# Node常用内置模块

王红元
coderwhy


实力IT教育

# 内置模块path

- path模块用于对路径和文件进行处理，提供了很多好用的方法。
- 并且我们知道在Mac OS、Linux和window上的路径时不一样的
  - window上会使用 \或者 \\ 来作为文件路径的分隔符，当然目前也支持 /；
  - 在Mac OS、Linux的Unix操作系统上使用 / 来作为文件路径的分隔符；
- 那么如果我们在window上使用 \ 来作为分隔符开发了一个应用程序，要部署到Linux上面应该怎么办呢？
  - 显示路径会出现一些问题；
  - 所以为了屏蔽他们之间的差异，在开发中对于路径的操作我们可以使用 path 模块；
- 可移植操作系统接口（英语：Portable Operating System Interface，缩写为POSIX）
  - Linux和Mac OS都实现了POSIX接口；
  - Window部分电脑实现了POSIX接口；

# path常见的API

- **从路径中获取信息**
  - dirname：获取文件的父文件夹；
  - basename：获取文件名；
  - extname：获取文件扩展名；
- **路径的拼接**
  - 如果我们希望将多个路径进行拼接，但是不同的操作系统可能使用的是不同的分隔符；
  - 这个时候我们可以使用path.join函数；
- **将文件和某个文件夹拼接**
  - 如果我们希望将某个文件和文件夹拼接，可以使用 path.resolve;
  - resolve函数会判断我们拼接的路径前面是否有 /或../或./；
  - 如果有表示是一个绝对路径，会返回对应的拼接路径；
  - 如果没有，那么会和当前执行文件所在的文件夹进行路径的拼接

# 在webpack中的使用

- 在webpack中获取路径或者起别名的地方也可以使用

JS craco.config.js > <unknown> > plugins

```
1  const CracoLessPlugin = require('craco-less');
2  const path = require("path");
3
4  const resolve = dir => path.resolve(__dirname, dir);
5
6  module.exports = {
7    plugins: [
8      {
9        plugin: CracoLessPlugin,
10       options: {…
17       },
18     }
19   ],
20   webpack: {
21     alias: {
22       "@": resolve("src"),
23       "components": resolve("src/components")
24     }
25   }
26 }
```

# 内置模块fs

- fs是File System的缩写，表示文件系统。
- 对于任何一个为服务器端服务的语言或者框架通常都会有自己的文件系统：
  - 因为服务器需要将各种数据、文件等放置到不同的地方；
  - 比如用户数据可能大多数是放到数据库中的（后面我们也会学习）；
  - 比如某些配置文件或者用户资源（图片、音视频）都是以文件的形式存在于操作系统上的；
- Node也有自己的文件系统操作模块，就是fs：
  - 借助于Node帮我们封装的文件系统，我们可以在任何的操作系统（window、Mac OS、Linux）上面直接去操作文件；
  - 这也是Node可以开发服务器的一大原因，也是它可以成为前端自动化脚本等热门工具的原因；

# fs的API介绍

- Node文件系统的API非常的多：
  - https://nodejs.org/dist/latest-v14.x/docs/api/fs.html
  - 我们不可能，也没必要一个个去学习；
  - 这个更多的应该是作为一个API查询的手册，等用到的时候查询即可；
  - 学习阶段我们只需要学习最常用的即可；
- 但是这些API大多数都提供三种操作方式：
  - 方式一：同步操作文件：代码会被阻塞，不会继续执行；
  - 方式二：异步回调函数操作文件：代码不会被阻塞，需要传入回调函数，当获取到结果时，回调函数被执行；
  - 方式三：异步Promise操作文件：代码不会被阻塞，通过 fs.promises 调用方法操作，会返回一个Promise，可以通过then、catch进行处理；

# 案例：获取一个文件的状态

- 我们这里以获取一个文件的状态为例：

```
// 1.方式一：同步读取文件
const state = fs.statSync('../foo.txt');
console.log(state);

console.log('后续代码执行');
```

```
// 2.方式二：异步读取
fs.stat("../foo.txt", (err, state) => {
  if (err) {
    console.log(err);
    return;
  }
  console.log(state);
})
console.log("后续代码执行");
```

```
// 3.方式三：Promise方式
fs.promises.stat("../foo.txt").then(state => {
  console.log(state);
  console.log(state.isDirectory());
}).catch(err => {
  console.log(err);
})
console.log("后续代码执行");
```

# 文件描述符

- 文件描述符（File descriptors）是什么呢？
  - 在 POSIX 系统上，对于每个进程，内核都维护着一张当前打开着的文件和资源的表格。
  - 每个打开的文件都分配了一个称为文件描述符的简单的数字标识符。
  - 在系统层，所有文件系统操作都使用这些文件描述符来标识和跟踪每个特定的文件。
  - Windows 系统使用了一个虽然不同但概念上类似的机制来跟踪资源。
- 为了简化用户的工作，Node.js 抽象出操作系统之间的特定差异，并为所有打开的文件分配一个数字型的文件描述符。
- fs.open() 方法用于分配新的文件描述符。
  - 一旦被分配，则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。

```
fs.open("../foo.txt", 'r', (err, fd) => {
  console.log(fd);
  fs.fstat(fd, (err, state) => {
    console.log(state);
  })
})
```

# 文件的读写

- 如果我们希望对文件的内容进行操作，这个时候可以使用文件的读写：
  - fs.readFile(path[, options], callback)：读取文件的内容；
  - fs.writeFile(file, data[, options], callback)：在文件中写入内容；

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

let content = "aaaaa";

fs.writeFile('../foo.txt', content, {flag: "a+"}, err => {
  console.log(err);
})
```

- 在上面的代码中，你会发现有一个大括号没有填写任何的内容，这个是写入时填写的option参数：
  - flag：写入的方式。
  - encoding：字符的编码；

# flag选项

- 我们先来看flag：
- flag的值有很多：https://nodejs.org/dist/latest-v14.x/docs/api/fs.html#fs_file_system_flags
  - w 打开文件写入，默认值；
  - w+打开文件进行读写，如果不存在则创建文件；
  - r+ 打开文件进行读写，如果不存在那么抛出异常；
  - r打开文件读取，读取时的默认值；
  - a打开要写入的文件，将流放在文件末尾。如果不存在则创建文件；
  - a+打开文件以进行读写，将流放在文件末尾。如果不存在则创建文件

# encoding选项

- 我们再来看看编码：
  - 我之前在简书上写过一篇关于字符编码的文章：https://www.jianshu.com/p/899e749be47c
  - 目前基本用的都是UTF-8编码；

- **文件读取：**
  - 如果不填写encoding，返回的结果是Buffer；

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

fs.readFile('../foo.txt', {encoding: 'utf-8'}, (err, data) => {
  console.log(data);
})
```

# 文件夹操作

- 新建一个文件夹
  - 使用fs.mkdir()或fs.mkdirSync()创建一个新文件夹：

```
const dirname = '../why';

if (!fs.existsSync(dirname)) {
  fs.mkdir(dirname, (err) => {
    console.log(err);
  })
}
```

- 获取文件夹的内容

```
function readFolders(folder) {
  fs.readdir(folder, {withFileTypes: true} ,(err, files) => {
    files.forEach(file => {
      if (file.isDirectory()) {
        const newFolder = path.resolve(dirname, file.name);
        readFolders(newFolder);
      } else {
        console.log(file.name);
      }
    })
  })
}
```

- 文件重命名

```
fs.rename('../why', '../coder', err => {
  console.log(err);
})
```

# 文件夹的复制

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

const srcDir = process.argv[2];
const destDir = process.argv[3];

let i = 0;

while (i < 30) {
  i++;
  const num = 'day' + (i + '').padStart(2, 0);
  const srcPath = path.resolve(srcDir, num);
  const destPath = path.resolve(destDir, num);
  if (fs.existsSync(destPath)) continue;
  fs.mkdir(destPath, (err) => {
    if (!err) console.log("文件创建成功开始拷贝:", num);

    // 遍历目录下所有的文件
    const srcFiles = fs.readdirSync(srcPath);
    for (const file of srcFiles) {
      if (file.endsWith('.mp4')) {
        const srcFile = path.resolve(srcPath, file);
        const destFile = path.resolve(destPath, file);
        fs.copyFileSync(srcFile, destFile);
        console.log(file, "拷贝成功");
      }
    }
  })
}
```

# events模块

- Node中的核心API都是基于异步事件驱动的：
  - 在这个体系中，某些对象（发射器（Emitters））发出某一个事件；
  - 我们可以监听这个事件（监听器 Listeners），并且传入的回调函数，这个回调函数会在监听到事件时调用；
- 发出事件和监听事件都是通过EventEmitter类来完成的，它们都属于events对象。
  - emitter.on(eventName, listener)：监听事件，也可以使用addListener；
  - emitter.off(eventName, listener)：移除事件监听，也可以使用removeListener；
  - emitter.emit(eventName[, ...args])：发出事件，可以携带一些参数；

```
const EventEmmiter = require('events');

// 监听事件
const bus = new EventEmmiter();

function clickHanlde(args) {
  console.log("监听到click事件", args);
}

bus.on("click", clickHanlde);

setTimeout(() => {
  bus.emit("click", "coderwhy");
  bus.off("click", clickHanlde);
  bus.emit("click", "kobe");
}, 2000);
```

# 常见的属性

- EventEmitter的实例有一些属性，可以记录一些信息：
  - emitter.eventNames()：返回当前 EventEmitter对象注册的事件字符串数组；
  - emitter.getMaxListeners()：返回当前 EventEmitter对象的最大监听器数量，可以通过setMaxListeners()来修改，默认是10；
  - emitter.listenerCount(事件名称)：返回当前 EventEmitter对象某一个事件名称，监听器的个数；
  - emitter.listeners(事件名称)：返回当前 EventEmitter对象某个事件监听器上所有的监听器数组；

```
console.log(bus.eventNames());
console.log(bus.getMaxListeners());
console.log(bus.listenerCount("click"));
console.log(bus.listeners("click"));
```

# 方法的补充

- emitter.once(eventName, listener)：事件监听一次
- emitter.prependListener()：将监听事件添加到最前面
- emitter.prependOnceListener()：将监听事件添加到最前面，但是只监听一次
- emitter.removeAllListeners([eventName])：移除所有的监听器

```
emitter.once('click', (args) => {
  console.log("a监听到事件", args);
})
```

```
emitter.prependOnceListener("click", (args) => {
  console.log("c监听到事件", args);
})
```

```
emitter.prependListener("click", (args) => {
  console.log("b监听到事件", args);
})
```

```
emitter.removeAllListeners();
emitter.removeAllListeners("click");
```