第七章：查询和读写文件

本章内容：

- 处理文件路径
- 从文件路径萃取信息
- 理解文件描述符
- 使用 fs.stat()获取文件信息
- 打开，读写，关闭文件
- 避免文件描述符泄露

Node 有一组数据流 API，可以像处理网络流那样处理文件，用起来很方便，但是它只允许顺序处理文件，不能随机读写文件。因此，需要使用一些更底层的文件系统操作。

本章覆盖了文件处理的基础知识，包括如何打开文件，读取文件某一部分，写数据，以及关闭文件。

Node 的很多文件 API 几乎是 UNIX（POSIX）中对应文件 API 的翻版，比如使用文件描述符的方式，就像 UNIX 里一样，文件描述符在 Node 里也是一个整型数字，代表一个实体在进程文件描述符表里的索引。

有 3 个特殊的文件描述符——1，2 和 3。他们分别代表标准输入，标准输出和标准错误文件描述符。标准输入，顾名思义，是个只读流，进程用它来从控制台或者进程通道读取数据。标准输出和标准错误是仅用来输出数据的文件描述符，他们经常被用来向控制台，其它进程或文件输出数据。标准错误负责错误信息输出，而标准输出负责普通的进程输出。

一旦进程启动完毕，就能使用这几个文件描述符了，它们其实并不存在对应的物理文件。你不能读写某个随机位置的数据，（**译者注：** 原文是 You can write to and read from specific positions within the file.根据上下文，作者可能少写了个“not”），只能像操作网络数据流那样顺序的读取和输出，已写入的数据就不能再修改了。

普通文件不受这种限制，比如 Node 里，你即可以创建只能向尾部追加数据的文件，还可以创建读写随机位置的文件。

几乎所有跟文件相关的操作都会涉及到处理文件路径，本章先会将介绍这些工具函数，然后再深入讲解文件读写和数据操作

处理文件路径

文件路径分为相对路径和绝对路径两种，用它们来表示具体的文件。你可以合并文件路径，可以提取文件名信息，甚至可以检测文件是否存在。

Node 里，可以用字符串来操处理文件路径，但是那样会使问题变复杂，比如你要连接路径的不同部分，有些部分以 “/” 结尾有些却没有，而且路径分割符在不同操作系统里也可能会不一样，所以，当你连接它们时，代码就会非常罗嗦和麻烦。

幸运的是，Node 有个叫 path 的模块，可以帮你标准化，连接，解析路径，从绝对路径转换到相对路径，从路径中提取各部分信息，检测文件是否存在。总的来说，path 模块其

实只是些字符串处理，而且也不会到文件系统去做验证（path.exists 函数例外）。

路径的标准化

在存储或使用路径之前将它们标准化通常是个好主意。比如，由用户输入或者配置文件获得的文件路径，或者由两个或多个路径连接起来的路径，一般都应该被标准化。可以用 path 模块的 normalize 函数来标准化一个路径，而且它还能处理“..”，“.”“//”。比如：

```
var path = require('path');
path.normalize('/foo/bar//baz/asdf/quux/..');
// => '/foo/bar/baz/asdf'
```

连接路径

使用 path.join()函数，可以连接任意多个路径字符串，只用把所有路径字符串依次传递给 join()函数就可以：

```
var path = require('path');
path.join('/foo', 'bar', 'baz/asdf', 'quux', '..');
// => '/foo/bar/baz/asdf'
```

如你所见，path.join()内部会自动将路径标准化。

解析路径

用 path.resolve()可以把多个路径解析为一个绝对路径。它的功能就像对这些路径挨个不断进行“cd”操作，和 cd 命令的参数不同，这些路径可以是文件，并且它们不必真实存在——path.resolve()方法不会去访问底层文件系统来确定路径是否存在，它只是一些字符串操作。

比如：

```
var path = require('path');
path.resolve('/foo/bar', './baz');
// => /foo/bar/baz
path.resolve('/foo/bar', '/tmp/file/');
// => /tmp/file
```

如果解析结果不是绝对路径，path.resolve()会把当前工作目录作为路径附加到解析结果前面，比如：

```
path.resolve('wwwroot', 'static_files/png/', '../gif/image.gif');
//  如果当前工作目录是/home/myself/node,  将返回
// => /home/myself/node/wwwroot/static_files/gif/image.gif'
```

计算两个绝对路径的相对路径

path.relative()可以告诉你如果从一个绝对地址跳转到另外一个绝对地址，比如：

```
var path = require('path');
path.relative('/data/orandea/test/aaa', '/data/orandea/impl/bbb');
// => ../../impl/bbb
```

从路径提取数据

以路径“/foo/bar/myfile.txt”为例，如果你想获取父目录（/foo/bar）的所有内容，或者读取同级目录的其它文件，为此，你必须用 path.dirname(filePath)获得文件路径的目录部分，比如：

```
var path = require('path');
path.dirname('/foo/bar/baz/asdf/quux.txt');
// => /foo/bar/baz/asdf
```

或者，你想从文件路径里得到文件名，也就是文件路径的最后那一部分，可以使用path.basename函数：

```
var path = require('path');
path.basename('/foo/bar/baz/asdf/quux.html')
// => quux.html
```

文件路径里可能还包含文件扩展名，通常是文件名中最后一个“.”字符之后的那部分字符串。

path.basename还可以接受一个扩展名字符串作为第二个参数，这样返回的文件名就会自动去掉扩展名，仅仅返回文件的名称部分：

```
var path = require('path');
path.basename('/foo/bar/baz/asdf/quux.html', '.html');
// => quux
```

要想这么做你首先还得知道文件的扩展名，可以用path.extname()来获取扩展名：

```
var path = require('path');
path.extname('/a/b/index.html');
// => '.html'
path.extname('/a/b.c/index');
// => ''
path.extname('/a/b.c/.');
// => ''
path.extname('/a/b.c/d.');
// => '.'
```

检查路径是否存在

目前为止，前面涉及到的路径处理操作都跟底层文件系统无关，只是一些字符串操作。然而，有些时候你需要判断一个文件路径是否存在，比如，你有时候需要判断文件或目录是否存在，如果不存在的话才创建它，可以用 path.exsits()：

```
var path = require('path');
path.exists('/etc/passwd', function(exists) {
    console.log('exists:', exists);
```

```
    // => true
});
path.exists('/does_not_exist', function(exists) {
    console.log('exists:', exists);
    // => false
});
```

注意：从Node0.8版本开始，exists从path模块移到了fs模块，变成了fs.exists，除了命名空间不同，其它都没变：

```
var fs = require('fs');
fs.exists('/does_not_exist', function(exists) {
    console.log('exists:', exists);
    // => false
});
```

path.exists()是个I/O操作，因为它是异步的，因此需要一个回调函数，当I/O操作返回后调用这个回调函数，并把结果传递给它。你还可以使用它的同步版本path.existsSync()，功能完全一样，只是它不会调用回调函数，而是直接返回结果：

```
var path = require('path');
path.existsSync('/etc/passwd');
// => true
```

fs 模块介绍

fs 模块包含所有文件查询和处理的相关函数，用这些函数，可以查询文件信息，读写和关闭文件。这样导入 fs 模块：

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

查询文件信息

有时你可能需要知道文件的大小，创建日期或者权限等文件信息，可以使用 fs.stath 函数来查询文件或目录的元信息：

```
var fs = require('fs');
fs.stat('/etc/passwd', function(err, stats) {
    if (err) { throw err;}
        console.log(stats);
});
```

这块代码片断会有类似下面的输出：

```
{ dev: 234881026,
ino: 95028917,
mode: 33188,
nlink: 1,
uid: 0,
gid: 0,
rdev: 0,
```

size: 5086,
blksize: 4096,
blocks: 0,
atime: Fri, 18 Nov 2011 22:44:47 GMT,
mtime: Thu, 08 Sep 2011 23:50:04 GMT,
ctime: Thu, 08 Sep 2011 23:50:04 GMT }

fs.stat()调用会将一个stats类的实例作为参数传递给它的回调函数，可以像下面这样使用stats实例：

- stats.isFile() —— 如果是个标准文件，而不是目录，socket，符号链接或者设备，则返回true，否则false
- stats.isDiretory() —— 如果是目录则返回tue，否则false
- stats.isBlockDevice() —— 如果是块设备则返回true，在大多数UNIX系统中块设备通常都在/dev目录下
- stats.isChracterDevice() —— 如果是字符设备返回true
- stats.isSymbolickLink() —— 如果是文件链接返回true
- stats.isFifo() —— 如果是个FIFO(UNIX命名管道的一个特殊类型)返回true
- stats.isSocket() —— 如果是个UNIX socket（TODO：google it）

打开文件

在读取或处理文件之前，必须先使用 fs.open 函数打开文件，然后你提供的回调函数会被调用，并得到这个文件的描述符，稍后你可以用这个文件描述符来读写这个已经打开的文件：

```
var fs = require('fs');
fs.open('/path/to/file', 'r', function(err, fd) {
    // got fd file descriptor
});
```

fs.open 的第一个参数是文件路径，第二个参数是一些用来指示以什么模式打开文件的标记，这些标记可以是 r，r+，w，w+，a 或者 a+。下面是这些标记的说明（来自 UNIX 文档的 fopen 页）

- r —— 以只读方式打开文件，数据流的初始位置在文件开始
- r+ —— 以可读写方式打开文件，数据流的初始位置在文件开始
- w ——如果文件存在，则将文件长度清 0，即该文件内容会丢失。如果不存在，则尝试创建它。数据流的初始位置在文件开始
- w+ —— 以可读写方式打开文件，如果文件不存在，则尝试创建它，如果文件存在，则将文件长度清 0，即该文件内容会丢失。数据流的初始位置在文件开始
- a —— 以只写方式打开文件，如果文件不存在，则尝试创建它，数据流的初始位置在文件末尾，随后的每次写操作都会将数据追加到文件后面。
- a+ ——以可读写方式打开文件，如果文件不存在，则尝试创建它，数据流的初始位置在文件末尾，随后的每次写操作都会将数据追加到文件后面。

读文件

一旦打开了文件，就可以开始读取文件内容，但是在开始之前，你得先创建一个缓冲区（buffer）来放置这些数据。这个缓冲区对象将会以参数形式传递给 fs.read 函数，并被 fs.read 填充上数据。

```
var fs = require('fs');
fs.open('./my_file.txt', 'r', function opened(err, fd) {
    if (err) { throw err }
    var readBuffer = new Buffer(1024),
    bufferOffset = 0,
    bufferLength = readBuffer.length,
    filePosition = 100;
    fs.read(fd,
        readBuffer,
        bufferOffset,
        bufferLength,
        filePosition,
        function read(err, readBytes) {
            if (err) { throw err; }
            console.log('just read ' + readBytes + ' bytes');
            if (readBytes > 0) {
                console.log(readBuffer.slice(0, readBytes));
            }
    });
});
```

上面代码尝试打开一个文件，当成功打开后(调用opened函数)，开始请求从文件流第100个字节开始读取随后1024个字节的数据（第11行）。

fs.read()的最后一个参数是个回调函数（第16行），当下面三种情况发生时，它会被调用：

- 有错误发生
- 成功读取了数据
- 没有数据可读

如果有错误发生，第一个参数（err）会为回调函数提供一个包含错误信息的对象，否则这个参数为null。如果成功读取了数据，第二个参数（readBytes）会指明被读到缓冲区里数据的大小，如果值是0，则表示到达了文件末尾。

注意：一旦把缓冲区对象传递给fs.open()，缓冲对象的控制权就转移给给了read命令，只有当回调函数被调用，缓冲区对象的控制权才会回到你手里。因此在这之前，不要读写或者让其它函数调用使用这个缓冲区对象；否则，你可能会读到不完整的数据，更糟的情况是，你可能会并发地往这个缓冲区对象里写数据。

写文件

通过传递给 fs.write()传递一个包含数据的缓冲对象，来往一个已打开的文件里写数据：

```
var fs = require('fs');
fs.open('./my_file.txt', 'a', function opened(err, fd) {
    if (err) { throw err; }
    var writeBuffer = new Buffer('writing this string'),
    bufferPosition = 0,
    bufferLength = writeBuffer.length, filePosition = null;
    fs.write(fd,
        writeBuffer,
        bufferPosition,
        bufferLength,
        filePosition,
        function wrote(err, written){
            if (err) { throw err; }
            console.log('wrote ' + written + ' bytes');
        });
});
```

这个例子里，第2（译者注：原文为3）行代码尝试用追加模式（a）打开一个文件，然后第7行代码（译者注：原文为9）向文件写入数据。缓冲区对象需要附带几个信息一起做为参数：

- 缓冲区的数据
- 待写数据从缓冲区的什么位置开始
- 待写数据的长度
- 数据写到文件的哪个位置
- 当操作结束后被调用的回调函数wrote

这个例子里，filePostion参数为null，也就是说write函数将会把数据写到文件指针当前所在的位置，因为是以追加模式打开的文件，因此文件指针在文件末尾。

跟read操作一样，千万不要在fs.write执行过程中使用哪个传入的缓冲区对象，一旦fs.write开始执行它就获得了那个缓冲区对象的控制权。你只能等到回调函数被调用后才能再重新使用它。

关闭文件

你可能注意到了，到目前为止，本章的所有例子都没有关闭文件的代码。因为它们只是些仅使用一次而且又小又简单的例子，当Node进程结束时，操作系统会确保关闭所有文件。

但是，在实际的应用程序中，一旦打开一个文件你要确保最终关闭它。要做到这一点，你需要追踪所有那些已打开的文件描述符，然后在不再使用它们的时候调用fs.close(fd[,callback])来最终关闭它们。如果你不仔细的话，很容易就会遗漏某个文件描述符。下面的例子提供了一个叫 openAndWriteToSystemLog 的函数，展示了如何小心的关闭文件：

```
var fs = require('fs');
function openAndWriteToSystemLog(writeBuffer, callback){
    fs.open('./my_file', 'a', function opened(err, fd) {
        if (err) { return callback(err); }
        function notifyError(err) {
```

```
                fs.close(fd, function() {
                    callback(err);
                });
            }
            var bufferOffset = 0,
            bufferLength = writeBuffer.length,
            filePosition = null;
            fs.write(fd, writeBuffer, bufferOffset, bufferLength, filePosition,
                function wrote(err, written) {
                    if (err) { return notifyError(err); }
                    fs.close(fd, function() {
                        callback(err);
                    });
                }
            );
        });
    }
    openAndWriteToSystemLog(
        new Buffer('writing this string'),
        function done(err) {
            if (err) {
                console.log("error while opening and writing:", err.message);
                return;
            }
            console.log('All done with no errors');
        }
    );
```

在这儿，提供了一个叫openAndWriteToSystemLog的函数，它接受一个包含待写数据的缓冲区对象，以及一个操作完成或者出错后被调用的回调函数，如果有错误发生，回调函数的第一个参数会包含这个错误对象。

注意那个内部函数notifyError，它会关闭文件，并报告发生的错误。

注意：到此为止，你知道了如何使用底层的原子操作来打开，读，写和关闭文件。然而，Node还有一组更高级的构造函数，允许你用更简单的方式来处理文件。

比如，你想用一种安全的方式，让两个或者多个write操作并发的往一个文件里追加数据，这时你可以使用WriteStream。

还有，如果你想读取一个文件的某个区域，可以考虑使用ReadStream。这两种用例会在第九章“数据的读，写流”里介绍。

小结

当你使用文件时，多数情况下都需要处理和提取文件路径信息，通过使用 path 模块你可以连接路径，标准化路径，计算路径的差别，以及将相对路径转化成绝对路径。你可以提取指定文件路径的扩展名，文件名，目录等路径组件。

Node 在 fs 模块里提供了一套底层 API 来访问文件系统，底层 API 使用文件描述符来操

作文件。你可以用 fs.open 打开文件，用 fs.write 写文件，用 fs.read 读文件，并用 fs.close 关闭文件。

当有错误发生时，你应该总是使用正确的错误处理逻辑来关闭文件——以确保在调用返回前关闭那些已打开的文件描述符。