第4章 文件系统操作

Node.js开发实战教程

内容导航

Contents

4.1 Node.js的文件系统基础

- 了解fs模块
- 同步文件操作与异步文件操作
- 文件路径与文件模式
- 文件系统标志与文件描述符

4.2 文件与目录基本操作

4.3 文件系统的高级操作

4.4 实战演练——操作表格文件

Section 00

【学习目标】

（1）了解fs模块；
（2）掌握文件系统的基础知识。

Section 01

【fs模块简介】

- fs模块提供一组文件操作API用于模仿标准POSIX函数与文件系统进行交互。导入该模块：

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

- fs模块共有5个类，分别是fs.Dirent（指示文件类型）、fs.FSWatcher（监视文件）、fs.ReadStream（读取流）、fs.WriteStream（写入流）和fs.Stats（文件的信息）。
- 名称具有“Sync”后缀的方法为同步方法，不具有该后缀的方法为异步方法。
- fs模块支持FS常量，包括文件可访问性、文件复制、文件打开、文件类型和文件模式的常量。

Section 02

【同步文件操作与异步文件操作】

- fs模块包含同步文件操作和异步文件操作两大类API方法，大多数文件操作提供同步和异步两种方式。
- 异步操作方法或函数的最后一个参数总是一个回调函数，第一个参数总是错误信息。

异步删除文件示例

```
fs.unlink('/tmp/hello.txt', (err) => {
  if (err) throw err;
  console.log('删除完成！');
});
```

同步删除文件示例

```
try{
    fs.unlinkSync('/tem/hello.txt');  //同步删除文件
    console.log('已成功删除！');
} catch (err) {
    //处理错误
}
```

Section 02

【同步文件操作与异步文件操作】

- 多个异步函数在同一层次执行，是无法保证执行顺序的，因此容易出错。
- 注意异步函数执行顺序。下表代码用于先修改文件名再查看文件属性。

多个异步函数在同一层次执行无法保证执行顺序

```
fs.rename('/tmp/hello.txt', '/tmp/world.txt', (err) => {
  if (err) throw err;
  console.log('重命名完成！ ');
});
fs.stat('/tmp/world.txt', (err, stats) => {
  if (err) throw err;
  console.log(`文件属性: ${JSON.stringify(stats)}`);
});
```

将一个函数放在另一个函数的回调函数中执行（回调嵌套）

```
fs.rename('/tmp/hello.txt', '/tmp/world.txt', (err) => {
  if (err) throw err;
  fs.stat('/tmp/world.txt', (err, stats) => {
    if (err) throw err;
    console.log(`文件属性: ${JSON.stringify(stats)}`);
  });
});
```

Section 02

【同步文件操作与异步文件操作】

- 异步操作方法支持同时处理多个任务，阻塞少、性能高、速度快。
- 同步操作将阻塞整个进程，直到所有任务完成。
- 同步操作方法不会延迟执行，当CPU完成准备时，同步操作方法会立即执行，用户只需等待它完成任务，但是在等待期间会阻塞其他任务的运行。
- 文件同步与异步操作比较的示例

程序代码

```
const fs = require('fs');//导入fs模块
fs.readFile('demo.txt', function (err, data) {// 异步读取
   if (err) {
       return console.error(err);
   }
   console.log("异步读取: " + data.toString());
});
var data = fs.readFileSync('demo.txt'); // 同步读取
console.log("同步读取: " + data.toString());
console.log("程序执行完毕。");
```

执行结果

```
C:\nodeapp\ch04\fstest>node sync-async.js
同步读取: 人生并不像火车要通过每个站似的经过每一个生活阶段。人生总是直向前行走，从不留下什么。
程序执行完毕。
异步读取: 人生并不像火车要通过每个站似的经过每一个生活阶段。人生总是直向前行走，从不留下什么。
```

Section 03

【文件路径】

▸▸▸ Node.js的文件路径表示方法

文件路径参数	说明	示例
字符串	用字符串（UTF-8字符序列）形式表示的路径可分为相对路径和绝对路径两种类型； 在POSIX系统上使用的绝对路径是从根目录（/）开始的； 在Windows上Node.js要遵循单驱动器工作目录的规则，绝对路径是从驱动器符号开始，而且路径分割使用反斜杠	POSIX系统：/open/some/file.txt Windows系统：C:\\temp\\myfile.html
Buffer对象	主要用于某些POSIX操作系统	Buffer.from('/open/some/file.txt')
URL对象	自Node.js 7.6.0开始新增的文件路径类型。目前Node.js仅支持使用file协议的URL 对象，file协议的URL始终是绝对路径。不同操作系统平台对URL对象的解析会有差别	const fileUrl = new URL('file:///tmp/hello'); fs.readFileSync(fileUrl);

Section 03

【文件路径】

▸▸▸ 使用path模块处理不同风格的路径

- 获取目录名：path.dirname(path)。
- 获取扩展名：path.extname(path)。
- 规范化路径：path.normalize(path)。
- 解析路径：path.resolve([...paths])。
- 连接路径：path.join([...paths])。

Section 04

【文件模式】

- 文件模式即文件权限，fs模块遵循POSIX文件操作规范，使用类Unix系统的权限表示方法。
- 访问权限

权限分配	所有者权限			所属组权限			其他用户权限		
权限项	读	写	执行	读	写	执行	读	写	执行
字符表示	r	w	x	r	w	x	r	w	x
数字表示	4	2	1	4	2	1	4	2	1

- Windows系统的文件权限默认是可读、可写、不可执行，所以权限位数字表示为0o666（八进制表示），转换十进制表示为438。

Section 05

【文件系统标志】

标志	说明
a	以追加模式打开文件。如果文件不存在则创建
ax	类似a标志，但是如果文件路径存在，则文件追加失败
a+	以读取追加模式打开文件，如果文件不存在则创建
ax+	类似a+标志，但是如果文件路径存在，则文件读取追加失败
as	以同步方式打开文件用于追加。如果文件不存在，则创建该文件
as+	以同步方式打开文件用于读取和追加。如果文件不存在，则创建该文件
r	以读取模式打开文件。如果文件不存在则抛出异常
r+	以读取和写入模式打开文件。如果文件不存在则抛出异常
rs	以同步方式读取文件。如果文件不存在则抛出异常
rs+	以同步方式读取和写入文件。它会指示操作系统绕过本地的文件系统缓存，除非需要，否则不建议使用此标志
w	以写入模式打开文件，如果文件不存在则创建
wx	类似w标志，但是如果文件路径存在，则文件写入失败
w+	以读写模式打开文件，如果文件不存在则创建
wx+	类似w标志，但是如果文件路径存在，则文件读写失败

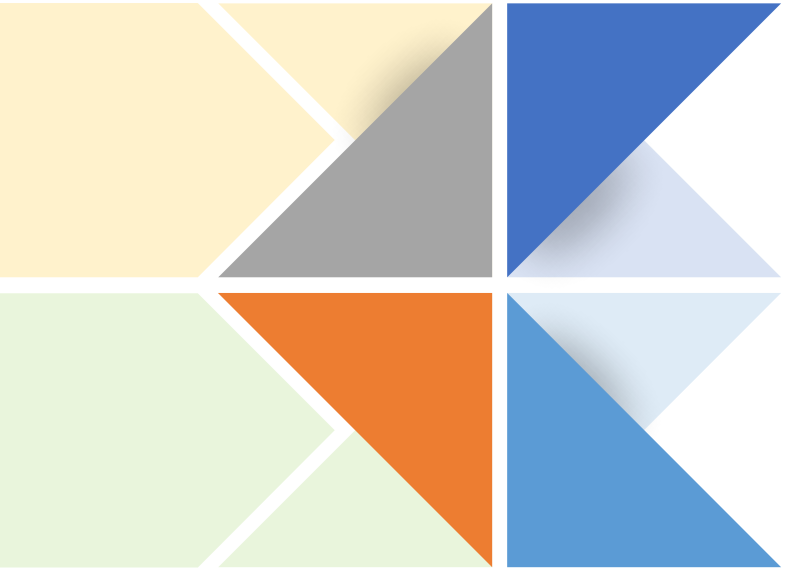
Section 06

【文件描述符】

- 操作系统为每个打开的文件分配一个名为文件描述符的数字，文件操作使用这些文件描述符来识别与追踪每个特定的文件。
- 在Node.js中每操作一个文件，文件描述符都是会自动递增的，文件描述符一般从3开始，因为前面的0、1、2是比较特殊的描述符，分别表示process.stdin(标准输入)、process.stdout(标准输出)和process.stderr(错误输出)。
- fs.open()方法用于分配新的文件描述符。一旦被分配，文件描述符可用于从文件读取数据、向文件写入数据或请求关于文件的信息。
- 大多数操作系统会限制在任何给定时间内可能打开的文件描述符的数量，因此操作完成时关闭描述符至关重要。

内容导航

Contents



Section 00

【学习目标】

（1）了解文件和目录的基本操作；

（2）掌握文件和目录基本操作的编程方法。

Section 01

【打开文件】

- 异步打开文件的用法：

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

  参数说明：

  - path:指定文件路径
  - flags:指定文件系统标志
  - mode:可选参数，用于设置文件模式，但仅限于创建文件的情况。文件创建默认权限为0o666(可读可写)。在windows系统上只能设置写权限。
  - callback:回调函数，带有两个参数err和fd，用于返回错误信息和表示文件描述符的整数。

- 同步打开文件的用法：

  fs.openSync(path, flags[, mode])

Section 01

【打开文件】

- 以异步方式打开文件的示例

```
const fs = require('fs');
console.log("准备打开文件");
//r+表示以读写模式打开，fd为返回的文件描述符
fs.open('demo.txt', 'r+', function(err, fd) {
    if (err) {
        return console.error(err);
    }
    console.log("文件打开成功！ ");
});
```

Section 02

【获取文件信息】

- 异步方式获取文件信息的用法：

  fs.stat(path[, options], callback)

  - path:文件路径
  - options:选项。使用bigint作为选项来指示返回的fs.Stats对象中的数值是否应为bigint类型，默认为false。
  - callback:回调函数。带有两个参数err和stats表示错误信息和fs.Stats对象。

- fs.Stats类提供的方法

方法	说明
stats.isBlockDevice()	如果是块设备则返回true，否则返回false
stats.isCharacterDevice()	如果是字符设备则返回true，否则返回false
stats.isDirectory()	如果是目录则返回true，否则返回false
stats.isFIFO()	如果是FIFO则返回true，否则返回false。FIFO是UNIX中的一种特殊类型的命令管道
stats.isFile()	如果是文件则返回true，否则返回false
stats.isSocket()	如果是套接字则返回true，否则返回false

Section 02

【获取文件信息】

- 异步读取文件信息的示例

程序代码

```
const fs = require('fs');
fs.stat("demo.txt", function (err, stats) {
    if(err) throw err;
    console.log(stats);//显示返回的fs.Stats对象
    console.log("读取文件信息成功！");
    // 检测文件类型
    console.log("是否为文件(isFile) ? " + (stats.isFile() ? '是':'否'));
    console.log("是否为目录(isDirectory) ? " + (stats.isDirectory() ? '是':'否'));
    // 读取文件属性
    console.log("文件大小：" + stats.size);
    console.log("创建时间：" + stats.birthtime);
});
```

执行结果

```
C:\nodeapp\ch04\fstest>node file_stat.js
Stats {
  dev: 2163518143,
  mode: 33206,
  nlink: 1,
  ......
  birthtime: 2019-05-06T02:41:34.882Z }
读取文件信息成功！
是否为文件(isFile) ? 是
是否为目录(isDirectory) ? 否
文件大小：63
创建时间： Mon May 06 2019 10:41:34 GMT+0800 (GMT+08:00)
```

Section 03 【读取文件】

▶▶▶ 读取指定的部分数据

➢ 异步方式读取文件指定数据的用法：

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

参数说明如下：

- fd:类型为integer，通过fs.open()方法返回的文件描述符，用于指定读取的文件。
- buffer:数据写入的缓冲区，类型为Buffer，也可以是TypedArray或DataView。
- offset:缓冲区写入的写入偏移量，类型为integer。
- length:要从文件中读取的字节数，类型为integer。
- position:文件读取的起始位置，类型为integer。如果position的值为null，则会从当前文件指针的位置读取，并更新文件位置。如果position值是整数，则文件位置将保持不变。
- callback:回调函数，有3个参数err、bytesRead和buffer，分别表示错误信息、读取的字节数和Buffer对象。

Section 03 【读取文件】

▶▶▶ 读取指定的部分数据

- 读取文件时需要先打开文件，再从fs.open()返回的文件描述符指向的文件中读取数据。
- 示例

```
const fs = require('fs');
fs.open("demo.txt","r",function(err,fd)
{
    if(err) throw err;
    console.log("打开文件成功。");
    var buf = Buffer.alloc(24);//分配缓冲区
    //开始读取字节
    fs.read(fd,buf,0,buf.length,0,function(err,bytes)  {
        if(err){
            fs.closeSync(fd);//关闭文件
            return console.log(err);
        }
        console.log("读取的字节长度:"+bytes);
        // 仅输出读取的字节
        if(bytes > 0){
            console.log("打开文件后读取的buff内容:"+buf.slice(0,
bytes).toString());
        }
        fs.closeSync(fd);//关闭文件
    });
});
```

Section 03

【读取文件】

▸▸▸ 读取文件的全部内容

- 异步方式读取文件全部内容的用法：

  fs.readFile(path[, options], callback)

- 示例

程序代码	执行结果

const fs = require("fs");
fs.readFile('demo.txt', (err, data) => {
  if (err) throw err;
  console.log(data);
});
	C:\nodeapp\ch04\fstest>node file_readall.js
	<Buffer ef bb bf e8 bf 99 e6 98 af e4 b8 80 e4 b8 aa e7 94 a8 e4 ba 8e e6 b5 8b
	e8 af 95 e8 af bb e5 8f 96 e6 96 87 e4 bb b6 e5 86 85 e5 ae b9 e7 9a 84 e6 96 ..
	. >

const fs = require("fs");
fs.readFile('demo.txt','utf8', (err, data) => {
  if (err) throw err;
  console.log(data);
});
	C:\nodeapp\ch04\fstest>node file_readall.js
	这是一个用于测试读取文件内容的文本文件。

Section 04 【写入文件】

▶▶▶ 覆盖式写入

- 异步方式覆盖式写入文件的用法：

  fs.writeFile(file, data[, options], callback)

- 示例

```
const fs = require('fs');
console.log("准备写入文件");
fs.writeFile('input.txt', '这样用于写入文件测试的内容', function(err) {
    if(err) throw err;
    console.log("数据写入成功！");
});
```

Section 04 【写入文件】

▸▸▸ 追加式写入

- 异步方式追加式写入文件的用法：

  fs.appendFile(path, data[, options], callback)

- 示例

```
const fs= require( ‘fs’ );
console.log("准备追加文件！ ");
fs.appendFile('input.txt','这是用于追加文件测试的内容', (err) => {
   if(err) throw err;
   console.log("数据追加成功！ ");
});
```

Section 04

【写入文件】

▸▸▸ 将数据写入文件指定的位置

- 写入Buffer对象的用法：

  fs.write(fd, buffer[, offset[, length[, position]]], callback)

- 写入字符串的用法：

  fs.write(fd, string[, position[, encoding]], callback)

Section 05

【文件的其他基本操作】

文件操作	用法
关闭文件	fs.close(fd, callback)
检查文件的可访问性	fs.access(path[, mode], callback)
文件重命名	fs.rename(oldPath, newPath, callback)
删除文件	fs.unlink(path, callback)
复制文件	fs.copyFile(src, dest[, flags], callback)

Section 06

【目录的基本操作】

▸▸▸ 创建目录

- 异步创建目录的用法：

```
fs.mkdir(path[, options], callback)
```

- 示例

```
fs.mkdir('/tmp/a/apple', { recursive: true }, (err) => {
  if (err) throw err;
});
```

▸▸▸ 删除目录

- 异步删除目录的用法：

```
fs.rmdir(path, callback)
```

Section 06

【目录的基本操作】

▸▸▸ 读取目录内容

- 异步读取目录内容的用法：

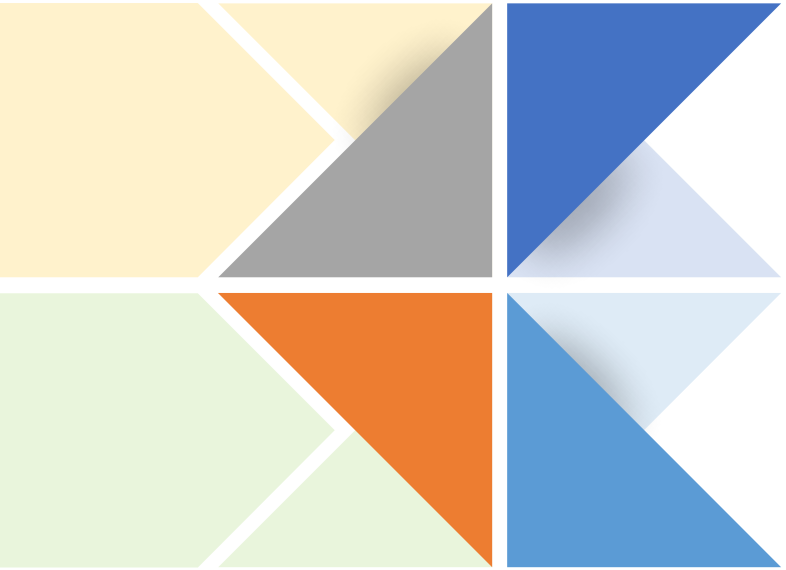
  fs.readdir(path[, options], callback)

- 示例

```
const fs= require('fs');
console.log("查看上一级目录的内容");
fs.readdir("../",function(err, files){
   if (err) throw err;
   files.forEach( function (file){
       console.log( file );
   });
});
```

内容导航

Contents



Section 00

【学习目标】

（1）使用Node.js文件操作的流接口；
（2）编程实现文件遍历和监视；
（3）掌握JSON格式的文件操作。

Section 01

【使用文件操作的流接口】

▸▸▸ 从流中读取数据

- 创建一个可读流的用法：

```
fs.createReadStream(path[, options])
```

- 通过流接口读取文件内容的示例

```
const fs = require('fs');
var data = '';
var readStream = fs.createReadStream('demo.txt'); // 创建可读流
readStream.setEncoding('UTF8'); // 设置字符编码格式为 utf8
//以下处理流事件data（当有数据可读时触发）、end（无数据可读时触发）和 error（发生错误时触发）
readStream.on('data', (chunk) => {
    data += chunk;
});
readStream.on('end', () =>{
    console.log(data);
});
readStream.on('error', (err) => {
    console.log(err.stack);
});
```

Section 01

【使用文件操作的流接口】

▶▶▶ 将数据写入流

➢ 创建一个可写流的用法：

```
fs.createWriteStream(path[, options])
var fs = require("fs");
var data =   '测试写入';
var writerStream = fs.createWriteStream('output.txt');
writerStream.write(data,'UTF8'); // 使用 utf8 编码写入数据
writerStream.end(); // 标记文件末尾
writerStream.on('finish', function() {
    console.log("写入完成。");
});
writerStream.on('error', function(err){
   console.log(err.stack);
});
console.log("程序执行完毕");
```

Section 01

【使用文件操作的流接口】

▸▸▸ 流的管道操作

```
const fs= require('fs');
var readStream = fs.createReadStream('src_file');//打开源文件准备读取
var writeStream = fs.createWriteStream('dest_file');//使用新的数据覆盖目标文件
readStream.pipe(writeStream);  //从源文件读取时，将读取的数据写入到目标文件
```

Section 01

【使用文件操作的流接口】

▸▸▸ 流的链式操作

- 压缩文件示例

```
const fs= require('fs');
const zlib = require('zlib');
fs.createReadStream('demo.txt')
  .pipe(zlib.createGzip())
  .pipe(fs.createWriteStream('demo.txt.gz'));
console.log("文件压缩完成!");
```

- 解压缩文件示例

```
const fs= require('fs');
const zlib = require('zlib');
fs.createReadStream('demo.txt.gz')
  .pipe(zlib.createGunzip())
  .pipe(fs.createWriteStream('demo.txt'));
console.log("文件解压缩完成!");
```

Section 02

【文件遍历】

▶▶▶ 文件遍历的实现思路

（1）使用fs.readdir()方法获取目录下的文件列表。

（2）遍历此文件列表，并使用fs.stat()方法获取文件信息。

（3）根据获取的信息判断是文件还是目录。

（4）如果是文件，打印其绝对路径。

（5）如果是目录，返回第1步重新从开始获取文件列表直至第5步操作。由于目录级数未知，所以可以使用递归方法来解决。

Section 02

【文件遍历】

▶▶▶ 文件遍历示例

```
const fs = require('fs');
const path = require('path');
var filePath = path.resolve('../');//这里遍历的是上一级目录
fileTraverse (filePath);  //调用文件遍历函数
/*定义文件遍历函数，参数filePath是需要遍历的文件路径 */
function fileTraverse (filePath){
    //根据文件路径读取文件，返回文件列表
    fs.readdir(filePath,function(err,files){
        if(err){
            console.warn(err)
        }else{
            //遍历读取到的文件列表
            files.forEach(function(file){
                var fullPath = path.join(filePath,file); //获取当前文件的绝对路径
                //根据文件路径获取文件信息，返回一个fs.Stats对象
```

Section 02

【文件遍历】

▶▶▶ 文件遍历示例

```
fs.stat(fullPath,function(eror,stats){
                if(eror){
                    console.warn('获取文件信息失败');
                }else{
                    if(stats.isFile()){   //如果是文件
                        console.log(fullPath);
                    }
                    if(stats.isDirectory()){//如果是目录
                        fileTraverse (fullPath);//递归，继续遍历该目录下的文件
                    }
                }
            })
        });
    }
});
}
```

Section 03

【文件监视】

▸▸▸ fs.watch()

➢ 用法：

fs.watch(filename[, options][, listener])

➢ 示例：监听上一级目录下文件变化

```
const fs = require('fs');
fs.watch('../', (eventType, filename) => {
    if (eventType=='rename'){
        console.log('发生重命名');
    }
    if (eventType=='change'){
        console.log('发生修改');
     }
    if (filename) {
        console.log(`文件名：  ${filename}`);
    } else {
      console.log('文件名未提供');
    }
});
```

Section 03

【文件监视】

fs.watchFile()

- 用法：

```
fs.watchFile(filename[, options], listener)
```

- 示例：监听某文件变化

```
const fs = require('fs');
//每隔1秒时间检测文件变化
fs.watchFile('demo.txt', {interval: 1000}, function (curr, prev) {
    console.log('当前的文件修改时间:' + curr.mtime + '当前的文件大小:' + curr.size);
    console.log("上一次文件修改时间:" + prev.mtime + '之前的文件大小:' + prev.size);
    fs.unwatchFile('demo.txt');    // 停止监听
});
```

选择fs.watch()还是fs.watchFile()

- fs.watch()方法更高效，但其API在各个平台上并非完全一致，在某些情况下不可用。
- 如果底层功能由于某些原因不可用，则fs.watch()方法将无法运行。此时就要考虑选用fs.watchFile()方法。
- fs.watchFile()方法较慢且不太可靠，但它是跨平台的，因而在网络文件系统中更可靠。

Section 04

【操作JSON文件】

▶▶▶ 从JSON文件中读取数据

```
const fs = require('fs');
fs.readFile('person.json',function(err,data){
    if(err) throw err;
    var person =JSON.parse(data.toString());//将字符串转换为JSON对象
    //对JSON数据进行处理
    for(var i = 0; i < person.length;i++){
       console.log('姓名：'+ person[i].name +'     年龄：'+ person[i].ages );
    }
});
```

Section 04

【操作JSON文件】

▶▶▶ 往JSON文件中添加数据

```
const fs = require('fs');
var newobj = { "name":"刘强", "ages":21 }
fs.readFile('person.json',function(err,data){
    if(err) throw err;
    var person =JSON.parse(data.toString());//将字符串转换为JSON对象
    person.push(newobj);//将新的对象加到数组对象中
    var str = JSON.stringify(person); //将JSON对象转换成字符串重新写入JSON文件中
    fs.writeFile('person.json',str,  function(err) {
       if(err) throw err;
       console.log("数据写入成功！ ");
    });
});
```

Section 04

【操作JSON文件】

▶▶▶ 更改JSON文件中的指定数据

```
for(var i = 0; i < person.length;i++){
    if(person[i].name == '张红'){
        person[i].ages = 22;
        break;
    }
}
var str = JSON.stringify(person); //将JSON对象转换成字符串重新写入JSON文件中
```

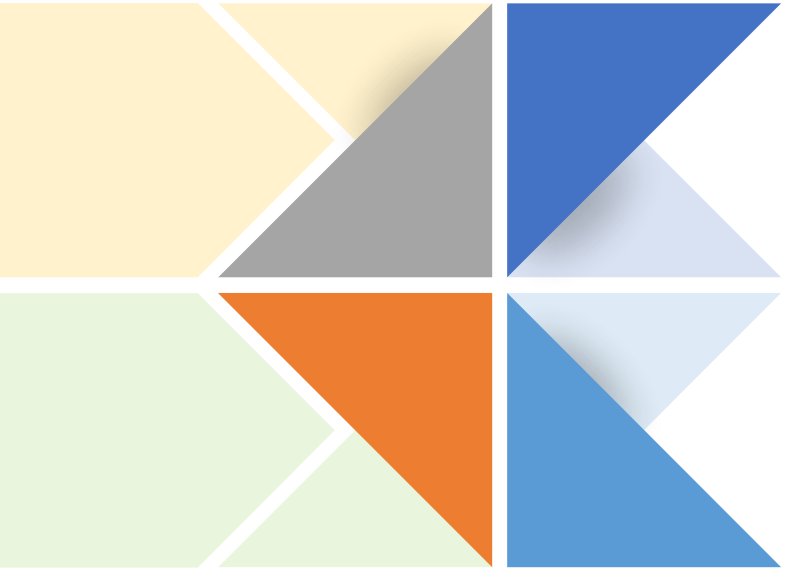
Section 05

【读取GBK格式的文件】

```
const fs = require('fs');
const iconv = require('iconv-lite');   // 加载编码转换模块
fs.readFile('gbksample.txt', function(err, data){
    if (err) throw err;
    console.log(data); // 输出字节数组
    var text = iconv.decode(data, 'gbk'); // 把数组转换为gbk中文
    console.log(text); // 输出中文内容
    //按默认的UTF8字符编码写入
    fs.writeFile('utf8sample.txt',text, function(err){
       if (err) throw err;
    });
});
```

内容导航

Contents



Section 00

【学习目标】

（1）了解CSV文件格式；
（2）使用CSV包编程处理CSV文件；
（3）掌握基于CSV格式的数据导入导出操作。

Section 01

【使用CSV包的准备工作】

- CSV包的子包
  - csv-generate：用于创建CSV字符串和JavasScript对象。
  - csv-parse：将CSV文本转换为数组或对象。
  - stream-transform：一个转换框架。
  - csv-stringify：将记录转换为CSV文本，相当于将数组或对象序列转换为CSV文本。
- 安装CSV包

Section 02

【从CSV文件中读取并解析数据】

▸▸▸ 使用流API

```
const fs = require('fs');
const parse = require('csv-parse'); //加载csv-parse模块
const output = []; //定义一个用于输出的数组
// 创建解析器，这里使用逗号作为CSV文件的字段分隔符
const parser = parse({
  delimiter: ','
});
// 有可读数据时，读取数据将其添加到输出数组
parser.on('readable', function(){
  let record;
  while (record = parser.read()) {
    output.push(record);
  }
})
```

Section 02

【从CSV文件中读取并解析数据】

▸▸▸ 使用流API

```
// 错误处理
parser.on('error', function(err){
  console.error(err.message);
})
// 数据解析完毕，这里获取的是一个JSON数组，可以进一步处理
parser.on('end', function(){
    console.log(output);
})
// 将从CSV文件读取的数据通过管道发送给上述可读流（解析器）
fs.createReadStream(__dirname+'/csvsample.csv').pipe(parser)
```

Section 02

【从CSV文件中读取并解析数据】

▸▸▸ 使用回调API

```
const fs = require('fs');
const parse = require('csv-parse');
fs.readFile('csvsample.csv', function (err, data) {
    if (err) throw err;
    parse(data,{delimiter: ','},function(err1,output){
      console.log(output);
    });
});
```

Section 02

【从CSV文件中读取并解析数据】

▸▸▸ 组合使用流API和回调API

- 输出流的用法

```
const stream = parse(input, options)
```

- 输入流的用法

```
const stream = parse(options, callback)
```

- 示例：组合方式读取CSV文件

```
const fs = require('fs');
const parse = require('csv-parse');
var parser = parse({delimiter: ','},function(err, data){
    console.log(data);
});
fs.createReadStream(__dirname+'/csvsample.csv').pipe(parser);
```

▸▸▸ 使用同步API

- 使用同步API时加载模块

```
const parse = require('csv-parse/lib/sync');
```

Section 03

【将记录转换为CSV文本并保存到CSV文件中】

- csv-stringify是一个将记录转换为CSV文本的转换器。
- 示例：写入CSV文件

```
const fs = require('fs');
const stringify = require('csv-stringify'); //加载csv-stringify模块
const data = [];
//创建转换流
const stringifier = stringify({
  delimiter: ':',record_delimiter:'windows'
});
//有可读数据时逐行处理
stringifier.on('readable', function(){
  let row;
  while(row = stringifier.read()){
    data.push(row);
  }
});
```

Section 03

【将记录转换为CSV文本并保存到CSV文件中】

- 示例：写入CSV文件

```
stringifier.on('error', function(err){
  console.error(err.message);
});
// 数据转换完毕，可以进一步处理，这里将其保存到CSV文件
stringifier.on('finish', function(){
  fs.writeFile('newcsv.csv', data,  function(err) {
    if (err) {
        return console.error(err);
    }
 });
})
// 将记录写到流
stringifier.write(['root','x','0','0','root','/root','/bin/bash']);
stringifier.write(['zxp','x','1022','1022','','/home/zxp','/bin/bash']);
stringifier.end();// 关闭流
```

Section 04

【对CSV数据进行转换处理】

- stream-transform提供一个简单的对象转换框架用于实现Node.js转换流API。
- 示例：转换CSV文件中的数据

```
const fs = require('fs');
const parse = require('csv-parse'); //加载csv-parse模块
const transform = require('stream-transform');
const output = [];
const transformer = transform(function(data){
  data.push(data.shift());
  return data;
})
transformer.on('readable', function(){
  while(row = transformer.read()){
    output.push(row);
  }
})
```

Section 04

【对CSV数据进行转换处理】

- 示例：转换CSV文件中的数据

```
transformer.on('error', function(err){
  console.error(err.message);
})
transformer.on('finish', function(){
  console.log(output);
})
fs.createReadStream(__dirname+'/csvsample.csv')
.pipe(parse({
    delimiter: ','
  }))
.pipe(transformer);
```

Section 05

【组合使用多个子模块来处理CSV数据】

- csv-generate用于创建CSV字符串和JavasScript对象以实现Node.js可读流API，可产生随机的或用户定义的数据集合。
- 可将它与上述子模块组合使用数据。

```
const csv = require('csv');
csv.generate  ({seed: 1, length: 20})
    .pipe(csv.parse())
    .pipe(csv.transform (function(record){
        return record.map(function(value){
        return value.toUpperCase()
    })}))
    .pipe(csv.stringify ())
    .pipe(process.stdout);
```

本章小结

Summary

本章的主要内容是Node.js文件系统的相关知识和操作技能，主要介绍了文件的同步操作和异步操作、文件路径、文件模式、文件系统标志和文件描述符的基本知识，讲解了文件与目录的基本操作方法，示范了文件流式操作、文件遍历和文件监视的实现方法以及实际应用中会涉及的JSON文件和GBK格式文件的操作方法，最后说明了如何使用第三方模块读写和转换CSV文件。

Thank you