  - root
  - base
  - name

    - ext
- path.basename(path) 返回 path 的最后一部分（同path.parse()中的base）
- path.dirname(path) 返回 path 的目录名（同path.parse()中的dir）
- path.extname(path) 返回 path 的扩展名，（同path.parse()中的ext）
- path.join([...paths]) 使用平台特定的分隔符把所有 path 片段连接到一起，并规范化生成的路径
- path.normalize(path) 规范化指定的 path，并处理 '..' 和 '.' 片段

events 模块

> 事件模块在 Nodejs 中有很多好处，但用法却可以很简单

- Nodejs 是单进程单线程应用程序，但是通过事件和回调支持并发，所以性能非常高。
- Nodejs的每一个API都是异步的，并作为一个独立线程运行，使用异步函数调用，并处理并发。
- Nodejs 基本上所有的事件机制都是用设计模式中观察者模式实现。
- Nodejs 单线程类似进入一个while(true)的事件循环，直到没有事件观察者退出，每个异步事件都生成一个事件观察者，如果有事件发生就调用该回调函数.



- 用法
  - 实例化一个事件实例 new events.EventEmitter();
  - 在实例对象上定义事件 on(eventname, function(){})
  - 通地 emit 方法触发事件 emit(eventname)

```
// 引入 events 模块
var events = require('events');
// 创建 eventEmitter 对象
var eventEmitter = new events.EventEmitter();

// 绑定事件及事件的处理程序
eventEmitter.on('connection', function(){
    console.log('连接成功。');
    // 触发 data_received 事件
    eventEmitter.emit('data_received');
});

// 使用匿名函数绑定 data_received 事件
eventEmitter.on('data_received', function(){
    console.log('数据接收成功。');
});

//用 eventEmitter 对象的 emit 方法来调用事件
eventEmitter.emit('connection');
console.log("程序执行完毕。");
```

fs 模块

> Buffer的理解：一个专门存放二进制数据的缓存区，类似与一个数组。 在处理像TCP流或文件流时，必须使用到二进制数据，js语言自身没有二进制数据类型，所以Nodejs推出Buffer类型用于解决文件读取与前后端数据传输的问题

- readFile(path,callback): 读取文件内容（异步）

```
var fs = require('fs');
fs.readFile('about.txt', function (err, data) {
    // err: 无法读取文件时的错误信息，读取成功时为null
    // data: 文件信息（Buffer）
    if (err) {
        return console.error(err);
    }
    console.log("异步读取: " + data.toString());
});
```

- readFileSync(path) : 读取文件内容（同步）

```
var fs = require('freadFileSyncs');
var data = fs.readFileSync('demoFile.txt');
console.log("同步读取: " + data.toString());
```

- writeFile(): 写入文件内容（覆盖）

```
var fs = require('fs');
//每次写入文本都会覆盖之前的文本内容
fs.writeFile('about.txt', '我们不一样',  function(err) {
    if (err) {
        return console.error(err);
    }
    console.log("数据写入成功！");
});
```

- appendFile(): 写入文件内容（追加）

```
var fs = require('fs');
fs.appendFile('about.txt', '有啥不一样', function (err) {
    if (err) {
        return console.error(err);
    }
    console.log("数据写入成功！");
});
```

- 图片读取（http服务器）

> 图片读取不同于文本，因为文本读出来可以直接用 console.log() 输出，但图片则需要在浏览器中显示，所以需要先搭建 web 服务，然后以字节方式读取的图片在浏览器中渲染。

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

//图片读取是以字节的方式
var content =  fs.readFileSync('001.jpg', "binary");

http.createServer(function(request, response){
  //图片在浏览器的渲染因为没有 img 标签，所以需要设置响应头为 image
  response.writeHead(200, {'Content-Type': 'image/jpeg'});
  response.write(content, "binary");
  response.end();
}).listen(3000,function(){
  console.log('Server running at http://localhost:3000/');
});
```

- 目录操作
  - 创建一个目录：fs.mkdir()
  - 列出目录内容：fs.readdir()

    - 重命名目录或文件：fs.rename()
    - 删除目录与文件：fs.rmdir(),fs.unlink()
    - 得到文件与目录的信息：fs.stat()
        - stats.isDirectory() 当前是否为文件夹
        - stats.isFile() 当前是否为文件
    - 检查文件或目录是否存在
        - fs.access()
        - fs.existsSync(path)

- stream流

    > Stream 是一个抽象接口，往往用于打开大型的文本文件，创建一个读取操作的数据流。Node 中有很多对象实现了这个接口。如http 服务器发起请求的request 对象就是一个 Stream。所有的 Stream 对象都是 EventEmitter 的实例

    - 所谓大型文本文件，指的是文本文件的体积很大，读取操作的缓存装不下，只能分成几次发送，每次发送会触发一个data事件，发送结束会触发end事件。

    - 读取流（以流的方式读取文件内容）

        - setEncoding()：设置读取编码格式
        - data事件：读取数据中
        - end事件：数据读取完毕

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

        // 创建可读流
        // 以流的方式读取input.txt中的内容
        var readerStream = fs.createReadStream('input.txt');
        // console.log(readerStream);
        ```

```
// 设置编码为 utf8。
readerStream.setEncoding('UTF8');

// 处理流事件 --> data, end, and error
readerStream.on('data', function(chunk) {
data += chunk;
});

readerStream.on('end',function(){
console.log(data);
});

readerStream.on('error', function(err){
console.log(err.stack);
});
```

- 写入流(以流的方式写入文件)
  - write()方法：写入内容方法
  - end()方法：标记写入结束
  - finish事件：写入完成后执行

```
//创建一个可以写入的流，写入到文件 output.txt 中

var fs = require("fs");
var data = '中国';

// 创建一个可以写入的流，写入到文件 output.txt 中
// var writerStream = fs.createWriteStream('output.txt', {'flags': 'a'}); //追加文本
var writerStream = fs.createWriteStream('output.txt');

// 使用 utf8 编码写入数据
writerStream.write(data,'UTF8');

// 标记文件末尾
```

```
writerStream.end();

// 处理流事件 --> data, end, and error
writerStream.on('finish', function() {
    console.log("写入完成。");
});

writerStream.on('error', function(err){
   console.log(err.stack);
});
```

- 管道流pipe

> 管道提供了一个输出流到输入流的机制。通常我们用于从一个流中获取数据并将数据传递到另外一个流中。我们把文件比作装水的桶，而水就是文件里的内容，我们用一根管子(pipe)连接两个桶使得水从一个桶流入另一个桶，这样就慢慢的实现了大文件的复制过程。

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

// 创建一个可读流
var readerStream = fs.createReadStream('input.txt');

// 创建一个可写流
var writerStream = fs.createWriteStream('output.txt');

// 管道读写操作
// 读取 input.txt 文件内容，并将内容写入到 output.txt 文件中
readerStream.pipe(writerStream);
```

- 链式流（多个pipe）

> 链式是通过连接输出流到另外一个流并创建多个对个流操作链的机制。链式流一般用于管道操作。接下来我们就是用管道和链式来压缩和解压文件。

- 压缩

```
var fs = require("fs");
//压缩和解压的模块
var zlib = require('zlib');

// 压缩 input.txt 文件为 input.txt.gz
// 以流的方式读取文本
fs.createReadStream('input.txt')
  .pipe(zlib.createGzip()) //把读取出来的文本调用压缩模块进行压缩
  .pipe(fs.createWriteStream('input.txt.zip'));//把压缩好的流进行保存
```

- 解压

```
var fs = require("fs");
//压缩和解压的模块
var zlib = require('zlib');

fs.createReadStream('input.txt.zip')
  .pipe(zlib.createGunzip())
  .pipe(fs.createWriteStream('input1.txt'));
```

> PS: 多文件的压缩与解压：tar模块

路由

> 在 BS 架构中，路由的概念都是一样的，可理解为根据客户端请求的 URL 响应不同的内容，一般根据 URL 中的路径、参数、锚点等信息进行响应。

应用场景

- 获取商品列表信息

    - 请求类型：get
    - url地址：http://localhost:88/list
- 获取单个商品信息
    - 请求类型：get
    - url地址：http://localhost:88/goods
    - 请求参数：id
- 修改商品信息
    - 请求类型：post
    - url地址：http://localhost:88/goods
    - 请求参数：
        - id
        - name
        - price
        - ...
- 如何访问不存在的路由则显示404页面
    - 404页面

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

http.createServer((request, response) => {
    // 处理url地址
    let urlObj = url.parse(request.url, true);

    // 获取访问路径
    let pathname = urlObj.pathname;

    // 获取请求类型
    let method = request.method.toUpperCase();
```

```
// 获取参数
let params = urlObj.query;

// POST请求处理方式
if(method == 'POST'){
    let postData = '';
    request.on('data', (_data) => {
        postData += _data;
    })
    request.on('end', () => {
        postData = qs.parse(postData);
        let result = {};
        switch(pathname){
            case '/login':
                //连接数据库，实现登陆逻辑
                result = {status: true};
                break;
            case '/register':
                //连接数据库，实现注册逻辑
                result = {status: true};
                break;
            default :
                result = {status: false, message: '没有对应的请求'};
                break;
        }
        response.end(JSON.stringify(result));
    })
}

// get请求处理方式
else if(method === 'GET'){
    let result = {};
    switch(pathname){
        case '/students':
            //连接数据库，获取学生信息
            result = {status: true, data: [], params};
```

```
                    break;
                case '/orders':
                    //连接数据库，获取订单信息
                    result = {status: true, data: [], params};
                    break;
                default :
                    result = {status: false, message: '没有对应的请求', params};
                    break;
            }
            response.end(JSON.stringify(result))
        }
    }).listen(88);
```

【案例】

- 利用原生模块搭建静态资源服务器
- 编写数据接口，商品列表、商品详情页数据的读取与修改

【练习】

- 制作本地相册
- 实现文件的上传、在线压缩与解压