第8章 Node.js基础模块一

课程提要

- NodeJS介绍
- NodeJS安装与使用
- NodeJS模块
- 回调函数
- 异步编程
- Buffer缓存区
- fs文件基本操作
- fs流

8.1 NodeJS介绍

Node.js 是一个基于 Chrome V8 引擎的 JavaScript 运行环境。 Node.js 使用了一个事件驱动、非阻塞式 I/O 的模型，使其轻量又高效。 Node.js 的包管理器 npm，是全球最大的开源库生态系统。

8.1.1 Node.js历史

- 2009年3月，Ryan Dahl 在其博客上宣布准备基于 V8 创建一个轻量级的 Web 服务器并提供一套库。
- 2009年5月，Ryan Dahl 在 GitHub 上发布了最初的版本。
- 2009年12月和2010年4月，两届 JSConf 大会都安排了 Node 的讲座。
- 2010年年底，Node 获得硅谷云计算服务商 Joyent 公司的资助，其创始人 Ryan Dahl 加入了 Joyent 公司全职负责 Node 的发展。
- 2011年7月，Node 在微软的支持下发布了其 Windows 版本。
- 2011年11月，Node 超越 Ruby on Rails，成为 GitHub 上关注度最高的项目（随后被 Bootstrap 项目超越，目前仍居第二）。
- 2012年1月底，Ryan Dahl 在 Node 架构设计满意的情况下，将掌门人的身份转交给 Isaac Z.Schlueter,自己转向一些研究项目。Isaac Z.Schlueter 是 Node 的包管理器 NPM 的作者，之后的 Node 的版本发布和 bug 修复等工作由他接手。
- 2015年09月 Node 基金会已发布 Node V4.0 版 与 io.js 合并后的第一个版本。
- 2016 年底 v6.0 支持95%以上的 es6 特性， v7.0通过 flag 支持 async 函数，99%的 es6 特性。
- 2017年2月发布v7.6版本，可以不通过 flag 使用 async 函数。
- 至目前，Node 已经更新了到了 v10.1.0 以上版本。

8.1.2 版本

在以往，Node 分为 io.js 与 Node.js。 1.x 到 3.x 版本被叫做 “io.js”，因为它们属于 io.js 的分支。从Node.js 4.0.0 开始，之前版本的 io.js 与 Node.js 0.12.x 合并到统一的 Node.js 发行版中。

8.1.3 作用

- 统一前后端编程语言环境
- 带来高性能的 I/O 操作
- 并行 I/O 可以更高效的利用分布式环境
- 并行 I/O 提高 Web 渲染能力
- 可用于云计算平台
- 可用于游戏开发
- 工具类应用

8.2 NodeJS安装与使用

8.2.1 NodeJS 下载

nodejs 官网：https://nodejs.org/en/

nodejs 中文网：http://nodejs.cn/

下载地址：http://nodejs.cn/download/

图8-1 node.js下载

8.2.2 NodeJS 安装

双击 NodeJS 安装包 node-v8.10.0-x64.msi， 并按照安装提示采用默认安装即可。安装过程图如下：

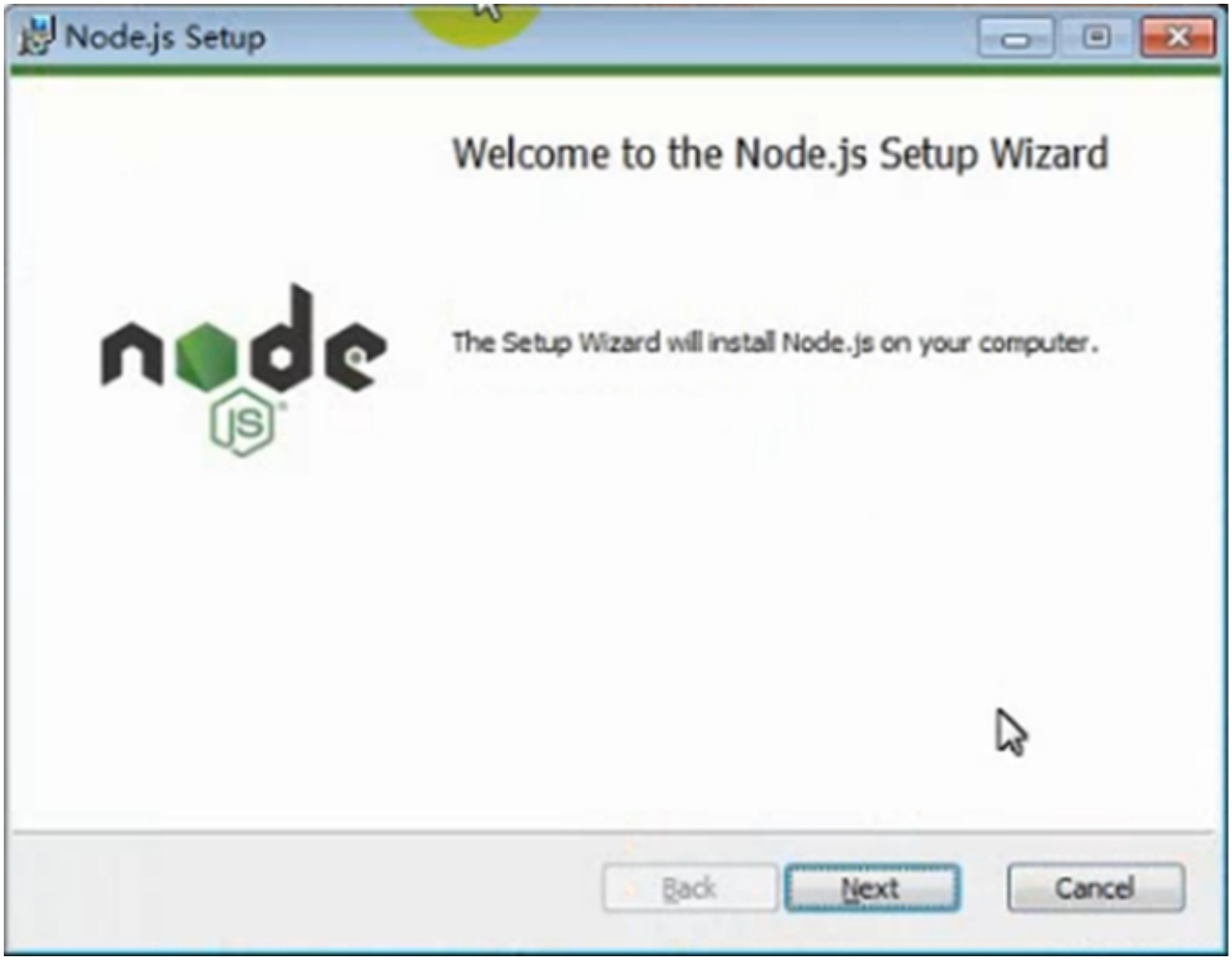
Node.js Setup
Welcome to the Node.js Setup Wizard
The Setup Wizard will install Node.js on your computer.
Back
Next
Cancel

图8-2 点击Next

图8-3 勾选“I accept the ...”，并点击Next

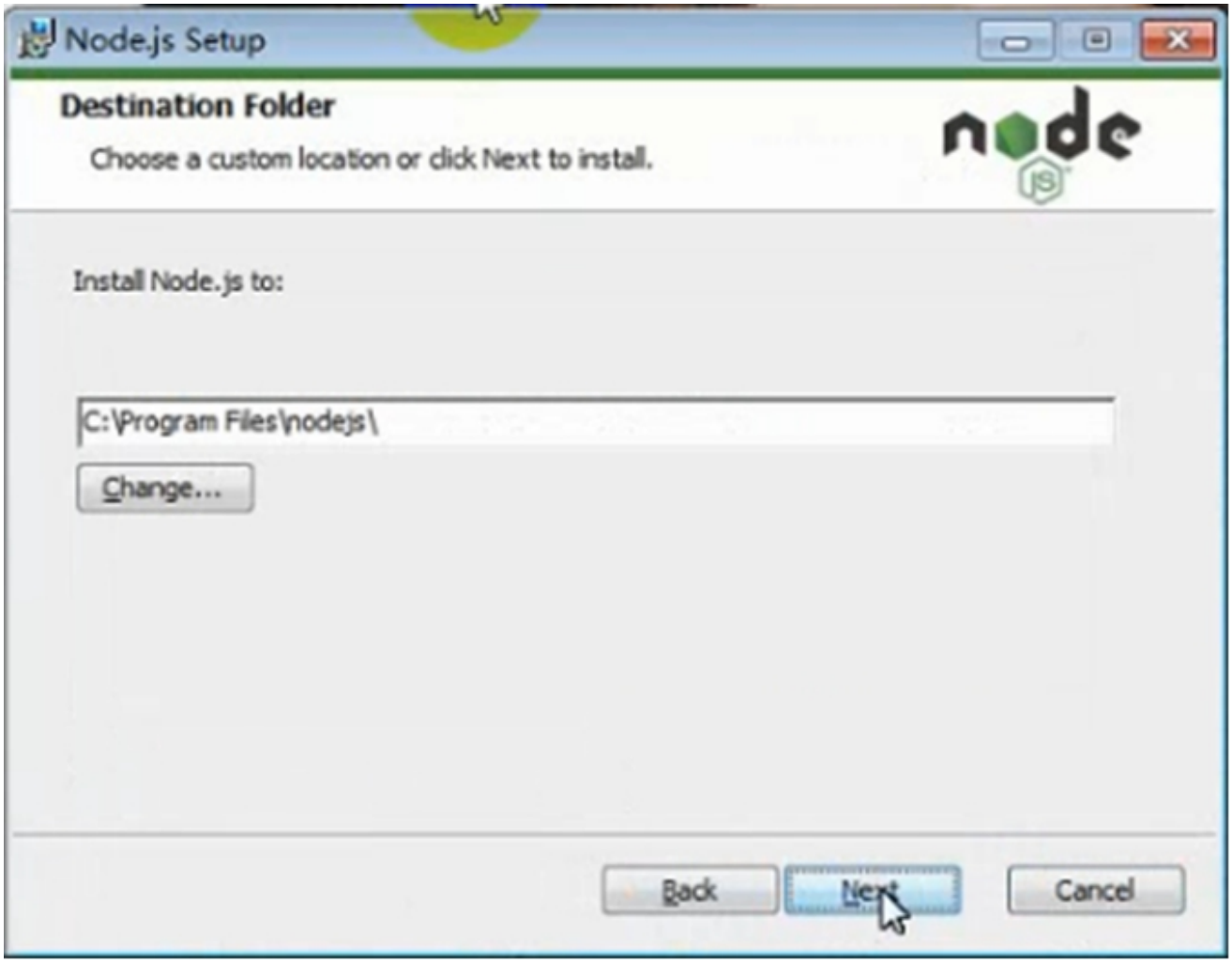
Node.js Setup
Destination Folder
Choose a custom location or click Next to install.
node
Install Node.js to:
C:\Program Files\nodejs\
Change...
Back
Next
Cancel

图8-4 点击Next

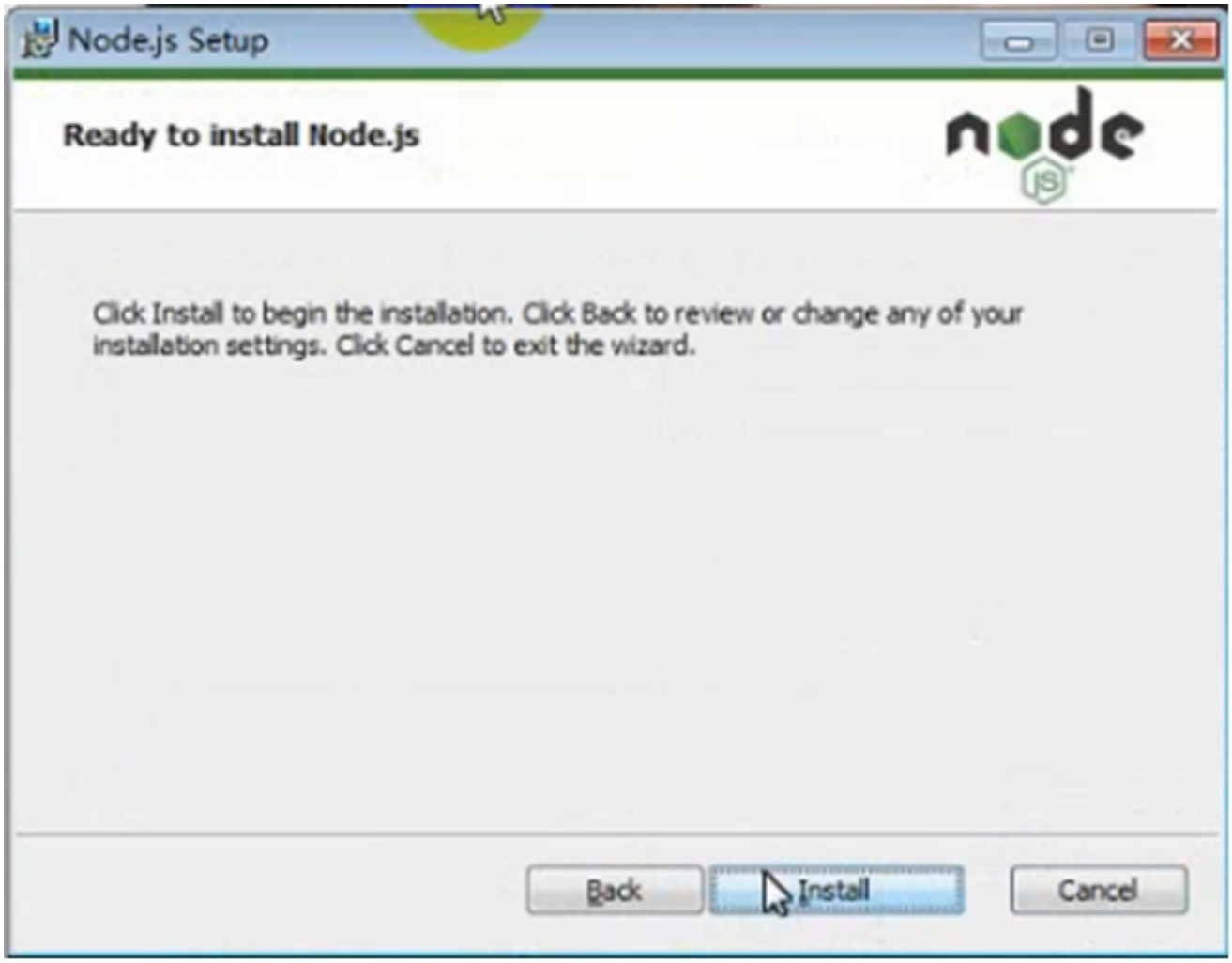

图8-5 点击Install

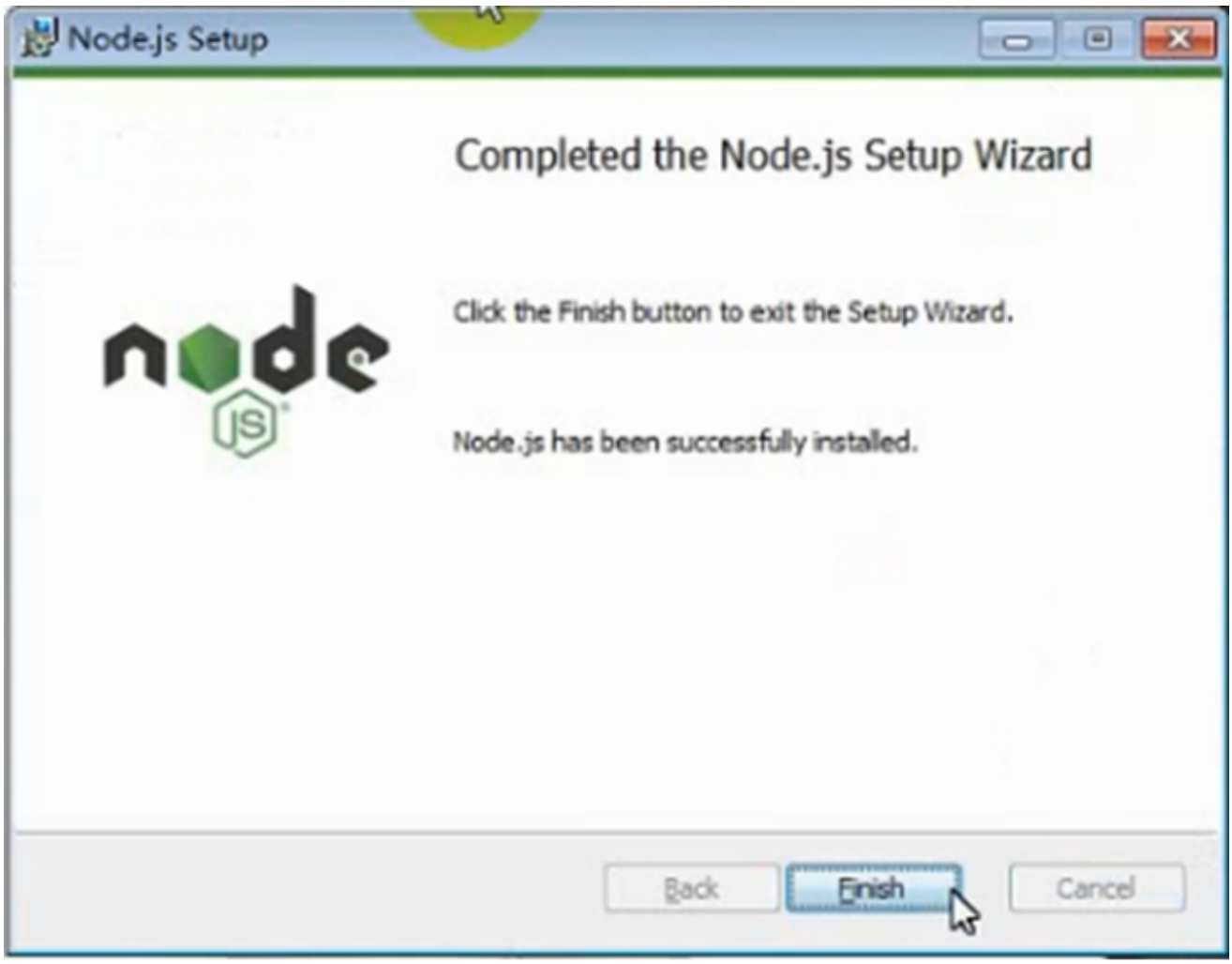

图8-6 点击Finish完成安装

8.3 NodeJS模块

由于 Node.js 的模块系统构建时期还没有 ES6 模块化机制，因此它采用的是 CommonJs 模块化来实现的。到后来为了兼容不同类型的模块系统，增加了对 AMD 和 ES6 模块化的支持。不过，我们通常使用 CommonJs 和 ES6 模块化。

8.3.1 module（模块）

在 Node.js 模块系统中，每个文件都被视为独立的模块。

模块导入语法：

```
const moduleName = require('yourModulePath');
```

模块导出语法：

```
// 方式一
module.exports = {
    name: 'apple',
    price: 8,
};

// 方式二
exports.name = 'apple';
exports.price = 8;
```

注意： 直接采用 exports 导出变量的时候，不能修改 exports 的指向，因此下面的写法是错误的。

```
exports = {
    name: 'apple',
    price: 8,
}
```

8.3.2 内置模块

- HTTP 模块：处理客户端的网络请求
- URL 模块：处理客户端请求过来的URL
- Query Strings 模块：处理客户端通过 get/post 请求传递过来的参数
- File System 模块：处理客户端与服务器之间的文件读取操作
- Path 模块：操作文件的路径，为文件操作服务
- Global 模块：全局共享的，不需要导入模块即可以使用

8.4 回调函数

回调函数就是一个通过函数指针调用的函数。如果你把函数的指针（地址）作为参数传递给另一个函数，当这个指针被用来调用其所指向的函数时，我们就说这是回调函数。回调函数不是由该函数的实现方直接调用，而是在特定的事件或条件发生时由另外的一方调用的，用于对该事件或条件进行响应。

```
function getSudentMsgByName(num, callback) {
    $.ajax({
        type: 'get',
        url: 'https://api.apiopen.top/getJoke',
        data: {
            page: 1,
            count: num,
            type: 'video'
        },
        success: function (res) {
            if (res) {
                callback(res); // 将结果res传递给回调函数callback
            }
        }
    })
}

getSudentMsgByName(2, function (data) {
```

```
    // handle
})
```

8.4.1 阻塞

由于 Node 保持了 JavaScript 在浏览器中单线程的特点，因此只要一讲到“阻塞”问题，不得不提的就是“单线程”。单线程在程序执行时，所走的程序路径按照连续顺序排下来，前面的必须处理好，后面的才会执行。阻塞就好比如“老板让办公室的小姐姐写一篇文章，小姐姐说自己是单线程的同步操作，必须等到昨天在网上买的鼠标到手之后才能写文章，在鼠标没到之前就干坐着等待”。因此，在 Node 开发环境当中，我们应当少用同步操作，以减少阻塞问题。

在浏览器环境中，最简单常见的一个同步阻塞案例：

```
alert('我是一个同步阻塞，当我未关闭的时候，后面的代码都不会被执行！');
```

8.4.2 非阻塞

结合上面阻塞的案例，非阻塞就好比如“饭店服务员给这一桌的顾客点完餐之后将菜单交给厨师就立马去给下一桌顾客点餐，而不是等待这一桌顾客的菜上完并吃完后才去服务下一桌顾客”。在 JavaScript 编程当中，通常在请求后台数据的时候采用异步 I/O 操作，此过程并不需要等待后台数据请求回来之后再往下执行任务，当所有任务完成之后，再通过轮询策略读取后台返回结果，因此这是一个非阻塞的单线程。

在浏览器环境中，最简单常见的一个非阻塞案例：

```
var img = new Image()

img.onload = function () {
// 图片正常加载完后执行这里
}

img.src = './img/icon.png';
```

8.5 异步编程

nodejs 保持了 Javascript 的单线程特点，为了高效使用硬件，nodejs 编程中大量使用了异步编程技术。因此异步编程对 nodejs 尤为重要。　　异步编程主要解决方案：

- 事件发布/订阅模式
- Promise/Deferred模式
- 流程控制库

8.5.1 事件发布/订阅模式

事件监听器模式是一种广泛用于异步编程的模式，是回调函数的事件化，又称发布/订阅模式。Node 自身提供的 events 模块是发布/订阅模式的一个简单实现，Node 中部分模块都继承自它，这个模块比前端浏览器中的大量 DOM 事件简单，不存在事件冒泡，也不存在 preventDefault()、stopPropagation()等控制事件传递的方法。它具有 addListener/on()、once()、removeListener()、removeAllListener()和 emit()等基本的事件监听模式的方法实现。事件发布/订阅模式的操作极其简单，**例如：**

```
const EventEmitter = require('events');
const myEmitter = new EventEmitter();

// 订阅
myEmitter.on('event', () => {
    console.log('A');
});
// 发布
myEmitter.emit('event');
```

8.5.2 Promise/Deferred模式

对于使用过 jQuery/ajax 技术的前端工程师而言，通过 Promise/Deferred 模式解决异步编程的方案并不陌生。在以往，异步的广泛使用使得回调、嵌套出现，一旦出现深度的嵌套，就会让我们后期的维护变得及其的困难。当我们使用 Promise/Deferred 模式能够在很大程度上解决这个问题。例如，我们可以通过该模式来封装一个自定义的请求后台数据的方法：

```
function request(option) {
    var def = $.Deferred(); // 这调用了jQuery的Deferred方法
    if (!option || !option.url) {
        throw Error('url is undefined.');
    }
    var _config = {
        type: 'get',
        dataType: 'json',
        success: function (res) {
            def.resolve(res); // 通过resolve来改变def对象的状态为“成功”
        },
        error: function (a, b, err) {
            def.reject(err); // 通过reject来改变def对象的状态为“失败”
        }
    };
    $.extend(_config, option, true);
    $.ajax(_config);
    return def.promise(); // 返回def对象的承诺 (promise)
}

// request调用
var option = {
    url: './admin/test.php',
    data: {
        name: 'Harrison'
    }
}
request(option).then(function () {
    // 成功时执行的回调函数
}, function () {
    // 失败时执行的回调函数
})
```

8.5.3 流程控制库

由于 Node 融入了 ES6 的特征，因此我们在 Node 编程时也可以采用 ES6 的流程控制模块 async 来解决异步编程问题。**例如：**

```
var option1 = {
    url: './admin/test.php',
    data: {
        name: 'Harrison'
    }
}
var option2 = {
    url: './admin/test.php',
    data: {
        name: 'Harrison'
    }
}

const myAsyncFn = async function () {
    const teacher = await request(option1); //request是一个异步函数
    const student = await request(option2);
    console.log(teacher); // 输出老师信息
    console.log(student); // 输出学生信息
}

myAsyncFn(option1, option2); // 调用async函数
```

8.6 Buffer缓存区

8.6.1 简介

Buffer 是 Node.js 中的一个用来读取或操作二进制数据流的全局变量，它可以在 TCP 流或文件系统操作等场景中处理二进制数据流。Buffer 类的实例类似于整数数组，但 Buffer 的大小是固定的、且在 V8 堆外分配物理内存。 Buffer 的大小在被创建时确定，且无法调整。

8.6.2 Buffer.from(array)

Buffer.from 方法返回一个新建的包含所提供的字节数组的副本的 Buffer。**例如：**

```
const buf = Buffer.from('hello')

console.log(buf); // <Buffer 68 65 6c 6c 6f>
console.log(buf.toString()); // hello
```

从上述案例我们可以看出，通过 Buffer.from 方法可以得到一个字符串的 Buffer 缓冲，而通过 toString 方法能够将 Buffer 转换成字符串。因此，我们可以通过这两个方法轻松的操作二进制数据流。

8.6.3 Buffer.alloc(size[, fill[, encoding]])

Buffer.alloc 方法能够为我们分配一个大小为 size 字节的新建的 Buffer。 如果 fill 为 undefined ，则该 Buffer 会用 0 填充。

```
const buf1 = Buffer.alloc(5);
const buf2 = Buffer.alloc(5, 3);
const buf3 = Buffer.alloc(6, '中文', 'utf-8')

console.log(buf1); // <Buffer 00 00 00 00 00>
console.log(buf2); // <Buffer 03 03 03 03 03>
console.log(buf3); // <Buffer e4 b8 ad e6 96 87>
console.log(buf3.toString()); // 中文
```

注意： 一个汉字的需要3个字节的空间来存储。Buffer.alloc 方法默认的编码方式就是“utf-8”，因此上面的案例中可以不写。

8.6.4 Buffer.allocUnsafe(size)

Buffer.allocUnsafe 方法可以为为我们分配一个大小为 size 字节的新建的 Buffer 。 如果 size 大于 buffer.constants.MAX_LENGTH 或小于 0，则抛出 RangeError 错误。 如果 size 为 0，则创建一个长度为 0 的 Buffer。

```
const buf1 = Buffer.allocUnsafe(10);
const buf2 = Buffer.allocUnsafe(10);
const buf3 = Buffer.allocUnsafe(10);

console.log(buf1); // <Buffer 06 00 00 00 00 00 00 00 e8 24>
console.log(buf2); // <Buffer 20 da 12 00 00 00 00 00 10 28>
console.log(buf3); // <Buffer 50 2b c1 02 00 00 00 00 00 00>
```

注意： 因为 Buffer.allocUnsafe 方法给我们分配的空间是未“重置”的，保留了硬盘当中之前存储过的垃圾信息，因此在使用的时候是“不安全的”，不过它的读取效率要比 Buffer.alloc 高。正因为这个特性，上述案例当中我们采用同样的方式申请了3个 buffer ，但最终得到的结果却没有相同的（因为同一块长度为10的磁盘不能同时被3个不同的对象来使用）。

8.6.5 Buffer 与字符编码

Buffer 实例一般用于表示编码字符的序列，比如 UTF-8 、 UCS2 、 Base64 、或十六进制编码的数据。 通过使用显式的字符编码，就可以在 Buffer 实例与普通的 JavaScript 字符串之间进行相互转换。**例如：**

```
const buf = Buffer.from('蜀山行者');

console.log(buf); // <Buffer e6 81 92 e4 bc 81 e6 95 99 e8 82 b2>

console.log(buf.toString()); // 蜀山行者
console.log(buf.toString('utf-8')); // 蜀山行者
console.log(buf.toString('base64')); // 5oGS5LyB5pWZ6IKy
```

由上面的案例，我们不难看出 toString 方法可以将我们的 Buffer 转换成指定格式（比如：utf-8 和 base64 格式，其中 utf-8 为 toString 方法默认编码格式）的 JavaScript 字符串。

Node.js 目前支持的字符编码包括：

- ascii - 仅支持 7 位 ASCII 数据。如果设置去掉高位的话，这种编码是非常快的。
- utf8 - 多字节编码的 Unicode 字符。许多网页和其他文档格式都使用 UTF-8 。

- utf16le - 2 或 4 个字节，小字节序编码的 Unicode 字符。支持代理对（U+10000 至 U+10FFFF）。
- ucs2 - 'utf16le' 的别名。
- base64 - Base64 编码。当从字符串创建 Buffer 时，按照 RFC4648 第 5 章的规定，这种编码也将正确地接受“URL 与文件名安全字母表”。
- latin1 - 一种把 Buffer 编码成一字节编码的字符串的方式（由 IANA 定义在 RFC1345 第 63 页，用作 Latin-1 补充块与 C0/C1 控制码）。
- binary - 'latin1' 的别名。
- hex - 将每个字节编码为两个十六进制字符。

8.7 fs文件基本操作

8.7.1 简介

Node.js 的 fs 是一个类似标准 POSIX 函数的用来操作文件的模块，它对所有的文件系统操作都有异步和同步两种形式。不过，需要注意的是“当采用异步形式的时候，回调函数的第一个参数都会保留给异常，如果操作成功完成，则第一个参数会是 null 或 undefined”。

8.7.2 基本用法

fs 模块引入及基本用法：

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

// 异步的方式
fs.readFile('./test.js', (err, data) => {
    //handle
})

const data = fs.readFileSync('./test.js'); // 同步的方式
```

8.7.3 fs.open(path, flags[, mode], callback) 异步的方式打开文件

fs.open 方法通常用来打开一个文件地址、URL 或者 Buffer。

其中 flags 参数可以是：

- r - 以读取模式打开文件。如果文件不存在则发生异常。
- r+ - 以读写模式打开文件。如果文件不存在则发生异常。
- rs+ - 以同步读写模式打开文件。命令操作系统绕过本地文件系统缓存。

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

fs.open('./public/test.js', 'rs+', (err, fd) => {
    if(err) {
        console.log('文件打开失败！');
        return false;
    }
    fs.write(fd, ...) // 在这里可以对上述打开的文件进行一系列的操作
            })
```

如果您想采用同步的方式来操作，则应该使用 fs.openSync()。

```
const fs = require('fs');
const fd = fs.openSync('./public/test.js', 'rs+');
```

8.7.4 fs.readFile(path[, options], callback) 异步的方式读取文件

fs.readFile 方法可以用来读取一个文件的具体内容，下面是一个采用异步的方式读取文件信息的案例。

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

fs.readFile('./test.js', 'utf-8', (err, data) => {
    if (err) throw err;
    console.log(data);// 这里将输出test.js的具体内容
})
```

如果您需要使用同步的方式来读取文件，可以使用 fs.readFileSync 方法来实现。由于 fs 模块对每一个文件操作都提供了同步和异步两种操作方式，这里以及下文都将只演示异步的方式（因为，方法的实现就差一个回调函数）。

8.7.5 fs.write(fd, buffer[, offset[, length[, position]]], callback) 异步的方式写入文件

通过 fs.write 方法可以将字符串或者 Buffer 写入一个指定的文件（fd）。**例如：**

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

fs.open('./test.txt', 'rs+', (err, fd) => {
    if(err) {
        console.log('文件打开失败! ');
        return false;
    }

    fs.write(fd, 'hello world', (err, written, str) => {
        if(err) throw err;
        console.log(written); // 11
        console.log(str); // hello world
    })
})
```

8.7.6 fs.close(fd, callback) 异步的方式关闭文件

与 fs.open 方法相反，fs.close 方法可以用来关闭一个已经打开的文件。**例如：**

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

fs.open('./public/test.js', 'rs+', (err, fd) => {
    if (err) {
        console.log('文件打开失败! ');
        return false;
```

```
    }

    // handle

    fs.close(fd, err => {
        if (err) throw err;
        // 关闭文件的回调函数体
    })
})
```

8.7.7 fs.unlink(path, callback) 异步的方式删除文件

fs.unlink 方法可以删除一个文件（不能删除目录）。**例如：**

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

fs.unlink('./test.txt', err => {
    if (err) throw err;
    console.log('删除成功! ');
})
```

8.7.8 fs.readdir(path[, options], callback) 读取一个目录

当我们需要知道某个目录下都有那些文件或者文件夹的时候，我们可以使用 fs.readdir 方法来实现。**例如：**

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

fs.readdir('./', (err, files) => {
    if (err) throw err;
    console.log(files.join(';')); // 这里将输出当前路径下的所有文件、文件夹名称并通过“; ”分隔
})
```

注意： fs 模块的每一个方法都提供了异步与同步的两种操作方式。其中，异步的方式性能高，是一种推荐的操作方式。如果您需要采用同步的方式，只要在异步方法名后面加上 Sync 即可，同时去掉末尾的 callback 参数。

8.8 fs流

8.8.1 fs流的简介

Stream 是一个抽象接口，Node 中有很多对象实现了这个接口。例如，对http 服务器发起请求的request 对象就是一个 Stream，还有stdout（标准输出）。

8.8.1.1 stream：流是一个抽象接口，有四种流类型：

- readable：可读
- writable：可写操作
- duplex：可读可写操作
- transform：操作被写入数据，然后读出结果

8.8.1.2 所有的stream对象都是EventEmitter 的实例，常用事件有：

- data：当有数据可读触发
- end：没有数据可读触发
- error：发生错误时触发
- finish：完成触发

8.8.2 fs.createReadStream(path[, options]) 创建一个可读流

fs.createReadStream 方法通常用来创建一个可读流。**例如：**

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

var data = '';

// 创建可读流
var readerStream = fs.createReadStream('./public/test.js');

// 设置编码为utf8。
readerStream.setEncoding('UTF8');

// 处理流事件-> data, end, and error
readerStream.on('data', function(chunk) {
    data += chunk;
});

readerStream.on('end',function(){
    console.log(data);
});

readerStream.on('error', function(err){
    console.log(err.stack);
});

console.log("程序执行完毕");
```

8.8.3 fs.createWriteStream(path[, options]) 创建一个可写流

fs.createWriteStream 方法可以创建一个可写入流对象，通过这个对象可以将自己想要写入的信息写到指定文件。**例如：**

```
var fs = require("fs");
var data = '这是我要写入到test.js文件中的数据';

// 创建一个可以写入的流，写入到文件 output.txt 中
var writerStream = fs.createWriteStream('./public/test.js');

// 使用 utf8 编码写入数据
writerStream.write(data, 'UTF8');

// 标记文件末尾
writerStream.end();

// 处理流事件 --> data, end, and error
```

```
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");
```

8.8.4 readerStream.pipe(writerStream) 将可读流信息导入可写流（即管道流）

readerStream.pipe 方法可以将我们创建的可读流文件信息一一导入到可写流当中。**例如：**

```
var fs = require("fs");

// 创建一个可读流
var readerStream = fs.createReadStream('input.txt');

// 创建一个可写流
var writerStream = fs.createWriteStream('output.txt');

// 管道读写操作
// 读取 input.txt 文件内容，并将内容写入到 output.txt 文件中
readerStream.pipe(writerStream);
console.log("程序执行完毕");
```

8.8.5 链式流

链式是通过连接输出流到另外一个流并创建多个流操作链的机制。链式流一般用于管道操作。例如我们可以用管道和链式来压缩和解压文件：

```
// 压缩
var fs = require("fs");
var zlib = require('zlib');

// 压缩 input.txt 文件为 input.txt.gz
fs.createReadStream('input.txt')
    .pipe(zlib.createGzip())
    .pipe(fs.createWriteStream('input.txt.gz'));
console.log("文件压缩完成。");

// 解压
var fs = require("fs");
var zlib = require('zlib');
// 解压 input.txt.gz 文件为 input.txt
fs.createReadStream('input.txt.gz')
    .pipe(zlib.createGunzip())
    .pipe(fs.createWriteStream('input.txt'));
console.log("文件解压完成。");
```

8.9 课程总结

- NodeJS介绍
  - Node.js历史
  - 版本
  - 作用
- NodeJS安装与使用
  - NodeJS 下载
  - NodeJS 安装
- NodeJS模块
  - module（模块）
  - 内置模块
- 回调函数
  - 阻塞
  - 非阻塞
- 异步编程
  - 事件发布/订阅模式
  - Promise/Deferred模式
  - 流程控制库
- Buffer缓存区
  - Buffer.from(array)
  - Buffer.alloc(size[, fill[, encoding]])
  - Buffer.allocUnsafe(size)
  - Buffer 与字符编码
- fs文件基本操作
  - 基本用法
  - fs.open(path, flags[, mode], callback) 异步的方式打开文件
  - fs.readFile(path[, options], callback) 异步的方式读取文件
  - fs.write(fd, buffer[, offset[, length[, position]]], callback) 异步的方式写入文件
  - fs.close(fd, callback) 异步的方式关闭文件
  - fs.unlink(path, callback) 异步的方式删除文件
  - fs.readdir(path[, options], callback) 读取一个目录
- fs流
  - fs.createReadStream(path[, options]) 创建一个可读流
  - fs.createWriteStream(path[, options]) 创建一个可写流
  - readerStream.pipe(writerStream) 将可读流信息导入可写流（即管道流）
  - 链式流

8.10 实训

1.为什么用Nodejs,它有哪些缺点？（简答题）

2.如何避免回调地狱？（简答题）

3. 什么是错误优先的回调函数？（简答题）

4. 通过fs模块的功能删除一个没用的文件。（编程题）

5.通过fs模块的功能实现文件的写入操作。（编程题）