# 一 fs模块

fs模块即：文件模块，是Node的核心模块，提供了操作文件的一些API。

文件模块针对同一个业务提供了 异步、同步两种操作方式，比如读取文件：readFile() readFileSync()。

## 1 打开文件open()

fs.open(path,flags,[mode],callback);

path：要打开的文件的路径（注意相对路径要 ./开头，为了兼容Linux和Win）

flags：打开文件的方式 读/写

mode：设置文件的模式 读/写/执行 分别对应 4/2/2

callback：回调

err：打开失败后保存的错误对象，如果成功，err为null

fd：被打开文件的标识

const fs = require('fs');  
fs.open('./1.html','r',function (err,fd) {  
 if(err){  
 console.log('err');  
 } else {  
 console.log('success');  
 console.log(fd); *//输出 3* }  
});

## 2 读取文件fs.readFile()

Node目前不支持中文GBK,GB2312编码，需要第三方模块iconv或者iconv-lite模块，其中iconv仅支持linux。

Node中读取文件一般使用fs.read()方法，该方法从一个特定的文件描述fd中读取数据

fs.read(fd,buffer,offset,length,position,callback);

fd：通过fs.open()方法返回的文件描述符

buffer：数据写入的缓冲区

offset：缓冲区写入的写入偏移量

length：要从文件中读取的字节数

position：文件读取的起始位置，如果为null，就会从当前文件指针的位置读取；

callback：有err，bytesRead，buffer三个参数，bytesRead表示读取的字节数，buffer为缓冲区对象。

const fs = require('fs');  
*//可选参数-字符集：{'charset':'utf8'}*fs.readFile('./1.html',function (err,data) {  
 if(err){  
 throw err;  
 }  
 *//data是个Buffer* console.log(data.toString());  
});

## 3 写入文件fs.wirteFile()

异步的将数据写入一个文件，如果文件不存在则创建，如果文件存在，则替换。data参数可以是一个string，也可以是一个buffer。

fs.writeFile(filename,data,[options],callback);

fs.appendFile() 该方法也可以将字符串或者缓冲区内容写入文件，

## 4 监听文件 fs.watch()

fs.wathc(filename,[options],[listener]);

观察置顶路径的改变，filename可以是文件或者目录，返回的对象是：fs.FSWatcher。第二个参数为布尔值，默认为true，代表只要文件被监听，就继续执行。

const fs = require('fs');  
fs.watch('./1.html',function (ev,fn) {  
 console.log(ev);  
 if(fn){  
 console.log(fn + '发生了改变');  
 } else {  
 console.log('...');  
 }  
});

## 5 文件元信息 fs.stat()

fs.stat(path,[callback]);

const fs = require('fs');  
fs.stat('./1.html',function (err,data) {  
 console.log(data);  
});

输出的结果是：

|  |
| --- |
| Stats {  dev: 139364547,  mode: 33206,  nlink: 1,  uid: 0,  gid: 0,  rdev: 0,  blksize: undefined,  ino: 11540474045137144,  size: 151,  blocks: undefined,  atimeMs: 1518349011803.3728,  mtimeMs: 1518349011803.3728,  ctimeMs: 1518349012232.9893,  birthtimeMs: 1518347359655.3713,  atime: 2018-02-11T11:36:51.803Z,  mtime: 2018-02-11T11:36:51.803Z,  ctime: 2018-02-11T11:36:52.233Z,  birthtime: 2018-02-11T11:09:19.655Z  } |

## 6其他常见API

fs.appendFile(name,data,[options],callback);

将数据添加到文件末尾（文件不存在则创建）

fs.unlink(path,callback); 删除一个文件

fs.exists(path,callback); 判断文件/目录是否存在

fs.rename(path1,path2,[callback]);重命名文件

fs.chomd(path,mode,[callback]); 修改权限,如mode取值为：’777’

获取文件元信息

fs.mkdir([path,[mode],callback); 创建文件夹

fs.readdir([path,callback); 读取文件夹

fs.rmdir([path,callback); 删除文件夹

# 二 Buffer类

### 1 Buffer实例的创建

Buffer类用于操作二进制数据，类内的参数用于分配长度，分配后不能更改。

new Buffer(size); //分配size个长度的数组

new Buffer(array) //分配特定数组的长度

new Buffer(string,[encoding]); //将字符串转换为16进制，编码可选，默utf8

buffer[index]:获取或设置在指定index索引未知的8位字节内容

|  |
| --- |
| **var** bf = **new** Buffer(**'test'**,**'utf-8'**); ***console***.log(bf); *//此length长度和字符串的长度有区别，指buffer的bytes大小* **for**(**var** i = 0; i < bf.**length**; i++){  ***console***.log(bf[i].toString(16));  ***console***.log(String.fromCharCode(bf[i])); } |

### 2 Buffer实例方法

buf.write(string,[offset],[length],[encoding]):根据参数offset，将参数string数据写入buffer

buf.toString([encoding],[length]):返回一个解码的string类型

buf.toJSON():返回一个JSON表示的Buffer实例，JSON.stringify将会默认调用来字符串序列化这个Buffer实例

buf.slice([start],[end]):返回一个新的buffer，这个buffer和老的buffer引用相同的内存地址

buf.copy(targetBuffer,[targetStart],[sourceStart],[sourceEnd])：进行buffer的拷贝，拷贝不会影响老的buffer。

|  |
| --- |
| **var** str = **'test'**; ***console***.log(**new** Buffer(str)); **var** bf = **new** Buffer(3); *//只能写入前三位;test* bf.write(str); ***console***.log(bf); |

### 3 Buffer的常用方法1

let buf = new Buffer('abcc--ddder--qwerqwer');  
  
*//查找*console.log(buf.indexOf('--')); *//输出4  
//截取*console.log(buf.slice(1,3).toString()); *//输出bc，不包含最后一个  
//切分 Buffer没有split方法  
//手写split方法，这样写的原则是 如果未来Buffer自带了，则使用自带的*Buffer.prototype.splite = Buffer.prototype.splite || function (o) {  
 let arr = [];  
 let current = 0;  
 let n = 0;  
 while ((n = this.indexOf(o,current)) != -1){  
 arr.push(this.slice(current,n));  
 current = n + o.length;  
 }  
 arr.push(this.slice(current));  
 return arr;  
};  
*//测试*console.log(buf.splite('--').map(buffer=>buffer.toString()));

### 4 Buffer的常用方法2

/\*  
对buffer进行查询  
 \*/  
let buffer = new Buffer('acc-=-ddder-=-qwer');  
  
*//查找indexOf 第一次出现-=-*console.log(new Buffer('-=-'));  
console.log(buffer.indexOf('-=-')); *//字符串被自动转成了buffer  
  
//截取slice 截取-=-第一次出现后的字符串*console.log(buffer.slice(6).toString()); *//一直截取到最后  
//注意：slice不包含结束位*console.log(buffer.slice(6,11).toString());  
  
*//切分split*

Buffer没有splite 切割操作，自己写一个：



# 三 Stream流

使用readFile方法是极其占用内存的（一次加载大文件），读取完毕后再发送，读和发没有同时进行，这样不但影响性能，也会让网络应用、磁盘应用之间永远只有一个在疯狂活动，应该读取一点，发送一点，这样网络、磁盘都在积极使用。

使用流可以解决上述问题：

读取流 fs.createReadStream req

写入流 fs.createWriteStream res

读写流 压缩、加密

const fs = require('fs');  
let rs = fs.createReadStream('./1.jpg'); *//读取流*let ws = fs.createWriteStream('./2.jpg'); *//写入流*rs.pipe(ws);

在服务器上应用：

const http = require('http');  
const fs = require('fs');  
http.createServer((req,res)=>{  
 let rs = fs.createReadStream('./1.jpg');  
 rs.pipe(res);  
}).listen(8000);

注意：上述案例中，读取1.jpgg，写入到2.jpg，流是有方向的，从读取端流向写入端

流的结束事件是finish不是end。其实req，res也是流，因为只有流才有on事件。

# 四 压缩模块 zlib

使用zlib模块压缩文件：

const zlib = require('zlib');  
const fs = require('fs');  
let rs = fs.createReadStream('./1.jpg');  
let ws = fs.createWriteStream('./2.gz');  
let gz = zlib.createGzip();  
rs.pipe(gz).pipe(ws);  
ws.on('finish',()=>{  
 console.log('完成');  
});

使用gzip与stream来优化服务端代码：使用下列代码书写后，响应的文件大小会极大缩小。

const http = require('http');  
const zlib = require('zlib');  
const fs = require('fs');  
http.createServer((req,res)=>{  
 let rs = fs.createReadStream(`文件地址`);

//没有该句会变成下载  
 res.setHeader('content-encoding','gzip');   
 let gz = zlib.createGzip();  
 rs.pipe(gz).pipe(res);  
 rs.on('error',err=>{  
 res.writeHeader(404);  
 res.write('Not found');  
 res.end();  
 });  
}).listen(8000);

# 五 静态资源

## 1 简单实现

将public文件夹映射成为一个静态文件目录。

const http = require('http');  
const fs = require('fs');  
const url = require('url');  
const path = require('path'); *//用来判断请求的文件的扩展名*http.createServer((req,res)=>{  
 let pathname = url.parse(req.url,true).pathname;  
 if(pathname == '/'){  
 res.end('hello world');  
 } else if(pathname.substring(0,8) == '/public/'){  
 fs.readFile('.' + pathname,(err,data)=>{  
 if(err){  
 res.writeHead(404,{'Content-Type':'text/html;charset=UTF8'});  
 res.end('404');  
 } else {  
 let mime = getMime(path.extname(pathname));  
 res.writeHead(200,{'Content-Type':mime});  
 res.end(data);  
 }  
 });  
 } else {  
 res.end('other page');  
 }  
}).listen(3000);  
  
function getMime(extname) {  
 switch (extname){  
 case '.html':  
 return 'text/html';  
 break;  
 case '.jpg':  
 return 'image/jpg';  
 break;  
 case '.png':  
 return 'image/png';  
 break;  
 }  
}

这里的getMIME是不严谨的，实际项目开发中要加载一个配置文件来读取对应的配置。

## 2 静态资源缓存

每次请求服务器的静态资源，都会造成IO上的浪费，那么我们可以使用缓存来优化性能。当浏览器中有缓存副本时，不确定该副本是否有效，会生成一个get请求，在该请求的header中包含一个if-modified-since时间参数。如果服务器端文件在这个时间参数后修改过了，服务器发送全部文件给客户端，如果没有，则返回304状态码，并不发送整个文件。

如果确定该副本有效，客户端不会发送GET请求。判断有效的方法是：服务端响应头上带有expires头。

Expires：是一个毫秒值，如果该值小于当前时间，则不缓存。

const http = require('http');  
const url = require('url');  
const fs = require('fs');  
  
http.createServer(function (req,res) {  
 let pathname = url.parse(req.url).pathname;  
 if(pathname == '/favicon.ico'){  
 return;  
 } else {  
 dealStatic(req,res,pathname);  
 }  
}).listen(80);  
  
  
function dealStatic(req,res,pathname) {  
 console.log('pathname=' + pathname);  
 let realPath = \_\_dirname + '\\public\\' + pathname.toString().substr(1);  
 console.log('realPath=' + realPath);  
 if(pathname == '/' || pathname == '/index'){  
 res.writeHead(200);  
 res.end('hi');  
 } else {  
 fs.exists(realPath,function (exists) {  
 if(!exists){ *//文件不存在* res.writeHead(404,{'Content-Type':'text/plain'});  
 res.end('404');  
 }else {  
  
 let mimeString = pathname.substring(pathname.lastIndexOf('.') + 1);  
 console.log('mimeString=' + mimeString);  
 let mimeType = null;  
 switch (mimeString){  
 case 'css': mimeType = 'text/css';  
 break;  
 case 'png': mimeType = 'image/png';  
 break;  
 default: mimeType = 'text/plain';  
 }  
  
 let fileInfo = fs.statSync(realPath);

*//获取服务器文件最后修改时间*  
 let lastModified = fileInfo.mtime.toUTCString();  *//设置7天缓存存在时间*

let CACHETIME = 60\*60\*24\*7; /\*  
 客户端请求时间 大于 Expires（date值）发送新请求  
 客户端请求时间 小于 Expires（date值）读取本地缓存  
 \*/  
 let date = new Date();

*//当前时间+缓存时间*  
 date.setTime(date.getTime() + CACHETIME\*1000); if(req.headers['if-modified-since'] && lastModified == req.headers['if-modified-since']){  
 console.log('执行了读取本地缓存');  
 res.writeHead(304,'Not Modified');  
 res.end('304');  
 } else {  
 fs.readFile(realPath,function (err,file) {  
 if(err){  
 res.writeHead(500);  
 res.end(err);  
 } else {  
 *//没有缓存，设置缓存* console.log('执行了发送服务器文件');  
 res.setHeader('Expires',date.toUTCString());  
 res.setHeader('Cache-Control','max-age=' + CACHETIME);  
 res.setHeader('Last-Modified',lastModified);  
 res.writeHead(200,{'Content-Type':mimeType});  
 res.end(file);  
 }  
 });  
 }  
 }  
 });  
 }  
}