link: null
title: 珠峰架构师成长计划
description: path是node中专门处理路径的一个核心模块
keywords: null
author: null
date: null
publisher: 珠峰架构师成长计划
stats: paragraph=121 sentences=328, words=1974

1. fs模块

- 在Node.js中，使用fs模块来实现所有有关文件及目录的创建、写入及删除操作。
- 在fs模块中，所有的方法都分为同步和异步两种实现。
- 具有 sync后缀的方法为同步方法，不具有 sync后缀的方法为异步方法。

2. 整体读取文件

2.1 异步读取

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

- options
    - encoding
    - flag flag 默认 = 'r'

2.2 同步读取

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

3. 写入文件

3.1 异步写入

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

- options
    - encoding
    - flag flag 默认 = 'w'
    - mode 读写权限，默认为0666

```
let fs = require('fs');
fs.writeFile('./1.txt',Date.now()+'\n',{flag:'a'},function(){
  console.log('ok');
});
```

3.2 同步写入

```
fs.writeFileSync(file, data[, options])
```

3.3 追加文件

> fs.appendFile(file, data[, options], callback)

```
fs.appendFile('./1.txt',Date.now()+'\n',function(){
  console.log('ok');
})
```

3.4 拷贝文件

```
function copy(src,target){
  fs.readFile(src,function(err,data){
    fs.writeFile(target,data);
  })
}
```

4. 从指定位置处开始读取文件

4.1 打开文件

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

- FileDescriptor 是文件描述符
- FileDescriptor 可以被用来表示文件
- in -- 标准输入(键盘)的描述符
- out -- 标准输出(屏幕)的描述符
- err -- 标准错误输出(屏幕)的描述符

```
fs.open('./1,txt','r',0600,function(err,fd){});
```

4.2 读取文件

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

```
const fs=require('fs');
const path=require('path');
fs.open(path.join(__dirname,'1.txt'),'r',0o666,function (err,fd) {
    console.log(err);
    let buf = Buffer.alloc(6);
     fs.read(fd,buf,0,6,3,function(err, bytesRead, buffer){
       console.log(bytesRead);
       console.log(buffer===buf);
       console.log(buf.toString());
     })
})
```

4.3 写入文件

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

```
const fs=require('fs');
const path=require('path');
fs.open(path.join(__dirname,'1.txt'),'w',0o666,function (err,fd) {
    console.log(err);
    let buf=Buffer.from('珠峰培训');
     fs.write(fd,buf,3,6,0,function(err, bytesWritten, buffer){
       console.log(bytesWritten);
       console.log(buffer===buf);
       console.log(buf.toString());
     })
})
```

4.4 同步磁盘缓存

> fs.fsync(fd,[callback]);

4.5 关闭文件

> fs.close(fd,[callback]);

```
let buf = Buffer.from('珠峰培训');
fs.open('./2.txt', 'w', function (err, fd) {
  fs.write(fd, buf, 3, 6, 0, function (err, written, buffer) {
    console.log(written);
    fs.fsync(fd, function (err) {
      fs.close(fd, function (err) {
          console.log('写入完毕!')
        }
      );
    });
  })
});
```

4.6 拷贝文件

```
let BUFFER_SIZE=1;
const path=require('path');
const fs=require('fs');
function copy(src,dest,callback) {
    let buf=Buffer.alloc(BUFFER_SIZE);
    fs.open(src,'r',(err,readFd)=>{
        fs.open(dest,'w',(err,writeFd) => {
            !function read() {
                fs.read(readFd,buf,0,BUFFER_SIZE,null,(err,bytesRead) => {
                    bytesRead&&fs.write(writeFd,buf,0,bytesRead,read);
                });
            }()
        })
    });
}
copy(path.join(__dirname,'1.txt'),path.join(__dirname,'2.txt'),()=>console.log('ok'));
```

5 目录操作

5.1 创建目录

> fs.mkdir(path[, mode], callback)

> 要求父目录必须存在

5.2 判断一个文件是否有权限访问

> fs.access(path[, mode], callback)

```
fs.access('/etc/passwd', fs.constants.R_OK | fs.constants.W_OK, (err) => {
  console.log(err ? 'no access!' : 'can read/write');
});
```

5.3 读取目录下所有的文件

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

5.4 查看文件目录信息

> fs.stat(path, callback)

- stats.isFile()
- stats.isDirectory()
- atime(Access Time)上次被读取的时间。
- ctime(State Change Time)：属性或内容上次被修改的时间。
- mtime(Modified time)：档案的内容上次被修改的时间。

5.5 移动文件或目录

```
fs.rename(oldPath, newPath, callback)
```

5.6 删除文件

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

5.7 截断文件

```
fs.ftruncate(fd[, len], callback)
```

```
const fd = fs.openSync('temp.txt', 'r+');

fs.ftruncate(fd, 4, (err) => {
  console.log(fs.readFileSync('temp.txt', 'utf8'));
});
```

5.8 监视文件或目录

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

```
let fs = require('fs');
fs.watchFile('1.txt', (curr, prev) => {

  if(Date.parse(prev.ctime)==0){
    console.log('创建');
  }else if(Date.parse(curr.ctime)==0){
    console.log('删除');
  }else if(Date.parse(prev.ctime) != Date.parse(curr.ctime)){
    console.log('修改');
  }
});
```

6 递归创建目录

6.1 同步创建目录

```
let fs=require('fs');
let path=require('path');
function makepSync(dir) {
    let parts=dir.split(path.sep);
    for (let i=1;ilet parent=parts.slice(0,i).join(path.sep);
        try {
            fs.accessSync(parent);
        } catch (error) {
            fs.mkdirSync(parent);
        }

    }
}
```

6.2 异步创建目录

```
function makepAsync(dir,callback) {
    let parts=dir.split(path.sep);
    let i=1;
    function next() {
        if (i>parts.length)
            return callback&&callback();
        let parent=parts.slice(0,i++).join(path.sep);
        fs.access(parent,err => {
            if (err) {
                fs.mkdir(parent,next);
            } else {
                next();
            }
        });
    }
    next();
}
```

6.3 Async+Await创建目录

```
async function mkdir(parent) {
    return new Promise((resolve,reject) => {
        fs.mkdir(parent,err => {
            if (err) reject(err);
            else resolve();
        });
    });
}

async function access(parent) {
    return new Promise((resolve,reject) => {
        fs.access(parent,err => {
            if (err) reject(err);
            else resolve();
        });
    });
}
async function makepPromise(dir,callback) {
    let parts=dir.split(path.sep);
    for (let i=1;ilet parent=parts.slice(0,i).join(path.sep);
        try {
            await access(parent);
        }catch(err) {
            await mkdir(parent);
        }

    }
}
```

7. 递归删除目录

7.1 同步删除目录(深度优先)

```
let fs=require('fs');
let path=require('path')
function rmSync(dir) {
    try {
        let stat = fs.statSync(dir);
        if (stat.isFile()) {
            fs.unlinkSync(dir);
        } else {
            let files=fs.readdirSync(dir);
            files
                .map(file => path.join(dir,file))
                .forEach(item=>rmSync(item));
            fs.rmdirSync(dir);
        }
    } catch (e) {
        console.log('删除失败!');
    }
}
rmSync(path.join(__dirname,'a'));
```

7.2 异步删除非空目录(Promise版)

```
function rmPromise(dir) {
    return new Promise((resolve,reject) => {
        fs.stat(dir,(err,stat) => {
            if (err) return reject(err);
            if (stat.isDirectory()) {
                fs.readdir(dir,(err,files) => {
                    let paths = files.map(file => path.join(dir,file));
                    let promises = paths.map(p=>rmPromise(p));
                    Promise.all(promises).then((() => fs.rmdir(dir,resolve)));
                });
            } else {
                fs.unlink(dir,resolve);
            }
        });
    });
}
rmPromise(path.join(__dirname,'a')).then(() => {
    console.log('删除成功');
})
```

7.3 异步串行删除目录(深度优先)

```
function rmAsyncSeries(dir,callback) {
    setTimeout(() => {
        fs.stat(dir,(err,stat) => {
            if (err) return callback(err);
            if (stat.isDirectory()) {
                fs.readdir(dir,(err,files) => {
                    let paths = files.map(file => path.join(dir,file));
                    function next(index) {
                        if (index>=files.length) return fs.rmdir(dir,callback);
                        let current=paths[index];
                        rmAsyncSeries(current,()=>next(index+1));
                    }
                    next(0);
                });
            } else {
                fs.unlink(dir,callback);
            }
        })
    },1000);
}

console.time('cost');
rmAsyncSeries(path.join(__dirname,'a'),err => {
     console.timeEnd('cost');
})
```

7.4 异步并行删除目录(深度优先)

```
function rmAsyncParallel(dir,callback) {
    setTimeout(() => {
        fs.stat(dir,(err,stat) => {
            if (err) return callback(err);
            if (stat.isDirectory()) {
                fs.readdir(dir,(err,files) => {
                    let paths=files.map(file => path.join(dir,file));
                    if (paths.length>0) {
                        let i=0;
                        function done() {
                            if (++i == paths.length) {
                                fs.rmdir(dir,callback);
                            }
                        }
                      paths.forEach(p=>rmAsyncParallel(p,done));
                    } else {
                        fs.rmdir(dir,callback);
                    }
                });
            } else {
                fs.unlink(dir,callback);
            }
        })
    },1000);
}
console.time('cost');
rmAsyncParallel(path.join(__dirname,'a'),err => {
     console.timeEnd('cost');
})
```

7.5 同步删除目录(广度优先)

```
function rmSync(dir){
    let arr=[dir];
    let index=0;
    while (arr[index]) {
        let current=arr[index++];
        let stat=fs.statSync(current);
        if (stat.isDirectory()) {
            let dirs=fs.readdirSync(current);
            arr=[...arr,...dirs.map(d => path.join(current,d))];
        }
    }
    let item;
    while (null != (item = arr.pop())) {
        let stat = fs.statSync(item);
        if (stat.isDirectory()) {
            fs.rmdirSync(item);
        } else {
            fs.unlinkSync(item);
        }
    }
}
```

7.6 异步删除目录(广度优先)

```
function rmdirWideAsync(dir,callback){
    let dirs=[dir];
    let index=0;
    function rmdir() {
        let current = dirs.pop();
        if (current) {
            fs.stat(current,(err,stat) => {
                if (stat.isDirectory()) {
                    fs.rmdir(current,rmdir);
                } else {
                    fs.unlink(current,rmdir);
                }
            });
        }
    }
    !function next() {
        let current=dirs[index++];
        if (current) {
            fs.stat(current,(err,stat) => {
                if (err) callback(err);
                if (stat.isDirectory()) {
                    fs.readdir(current,(err,files) => {
                        dirs=[...dirs,...files.map(item => path.join(current,item))];
                        next();
                    });
                } else {
                    next();
                }
            });
        } else {
            rmdir();
        }

    }();
}
```

8. 遍历算法

- 目录是一个树状结构，在遍历时一般使用深度优先+先序遍历算法
- 深度优先，意味着到达一个节点后，首先接着遍历子节点而不是邻居节点
- 先序遍历，意味着首次到达了某节点就算遍历完成，而不是最后一次返回某节点才算数
- 因此使用这种遍历方式时，下边这棵树的遍历顺序是A > B > D > E > C > F。

```
      A
     / \
    B   C
   / \   \
  D   E   F
```

8.1 同步深度优先+先序遍历#

```
function deepSync(dir){
  console.log(dir);
  fs.readdirSync(dir).forEach(file=>{
      let child = path.join(dir,file);
      let stat = fs.statSync(child);
      if(stat.isDirectory()){
          deepSync(child);
      }else{
          console.log(child);
      }
  });
}
```

8.2 异步深度优先+先序遍历#

```
function deep(dir,callback) {
  console.log(dir);
  fs.readdir(dir,(err,files)=>{
      !function next(index){
          if(index == files.length){
              return callback();
          }
          let child = path.join(dir,files[index]);
          fs.stat(child,(err,stat)=>{
              if(stat.isDirectory()){
                  deep(child,()=>next(index+1));
              }else{
                  console.log(child);
                  next(index+1);
              }
          })
      }(0)
  })
}
```

8.3 同步广度优先+先序遍历#

```
function wideSync(dir){
  let dirs = [dir];
  while(dirs.length>0){
      let current = dirs.shift();
      console.log(current);
      let stat = fs.statSync(current);
      if(stat.isDirectory()){
          let files = fs.readdirSync(current);
          files.forEach(item=>{
              dirs.push(path.join(current,item));
          });
      }
  }
}
```

** 8.4 异步广度优先+先序遍历 #**

```
function wide(dir, cb) {
    console.log(dir);
    cb && cb()
    fs.readdir(dir, (err, files) => {
        !function next(i){
            if(i>= files.length) return;
            let child = path.join(dir,files[i]);
            fs.stat(child,(err,stat)=>{
                if(stat.isDirectory()){
                    wide(child, () => next(i+1));
                } else {
                    console.log(child);
                    next(i+1);
                }
            })
        }(0);
    })
}
wide(path.join(__dirname,'a'));
```

8. path模块

path是node中专门处理路径的一个核心模块

- path.join 将多个参数值字符串结合为一个路径字符串
- path.basename 获取一个路径中的文件名
- path.extname 获取一个路径中的扩展名
- path.sep 操作系统提定的文件分隔符
- path.delimiter 属性值为系统指定的环境变量路径分隔符
- path.normalize 将非标准的路径字符串转化为标准路径字符串 特点：
    - 可以解析 . 和 ..
    - 多个杠可以转换成一个杠
    - 在windows下反杠会转化成正杠
    - 如结尾以杠结尾的，则保留斜杠
- resolve
    - 以应用程序根目录为起点
    - 如果参数是普通字符串，则意思是当前目录的下级目录
    - 如果参数是.. 回到上一级目录
    - 如果是/开头表示一个绝对的根路径

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

console.log(path.normalize('./a///b//..\\c//e//..//'));

console.log(path.join(__dirname,'a','b'));

console.log(path.resolve());
console.log(path.resolve('a','/c'));

console.log(path.relative(__dirname,'/a'));

console.log(path.dirname(__filename));
console.log(path.dirname('./1.path.js'));

console.log(path.basename(__filename));
console.log(path.basename(__filename,'.js'));
console.log(path.extname(__filename));

console.log(path.sep);
console.log(path.win32.sep);
console.log(path.posix.sep);
console.log(path.delimiter);
```

9. flags

符号 含义 r 读文件，文件不存在报错 r+ 读取并写入，文件不存在报错 rs 同步读取文件并忽略缓存 w 写入文件，不存在则创建，存在则清空 wx 排它写入文件 w+ 读取并写入文件，不存在则创建，存在则清空 wx+ 和w+类似，排他方式打开 a 追加写入 ax 与a类似，排他方式写入 a+ 读取并追加写入，不存在则创建 ax+ 作用与a+类似，但是以排他方式打开文件

10. 助记

- r 读取
- w 写入
- s 同步
-
  - 增加相反操作
- x 排他方式
- r+ w+的区别?
  - 当文件不存在时，r+不会创建，而会导致调用失败，但w+会创建。
  - 如果文件存在，r+不会自动清空文件，但w+会自动把已有文件的内容清空。

11. linux权限

文件类型与权限 链接占用的节点(i-node) 文件所有者 文件所有者的用户组 文件大小 文件的创建时间 最近修改时间 文件名称