第2章 Node.js编程基础

Node.js开发实战教程

内容导航

Contents

Section 00

【学习目标】

（1）了解JavaScript的基本语法；
（2）能读懂JavaScript代码。

Section 01

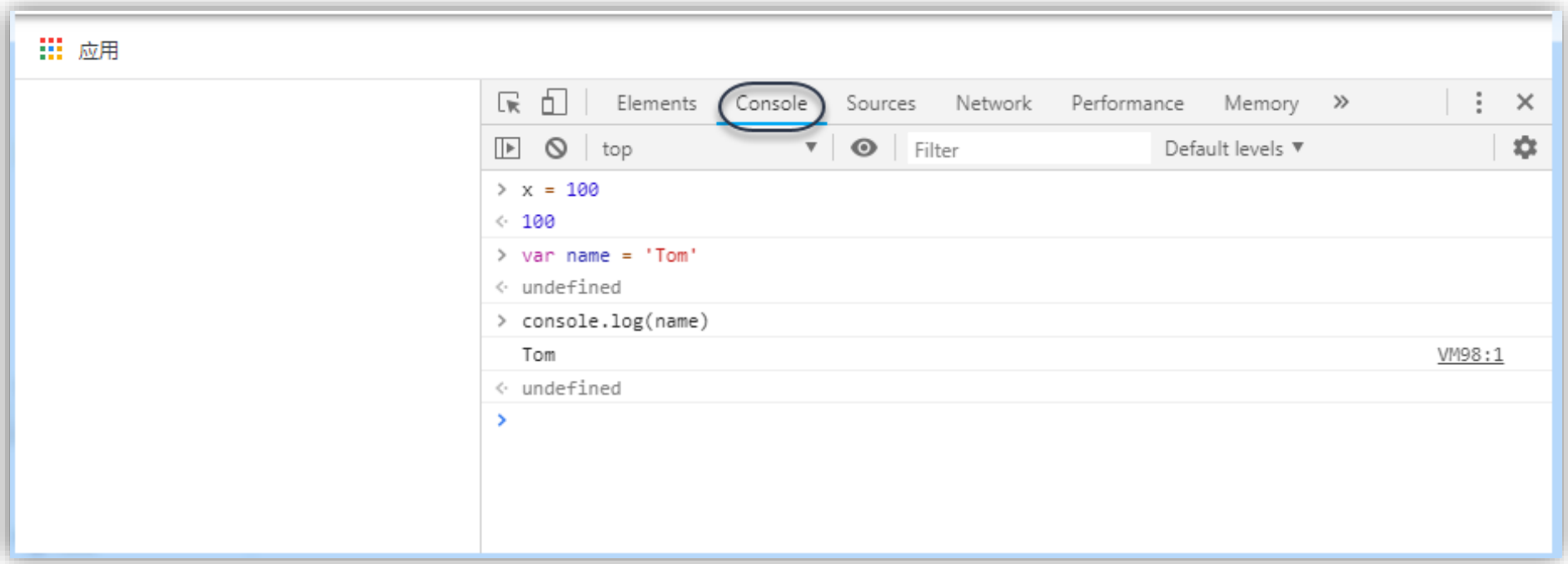
【JavaScript版本】

- ECMA组织制定了JavaScript语言的标准——ECMAScript（简称ES）。
- 大多数场合使用JavaScript名称，可将JavaScript看作是ES标准的实现语言。
- 主要版本：
  - 2009年——ES5；
  - 2015年——ES6（ES2015）；
  - ES7（ES2016）、ES8（ES2017）、ES9（ES2018）和ES10（ES2019）。
- ES6是目前的主流版本，Node.js自6.0版本开始全面支持ES6。
- Node.js自7.6版本开始就默认支持async/await异步编程。

Section 02

【JavaScript运行环境】

- Node.js REPL交互式运行环境
- 浏览器控制台



Section 03

【JavaScript语句与注释】

▸▸▸ 语句

- JavaScript每条语句都以分号“;”结束。
- 一行代码可包含多条语句。
- 一行语句太长，则可以使用续行符“\”进行换行。

▸▸▸ 语句块

- 语句块是一组语句的集合，作为一个整体使用大括号“{}”封装。
- 语句块可以嵌套，形成层级结构。

▸▸▸ 注释

- 行注释

```
// 我是单独一行注释
alert('Hello World'); // 我是句尾的行注释
```

- 块注释

```
/* 我要开始注释啦
我要结束注释啦*/
```

Section 04 【变量】

▸▸▸ 变量的命名

- JavaScript的变量可以是任意数据类型。
- 变量名可以是大小写英文字母、数字、符号“$”或“_”的任意组合，但不能以数字开头。

▸▸▸ 变量的声明与赋值

- 弱类型的编程语言，所有数据类型都可以用var关键字声明。在定义变量时无须指定变量类型。

```
var hello; // 声明一个名为hello的变量，此时该变量的值为undefined，表示未定义
```

- 使用等号对变量进行赋值，可以将任意数据类型赋值给变量。

```
hello='我是个字符串'; //此时变量的值为“我是个字符串”
```

- 可以在声明变量的同时对变量进行赋值。
- 可以反复赋值同一个变量。

Section 04 【变量】

▸▸▸ 变量提升

- 变量可以在声明之前使用，值为undefined。

```
console.log(temp);// 返回undefined
var temp = '你好';
```

- ES6用let关键字改变这种行为，变量一定要在声明之后使用。

▸▸▸ 变量泄露

- 用来计数的循环变量使用var关键字声明后会泄露为全局变量。

```
var temp = 'Hello!';
for (var i = 0; i < temp.length; i++) {
    console.log(temp[i]);
}
console.log(i); // 返回数字6
```

- 改用let关键字来声明循环变量避免变量泄露。

Section 04 【变量】

▸▸▸ 全局作用域和函数作用域

- ES5中只有全局作用域（顶层作用域）和函数作用域。

```
var temp = '你好！';//全局作用域
function testScope() {
    var temp = '早上好！'; //函数作用域
    console.log(temp);
}
testScope(); //返回函数作用域中的“早上好”！
console.log(temp); //返回全局变量的“你好”！
```

Section 04 【变量】

▸▸▸ 全局作用域和函数作用域

- ES5中只有全局作用域（顶层作用域）和函数作用域。

```
var temp = '你好！';//全局作用域
function testScope() {
    var temp = '早上好！'; //函数作用域
    console.log(temp);
}
testScope(); //返回函数作用域中的“早上好”！
console.log(temp); //返回全局变量的“你好”！
```

Section 04 【变量】

▸▸▸ 块级作用域与let关键字

- ES6引入块级作用域，使用let关键字声明的变量只能在当前块级作用域中使用。

```
function testBlockScope() {
    let name = '小明';
    if (true) {
        let name = '小红';
        console.log(name); // 返回“小红”
    }
    console.log(name); // 返回“小明”
}
```

▸▸▸ 使用const关键字声明只读常量

- ES6引入const关键字声明只读的常量。

```
const PI = 3.1415;
```

Section 05 【数据类型】

▸▸▸ 数值（Number）

- 不区分整数和浮点数，统一用数值表示。
- 十六进制数使用0x作为前缀。二进制和八进制数值分别使用前缀0b（或0B）和0o（或0O）。
- 无法计算结果时就可用NaN表示；Infinity表示无限大。

▸▸▸ 字符串（String）

- 字符串是用单引号“'”或双引号“"”括起来的任意文本。
- ES6提供模板字符串，可使用反引号包括整个模板字符串，使用${}将变量括起来。

```
var msg = `服务器侦听监听地址和端口：${srvip}:${port}，请注意！`;
```

- 模板字符串中也可以不嵌入任何变量，通常用于按实际格式输出（如换行）。

Section 05

【数据类型】

布尔值（Boolean）

- 布尔值只有true、false两种，经常用于条件判断中。
- 在比较是否相等时，建议使用===而不要使用==。

null和undefined

- null表示一个空值，即什么也没有。
- undefined表示“未定义”，仅用于判断函数参数是否正常传递。

数组（Array）

- JavaScript的数组可以包括任意数据类型。
- 数组用[]表示，元素之间用逗号分隔。
- 数组的元素可以通过索引来访问，注意索引的起始值为0。

Section 05 【数据类型】

▸▸▸ 对象（Object）

- 对象是一组由键值对组成的无序集合，用{}表示，键值对之间用逗号分隔。

```
var myObj = {
  isobj: true,
  num: [1,2,3],
  desp: '对象好像可以无所不包'
};
```

- 键均为字符串类型，而值可以是任意数据类型。
- 获取一个对象的属性可用“对象名.属性（键）名”的方式。
- ES6允许将表达式作为对象的属性名，即把表达式放在方括号内。

```
let numproperty = 'num';
var myObj = {
    [numproperty]: [1,2,3],
  ['des'+'cription']: '我是个对象'
};
```

```
myObj[numproperty]  //[1,2,3]
myObj[‘num’]  //[1,2,3]
myObj[‘description’]
```

Section 05 【数据类型】

▸▸▸ 符号（Symbol）

- ES6引入数据类型Symbol，用于表示独一无二的值，其值通过Symbol()函数自动生成。
- Symbol值用于对象的属性名，可以有3种表示方法。

```
let welcome = Symbol();//自动产生一个值
// 第1种表示方法
let myObj = {};
myObj[welcome] = '欢迎光临';
// 第2种表示方法
let myObj = { [welcome]: '欢迎光临' };
// 第3种表示方法
let myObj = {};
Object.defineProperty(myObj, welcome, { value: '欢迎光临' });
```

Section 05 【数据类型】

▸▸▸ 映射（Map）

- ES6引入Map数据结构，与对象类似，但各种类型的数据（甚至对象）都可以作为键。
- Map本身是一个构造函数，用于生成Map数据结构。

```
const myMap = new Map();
```

- 可以使用Map结构的set方法添加成员

```
const myObj = {welcome: '欢迎光临'};
myMap.set(myObj,'我是一个对象');
```

- 使用Map结构的get方法读取键（成员）

```
myMap.get(myObj); //结果为'我是一个对象
```

- Map()函数也可以将一个数组作为参数，该数组的成员是表示键值对的数组

```
const myMap = new Map([
  [ ‘name’ ,’ 王刚’ ],
  [ ‘title’ ,’ 博士’ ]
]);
myMap.get( ‘name’ ); //返回 “王刚”
```

Section 05 【数据类型】

▸▸▸ 映射（Map）

- Map结构的实例支持遍历方法。
  - keys()：返回键名的遍历器。
  - values()：返回键值的遍历器。
  - entries()：返回键值对的遍历器。
  - forEach()：使用回调函数遍历每个成员。

Section 05 【数据类型】

▸▸▸ 集合（Set）

- 集合是无重复的、无序的数据结构，类似于数组，即没有重复的值。
- 集合是一个构造函数，用于生成Set数据结构。

```
const mySet = new Set();
```

- 可以通过add()方法向Set结构加入成员。
- Set()函数可将Iterable类型的数据结构（数组、集合或映射）作为参数，用于初始化集合。

```
const mySet = new Set([1, 2, 3, 4, 4]);//会自动过滤掉其中一个数字4
```

- Set结构中的元素可以看作是键，与Map结构不同的是，它只有键名没有键值。
- Set结构使用与Map结构相同的4种遍历方法来遍历成员。

遍历数组可以采用下标循环，而遍历映射和集合就无法使用下标。为了统一集合类型，ES6引入了新的Iterable类型，数组、映射和集合都属于Iterable类型。这种类型的集合可以通过新的for ... of循环来遍历。更好的遍历方式是使用Iterable类型内置的forEach方法。

forEach用法

- 以数组arr为例

```
arr.forEach(function(element,index,array){....});
```

- 以映射map为例

```
map.forEach(function(value,key,map){....});
```

- 以集合set为例

```
set.forEach(function(element,sameElement,set){....});
```

Section 06 【流程控制】

▸▸▸ 分支结构

- if () { … } else { … }
- if () { … } else if () { … } else { … }
- switch … case

```
switch(变量)
{
    case 值1:
        代码1;
        break;
    case 值2:
        代码2;
        break;
    default:
        如果以上条件都不满足，则执行该代码;
}
```

Section 06 【流程控制】

▸▸▸ 循环结构

- for
- for ... in

```
for (var key in obj) {
    console.log(key);
}
```

- while
- do ... While
- break
- continue

循环结构中的条件需使用括号括起。

Section 07

【JavaScript的函数】

▸▸▸ 函数声明

- 声明函数需要使用function关键字。

```
function sumAge(x,y) {
    return  x + y;
}
```

- ES6直接支持默认参数

```
sumAge(x=19,y=20)
```

- 在ES5中设置默认参数需要采用变通的方法

```
function sumAge (x,y) {
    x = arguments[0]===undefined ? 21 : arguments[0]; //设置参数x的默认值为21
    y = arguments[1]===undefined ? 20 : arguments[1]; //设置参数y的默认值为20
    return  x + y;
}
```

Section 07

【JavaScript的函数】

▸▸▸ 函数调用

➢ 声明函数之后使用“函数名(参数)”的形式，按顺序传入参数进行调用。

```
sumAge(22,20); //两人加起来42岁
```

▸▸▸ arguments对象

➢ JavaScript的函数默认带有arguments对象，利用该对象可以获得调用者传入的所有参数。

```
function sumAge (){
    var sum = 0;
    var numcount= arguments.length
    for (var i=0;i<numcount;i++){
      sum += arguments[i];
    }
    return sum;
}
```

▸▸▸ rest参数

➢ ES6引入rest参数（形式为“...变量名”），用于获取函数的多余参数，以代替arguments对象。

```
function sumAge(...values){
    var  sum = 0;
    for (var val of values){
        sum += val;
    }
    return sum;
}
```

Section 07

【JavaScript的函数】

▸▸▸ 匿名函数

- 声明函数时省略函数名。

```
var sumAge = function (x,y) {
    return  x + y;
};
```

Section 07

【JavaScript的函数】

▸▸▸ 箭头函数

➢ 使用箭头符号（=>）定义函数。

示例	箭头函数	普通函数

一个参数	var f = x => x;	var f = function (x) {
		  return x;
		};

多个参数	var sumAge = (x, y) => x + y;	var sumAge = function (x,y) {
		  return x + y;
		};

无参数	var f = () => 20;	var f = function () { return 20 };

代码块包含多条语句	var diffAge = (x, y) => {
	  var diff = x – y;
	  return Math.abs(diff);
	}
		var diffAge = function (x,y) {
		  var diff = x – y;
		  return Math.abs(diff);
		}

Section 07

【JavaScript的函数】

▸▸▸ 高阶函数

- 以一个或多个函数作为参数的函数。

```
function diffAge(m, n, abs) {
    return abs(m-n);
}
diffAge(19, 22, Math.abs);
```

Section 07

【JavaScript的函数】

▸▸▸ 闭包

- 当函数作为返回值，或者作为参数传递时，该函数就被称为闭包。
- 闭包是能够读取其他函数内部变量的函数，可以使用函数之外定义的变量。

```
var basePrice = 10.00;//起步价
var baseMiles = 3.00;        //起步里程
function taxiPrice(unitPrice, Mileage) {
  function totalPrice() {   //计算总费用； 这是定义在一个函数内部的函数
    if ( Mileage > baseMiles) {   //超过起步里程
      return  unitPrice*Mileage;  //单价与里程相乘
    }
    else{                    //在起步里程内
      return basePrice;
    }
  }
  return totalPrice ();
}
taxiPrice(2.00,6.00);//打车费用12.00
```

Section 08

【JavaScript的类】

ES5通过构造函数生成实例对象	ES6引入类作为对象的模板，通过class关键字定义类

function Visitor(name, sex) { //来宾信息
  this.name = name;
  this.sex = sex;
}
Visitor.prototype.getInfo = function () {
  return this.name + ', ' + this.sex;
};
var visitor = new Visitor('张勇', '先生');
	class Visitor{
	  constructor(name, sex) {
	  this.name = name;
	  this.sex = sex;
	  }
	  getInfo() {
	  return this.name + ', ' + this.sex;
	  }
	}
	var visitor = new Visitor('张勇', '先生');

prototype 属性使您有能力向对象添加属性和方法。

Section 09

【严格模式】

- 变量必须声明后使用。
- 不能出现两个命名参数同名的情况，否则报错。
- 不能使用with语句。
- 不能对只读属性赋值，否则报错。
- 不能使用前缀0表示八进制数，否则报错。
- 不能删除不可删除的属性，否则报错。
- 不能删除变量，只能删除属性。
- arguments不能被重新赋值。
- 不能使用arguments.callee和arguments.caller属性。
- 禁止this关键字指向全局对象。
- 不能使用fn.caller和fn.arguments属性获取函数调用的堆栈。
- 增加了保留字（如protected、static和interface）。

Section 10

【JavaScript编程规范】

▸▸▸ 代码格式

- 每条JavaScript语句应该以分号结束。
- 缩进使用2个半角空格或4个半角空格，而不使用Tab键。
- 每行仅声明一个变量，而不要声明多个变量。
- 字符串尽量使用单引号。
- 符号“{”应在行末，表示代码块的开始，符号“}”应在行首，表示代码块的结束。

▸▸▸ 命名规范

- 变量推荐使用小驼峰命名法。
- 常量名建议全部大写。
- 函数也采用小驼峰命名法，建议前缀部分为动词。
- 类和构造函数采用大驼峰命名法，同时建议前缀部分为名词。
- 文件命名尽量采用下划线分割单词。

Section 10

【JavaScript编程规范】

函数

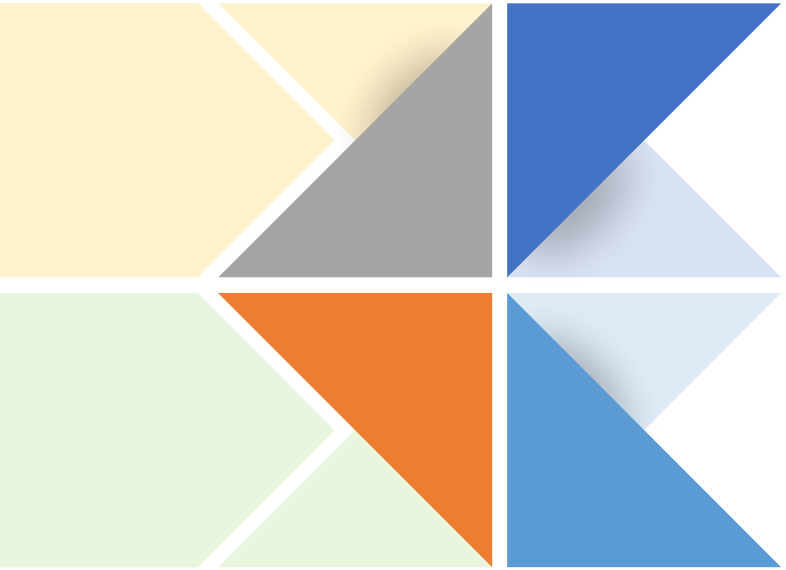
- 函数的实现代码尽可能短小精悍，便于阅读。
- 避免多余的else语句，尽早执行return语句。
- 尽可能为闭包命名，便于调试跟踪。
- 不要嵌套闭包。
- 使用方法链时，每行仅调用一个方法，并使用缩进表明方法的并列关系。

注释

- 注释要尽量简单，清晰明了，同时注重注释的意义。
- 尽可能从更高层次说明代码的功能。
- 尽可能使用英文注释。

内容导航

Contents



Section 00

【学习目标】

（1）理解Node.js的回调函数；
（2）掌握回调函数的使用。

Section 01

【什么是回调函数】

```
function main(info, callback){//我是主函数，参数列表中的callback是一个回调函数
    console.log('还在回家的路上');
    console.log('到家了，发条信息吧');
    callback (info);//调用回调函数
}
function sendMsg(msg){  //我是回调函数
    console.log(msg);
}
main('亲爱的，我到家了！',sendMsg);//执行主函数
```

Section 02

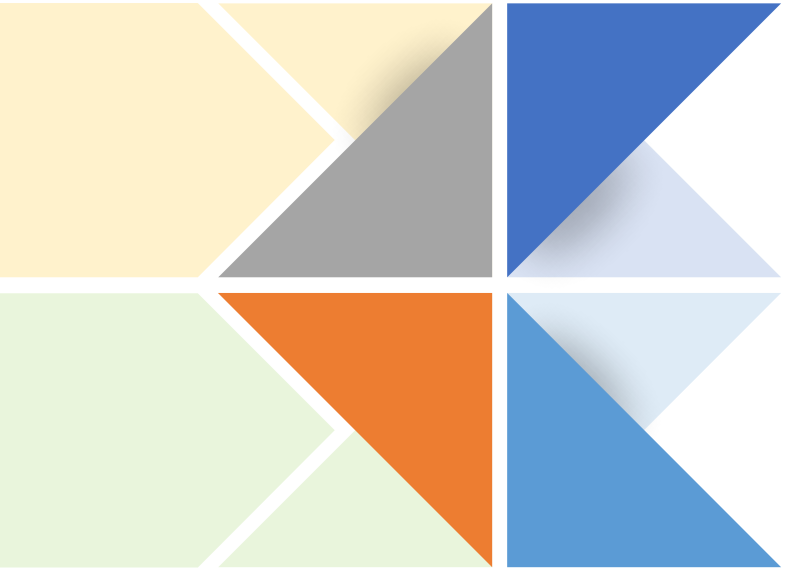
【回调函数示例】

程序代码	执行结果

const fs = require("fs");//引入fs（filesystem）模块
//异步读取文件内容
fs.readFile('demo.txt', function (err, data) {
　　if (err) return console.error(err); //读取失败则报错
　　console.log(data.toString());//读取成功则输出文件内容
});
console.log("Node程序已经执行结束!");
	C:\nodeapp\ch02>node test_callback.js
	Node程序已经执行结束!
	周董好久没有发新歌了，左盼右盼，终于等来了《说好不哭》

内容导航

Contents



Section 00 【学习目标】

（1）理解Node.js的事件机制；
（2）掌握事件的监听与触发的用法。

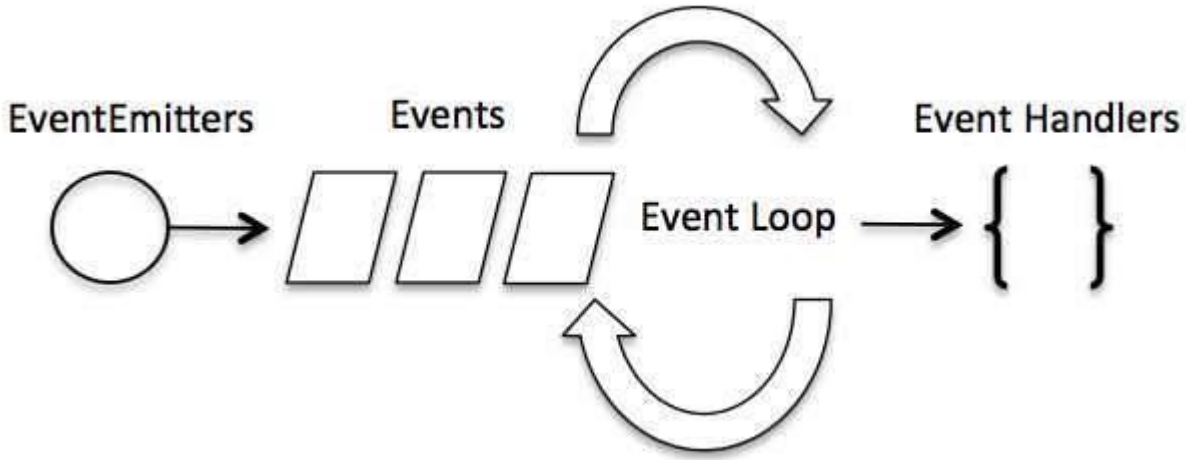
Section 01 【事件循环】

- Node.js 是单进程单线程应用程序，但是因为 V8 引擎提供的异步执行回调接口，通过这些接口可以处理大量的并发，所以性能非常高。
- Node.js 几乎每一个 API 都是支持回调函数的。
- Node.js 基本上所有的事件机制都是用设计模式中观察者模式实现。
- Node.js 单线程类似进入一个while(true)的事件循环，直到没有事件观察者退出，每个异步事件都生成一个事件观察者，如果有事件发生就调用该回调函数.

Section 00 【学习目标】

（1）理解Node.js的事件机制；
（2）掌握事件的监听与触发的用法。

Section 01 【事件循环】



Section 02

【Node.js事件的监听与触发】

▶▶▶ 事件监听与触发

Node.js 有多个内置的事件，我们可以通过引入 events 模块，并通过实例化 EventEmitter 类来绑定和监听事件， EventEmitter 类的核心就是事件触发与事件监听器功能的封装。

```
const EventEmitter = require('events'); //引入事件模块
const myEmitter = new EventEmitter.EventEmitter();//创建EventEmitter对象用于监听
//注册seen事件用于监视，有人来了就报告
myEmitter.on('seen', () => {
  console.log('报告，有人来了');
});
myEmitter.emit('seen'); //触发（发射）seen事件进行报告
```

Section 02

【Node.js事件的监听与触发】

▸▸▸ EventEmitter类常用API

- EventEmitter.on(event,listener)、emitter.addListener(event,listener)：为指定事件注册一个监听器，参数event和listener分别表示事件名称和回调函数。
- EventEmitter.once(event,listener)：为指定事件注册一个单次监听器，即监听器最多触发一次，触发后立刻解除该监听器。
- EventEmitter.emit(event,[arg1],[arg2],[…..])：触发由event参数指定的事件，传递若干可选参数到事件监听器的参数表。
- EventEmitter.removeListener(event,listener)：删除指定事件的某个监听器，参数listener必须是该事件已经注册过的监听器。
- emitter.listeners(event)：返回由event参数指定的事件的监听器的数组。
- emitter.setMaxListeners(n)：设置emitter实例的最大事件监听数，默认是10个，设置0为不限制。
- emitter.removeAllListeners(event)：删除所有由event参数指定的事件的监听器。

Section 02

【Node.js事件的监听与触发】

▸▸▸ EventEmitter类常用API

为一个事件注册多个监听器。

```
const EventEmitter = require('events').EventEmitter;     // 加载事件模块
var event = new EventEmitter();     // 实例化事件模块
// 注册事件(seen)
event.on('seen', function(who) {
    console.log('报告，来人是一位', who);
});
// 再次注册事件(seen)
event.on('seen', function() {
    console.log('欢迎光临！');
});
event.emit('seen', '女士');     // 发射(触发)事件(seen)
```

Section 02

【Node.js事件的监听与触发】

▸▸▸ 处理error事件

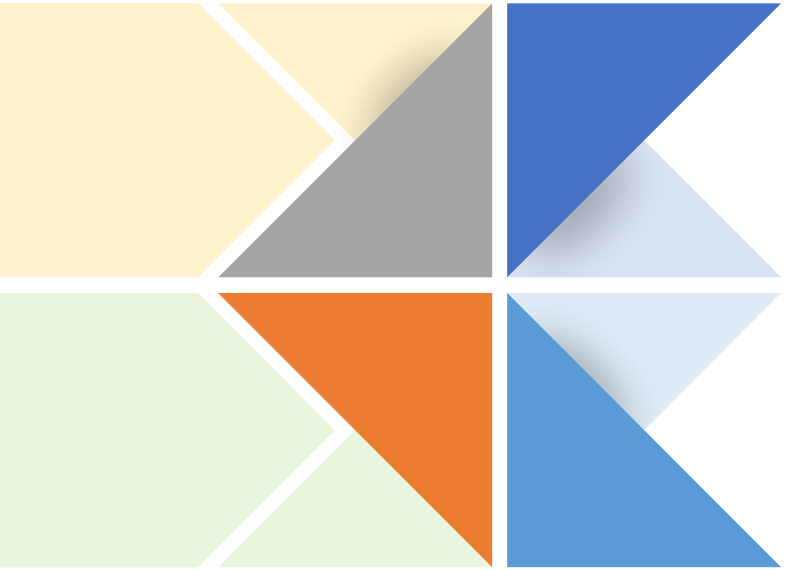
- EventEmitter 定义了一个特殊的事件 error，它包含了错误的语义，我们在遇到 异常的时候通常会触发 error 事件。
- 当 error 被触发时，EventEmitter 规定如果没有响 应的监听器，Node.js 会把它当作异常，退出程序并输出错误信息。
- 我们一般要为会触发 error 事件的对象设置监听器，避免遇到错误后整个程序崩溃。

```
var events=require('events');
var myEvent = new events.EventEmitter();
myEvent.emit('error', new Error('whoops!'));
```

```
var events=require('events’);
var myEvent = new events.EventEmitter
();
myEvent.on('error', (err) => {
    console.log('whoops! there was an error');
});
myEvent.emit('error', new Error('whoops!'));
```

内容导航

Contents



Section 00

【学习目标】

（1）了解Node.js的全局对象和全局变量；
（2）掌握console和process的用法。

Section 01

【全局变量】

- __filename指向当前正在执行的脚本文件名。
- __dirname指向当前运行的脚本所在的目录。

运行以下语句，对比结果：

```
console.log(__filename);
console.log(__dirname);
```

Section 02

【console模块】

▸▸▸ console 模块提供了一个简单的调试控制台，类似于 Web 浏览器提供的 JavaScript 控制台。

▸▸▸ 该模块导出两个特定组件：

- Console 类，包含 console.log()、 console.error() 和 console.warn() 等方法，可用于写入任何 Node.js 流。

- 全局的 console 实例，配置为写入 process.stdout 和 process.stderr。使用时无需调用 require('console')。

Section 02

【console模块】

▸▸▸ 全局console实例

```
console.log('hello world'); // 打印hello world到标准输出流
console.log('hello %s', 'world');// 打印hello world到标准输出流
console.error(new Error('错误信息')); 打印 [Error: 错误信息] 到标准错误流
const name = 'Robert';
console.warn(`Danger ${name}! Danger!`);// 打印Danger Robert! Danger!到标准错误流
```

▸▸▸ Console类

```
const out = getStreamSomehow();
const err = getStreamSomehow();
const myConsole = new console.Console(out, err);
myConsole.log('hello world'); // 打印hello world到out流
myConsole.log('hello %s', 'world'); //打印hello world到out流
myConsole.error(new Error('错误信息')); //打印 [Error: 错误信息] 到 err流
const name = 'Robert';
myConsole.warn(`Danger ${name}! Danger!`); 打印Danger Robert! Danger!到 err流
```

Section 03

【process对象】

process对象是Node的一个全局对象，提供当前Node进程的信息。它可以在脚本的任意位置使用，不必通过require命令加载。该对象部署了EventEmitter接口。

▸▸▸ 进程事件

```
process.on('exit', function(code) {
    // 以下代码永远不会执行
    setTimeout(function(){
      console.log("该代码不会执行");
    }, 0);
    console.log('退出码为:',code);
});
console.log("程序执行结束");
```

▸▸▸ 退出状态码

- 如果没有异步操作任务正在等待执行，则Node.js会以状态码0正常退出。
- 其他情形使用相应的状态码。

Section 03

【process对象】

process对象的属性

- 获取process对象的属性

```
process.stdout.write("Hello World!" + "\n");// 将字符串输出到终端
//通过参数读取
process.argv.forEach(function(val, index, array) {
    console.log(index + ': ' + val);
});
console.log(process.execPath); // 获取执行路径
console.log(process.platform); // 获取平台信息
```

- 获取命令行参数

```
console.log('读取命令行参数：', process.argv);
console.log('第1个参数：', process.argv[2]);
```

Section 03

【process对象】

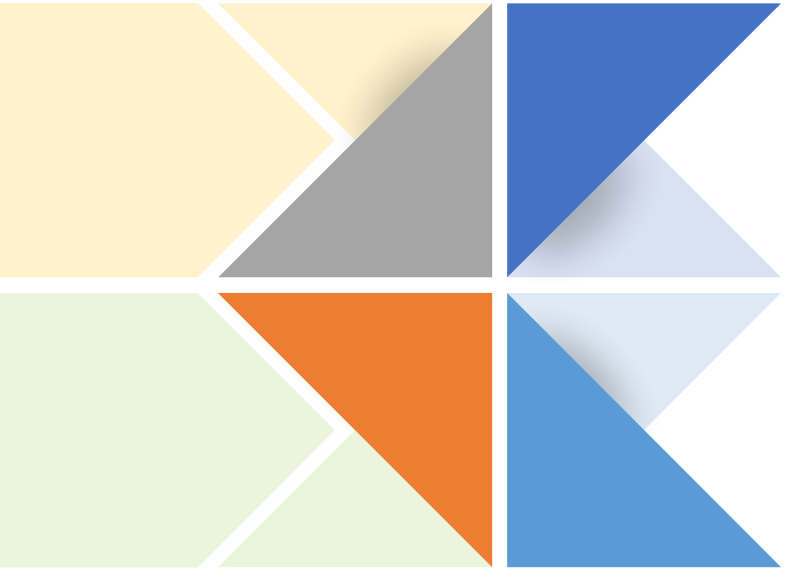
process对象的方法

- process对象提供多种方法用于控制与系统的交互

```
console.log('当前目录: ' + process.cwd());
console.log('当前版本: ' + process.version);
console.log(process.memoryUsage());
```

内容导航

Contents



Section 00 【学习目标】

（1）了解Node.js的定时器；
（2）掌握有关定时器的类和方法的使用。

Section 01 【设置定时器】

▸▸▸ 一次性定时器

- 基本用法

```
setTimeout(callback, delay[, ...args])
```

- 示例

```
setTimeout(function(){
    console.log('我是一个一次性的定时器');
},1000);
```

Section 01

【设置定时器】

▸▸▸ 周期性定时器

- 基本用法

```
setInterval(callback, delay[, ...args])
```

- 示例

```
setInterval (function(){
   console.log('我是一个周期性的定时器');
},1000);
```

▸▸▸ 即时定时器

- 基本用法

```
setImmediate(callback[, ...args])
```

Section 02

【取消定时器】

- 分别用clearTimeout()、clearInterval()和clearImmediate()方法取消相应定时器，防止该定时器触发。

```
var testInterval=setInterval(testFunc,2000);
...
clearInterval(testInterval);
```

Section 03

【Timeout和Immediate类】

- Node.js内置两个有关定时器的类Timeout和Immediate，可用于创建相应的对象。
- Timeout对象在内部创建，并由setTimeout()或setInterval()方法返回，可以传递给clearTimeout()或clearInterval()以取消定时器。
- Immediate对象也在内部创建，并由setImmediate()方法返回。它可以传递给clearImmediate()以取消即时定时器。

Section 04

【setImmediate()方法与setTimeout()方法的对比】

	程序代码	执行结果

在一个I/O周期（即主模块）内调用的比较	setTimeout(() => {
	  console.log('一次性');
	}, 0);
	setImmediate(() => {
	  console.log('即时性');
	});
		C:\nodeapp\ch02>node timeout_vs_immediate1.js
		一次性
		即时性
		C:\nodeapp\ch02>node timeout_vs_immediate1.js
		即时性
		一次性

同一个I/O循环内调用的比较	const fs = require('fs');
	fs.readFile(__filename, () => {
	  setTimeout(() => {
	  console.log('一次性');
	  }, 0);
	  setImmediate(() => {
	  console.log('即时性');
	  });
	});
		C:\nodeapp\ch02>node timeout_vs_immediate2.js
		即时性
		一次性
		C:\nodeapp\ch02>node timeout_vs_immediate2.js
		即时性
		一次性

Section 05

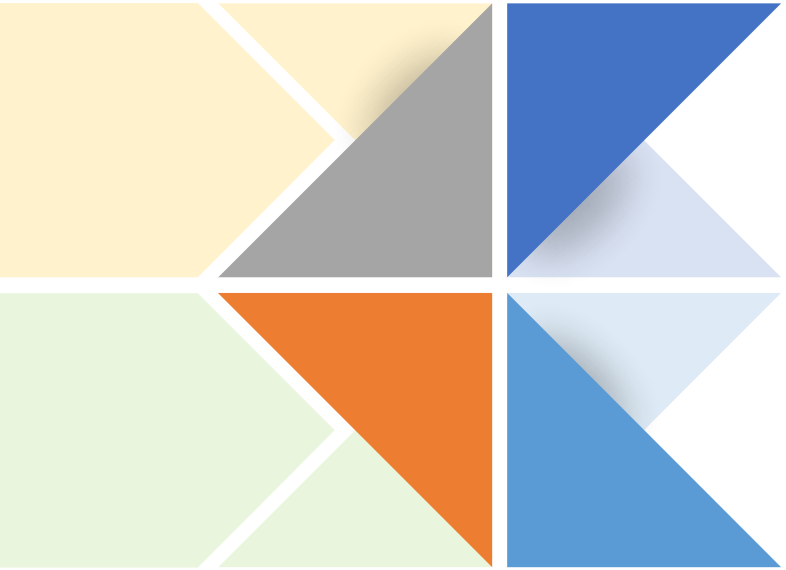
【process.nextTick()与setImmediate()的对比】

- process.nextTick()在当前阶段立即执行。
- setImmediate()在下一次迭代或事件循环的tick事件上被触发。
- process.nextTick()的回调函数执行的优先级要高于setImmediate()。

程序代码	执行结果
console.log('开始'); process.nextTick(() => { console.log('下一个时间点的回调'); }); console.log('调度');	开始、调度、下一个时间点的回调

内容导航

Contents



Section 00

【学习目标】

（1）理解Buffer的概念；
（2）掌握Buffer数据类型的使用。

Section 01

【创建Buffer实例】

- 可以将Buffer视为一种用来处理二进制数据的数据类型。
- Buffer类的实例（即对象）类似于整数数组，但实例对应于固定大小的原始内存分配，其大小在创建时被确定且无法更改。

Section 01

【创建Buffer实例】

➢ Buffer实例创建使用Buffer.from()、Buffer.alloc()或Buffer.allocUnsafe()方法。

■ Buffer 提供了以下 API 来创建 Buffer 类：

- **Buffer.alloc(size[, fill[, encoding]])：** 返回一个指定大小的 Buffer 实例，如果没有设置 fill，则默认填满 0
- **Buffer.allocUnsafe(size)：** 返回一个指定大小的 Buffer 实例，但是它不会被初始化，所以它可能包含敏感的数据
- **Buffer.allocUnsafeSlow(size)**
- **Buffer.from(array)：** 返回一个被 array 的值初始化的新的 Buffer 实例（传入的 array 的元素只能是数字，不然就会自动被 0 覆盖）
- **Buffer.from(arrayBuffer[, byteOffset[, length]])：** 返回一个新建的与给定的 ArrayBuffer 共享同一内存的 Buffer。
- **Buffer.from(buffer)：** 复制传入的 Buffer 实例的数据，并返回一个新的 Buffer 实例
- **Buffer.from(string[, encoding])：** 返回一个被 string 的值初始化的新的 Buffer 实例

Section 01

【创建Buffer实例】

```
// 创建一个包含数组[0x1, 0x2, 0x3]的Buffer实例
    const buf1 = Buffer.from([1, 2, 3]);
    // 创建一个包含 UTF-8 字节 [0x74, 0xc3, 0xa9, 0x73, 0x74] 的 Buffer实例
    const buf2 = Buffer.from('tést');
    // 创建一个包含 Latin-1（说明见2.6.2节）字节 [0x74, 0xe9, 0x73, 0x74] 的
Buffer实例
    const buf3 = Buffer.from('tést', 'latin1');
    // 创建一个长度为 10、且用零填充的 Buffer实例
    const buf4 = Buffer.alloc(10);
    // 创建一个长度为 10、且用 0x1 填充的 Buffer实例
    const buf5 = Buffer.alloc(10, 1);
    /* 创建一个长度为 10、且未初始化的 Buffer实例。这个方法比调用
Buffer.alloc()更快，
    但返回的 Buffer 实例可能包含旧数据，因此需要使用 fill() 或 write() 重写。*/
    const buf6 = Buffer.allocUnsafe(10);
```

Section 02

【Buffer用于编码转换】

- Buffer实例一般用于表示编码字符的序列，如UTF-8、UCS2、Base64或十六进制编码的数据。
- 在文件操作和网络操作中，如果没有显式声明编码格式，返回数据的默认类型为Buffer。
- 通过使用显式字符编码将Buffer实例与JavaScript字符串相互转换。
- 在创建Buffer实例时指定存入字符串的字符编码

```
const buf = Buffer.from('hello world', 'ascii');
```

- 将已创建的Buffer实例转换成字符串的用法

```
buf.toString([encoding[, start[, end]]])
```

- 示例

```
const buf = Buffer.from('tést');
console.log(buf.toString('hex'));// 输出结果: 74c3a97374
console.log(buf.toString('utf8', 0, 3));//输出结果：té
```

Section 03

【将Buffer实例转换为JSON对象】

- 使用buf.toJSON()方法将Buffer实例转换为JSON对象，适用于将二进制数据转换为JSON格式。
- 示例

```
const buf = Buffer.from([0x1, 0x2, 0x3, 0x4, 0x5]);
const json = JSON.stringify(buf);
console.log(json);  // 输出：{"type":"Buffer","data":[1,2,3,4,5]}
const copy = JSON.parse(json, (key, value) => {
    return value && value.type === 'Buffer' ?
        Buffer.from(value.data) :
        value;
});
console.log(copy);  // 输出：  <Buffer 01 02 03 04 05>
```

Section 04

【Buffer实例基本操作】

写入Buffer实例

- 使用buf.write()方法将字符串写入Buffer实例

```
buf.write(string[, offset[, length]][, encoding])
```

- 示例

```
const buf = Buffer.alloc(256);
const len = buf.write('\u00bd + \u00bc = \u00be', 0);
console.log(`${len} 个字节: ${buf.toString('utf8', 0, len)}`);
// 输出: 12 个字节: ½ + ¼ = ¾
```

从Buffer实例读取数据

- 使用buf.toString()方法从Buffer实例读取字符串
- 使用其他专用方法从Buffer实例读取其他类型的数据

```
const buf = Buffer.from([-1, 5]);
console.log(buf.readInt8(0));// 输出结果: -1
console.log(buf.readInt8(1));// 输出结果: 5
console.log(buf.readInt8(2));// 抛出异常 ERR_OUT_OF_RANGE（超出范围）
```

Section 04

【Buffer实例基本操作】

▸▸▸ Buffer实例合并

➢ 使用Buffer.concat()方法

```
Buffer.concat(list[, totalLength])
```

参数描述如下：
list - 用于合并的 Buffer 对象数组列表。
totalLength - 指定合并后Buffer对象的总长度。
返回一个多个成员合并的新 Buffer 对象。

```
var buffer1 = Buffer.from((‘你好'));
var buffer2 = Buffer.from((Node.js'));
var buffer3 = Buffer.concat([buffer1,buffer2]);
console.log("buffer3 内容: " + buffer3.toString());
```

Section 04

【Buffer实例基本操作】

▸▸▸ Buffer实例复制

➢ 使用buf.copy()方法

```
buf.copy(target[, targetStart[, sourceStart[, sourceEnd]]])
```

■ 参数描述如下：

- **target** - 要拷贝的 Buffer 对象。
- **targetStart** - 数字, 可选, 默认: 0
- **sourceStart** - 数字, 可选, 默认: 0
- **sourceEnd** - 数字, 可选, 默认: buffer.length
- 无返回值

```
var buf1 = Buffer.from('abcdefghijkl');
var buf2 = Buffer.from( 'OOOO');
//将 buf2 插入到 buf1 指定位置上
buf2.copy(buf1, 2);
console.log(buf1.toString());
```

Section 04

【Buffer实例基本操作】

▸▸▸ Buffer实例切片

➢ 使用buf.slice()方法

```
buf.slice([start[, end]])
```

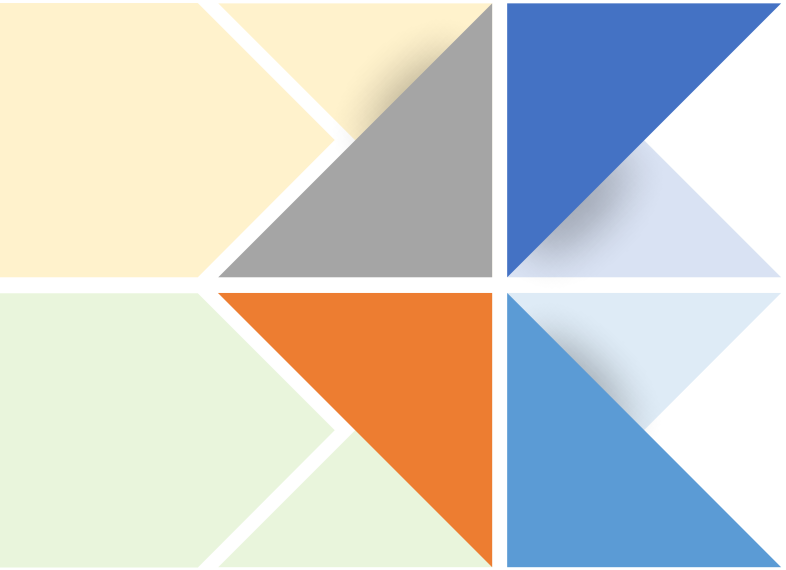
■ 参数描述如下：

- **start** - 数字, 可选, 默认: 0
- **end** - 数字, 可选, 默认: buffer.length
- 返回一个新的缓冲区，它和旧缓冲区指向同一块内存，但是从索引 start 到 end 的位置剪切。

```
var buffer1 = Buffer.from( ‘helloworld’ );
// 剪切缓冲区
var buffer2 = buffer1.slice(0,2);
console.log("buffer2 content: " + buffer2.toString());//he
```

内容导航

Contents



Section 00

【学习目标】

（1）理解流的概念；
（2）掌握流的使用方法。

Section 01

【理解流的概念】

- 流可以看作是某段时间内从一个点移动到另一个点的数据序列。
- Node.js中的流用于管理和处理数据，使用流完成对大量数据的操作以及逐段处理的操作。
- 流是Node.js中处理流式数据的抽象接口。
- stream模块用于构建实现了流接口的对象

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

- stream模块主要用于开发人员创建新类型的流实例。几乎所有的Node.js应用程序都在某种程度上使用了流。应用程序只需写入数据到流，或者从流消费数据，并不需要直接实现流的接口，通常也不需要直接加载stream模块。

Section 01

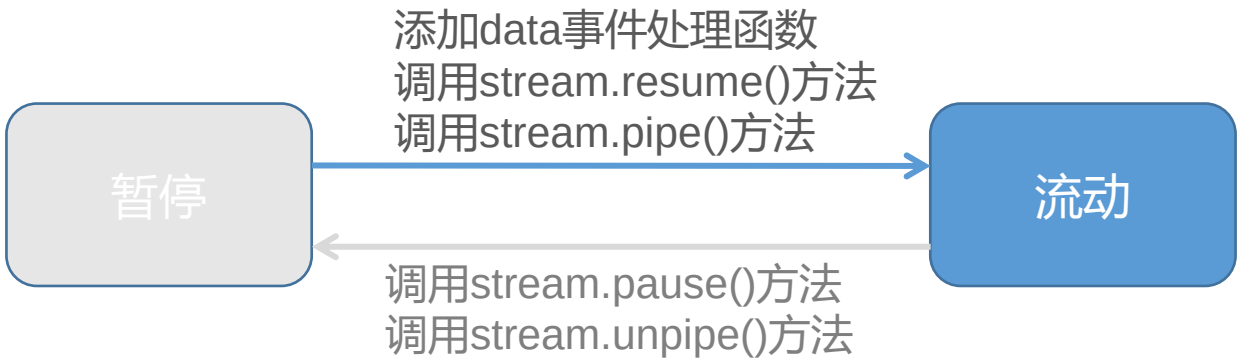
【理解流的概念】

- 流的基本类型：
  - 可写流（Writable）
  - 可读流（Readable）
  - 双工流（Duplex）
  - 转换流（Transform）
- 流的实现类型
  - 普通的流：基于字符串和Buffer（或Uint8Array）工作
  - 对象模式的流：使用其他类型的JavaScript值工作
- 可写流和可读流都会在内部的缓冲器中存储数据，可缓冲的数据大小取决于传入流构造函数的highWaterMark选项（相当于水位线）。
- 双工流和转换流都是可读又可写的，各自维护着两个相互独立的内部缓冲器用于读取和写入。在维护数据流时，读取和写入两端可以各自独立地工作。

Section 02

【可读流】

- 可读流是对提供数据的来源的一种抽象。所有可读流都实现了stream.Readable类定义的接口。
- 可读流有两种模式：流动（Flowing）和暂停（Paused）。



- stream.Readable类定义的主要事件
  - data：当有数据可读时被触发。
  - end：没有更多的数据可读时被触发。
  - close：当流或其底层资源被关闭时被触发。
- stream.Readable类定义的主要方法
  - readable.read([size])：从内部缓冲区拉取并返回数据。
  - readable.pause()：使流动模式的流停止触发data事件，并切换出流动模式。
  - readable.setEncoding(encoding)：为从可读流读取的数据设置字符编码。

Section 02

【可读流】

- 可读流操作示例

```
const fs = require('fs')
//以流的方式读取文件
var readStream=fs.createReadStream('demo.txt');
var str='';//保存数据
readStream.on('data',function(chunk){
    str+=chunk;
})
//读取完成
readStream.on('end',function(chunk){
    console.log(str);
})
//读取失败
readStream.on('error',function(err){
    console.log(err);
})
```

Section 03

【可写流】

- 可写流是对数据被写入的目的地的一种抽象。
- 所有可写流都实现了stream.Writable类定义的接口。
- stream.Writable类定义的主要事件
  - close：当流或其底层资源被关闭时被触发。
  - error：写入数据发生错误时被触发。
  - finish：调用stream.end()方法且缓冲数据都已传给底层系统之后被触发。
- stream.Writable类定义的主要方法
  - writable.write(chunk[, encoding][, callback])：写入数据到流，并在数据被完全处理之后调用回调函数。

示例：

```
const myStream = getWritableStreamSomehow();
myStream.write( '一些数据' );
myStream.write( '更多数据' );
myStream.end( '完成写入数据' );
```

Section 03

【可写流】

- 可写流操作示例

```
const fs = require('fs')
var str = '这首歌真的很好听呢';
// 创建一个可以写入的流，写入到文件output.txt 中
var writerStream = fs.createWriteStream('output.txt');
// 使用 utf8 编码写入数据
writerStream.write(str,'UTF8');
// 标记文件末尾
writerStream.end();
// 处理流事件
writerStream.on('finish', function() {
    console.log('写入完成!');
});
writerStream.on('error', function(err){
    console.log('写入失败');
});
```

Section 04

【管道读写操作】

- 管道操作将从一个流中获取的数据传递到另外一个流中。
- 可读流提供的readable.pipe()方法在可读流与可写流之间架起桥梁

```
readable.pipe(destination[, options])
```

- 将可读流的所有数据通过管道推送到文件的示例

```
const readable = getReadableStreamSomehow();//创建一个可读流
const writable = fs.createWriteStream('file.txt'); // 创建一个可写流
readable.pipe(writable); //管道读写操作，将readable 的所有数据都推送到文件file.txt
```

- 在单个可读流上绑定多个可写流，对流进行链式管道操作的示例

```
const fs = require('fs' );
const zlib = require( 'zlib' );
const r = fs.createReadStream('file.txt');
const z = zlib.createGzip();
const w = fs.createWriteStream('file.txt.gz');
r.pipe(z).pipe(w);// 链式管道操作两个可写流
```

Section 04

【管道读写操作】

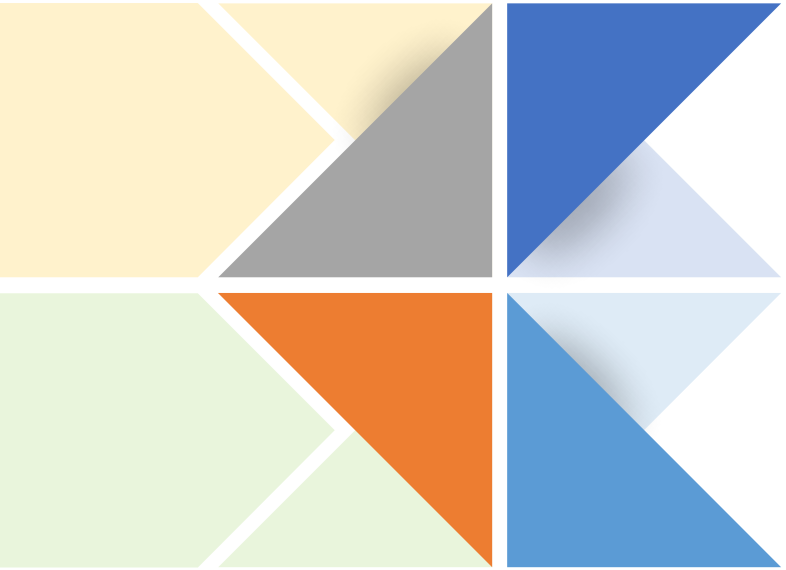
- 默认当来源可读流触发end事件时，目标可写流也会调用stream.end()结束写入。
- 禁用默认行为，end选项设为false，目标流就会保持打开状态

```
reader.pipe(writer, { end: false });
reader.on('end', () => {
  writer.end('结束');
});
```

- 可读流发生错误，目标可写流不会自动关闭，需要手动关闭所有流以避免内存泄漏。
- readable.unpipe()方法用于解绑之前使用stream.pipe()方法绑定的可写流。

内容导航

Contents



- 综合运用本章所学知识
- 搭建图片浏览的静态文件服务器

Section 00 【学习目标】

（1）尝试开发一个简单的应用程序；
（2）实现一个静态文件服务器来提供图片浏览服务。

Section 01 【实现思路】

▸▸▸ 搭建B/S架构

➢ 应用程序基于B/S架构，由Node.js内置的核心模块http提供Web服务。

```
const server = http.createServer(function(req, res){
    //请求处理并返回结果
});
server.listen(8000);
```

Section 01

【实现思路】

▸▸▸ 解析资源文件路径

- 获取请求的URL并返回文件路径名

```
var pathName = url.parse(req.url).pathname;
```

- 从文件路径名取得资源文件的绝对路径需要用到全局变量__dirname

```
var filePath = path.resolve(__dirname + pathName);
```

- 可以通过path.extname属性来获取文件的扩展名来确定文件类型

▸▸▸ 处理不同类型的资源文件

- Content-Type的格式：

```
Content-Type：type/subtype ;parameter
```

- 由一个键值对集合来指定不同扩展名与Content-Type类型之间的对应关系

```
var mime = {
        ".jpeg": "image/jpeg",
        ".jpg": "image/jpeg",
}
```

Section 01

【实现思路】

▸▸▸ 文件读取

- 读取文件的状态

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

- 资源文件可以使用fs.createReadStream()方法打开。
- 静态文件服务器通常使用gzip压缩文件以提高传输效率，内置模块zlib提供gzip压缩功能。
- 可以在文件流发送到HTTP响应之前增加一个压缩文件的管道操作：

```
stream.pipe(zlib.createGzip()).pipe(res);
```

- 让浏览器知道已经开启gzip压缩，需要在HTTP消息头中提供相应的内容编码信息：

```
res.writeHead(200, { "content-encoding": 'gzip' });
```

Section 02

【编写代码】

- 第1部分

```
const http = require('http');          //加载http模块
......
var curDir = '';                       //当前目录名
//创建HTTP服务
const server = http.createServer(function(req, res){
    //定义mime对象设置相应的响应头类型，这里仅列出少量的扩展名用于测试
    var mime = {
        ".jpeg": "image/jpeg",
        ".jpg": "image/jpeg",
        ".png": "image/png",
        ".tiff": "image/tiff",
        ".pdf": "application/pdf"
    };
    //获取请求URL并转换请求路径
    var pathName = url.parse(req.url).pathname;
```

Section 02

【编写代码】

- 第2部分

```
//获取请求URL并转换请求路径
var pathName = url.parse(req.url).pathname;
//对路径进行解码以防中文乱码
var pathName = decodeURI(pathName);
//获取资源文件的绝对路径，这里用到全局变量__dirname
var filePath = path.resolve(__dirname + pathName);
console.log(filePath);//控制台显示绝对路径
//获取文件的扩展名
var extName = path.extname(pathName);
//为简化处理，没有扩展名的或未知类型使用text/plain表示
var contentType = mime[extName] || "text/plain";
```

Section 02

【编写代码】

- 第3部分

```
//通过读取文件状态来决定如何读取静态文件
  fs.stat(filePath, function(err, stats){
  ......
     //如果路径是目录
     if (!err && stats.isDirectory()) {
        var html = "<head><meta charset = 'utf-8'/></head><body><ul>";
        curDir= path.basename(path.relative(__dirname,filePath));//获取当前目录
          ......
             for (var file of files) {
                //这里用到了ES6模板字符串
                var curPath = path.join(curDir,file);
                html += `<li><a href='${filecurPath}'>${file}</a></li>`;
             }
             html += '</ul></body>';
             res.writeHead(200, {'content-type': "text/html"});
             res.end(html);
          }
     });
  }
```

Section 02 【编写代码】

- 第4部分

```
//声明函数流式读取文件
    function readFile(filePath, contentType){
        //设置HTTP消息头
        res.writeHead(200, {'content-type': contentType,'content-encoding':'gzip'});
        //创建流对象读取文件
        var stream = fs.createReadStream(filePath);
        //流式读取错误处理
        stream.on('error', function() {
            res.writeHead(500, { 'content-type': contentType });
            res.end("<h1>500 服务器错误</h1>");
        });
        //链式管道操作将文件内容流到客户端
        stream.pipe(zlib.createGzip()).pipe(res);
    }
});
});
```

Section 02

【编写代码】

- 第5部分

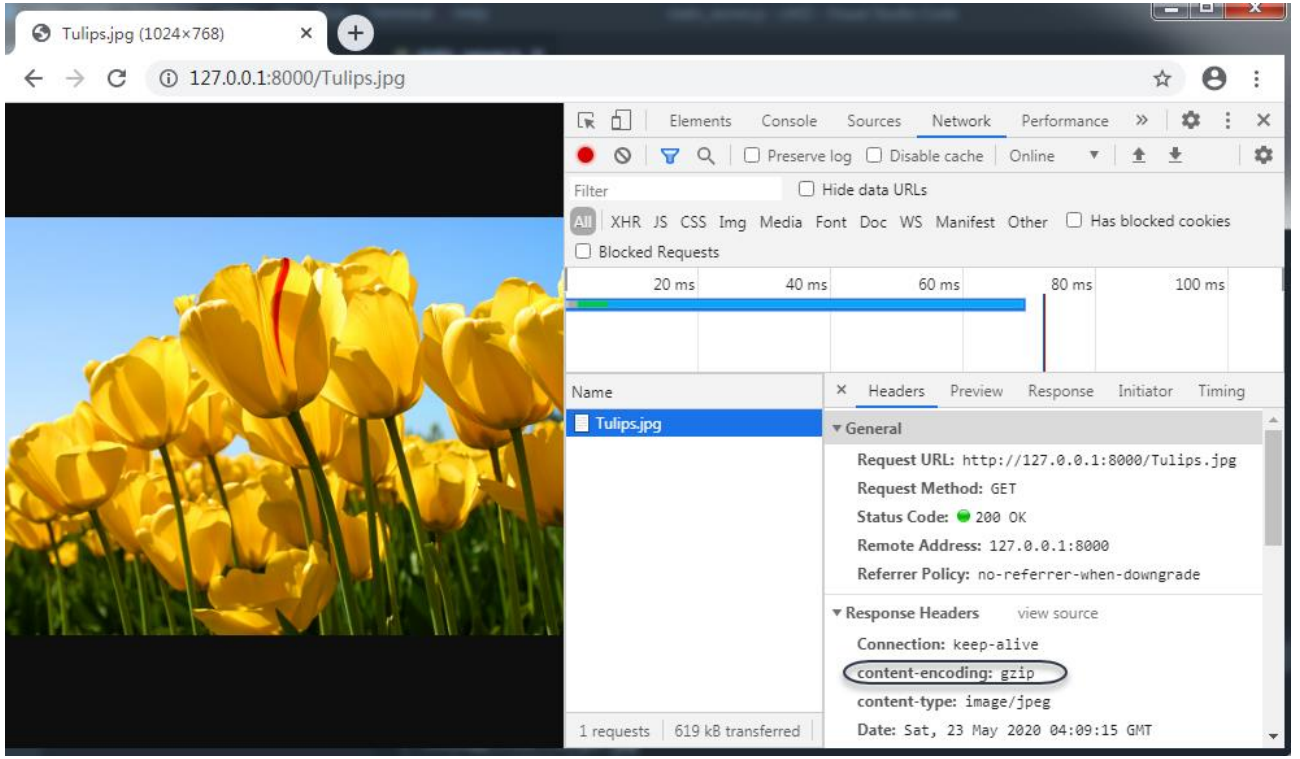
```
var port = 8000; //指定服务器监听的接口
server.listen(port, function() {
    console.log(`图片服务器正运行在端口:${port}`);
    console.log(`访问网址: http://localhost:${port}`);
});
```

Section 03

【运行程序】

```
C:\nodeapp\ch02>node static_server
图片服务器正运行在端口:8000
访问网址: http://localhost:8000
C:\nodeapp\ch02
C:\nodeapp\ch02\favicon.ico
C:\nodeapp\ch02\Tulips.jpg
```

验证gzip编码



本章小结

Summary

本章的主要内容是Node.js编程的基础知识，包括JavaScript语法、Node.js的回调函数、事件机制、全局对象和全局变量、定时器、Buffer类、流，最后是一个使用原生Node.js实现静态服务器的案例。

Thank you