第1讲 ES6 语法

|  |  |  |  |
| --- | --- | --- | --- |
| 知识模块 | 课程内容 | 重点 | 课程目标 |
| ES6 | ECMAScript简介、ES5回顾 | ☆ | 掌握：ES5的特性与ES6的区别  掌握：let和const关键字  掌握：对象解构赋值，函数形参解构赋值  掌握：模板字符串用法  掌握：对象的属性名简写和属性名表达式  掌握：函数参数默认值、箭头函数this用法 |
| ES6简介 | ☆ |
| let和const | ★ |
| 解构赋值 | ★ |
| 模板字符串 | ☆ |
| 数组、对象扩展 | ★ |
| 函数扩展 | ★ |

1.1 本讲重点

● 了解ECMA的版本，ECMA和JavaScript的关系

● 掌握let和const的用法，什么是块作用域，什么是函数作用域，let暂时性死区，不存在变量声明提升，同一块作用域内不能重复声明

● 掌握解构赋值中“模式”的概念，掌握解构赋值默认值，对象解构赋值，函数形参解构赋值

● 掌握模板字符串的用法，了解标签模板的作用和写法

● 掌握扩展、剩余参数运算符，掌握对象的属性名简写和属性名表达式写法

● 掌握函数形参的默认值，箭头函数和普通函数的区别，箭头函数的this用法

1.2 ECMAScript简介

ECMAScript 6.0（以下简称 ES6）是 JavaScript 语言的下一代标准，已经在 2015年 6 月正式发布了。它的目标，是使得 JavaScript 语言可以用来编写复杂的大型应用程序，成为企业级开发语言。

一个常见的问题是，ECMAScript 和 JavaScript 到底是什么关系？

要讲清楚这个问题，需要回顾历史。1996 年 11 月，JavaScript 的创造者Netscape 公司，决定将 JavaScript 提交给国际标准化组织 ECMA，希望这种语言能够成为国际标准。次年，ECMA 发布 262 号标准文件（ECMA-262）的第一版，规定了浏览器脚本语言的标准，并将这种语言称为 ECMAScript，这个版本就是 1.0 版。

该标准从一开始就是针对 JavaScript 语言制定的，但是之所以不叫 JavaScript，有两个原因。一是商标，Java是Sun公司的商标，根据授权协议，只有Netscape公司可以合法地使用 JavaScript 这个名字，且 JavaScript 本身也已经被 Netscape 公司注册为商标。二是想体现这门语言的制定者是 ECMA，不是 Netscape，这样有利于保证这门语言的开放性和中立性。

因此，ECMAScript 和 JavaScript 的关系是，前者是后者的规格，后者是前者的一种实现（另外的ECMAScript方言还有Jscript和ActionScript）。日常场合，这两个词是可以互换的。

1.3 ES5回顾

1.3.1 严格模式

JavaScript 严格模式（strict mode）即在严格的条件下运行。

设立严格模式的原因：

● 消除JavaScript语法的一些不合理、不严谨之处，减少一些怪异行为;

● 消除代码运行的一些不安全之处，保证代码运行的安全；

● 提高编译器效率，增加运行速度；

● 为未来新版本的JavaScript做好铺垫。

"严格模式"体现了JavaScript更合理、更安全、更严谨的发展方向，IE 10在内的主流浏览器。"use strict";//是进入严格模式的标志(老版本的浏览器会把它当作一行普通字符串，加以忽略。)将"use strict"放在脚本文件的第一行，则整个脚本都将以"严格模式"运行。如果这行语句不在第一行，则无效，整个脚本以"正常模式"运行。

"use strict"

使用需要记住的几个变化：

● 声明定义变量必须用var

'use strict';

age = 12;

console.log(age);//报错

● 禁止自定义的函数中的this关键字指向全局对象

'use strict';

function Person(name, age) {

this.name = name;

this.age = age;

}

Person('ujiuye', 39);//报错

● 创建eval作用域, 更安全

'use strict';

var name = 'ujiuye';

eval('var name = "ujiuyeweb";alert(name)');//ujiuyeweb

console.log(name);//ujiuye

1.3.2 JSON对象

JSON对象有两个方法，并且JSON 通常用于与服务端交换数据。分别为JSON.parse()和JSON.stringify()

在接收服务器数据时一般是字符串。我们可以使用 JSON.parse() 方法将数据转换为 JavaScript 对象。json.stringfy()将对象、数组转换成字符串。

示例：

var obj = {

name : 'ujiuye',

age : 18

};

obj = JSON.stringify(obj);

console.log( typeof obj);//string

obj = JSON.parse(obj);

console.log(obj);//{name: "ujiuye", age: 18}

1.3.3 Object扩展

ES5给Object扩展了好一些静态方法, 常用的2个:

● Object.create(prototype, [descriptors])

作用: 以指定对象为原型创建新的对象，为新的对象指定新的属性, 并对属性进行描述。

value : 指定值

writable : 标识当前属性值是否是可修改的, 默认为false

configurable: 标识当前属性是否可以被删除 默认为false

enumerable： 标识当前属性是否能用for in 枚举 默认为false

var person1 = { name:'ujiuye',age:18 }

var person2 = Object.create( person1,{

sex:{

value:'男',

writable:true,//可修改的

// configurable:true//可以删除此属性

enumerable:true//是否可枚举，是否可遍历（循环、迭代）

},

sex2:{

value:'男',

writable:true,//可修改的

// configurable:true//可以删除此属性

enumerable:true//是否可枚举，是否可遍历（循环、迭代）

}

} )//新增的属性不会加到原型上

console.log(person2.sex);//男

person2.sex = '女'

console.log(person2.sex);//女

delete person2.sex

console.log(person2,'delete');//已删除sex

for( var attr in person2 ){

console.log(attr,person2[attr]);//遍历出sex2属性

}

● Object.defineProperties(object, descriptors)

作用: 为指定对象定义扩展多个属性

get ：用来获取当前属性值得回调函数

set ：修改当前属性值得触发的回调函数，并且实参即为修改后的值

var person = {

firstname:'Marry',

lastname:'LuCi'

}

Object.defineProperties( person,{

fullName:{

get: function () {

return this.firstname +' '+ this.lastname

},

set: function ( value ) {

//当我们在外部设置此属性的时候，会调用一次set方法

var names = value.split(' ')

this.firstname = names[0]

this.lastname = names[1]

}

}

} )

person.fullName = 'L Jams' //

console.log( person )//{firstname: "L", lastname: "Jams"}

1.3.4 Array扩展

数组的扩展主要体现在一些原型方法上一定要合理的使用，在后续的框架当中也会经常使用。

● Array.prototype.forEach(function(item, index){}) : 遍历数组

var arr = [11,22,33]

arr.forEach(function (item, index) {

console.log(item, index);

});//依次打印 11 0 | 22 1 | 33 2

● Array.prototype.map(function(item, index){}) : 遍历数组返回一个新的数组，返回加工之后的值

var arr = [11,22,33]

var arr1 = arr.map(function (item, index) {

return item + 10

});

console.log(arr, arr1);//[11, 22, 33] [21, 32, 43]

● Array.prototype.filter(function(item, index){}) : 遍历过滤出一个新的子数组， 返回条件为true的值

var arr = [1,3,-5,10,6]

var arr2 = arr.filter(function (item, index) {

return item > 4

});

console.log(arr, arr2);//[1, 3, -5, 10, 6]  [10, 6]

1.3.5 Function扩展

● Function.prototype.bind(obj) :

作用: 将函数内的this绑定为obj, 并将函数返回

function fun(age) {

this.name = '优就业';

this.age = age;

}

var obj = {};

fun.bind(obj, 18)();

console.log(obj.name, obj.age);//优就业 18

1.4 ES6简介

1.4.1 let

ES6新增的let用来定义变量，在ES5中使用var定义变量；其中，let相比var的区别之一就是在var的函数作用域基础上，新增了块作用域。

块级作用域

了解块作用域必须要知道什么是块？其实就是语句块，它的表现形式为{},大括号包裹的内容为语句块，例如：

var age = 20

if(age > 20){

console.log(‘成年人’);

}//此时的大括号就是语句块，它的意义是多条语句能被if判断影响到

for(let i=0;i<10;i++){

//TODO

}//此时的大括号也是语句块，它的意义是多条语句能被for循环影响到

{var name = ‘zhangsan’;}//此时的大括号也是语句块，只是它在此时的意义不大，因为var的作用域并不是这个语句块，而是函数作用域

function fn(x){

var a = 10;

let b = 20;

}//此时的大括号也算语句块范畴，但它的意义是定义函数体，构成函数作用域，其中var声明的变量、包括fn的形参都是属于这个函数作用域的;let声明的参数b也属于这个作用域。

所以，块作用域相当于是对函数作用域的一种补充和细化，关于let的块作用域和var的函数作用域要区别清楚。var的函数作用域意味着，var声明的变量在函数内部是局部变量，在函数外部为全局变量；函数外部情况：不管是if语句块还是for语句块中使用var声明变量都属于全局变量。

基本用法：

ES6 新增了let命令，用来声明变量。它的用法类似于var。

let a = 10;

var b = 10;

console.log(a,b);//10 10

上面代码中，分别用let、var声明了变量，然后调用这两个变量，都能得到正确的值，但是两者之间还有区别的，如下例所示：

{

let c = 10;

var d = 10;

}

console.log(d); //10

console.log(c); //c is not defined

上面代码中分别用let、var声明了变量，在代码块外面调用，结果let声明的变量报错，var声明的变量返回了正确的值。这表明，let声明的变量只在它所在的代码块有效。

注意：变量名命名规则

● 见名知义

● 不能使用关键字与保留字

● 遵循驼峰标识

● 变量名不能重（会报错）

let和var的区别：

1、不存在变量提升

var命令会发生”变量提升“现象，即变量可以在声明之前使用，值为undefined。这种现象多多少少是有些奇怪的，按照一般的逻辑，变量应该在声明语句之后才可以使用。为了纠正这种现象，let命令改变了语法行为，它所声明的变量一定要在声明后使用，否则报错。

//var

console.log(a); //undefined

var a = 10;

//let

console.log(b); // b is not defined

let b = 10;

上面代码中，变量a用var命令声明，会发生变量提升，即脚本开始运行时，变量a已经存在了，但是没有值，所以会输出undefined。变量b用let命令声明，不会发生变量提升。这表示在声明它之前，变量b是不存在的，这时如果用到它，就会抛出一个错误。

2、暂时性死区(TDZ)

只要块级作用域内存在let命令，它所声明的变量就“绑定”（binding）这个区域，不再受外部的影响。

ES6 明确规定，如果区块中存在let和const命令，这个区块对这些命令声明的变量，从一开始就形成了封闭作用域。凡是在声明之前就使用这些变量，就会报错。

if (true) {

tmp = 'abc'; // ReferenceError

console.log(tmp); // ReferenceError

let tmp; // TDZ结束

console.log(tmp); // undefined

tmp = 123;

console.log(tmp); // 123

}

typeof x; // ReferenceError

let x;

function bar(x = y, y = 2) {

return [x, y];

}

bar(); // 报错,x=y中的y没有声明

所以，暂时性死区的问题矛头都指向了一个原则：先声明，后使用；不存在变量声明提升。

3、不能重复声明

基于暂时性死区的问题，在同一块级作用域内不能重复声明同名变量，例如：

{

var a = 10;

let a = 10/报错

}

{

let name = 20;

...

let name = 20;//报错

}

function (x){

let x = 10;//报错，形参和x和let声明的x属于同一作用域

}

块作用域嵌套

块级作用域允许任意嵌套，外层作用域无法读取内层作用域的变量，而且内外层作用域可以声明同名变量。

{

let a = 'admin'

console.log(a);//admin

{

console.log(a);//报错，这个问题属于暂时性区+不存在变量声明提升

let a = 'ujiuye'

console.log(a);//ujiuye

}

//此时访问内容为ujiuye的变量a会失败,因为此时它的块作用域已结束

}

console.log(a)//a is not defined，此时两个变量a的块作用域都已经结束

循环中使用let

循环操作非常适合let命令，

for(let i = 0; i < 10;i++){

console.log(i);

}

console.log(i); // i is not defined

上面代码中，计数器i只在for循环体内有效，在循环体外引用就会报错。

var arr = arr1 = [];

for(var i = 0;i < 10;i++){

arr[i] = function () {

console.log(i);

}

}

arr[6](); //10

for(let j = 0;j < 10;j++){

arr1[j] = function () {

console.log(j);

}

}

arr1[6]();//6

上面代码中，用var声明的变量i，在全局范围内都有效，所以全局只有一个变量i。每一次循环，变量i的值都会发生改变，而循环内被赋给数组arr的函数内部的console.log(i)，里面的i指向的就是全局的i。也就是说，所有数组arr的成员里面的i，指向的都是同一个i，导致运行时输出的是最后一轮的i的值，也就是 10。

用let声明的变量j，当前的j只在本轮循环有效，所以每一次循环的j其实都是一个新的变量，所以最后输出的是6。你可能会问，如果每一轮循环的变量j都是重新声明的，那它怎么知道上一轮循环的值，从而计算出本轮循环的值？这是因为 JavaScript 引擎内部会记住上一轮循环的值，初始化本轮的变量j时，就在上一轮循环的基础上进行计算。

另外，for循环还有一个特别之处，就是设置循环变量的那部分是一个父作用域，而循环体内部是一个单独的子作用域。

for(let k = 0; k <3;k++){

let k = "ujiuye";

console.log(k);//输出3次ujiuye

}

上面代码正确运行，输出了 3 次ujiuye。这表明函数内部的变量k与循环变量k不在同一个作用域，有各自单独的作用域。

let在循环中定义的拆解明示：

for(let j = 0;j < 10;j++){

arr1[j] = function () {

console.log(j);

}

}

拆解为每一次的单独定义如下：

{

let j = 0;

{

a[j] = function(){console.log(j)}

}

}

{

let j = 1;

{

a[j] = function(){console.log(j)}

}

}

...

**小结：**掌握let和var的区别，什么是块作用域，什么是函数作用域，let和var区别：let暂时性死区；不存在变量声明提升；同一块作用域内不能重复声明。

1.4.2 const

const声明的常量，后期不能修改常量的值，并且在声明的同时要立刻初始化，不能后面赋值修改，其声明常量的作用域与let相同，都属于块作用域。let中存在的暂时性死区，不存在变量声明提升和同一块作用域内不能重复声明，const也同时存在。

const使用形式如下：

//1.常量 const声明的常量，不能修改该常量的值

const PI = 3.1414;

console.log(PI);

PI = 3; //报错，Assignment to constant variable

console.log(PI);

//2.用const声明的常量，需要立即初始化，不能留到后面赋值

const a; //Missing initializer in const declaration

//3.作用域与let相同，在代码块内有效

if(true){

const block = 10;

}

console.log(block); //block is not defined

//4.const声明的常量也不存在常量提升

console.log(up); // up is not defined

const up = 10;

使用const定义常量注意两点：1、声明必须赋值，否则报错；2、定义后不可再赋值修改，否则报错。

1.4.3 模板字符串

目前的形式来看，前后端分离设计的理念已经很明确了；一般来说我们的web前端ajax请求数据，渲染数据的操作形式有很多，可以是字符串拼接的形式，如下：

传统JavaScript输出模板通常是：

var n = 10;

$("p").append("there are <b>"+n+"</b>"+

"ujiuye IT"+

"<em>"+n+"</em>");

这种写法相当繁琐不方便，如果字符串特别长的时候也许还会弄错+和“”的位置；而且这样写的代码后期维护会很难。

es6引入模板字符串的概念后，我们可以使用模板字符串解决类似的问题；模板字符串是增强版字符串，用反引号`表示，里面的变量用${}包起来就可以实现。

var n = 10;

$("div").append(`there are <b>${n}</b>`);

它可以当作普通字符串使用，也可以用来定义多行字符串。

var str = `<div>

<h1>标题</h1>

</div>`;

console.log(str);

console.log(`${n}`);

${}中嵌入的内容为表达式，表达式的定义如下：由变量、常量或变量、常量加运算符或函数调用组成的运算单元。所以下面形式都合法：

var a = 1;

const PI = 3.14;

function fn() {

return "hello";

}

console.log(`${a}`);

console.log(`${a + 1}`);

console.log(`${PI}`);

console.log(`${a + PI}`);

console.log(`${123}`);

console.log(`${fn()}`);

1.4.4 对象扩展

对象属性名简写

ES6 允许在对象中直接写入变量和函数，作为对象的属性和方法。这样的书写更加简洁。

let foo = 'bar';

let baz = {foo};//此时的{foo}为简写形式，key和value都是foo，key为字符串foo，value为bar

console.log(baz) // {foo: "bar"}

// 等同于

const baz = {'foo': foo};

除了属性简写，方法也可以简写。

const o = {

fn() {

return "Hello!";

}

};

// 等同于

const o = {

fn: function() {

return "Hello!";

}

};

对象属性名表达式

目前来说我们定义对象的属性形式有：

// 方法一obj.foo = true;

// 方法二obj['a' + 'bc'] = 123;

不论是方法一还是方法二，对象的属性名都是以字符串的形式存在的。ES6支持对象属性名为表达式(注意回忆前面内容中对表达式的定义)的做法，如下：

let lastWord = 'last word';

const a = {

'first word': 'hello',

[lastWord]: 'world' //lastWord是一个变量

};

a['first word'] // "hello"

a[lastWord] // "world"

a['last word'] // "world"

另外，对象属性名表达式的做法还可以定义方法名，如下：

let obj = {

['h' + 'ello']() {

return 'hi';

}};

obj.hello() // hi

这非常有用，举个例子，我们想组织一个这样的对象：{A:[],B:[],C:[],D:[]...}，做法如下：

var obj = {};

for(let i=0;i<26;i++){

let char = String.fromCharCode(65+i);

obj[char] = [];

}

console.log(obj) //{A:[],B:[],C:[],D:[]...}

1.4.5 函数扩展

es6允许用箭头（=>）声明函数，更加快捷方便。()=>{}等价于function (){}。

箭头函数与普通函数的区别

●没有参数时

语法：变量名 = ()=>{函数体}

//1.没有参数的情况 ()=>{}

var f =()=>{

console.log("10"); //"10"

}

f();

●有参数时

语法：变量名=(参数1，参数2)=>{函数体}，如果只有一个参数，可以将（）省略，两个或两个以上参数不能省略(),每个参数之间用’,’逗号隔开。

//2.一个参数的情况

var f = n =>{

var sum = 0;

for(let i = 0;i <= n;i++){

sum += i;

}

console.log(sum); //5050

}

f(100);

//3.多个参数的情况，不能省略(),多个参数之间用,隔开

var f4 = (n,m)=>{

console.log(n\*m);

}

f4(10,20);

●有返回值时

语法：变量名 = (参数)=>{return 返回值}

//普通函数返回值

function fun5(n,m) {

return n\*m;

}

console.log(fun5(2,3)); //6

//箭头函数返回值

var fu = (n,m)=>{

return n\*m;

}

console.log(fu(10,20)); //200

如果函数体中就一条语句，可以省略大括号｛｝，如果这一条语句是返回值，还可以省略return关键字。

//如果这一条语句是返回值，则可以省略return关键字

var f1 =(n,m)=> n\*m; //省略return

console.log(f1(10,20)); //200

箭头函数中的this

●在全局函数的中的this

//1.在window中定义，this指向window

function fun(){

console.log(this); //window

}

fun();

var fun1 = n =>{

console.log(this); //window

}

fun1();

Window中声明的函数,es5和es6中的this都指向window。

●在事件处理函数中的this

var oDiv = document.getElementById("box");

oDiv.onclick = function () {

console.log(this); //div

}

oDiv.onclick = ()=>{

console.log(this); //window

}

如果是作为事件处理函数，es5中的this指向的是当前触发事件的对象，es6中的this指向的是window。

●在对象方法中的this

//在构造函数中给对象添加了eat方法，调用这个方法会返回一个函数

function Person() {

this.eat = function () {

return function () {

console.log(this);

}

}

}

var p = new Person();

p.eat()(); //函数返回到外面，全局调用，this-->window

//箭头函数里面的this，在那定义就指向谁

function Dog() {

this.eat = function () {

return ()=>{

console.log(this);

}

}

}

var d = new Dog();

d.eat()(); //在Dog构造函数里面调用，this--->Dog

函数声明在对象方法中，es5中因为函数传出到全局所以指向的是window，es6中函数在那定义的就指向谁，所以this指向Dog。

●this对象

函数体内的this对象，就是定义时所在的对象，而不是使用时所在的对象。

function Dog() {

this.eat = function () {

return ()=>{

console.log(this);

}

}

}

var d = new Dog();

d.eat()(); //在Dog构造函数里面调用，this--->Dog

●不能作为构造函数

不可以当作构造函数，也就是说，不可以使用new命令，否则会抛出一个错误。

var Person = (name)=>{

this.name = name;

}

var p1 =new Person("ujiuye"); //Person is not a constructor 不是构造函数

console.log(p1.name);

●不可以使用arguments对象

不可以使用arguments对象，该对象在函数体内不存在。如果要用，可以用Rest参数代替。

var arg = (a,b,c,d)=>{

console.log(arguments); //arguments is not defined

}

arg(1,2,3,4);

var arg1 = (...num)=>{

console.log(num); //[1, 2, 3, 4]

}

arg1(1,2,3,4);

**小结：**箭头函中不好理解的内容为this的指向问题，箭头函数内的this指向的是定义时所在的对象而不是使用对象，箭头函数的this也不绑定当前调用者；这两句内容不太好理解，看下面的例子，这个例子可以有四种变化，如下：

第一类：

var title = "window";

    var obj = {

      title: "对象",

      fn: function() {

        // 情况一

        // setTimeout(() => {

         //   console.log(this.title); //对象

        // }, 50);

        // 情况二

        setTimeout(function() {

          console.log(this.title); //window

        }, 50);

      }

};

obj.fn()

第二类：

var title = "window";

    var obj = {

      title: "对象",

      fn: () => {

        // 情况三

        // setTimeout(() => {

        //   console.log(this.title);//window

        // }, 50);

        // 情况四

        setTimeout(function() {

          console.log(this.title); //window

        }, 50);

      }

    };

obj.fn()

两类问题，四种情况；说明箭头的内部的this与上下文环境有关，这个上下文的理解就是箭头函数的代码写在谁的作用域里面，比如，情况一的箭头函数是写在fn函数里面的，此时箭头函数的上下文就是fn，fn函数不是箭头函数，属于obj对象的一个属性，也就是说fn函数里面的this指向是obj，所以箭头函数的的this也指向obj对象；情况三的箭头函数是嵌套使用的，fn本身是箭头函数，fn的上下文为obj对象，obj对象是定义在window作用域里面的，所以此时fn里面的this指向应该是window。

箭头函数常见有两种不适用场景：不适合在对象内部的函数上使用，不适合在dom的事件回调函数上使用。

箭头函数和普通函数的区别以及特性如下：

1、箭头函数是匿名函数，不绑定自己的this,arguments,super,new.target

2、箭头函数会捕获其所在上下文的this值，作为自己的this值，在使用call/apply绑定时，相当于只是传入了参数，对this没有影响

3、箭头函数不绑定arguments，取而代之用rest参数…解决

4、箭头函数当方法使用的时候，没有定义this绑定

5、箭头函数不能作为构造函数，和 new 一起用就会抛出错误

6、箭头函数没有原型属性

7、不能简单返回对象字面量

1.4.6 剩余参数运算符

ES6 引入 rest 参数（形式为...变量名），用于获取函数的多余参数，这样就不需要使用arguments对象了（箭头函数内不能使用arguments）。rest 变量是一个数组。如下：

function add(...values) {

let sum = 0;

for (var val of values) {

sum += val;

}

return sum;}

add(2, 5, 3) // 10

箭头函数内部无法使用arguments来获取实参，可以使用剩余参数运算符接受不固定参数数量的参数。

**注意：**rest 参数之后不能再有其他参数（即只能是最后一个参数），否则会报错。如下：

// 报错function f(a, ...b, c) {

// ...}

所以使用形参列表中的剩余参数需要指定为形参列表中的最后一个参数。

1.4.7 扩展运算符

扩展运算符（spread）是三个点（...）。它好比 rest 参数的逆运算，将一个数组转为用逗号分隔的参数序列。

//将一个数组转为用逗号分隔的参数序列。

console.log(...[1, 2, 3]); //1 2 3

console.log(1, ...[2, 3, 4], 5); //1 2 3 4 5

//一般用于函数调用

function add(x, y) {

return x + y;

}

const numbers = [2,3];

console.log(add(...numbers)) // 5

应用

可以复制数组，将类似数组转换为真正的数组

//1.复制数组

const a1 = [1, 2];

const a2 = [...a1];

//2.将类似数组转为真正的数组

console.log([...'hello']); //["h", "e", "l", "l", "o"]

console.log([..."𠮷"].length);//1能够正确识别四个字节的 Unicode 字符

var oDiv = document.getElementsByTagName("div");

console.log([...oDiv]); //[div, div, div]

1.4.8 函数参数默认值

ES6 之前，不能直接为函数的参数指定默认值，只能采用变通的方法。

function log(x, y) {

y = y || 'World';

console.log(x, y);

}

log('Hello') // Hello World

log('Hello', 'China') // Hello China

log('Hello', '') // Hello World

上面代码检查函数log的参数y有没有赋值，如果没有，则指定默认值为World。这种写法的缺点在于，如果参数y赋值了，但是对应的布尔值为false，则该赋值不起作用。就像上面代码的最后一行，参数y等于空字符，结果被改为默认值。

ES6 允许为函数的参数设置默认值，即直接写在参数定义的后面。

function log(x, y = 'World') {

console.log(x, y);

}

log('Hello') // Hello World

log('Hello', 'China') // Hello China

log('Hello', '') // Hello

除了简洁，ES6 的写法还有两个好处：首先，阅读代码的人，可以立刻意识到哪些参数是可以省略的，不用查看函数体或文档；其次，有利于将来的代码优化，即使未来的版本在对外接口中，彻底拿掉这个参数，也不会导致以前的代码无法运行。

参数变量是默认声明的，所以不能用let或const再次声明。

function foo(x = 5) {

let x = 1; // error，因为形参的x和当前这个x同属于一个块作用域，这违反了变量不能重复声明的原则

const x = 2; // error

}

1.5 高频面试

● call() 和 apply()的区别和作用？

1、apply()函数有两个参数：第一个参数是上下文，第二个参数是参数组成的数组。如果上下文是null，则使用全局对象代替。

如：function.apply(this,[1,2,3]);

2、call()的第一个参数是上下文，后续是实例传入的参数序列。

如：function.call(this,1,2,3);

● 箭头函数有哪些特性

箭头函数会捕获其所在上下文的this值，作为自己的this值，在使用call/apply绑定时，相当于只是传入了参数，对this没有影响

箭头函数是匿名函数，不绑定自己的this,arguments,super,new.target

箭头函数不绑定arguments，取而代之用rest参数…解决

箭头函数当方法使用的时候，没有定义this绑定

箭头函数不能作为构造函数，和 new 一起用就会抛出错误

箭头函数没有原型属性

不能简单返回对象字面量

● let的应用场景

注意块级作用域

暂时性死区

不存在变量提升

同一作用域内不允许重复

● 箭头函数中this的指向

箭头函数会捕获其所在上下文的this值，作为自己的this值，在使用call/apply绑定时，相当于只是传入了参数，对this没有影响（通俗的来说，this的指向要看其父级所在的作用域）

● 我们如何解决函数传参顺序不一致的问题

function fn({name,age,sex}){

console.log(`他叫${name},今年${age}岁了，是个${sex}孩`)

}

fn ({sex:"男",age:30,name:"里斯"})

第2讲 ES6 语法

|  |  |  |  |
| --- | --- | --- | --- |
| 知识模块 | 课程内容 | 重点 | 课程目标 |
| ES6 | Symbol类型 | ☆ | 了解：Symbol的作用和场景  了解：set和map的用法和场景  了解：iterator遍历器  了解：generator函数  掌握：Promise对象  掌握：async...await函数 |
| Set和Map | ☆ |
| iterator遍历器 | ☆ |
| generator函数 | ☆ |
| Promise | ★ |
| async...await | ★ |

2.1 本讲重点

● 了解Symbol类型的作用和写法

● 掌握Set集合过滤重复数组的做法

● 了解iterator和generator的用法和作用

● 掌握Promise的基本使用和Promise.all的使用，掌握async…await的用法

2.2 Symbol类型

ES5 的对象属性名都是字符串，这容易造成属性名的冲突。比如，你使用了一个他人提供的对象，但又想为这个对象添加新的方法（mixin 模式），新方法的名字就有可能与现有方法产生冲突。如果有一种机制，保证每个属性的名字都是独一无二的就好了，这样就从根本上防止属性名的冲突。这就是 ES6 引入Symbol的原因。

ES6 引入了一种新的原始数据类型Symbol，表示独一无二的值。它是 JavaScript 语言的第七种数据类型，前六种是：undefined、null、布尔值（Boolean）、字符串（String）、数值（Number）、对象（Object）。

Symbol 值通过Symbol函数生成。这就是说，对象的属性名现在可以有两种类型，一种是原来就有的字符串，另一种就是新增的 Symbol 类型。凡是属性名属于 Symbol 类型，就都是独一无二的，可以保证不会与其他属性名产生冲突。需要注意的是Symbol类型的值表现形式不是字符串或数字等形式，也不是加密后的内容，输出表现为symbol()。

2.2.1 基本用法

let s = Symbol();

typeof s

// "symbol"

上面代码中，变量s就是一个独一无二的值。typeof运算符的结果，表明变量s是 Symbol 数据类型，而不是字符串之类的其他类型，if判断两个symbol类型变量是不可能相等的。

注意，Symbol函数前不能使用new命令，否则会报错。这是因为生成的 Symbol 是一个原始类型的值，不是对象。也就是说，由于 Symbol 值不是对象，所以不能添加属性。

Symbol函数可以接受一个字符串作为参数，表示对 Symbol 实例的描述，主要是为了在控制台显示，或者转为字符串时，比较容易区分。这个描述(description)没有任何实际意义，它相当于是代码中的注释，起到一个描述和解释的作用。

let s1 = Symbol('foo');

let s2 = Symbol('bar');

s1 // Symbol(foo)

s2 // Symbol(bar)

s1.toString() // "Symbol(foo)"

s2.toString() // "Symbol(bar)"

注意，Symbol函数的参数只是表示对当前 Symbol 值的描述，因此相同参数的Symbol函数的返回值是不相等的。

// 没有参数的情况

let s1 = Symbol();

let s2 = Symbol();

s1 === s2 // false

// 有参数的情况

let s1 = Symbol('foo');

let s2 = Symbol('foo');

s1 === s2 // false

2.2.2 作为属性名的 Symbol

由于每一个 Symbol 值都是不相等的，这意味着 Symbol 值可以作为标识符，用于对象的属性名，就能保证不会出现同名的属性。这对于一个对象由多个模块构成的情况非常有用，能防止某一个键被不小心改写或覆盖。

let mySymbol = Symbol();

// 第一种写法

let a = {};

a[mySymbol] = 'Hello!';

// 第二种写法

let a = {

[mySymbol]: 'Hello!'

};

**注意：**for in, for of遍历时不会遍历symbol属性。

2.2.3 Symbol模拟私有方法

JavaScript内部无法实现面向对象中真正的私有数据，可以使用symbol来模拟私有，保证外界无法访问：

function Animal() {

 // innnerName是私有方法名的定义,使用了symbol产生一个独一无二的数据

var innerName = Symbol("inner方法名");

 //   通过对象属性名表达式给this设置方法，方法名为innerName的Symbol

this[innerName] = function() {

console.log("我是私有方法");

};

this.sayHi = function() {

this[innerName]();

console.log("hello");

};

}

let animal = new Animal();

animal.sayHi();

2.3 set和map数据结构

2.3.1 Set

ES6 提供了新的数据结构 Set。它类似于数组，但是成员的值都是唯一的，没有重复的值。Set本身是一个构造函数，用来生成 Set 数据结构。Set集合的主要作用是用来过滤数组重复数据。

语法：

let set = new Set(数组);

Set 实例提供的操作属性、方法（用于操作数据）

● size：类似数组length属性，size用于返回Set集合的内容长度。

● add(value)：添加某个值，返回 Set 结构本身。

● delete(value)：删除某个值，返回一个布尔值，表示删除是否成功。

● has(value)：返回一个布尔值，表示该值是否为Set的成员。

● clear()：清除所有成员，没有返回值。

● forEach/for...of：遍历Set集合使用forEach或for...of

示例如下：

let set = new Set([1,3,3,2,3,4,1,5]);

set.forEach(val=>{

console.log(val)

})

console.log(set);

set.add('abc');

console.log(set, set.size);

//delete(value)

set.delete(2);

console.log(set);

//has(value)

console.log(set.has(2));//false

console.log(set.has(1));//true

//clear()

set.clear();

console.log(set);

2.3.2 Map

JavaScript 的对象（Object），本质上是键值对的集合（Hash 结构），但是传统上只能用字符串(还可以是Symbol类型数据)当作键。这给它的使用带来了很大的限制。

const data = {};

const element = document.getElementById('myDiv');

data[element] = 'metadata';

data['[object HTMLDivElement]'] // "metadata"

上面代码原意是将一个 DOM 节点作为对象data的键，但是由于对象只接受字符串作为键名，所以element被自动转为字符串[object HTMLDivElement]。

为了解决这个问题，ES6 提供了 Map 数据结构。它类似于对象，也是键值对的集合，但是“键”的范围不限于字符串，各种类型的值（包括对象）都可以当作键。也就是说，Object 结构提供了“字符串—值”的对应，Map 结构提供了“值—值”的对应，是一种更完善的 Hash 结构实现。如果你需要“键值对”的数据结构，Map 比 Object 更合适。

作为构造函数，Map 也可以接受一个数组作为参数。该数组的成员是一个个表示键值对的数组。

Map 实例提供的操作属性、方法（用于操作数据）

● size：类似数组length属性，size用于返回Map集合的内容长度。

● set(key,value)：添加一个键值数据。

● get(key)：根据key获取value。

● has(key)：返回一个布尔值，表示map集合是否包含当前key的成员。

● delete(key)：根据指定的key删除数据。

● clear()：清除所有成员，没有返回值。

● forEach/for...of：遍历Set集合使用forEach或for...of

let map = new Map([['bbcc', 12],[25, 'age']]);

map.forEach(val=>{

console.log(val)

})

console.log(map);

map.set('男', '性别');

console.log(map);

console.log(map.get(25));//age

//delete(key)

map.delete('男');

console.log(map);

console.log(map.has('男'));//false

console.log(map.has('bbcc'));//true

map.clear();

console.log(map);

2.4 Iterator遍历器

2.4.1 Iterator（遍历器）的概念

JavaScript 原有的表示“集合”的数据结构，主要是数组（Array）和对象（Object），ES6 又添加了Map和Set。这样就有了四种数据集合，用户还可以组合使用它们，定义自己的数据结构，比如数组的成员是Map，Map的成员是对象。这样就需要一种统一的接口机制，来处理所有不同的数据结构。

遍历器（Iterator）就是这样一种机制。它是一种接口，为各种不同的数据结构提供统一的访问机制。任何数据结构只要部署 Iterator 接口，就可以完成遍历操作（即依次处理该数据结构的所有成员）。

Iterator 的作用有三个：一是为各种数据结构，提供一个统一的、简便的访问接口；二是使得数据结构的成员能够按某种次序排列；三是 ES6 创造了一种新的遍历命令for...of循环，Iterator 接口主要供for...of消费（意味着for...of是iterator接口的上层表现形式，我们在使用for...of的遍历集合的时候，底层运行逻辑为遍历部署了iterator接口的集合）。

Iterator 的遍历过程是这样的。

（1）创建一个指针对象，指向当前数据结构的起始位置。也就是说，遍历器对象本质上，就是一个指针对象。

（2）第一次调用指针对象的next方法，可以将指针指向数据结构的第一个成员。

（3）第二次调用指针对象的next方法，指针就指向数据结构的第二个成员。

（4）不断调用指针对象的next方法，直到它指向数据结构的结束位置。

每一次调用next方法，都会返回数据结构的当前成员的信息。具体来说，就是返回一个包含value和done两个属性的对象。其中，value属性是当前成员的值，done属性是一个布尔值，表示遍历是否结束。

下面是一个模拟next方法返回值的例子。

var it = makeIterator(['a', 'b']);

it.next() // { value: "a", done: false }

it.next() // { value: "b", done: false }

it.next() // { value: undefined, done: true }

function makeIterator(array) {

var nextIndex = 0;

return {

next: function() {

return nextIndex < array.length ?

{value: array[nextIndex++], done: false} :

{value: undefined, done: true};

}

};

}

2.4.2 默认 Iterator 接口

Iterator 接口的目的，就是为所有数据结构，提供了一种统一的访问机制，即for...of循环。当使用for...of循环遍历某种数据结构时，该循环会自动去寻找 Iterator 接口。

一种数据结构只要部署了 Iterator 接口，我们就称这种数据结构是“可遍历的”（iterable）。

ES6 规定，默认的 Iterator 接口部署在数据结构的Symbol.iterator属性，或者说，一个数据结构只要具有Symbol.iterator属性，就可以认为是“可遍历的”（iterable）。Symbol.iterator属性本身是一个函数，就是当前数据结构默认的遍历器生成函数。执行这个函数，就会返回一个遍历器。至于属性名Symbol.iterator，它是一个表达式，返回Symbol对象的iterator属性，这是一个预定义好的、类型为 Symbol 的特殊值。

const obj = {

[Symbol.iterator] : function () {

return {

next: function () {

return {

value: 1,

done: true

};

}

};

}

};

ES6 的有些数据结构原生具备 Iterator 接口（比如数组），即不用任何处理，就可以被for...of循环遍历。原因在于，这些数据结构原生部署了Symbol.iterator属性（详见下文），另外一些数据结构没有（比如对象）。凡是部署了Symbol.iterator属性的数据结构，就称为部署了遍历器接口。调用这个接口，就会返回一个遍历器对象。

原生具备 Iterator 接口的数据结构包括Array、Map、Set、String、TypedArray、函数的 arguments 对象、NodeList 对象。

下面的例子是数组的Symbol.iterator属性。

let arr = ['a', 'b', 'c'];

let iter = arr[Symbol.iterator]();

iter.next() // { value: 'a', done: false }

iter.next() // { value: 'b', done: false }

iter.next() // { value: 'c', done: false }

iter.next() // { value: undefined, done: true }

2.5 Generator函数

2.5.1 简介

Generator 函数是 ES6 提供的一种异步编程解决方案，语法行为与传统函数完全不同。Generator 函数有多种理解角度。语法上，首先可以把它理解成，Generator 函数是一个状态机，封装了多个内部状态。

执行 Generator 函数会返回一个遍历器对象，也就是说，Generator 函数除了状态机，还是一个遍历器对象生成函数。返回的遍历器对象，可以依次遍历 Generator 函数内部的每一个状态。

形式上，Generator 函数是一个普通函数，但是有两个特征。一是，function关键字与函数名之间有一个星号；二是，函数体内部使用yield表达式，定义不同的内部状态。

function\* helloGenerator() {

yield 'hello';

yield 'Generator';

return 'ending';

}

var hw = helloGenerator();

上面代码定义了一个 Generator 函数helloGenerator，它内部有两个yield表达式（hello和Generator），即该函数有三个状态：hello，Generator和 return 语句（结束执行）。

然后，Generator 函数的调用方法与普通函数一样，也是在函数名后面加上一对圆括号。不同的是，调用 Generator 函数后，该函数并不执行，返回的也不是函数运行结果，而是一个指向内部状态的指针对象，也就是上一章介绍的遍历器对象（Iterator Object）。

下一步，必须调用遍历器对象的next方法，使得指针移向下一个状态。也就是说，每次调用next方法，内部指针就从函数头部或上一次停下来的地方开始执行，直到遇到下一个yield表达式（或return语句）为止。换言之，Generator 函数是分段执行的，yield表达式是暂停执行的标记，而next方法可以恢复执行。

hw.next()

// { value: 'hello', done: false }

hw.next()

// { value: 'Generator ', done: false }

hw.next()

// { value: 'ending', done: true }

hw.next()

// { value: undefined, done: true }

页面代码一共调用了四次next方法。

第一次调用，Generator 函数开始执行，直到遇到第一个yield表达式为止。next方法返回一个对象，它的value属性就是当前yield表达式的值hello，done属性的值false，表示遍历还没有结束。

第二次调用，Generator 函数从上次yield表达式停下的地方，一直执行到下一个yield表达式。next方法返回的对象的value属性就是当前yield表达式的值Generator，done属性的值false，表示遍历还没有结束。

第三次调用，Generator 函数从上次yield表达式停下的地方，一直执行到return语句（如果没有return语句，就执行到函数结束）。next方法返回的对象的value属性，就是紧跟在return语句后面的表达式的值（如果没有return语句，则value属性的值为undefined），done属性的值true，表示遍历已经结束。

第四次调用，此时 Generator 函数已经运行完毕，next方法返回对象的value属性为undefined，done属性为true。以后再调用next方法，返回的都是这个值。

总结一下，调用 Generator 函数，返回一个遍历器对象，代表 Generator 函数的内部指针。以后，每次调用遍历器对象的next方法，就会返回一个有着value和done两个属性的对象。value属性表示当前的内部状态的值，是yield表达式后面那个表达式的值；done属性是一个布尔值，表示是否遍历结束。

2.5.2 yield 表达式

由于 Generator 函数返回的遍历器对象，只有调用next方法才会遍历下一个内部状态，所以其实提供了一种可以暂停执行的函数。yield表达式就是暂停标志。

遍历器对象的next方法的运行逻辑如下：

（1）遇到yield表达式，就暂停执行后面的操作，并将紧跟在yield后面的那个表达式的值，作为返回的对象的value属性值。

（2）下一次调用next方法时，再继续往下执行，直到遇到下一个yield表达式。

（3）如果没有再遇到新的yield表达式，就一直运行到函数结束，直到return语句为止，并将return语句后面的表达式的值，作为返回的对象的value属性值。

（4）如果该函数没有return语句，则返回的对象的value属性值为undefined。

需要注意的是，yield表达式后面的表达式，只有当调用next方法、内部指针指向该语句时才会执行，因此等于为 JavaScript 提供了手动的“惰性求值”（Lazy Evaluation）的语法功能。

function\* gen() {

yield 12 + 45

}

上面代码中，yield后面的表达式12+ 45，不会立即求值，只会在next方法将指针移到这一句时，才会求值。

yield表达式与return语句既有相似之处，也有区别。相似之处在于，都能返回紧跟在语句后面的那个表达式的值。区别在于每次遇到yield，函数暂停执行，下一次再从该位置继续向后执行，而return语句不具备位置记忆的功能。一个函数里面，只能执行一次（或者说一个）return语句，但是可以执行多次（或者说多个）yield表达式。正常函数只能返回一个值，因为只能执行一次return；Generator 函数可以返回一系列的值，因为可以有任意多个yield。从另一个角度看，也可以说 Generator 生成了一系列的值，这也就是它的名称的来历（英语中，generator 这个词是“生成器”的意思）。

2.5.3 next 方法的参数

yield表达式本身没有返回值，或者说总是返回undefined。next方法可以带一个参数，该参数就会被当作上一个yield表达式的返回值。

function\* main(){

var r1 = yield 'hello'

console.log(r1)

var r2 = yield 'ujiuye'

console.log(r2)

}

var ge = main()

ge.next()

ge.next('HELLO')

ge.next('UIJUYE')

上述代码依次输出 HELLO和UJIUYE

2.6 Promise对象

Promise 是异步编程的一种解决方案，比传统的解决方案——回调函数和事件——更合理和更强大。它由社区最早提出和实现，ES6 将其写进了语言标准，统一了用法，原生提供了Promise对象。

所谓Promise，简单说就是一个容器，里面保存着某个未来才会结束的事件（通常是一个异步操作）的结果。从语法上说，Promise 是一个对象，从它可以获取异步操作的消息。Promise 提供统一的 API，各种异步操作都可以用同样的方法进行处理。

Promise对象有以下两个特点：

（1）对象的状态不受外界影响。Promise对象代表一个异步操作，有三种状态：pending（进行中）、fulfilled（已成功）和rejected（已失败）。只有异步操作的结果，可以决定当前是哪一种状态，任何其他操作都无法改变这个状态。

（2）一旦状态改变，就不会再变，任何时候都可以得到这个结果。Promise对象的状态改变，只有两种可能：从pending变为fulfilled和从pending变为rejected。只要这两种情况发生，状态就凝固了，不会再变了，会一直保持这个结果，这时就称为resolved（已定型）。

有了Promise对象，就可以将异步操作以同步操作的流程表达出来，避免了层层嵌套的回调函数。此外，Promise对象提供统一的接口，使得控制异步操作更加容易。

Promise也有一些缺点。首先，无法取消Promise，一旦新建它就会立即执行，无法中途取消。其次，如果不设置回调函数，Promise内部抛出的错误，不会反应到外部。第三，当处于pending状态时，无法得知目前进展到哪一个阶段（刚刚开始还是即将完成）。

2.6.1 Promise对象的基本用法

es6 规定，Promise对象是一个构造函数，用来生成Promise实例。

**例**：

var promis = new Promise(function (resolve,reject) {

/\*异步执行代码：定时器或ajax\*/

if(true/\*异步操作成功\*/){

resolve(value);

}else {

reject(value);

}

})

Promise构造函数接受一个函数作为参数，该函数的两个参数分别是resolve和reject。它们是两个函数，由 JavaScript 引擎提供，不用自己部署。

resolve函数的作用是，将Promise对象的状态从“未完成”变为“成功”（即从 pending 变为 resolved），在异步操作成功时调用，并将异步操作的结果，作为参数传递出去；reject函数的作用是，将Promise对象的状态从“未完成”变为“失败”（即从 pending 变为 rejected），在异步操作失败时调用，并将异步操作报出的错误，作为参数传递出去。

Promise实例生成以后，可以用then方法分别指定resolved状态和rejected状态的回调函数。

例：

promise.then(function(value) {

// success

}, function(error) {

// failure

});

then方法可以接受两个回调函数作为参数。第一个回调函数是Promise对象的状态变为resolved时调用，第二个回调函数是Promise对象的状态变为rejected时调用。其中，第二个函数是可选的，不一定要提供。这两个函数都接受Promise对象传出的值作为参数（resolve和reject函数调用时所携带的参数）。

例：

function getJson(url) {

var promise = new Promise(function (resolve,reject) {

var ajax = new XMLHttpRequest();

ajax.open("GET",url);

ajax.send();

ajax.onreadystatechange = function () {

if(ajax.readyState != 4){

return;

}

if(ajax.status == 200){

resolve(ajax.response);

}else {

reject(new Error(ajax.statusText));

}

}

});

return promise;

}

//调用，then回调

getJson("file.json").then(function (json) {

console.log(json);

},function (text) {

console.log(text);

});

上面代码中，getJson是对XMLHttpRequest对象的封装，用于发出一个针对JSON数据的HTTP请求，并返回一个Priomis对象。

需要注意的是，resolve等函数都带有参数，这些参数会传递给回调函数，reject函数的参数通常是Error对象的实例，表示抛出的错误。

2.6.2 Promise.prototype.then()

Promise实例具有then方法，因此then方法是被定义在对象原型上的，他的作用是为Promise实例添加状态改变时的回调函数。

语法：Promise实例.then（function (){成功时调用},function(){失败时调用}可选）。

例：

getJson("file.json").then(function (json) {

console.log(json);

});

then方法也是会有返回值，返回是一个新的Promise对象，因此我们可以采用链式操作，即then方法后面调用另一个then方法。

例：

getJson("file.json").then(function (json) {

return getJson("file.txt");

}).then(function () {

console.log("Fd");

},function () {

console.log("fd");

})

2.6.3 Promise.prototype.catch()

Promise.prototype.catch方法是.then(null, rejection)或.then(undefined, rejection)的别名，用于指定发生错误时的回调函数。

getJSON('/posts.json').then(function(posts) {

// ...

}).catch(function(error) {

// 处理 getJSON 和 前一个回调函数运行时发生的错误

console.log('发生错误！', error);

});

上面代码中，getJSON方法返回一个 Promise 对象，如果该对象状态变为resolved，则会调用then方法指定的回调函数；如果异步操作抛出错误，状态就会变为rejected，就会调用catch方法指定的回调函数，处理这个错误。另外，then方法指定的回调函数，如果运行中抛出错误，也会被catch方法捕获。

一般来说，不要在then方法里面定义 Reject 状态的回调函数（即then的第二个参数），应该总是使用catch方法。

// bad

promise.then(function(data) {

// success

}, function(err) {

// error

});

// good

promise.then(function(data) {

// success

}).catch(function(err) {

// error

});

上面代码中，第二种写法要好于第一种写法，理由是第二种写法可以捕获前面then方法执行中的错误，也更接近同步的写法（try/catch）。因此，建议总是使用catch方法，而不使用then方法的第二个参数。

2.6.4 Promise.all()

Promise.all()方法用于将多个 Promise 实例，包装成一个新的 Promise 实例。

const p = Promise.all([p1, p2, p3]);

上面代码中，Promise.all()方法接受一个数组作为参数，p1、p2、p3都是 Promise 实例。p的状态由p1、p2、p3决定，分成两种情况：

（1）只有p1、p2、p3的状态都变成fulfilled，p的状态才会变成fulfilled，此时p1、p2、p3的返回值组成一个数组，传递给p的回调函数。

（2）只要p1、p2、p3之中有一个被rejected，p的状态就变成rejected，此时第一个被reject的实例的返回值，会传递给p的回调函数。

注意，如果作为参数的 Promise 实例，自己定义了catch方法，那么它一旦被rejected，并不会触发Promise.all()的catch方法。

2.6.5 Promise.race()

Promise.race()方法同样是将多个 Promise 实例，包装成一个新的 Promise 实例。

const p = Promise.race([p1, p2, p3]);

上面代码中，只要p1、p2、p3之中有一个实例 率先改变状态，p的状态就跟着改变。那个率先改变的 Promise 实例的返回值，就传递给p的回调函数。

Promise.race()方法的参数与Promise.all()方法一样，如果不是 Promise 实例，就会先调用下面讲到的Promise.resolve()方法，将参数转为 Promise 实例，再进一步处理。

下面是一个例子，如果指定时间内没有获得结果，就将 Promise 的状态变为reject，否则变为resolve。

const p = Promise.race([

fetch('/resource-that-may-take-a-while'),

new Promise(function (resolve, reject) {

setTimeout(() => reject(new Error('request timeout')), 5000)

})]);

p.then(console.log).catch(console.error);

上面代码中，如果 5 秒之内fetch方法无法返回结果，变量p的状态就会变为rejected，从而触发catch方法指定的回调函数。

2.7 async函数

2.7.1 简介

ES2017 标准引入了 async 函数，使得异步操作变得更加方便。async 函数其实就是 Generator 函数的语法糖。async函数就是将 Generator 函数的星号（\*）替换成async，将yield替换成await

async函数返回一个 Promise 对象，可以使用then方法添加回调函数。当函数执行的时候，一旦遇到await就会先返回，等到异步操作完成，再接着执行函数体内后面的语句。

function timeout(ms) {

return new Promise((resolve) => {

setTimeout(resolve, ms);

});

}

async function asyncPrint(value, ms) {

await timeout(ms);

console.log(value);

}

asyncPrint('hello world', 1000);

上面代码指定 1000 毫秒以后，输出hello world。

2.7.2 async与promise

async函数返回一个 Promise 对象。async函数内部return语句返回的值，会成为then方法回调函数的参数。resolve函数传递的参数返回值给await。

// await

async function awaitTest() {

let result = await Promise.resolve('执行成功');

console.log(result);

// let result2 = await Promise.reject('执行失败');

// console.log(result2);

let result3 = await Promise.resolve('再执行一次');//

console.log(result3);

}

//awaitTest();

**注意：**await不能单独存在，必须要出现在async修饰的函数内，否则报错。await常常用来等待异步运行，在实际情况下更多的可能是ajax操作。

2.8 高频面试

● 手写一个promise

var p = new Promise((resolve,reject)={

if(1){

resolve(‘成功’)

}

else{

reject(‘失败’);

}

})

p.then((res)=>{}).catch((err)=>{})

● class定义类

class Aniaml{

static Type = ‘动物’

constructor(newName){

this.name = newName

}

eat(){conosle.log(‘吃东西’)}

static sayHi(){

console.log(‘大家好’)

}

}

● class实现继承

class Tigger extends Animal{

constructor(props){

super(props)

}

}

● 使用promise封装ajax

● async和await的基本用法

async function asyncPrint(value, ms) {

await timeout(ms);

console.log(value);

}

asyncPrint('hello world', 1000);

● 简单介绍一下promise

从语法上说，Promise 是一个对象，从它可以获取异步操作的消息。Promise对象有以下两个特点：

（1）Promise有三种状态：pending（进行中）、fulfilled（已成功）和rejected（已失败）。

（2）一旦状态改变，就不会再变，任何时候都可以得到这个结果

第3讲 ES6 语法

|  |  |  |  |
| --- | --- | --- | --- |
| 知识模块 | 课程内容 | 重点 | 课程目标 |
| ES6 | 面向对象介绍 | ★ | 掌握：面向对象的概念关系，类、对象、属性和方法  掌握：class的基本用法和语义  掌握：成员和静态的使用场景，各自不同的写法和兼容情况  掌握：ES6继承的写法，constructor的作用  掌握：super对象和super方法的使用场景  掌握：深度克隆原理和json克隆的用法 |
| Class的基本用法 | ★ |
| 成员和静态的区别 | ★ |
| ES6继承的写法 | ★ |
| Super用法 | ★ |
| 深度克隆 | ★ |

3.1 本讲重点

● 掌握面向对象中的概念关系，类，对象，属性，和方法；掌握成员和静态的区别和使用场景

● 掌握class的基本语义和继承的写法

● 掌握super的对象和super方法的使用场景

● 掌握深度克隆和json克隆

3.2 Class的基本用法

3.2.1 面向对象介绍

计算机的发明是为了解决人们现实生产、生活中遇到的问题；我们在现实生活中遇到问题的解决措施往往都是步骤化、过程化的，也就是说通常我们看待问题的时候会把问题分解为若干个小问题，按照一定步骤串联，并依次解决；在程序中这种解决问题的思路我们称之为面向过程的编程，即解决问题的时候从问题的过程下手，把问题步骤化、过程化；比如：我们需要请求一组数据，然后渲染到页面上。这里面我们可能需要事先封装两个方法方便使用，一个方法用于请求数据，一个方法用于渲染数据；这两个方法串联起来按照先后调用就能够完成我们的目标。

但这也仅仅是一个比较简单的任务，如果我们遇到的问题很大，想象一下我们现在要为飞机写一套飞控系统的程序，事实上，可能程序还没写，我们已经被飞机上眼花缭乱的功能按键震惊。这样的系统功能应该是非常复杂，如果我们把所有可见和不可见的功能都封装为函数，这些函数的数量将相当庞大，再把这些函数串联起来协同工作，绝对是剪不断理还乱的事情。

随着社会的发展进步，人们的生活需求越来越高，软件的发展也越来越复杂、庞大、精细化；面向过程思路在解决问题是已经是捉襟见肘；上世纪70年代，图灵奖获得者  
Alan Curtis Kay（艾伦·凯）提出了面向对象的思想，可以说，他是面向对象思想的鼻祖。图灵奖由[美国计算机协会](https://baike.baidu.com/item/%E7%BE%8E%E5%9B%BD%E8%AE%A1%E7%AE%97%E6%9C%BA%E5%8D%8F%E4%BC%9A/1896077" \t "https://baike.baidu.com/item/%E5%9B%BE%E7%81%B5%E5%A5%96/_blank)（ACM）于1966年设立，专门奖励那些对计算机事业作出重要贡献的个人。一般每年只奖励一名计算机科学家，只有极少数年度有两名合作者或在同一方向作出贡献的科学家共享此奖。因此它是计算机界最负盛名、最崇高的一个奖项，有“计算机界的[诺贝尔奖](https://baike.baidu.com/item/%E8%AF%BA%E8%B4%9D%E5%B0%94%E5%A5%96/187878" \t "https://baike.baidu.com/item/%E5%9B%BE%E7%81%B5%E5%A5%96/_blank)”之称。

面向对象编程在解决问题的时候更注重的是关系，面向对象编程中有这样几个概念：类、对象、属性、方法。我们以现实社会来做比喻，人类就是一个类的概念，我们为什么称之为人类？因为人类的属性和方法有别于其他物种，人类的属性我们可以认为是一些静态特征，比如：胳膊、手，腿、脚等等；人类的方法我们可以认为是一些动态特征，比如：直立行走，制造工具，弹跳等。再比如猫类，猫类具备的属性和方法与人类就不一样了；那么猫类和人类之间怎么会有关联？事实上我们有养猫的习惯，某个猫可能属于某个人，我们可以说这只猫是这个人的宠物，是一种归属关系。也就是说这个人有一个特殊的属性就是宠物，他的宠物里面有一只猫。所以，面向对象描述问题的思路就是这样的。

这样来看，类似乎是一种封装，它内部封装了属性（变量）和方法（函数），这是一种比函数封装更加高级的做法，面向对象的三大典型特征之一就是封装。

我们在进行面向对象编程的时候应该首先定义一个类，类里面描述这个类应该具备的属性和方法（想象一下我们刚才的人类和猫类），类描述完毕之后就是对类的使用了，类这个概念是抽象的，现实生活中的每个人都是类的一个实例化表现，所以我们经常听到一个概念叫做——实例化一个对象；（类）抽象到（对象）实例的过程就称作实例化。所以，每个人类对象都应该具备人类这个类所描述的属性和方法。这些属性和方法，我们称之为成员属性和成员方法，后面我们还会说到静态属性和静态方法以及继承的概念。接下来我们还是先看一下ES6中是怎么定义类、属性、方法和对象吧！

3.2.2 Class基础语法

JavaScript 语言中，生成实例对象的传统方法是通过构造函数。如下示例：

function Person( name ){

this.name = name

}

Person.prototype.say= function () {

console.log(this.name+'说明')

}

var p1 = new Person('小明')

p1.say()

ES6 提供了更接近传统语言的写法，引入了 Class（类）这个概念，作为对象的模板。通过class关键字，可以定义类。

基本上，ES6 的class可以看作只是一个语法糖，它的绝大部分功能，ES5 都可以做到，新的class写法只是让对象原型的写法更加清晰、更像面向对象编程的语法而已。上面的代码用 ES6 的class改写，就是下面这样。

class Person {

//调用类的构造方法

constructor(name, age){

this.name = name;

this.age = age;

}

//定义一般的方法

showName(){

console.log(this.name, this.age);

}

}

let person = new Person('优就业', 10);

console.log(person, person.showName());

上面代码定义了一个“类”，可以看到里面有一个constructor方法，这就是构造方法，而this关键字则代表实例对象。也就是说，ES5 的构造函数Person，对应 ES6 的Person类的构造方法。

这是一个基本示例，从上文关于面向对象的介绍中我们知道了类内部是描述属性和方法的，这些属性和方法在ES6的class语法中写的时候有哪些注意事项呢？我们在静态和成员中进行介绍。

3.3 成员和静态的区别

关于成员和静态的区别，我们需要各自了解什么是成员和静态之后才能有所对比；成员和静态只是两个概念，我们认为由对象调用的属性和方法叫做成员属性和成员方法，由类直接调用的属性和方法叫做静态属性和静态方法，也就是说成员与静态的区别可以看做是对象和类的区别；从这里我们知道了几个概念：成员属性，成员方法，静态属性，静态方法；接下来我们通过下面案例来学习这几个概念。

声明一个Product类，如下：

class Product {

// var title = '产品-1';//错误的成员属性写作形式

// this.title = '产品-1'//错误的成员属性写作形式

title = "产品-1"; //正确的成员属性

constructor() {

var joke = "局部变量"; //正确的局部变量

this.brand = "品牌"; //正确的成员属性

}

// 正确的成员方法书写形式

funs() {

console.log(this.title); //产品-1

console.log(this.brand); //品牌

// console.log(joke); //Uncaught ReferenceError: joke is not defined

console.log(this.joke); //undefined

}

// 错误的方法写作形式

// function funs(){

// console.log("功能描述");

// }

// 错误的方法写作形式

// this.funs = function(){

// console.log("功能描述");

// }

// 错误的方法写作形式

// var funs = function(){

// console.log("功能描述");

// }

}

let p = new Product();

console.log(p.title); //产品-1

p.funs();

通过例子我们了解到，成员属性可以写到class内部的所有方法之外（类顶层），不需要var，this的修饰；也可以直接在constructor方法内使用this来定义；这是ES6定义成员属性的两种方式。

第一种定义形式，也就是写在所有方法之外的形式（类顶层）有一定的兼容问题，部分node版本中无法兼容使用。 但这种形式符合先声明后使用的原则，一目了然，如果能够兼容，推荐使用。

第二种定义形式，是将成员属性定义在constructor方法里面的，事实上constructor起到的作用是初始化，初始化程序内部的成员属性状态，初始化一些方法的调用。所以如果条件允许，推荐使用第一种定义形式，这会让你的代码看上去更直观、优雅。

关于成员方法，上面例子也给出了用法，不要试图给class类内部的方法加function修饰，使用方法名和形参列表即可。不要在class类内部的任意方法体外写分号或逗号，会引起错误。需要注意的是在方法内使用let或var定义的仍旧是变量，只需要注意它们的作用域即可。类内部通过this使用成员属性，局部变量无法通过this使用。

接下来我们再通过一个例子来了解静态属性和静态方法，继续完善上面的Product类，如下：

class Product {

static type = "PRO"; //正确的静态属性

// var title = '产品-1';//错误的成员属性写作形式

// this.title = '产品-1'//错误的成员属性写作形式

title = "产品-1"; //正确的成员属性

constructor() {

var joke = "局部变量"; //正确的局部变量

this.brand = "品牌"; //正确的成员属性

}

// 正确的成员方法书写形式

funs() {

console.log(this.title); //产品-1

console.log(this.brand); //品牌

// console.log(joke); //Uncaught ReferenceError: joke is not defined

console.log(this.joke); //undefined

}

//静态方法

static pType() {

console.log(this.type); //PRO

console.log(this.title); //undefined

}

// 错误的方法写作形式

// function funs(){

// console.log("功能描述");

// }

// 错误的方法写作形式

// this.funs = function(){

// console.log("功能描述");

// }

// 错误的方法写作形式

// var funs = function(){

// console.log("功能描述");

// }

}

Product.category = "产品分类";

Product.pCategory = function() {

console.log(this.type); //PRO

console.log(this.category); //产品分类

};

let p = new Product();

// console.log(p.type); //undefined

// p.pType(); //Uncaught TypeError: p.pType is not a function

// p.pCategory(); //Uncaught TypeError: p.pCategory is not a function

console.log(Product.type); //PRO

Product.pType();

Product.pCategory();

通过上面的例子，我们发现静态属性和静态方法的写法也有两种，可以通过增加static关键字修饰，也可以在class类外直接给类名字增加属性。

第一种写法是通过在属性或方法前面增加一个static关键字来修饰，让属性或方法变成静态属性或方法。

第二种写法在ES5中就有，直接给类名字增加属性即可。

其中第一种写法在node的部分版本中仍旧有兼容问题，不过在条件允许的情况下，还是推荐第一种写法，这样的写法看上去更直观、优雅。

那么成员和静态的区别是什么呢？

其实，很简单。成员的属性和成员的方法，只能由对象使用；静态的属性和方法只能由类名字直接使用。需要注意this的用法，我们都知道this表示当前调用者，但是在一些具体的场景中仍旧会弄错，比如在static修饰的方法中，this表示什么？在成员方法中this又表示什么？this在static修饰的方法中表示当前调用者，即类本身；在成员方法中表示当前调用者，即当前调用对象。

我们应该使用成员还是静态呢？毕竟成员和静态都可以描述属性和方法。这要看它们的特性来决定，比如我们定义了一个人类person，使用person类的时候我们该以成员的形式还是类的形式来使用？如果我们要定义三个人：小明，小华和小李。这必然要使用成员的形式；因为我们需要声明三个对象，这三个对象的属性和方法可能都是不同的；每个对象都具备一份独立的成员属性和成员方法的拷贝；也就是小明和小李的name属性不会冲突，都是各自对象的独立属性。因为类的声明只能有一次，再次声明同名的类会报错，所以我们认为类在特定环境下是唯一的。有了这份认识，我们就可以认为类的属性和方法是唯一的；也就是说静态属性和方法是唯一的，重名会被覆盖。很显然，一些没有必要随着每次声明对象而产生的属性和方法就可以使用静态属性和静态方法；比如person有个type属性为“人类”，不论是小明还是小华都是人类，所以这个属性可以放在静态属性中，这样可以避免反复的内存占用（每个对象的属性都是要占用内存空间的），节省内存。如果我们把Math看做是一个类，考虑一下Math.sin(),Math.cos()的用法，如果换成是成员的操作该怎么做？是否有必要？是否便利？

3.4 ES6继承的写法

ES6 Class 可以通过extends关键字实现继承，这比 ES5 的通过修改原型链实现继承，要清晰和方便很多。

语法：

class Father{

}

class Son extends Father {

}

上面代码定义了一个Son 类，该类通过extends关键字，继承了Father类的所有属性和方法。但是由于没有部署任何代码，所以这两个类完全一样，等于复制了一个Father类。下面，我们是具体实例。

class Students extends Person{

constructor(name, age, city){

super(name, age);//调用父类的构造方法

this.city = city;

}

showName(){//在子类自身定义方法

console.log(this.name, this.age, this.city);

}

}

let str = new Students ('中公教育', 20, '北京');

console.log(str);

str.showName();

子类必须在constructor方法中调用super方法，否则新建实例时会报错。这是因为子类自己的this对象，必须先通过父类的构造函数完成塑造，得到与父类同样的实例属性和方法，然后再对其进行加工，加上子类自己的实例属性和方法。如果不调用super方法，子类就得不到this对象。

这很重要，子类可以不实现constructor方法，但是如果实现constructor方法则必须在constructor的第一行调用super方法初始化父类，否则会报错。

父类已经实现的属性和方法，子类直接使用即可，不需要再次定义；如果父类没有实现，子类在增加属性的时候应该实现constructor方法，增加属性的同时，初始化父类的属性。子类增加方法没有特殊。如果子类实现了父类的同名方法，这种情形叫做覆盖；之所以出现这种做法，是因为父类的功能无法满足子类需求，子类需要另行实现。这样一来就有一个问题，子类在调用方法的时候无法调用到父类的同名方法，因为子类已经覆盖了这个方法；可以使用super对象来解决。

3.5 Super用法

super在ES6的class中有两个用法：

1、super方法：super方法仅能出现在子类继承父类时的constructor方法内，且应该写在第一行的位置。作用是初始化父类的属性。

2、super对象：super对象用在子类覆盖的父类同名方法内；用来调用父类被覆盖的同名方法，比如：super.父类方法()

3.6 深度克隆

浅拷贝只是复制了对象的引用地址，两个对象指向同一个内存地址，所以修改其中任意的值，另一个值都会随之变化，这就是浅拷贝（例：assign()）。深拷贝是将对象及值复制过来，两个对象修改其中任意的值另一个值不会改变，这就是深拷贝（例：JSON.parse()和JSON.stringify()，但是此方法无法复制函数类型）。最终的方案是：**浅拷贝+递归**实现深拷贝。

function deepCopy( obj ){

if( Object.prototype.toString.call( obj ).slice(8,-1) == 'Object' ){

var result = {}

}else if( Object.prototype.toString.call( obj ).slice(8,-1) == 'Array' ){

var result = []

}//判断数据类型类型 如果是数组则初始一个 [] 如果是一个Object则初始一个 {}

//浅拷贝，但是+ 递归思想，就实现了深拷贝

for( var attr in obj ){

if( typeof obj[attr] == 'object'){

result[attr] = deepCopy( obj[attr] )

}else{

result[attr] = obj[attr]

}

}

return result

}

3.7 高频面试

● 实现深拷贝

let obj = {name:’zhangsan’,age:22}

let obj\_str = JSON.stringfy(obj);

let obj\_clone = JSON.parse(obj\_str);

● ES6如何实现继承,ES5如何实现

ES6:

class Father{

}

class Son extends Father {

}

ES5: 原型链继承

function Dad(){

}

function Son(){

}

Son.prototype=new Dad()

● js继承方式及其优缺点

1、原型链继承的缺点

一是字面量重写原型会中断关系，使用引用类型的原型，并且子类型还无法给超类型传递参数。

2、借用构造函数（类式继承）

借用构造函数虽然解决了刚才两种问题，但没有原型，则复用无从谈起。所以我们需要原型链+借用构造函数的模式，这种模式称为组合继承

3、组合式继承

组合式继承是比较常用的一种继承方法，其背后的思路是 使用原型链实现对原型属性和方法的继承，而通过借用构造函数来实现对实例属性的继承。这样，既通过在原型上定义方法实现了函数复用，又保证每个实例都有它自己的属性。

第4讲 Node.js 基础

|  |  |  |  |
| --- | --- | --- | --- |
| 知识模块 | 课程内容 | 重点 | 课程目标 |
| Node.js基础 | Node.js简介及安装 | ★ | 掌握：Node.js环境搭建  掌握：模块的基本使用  掌握：CommonJS规范  掌握：事件订阅与发布机制  掌握：包及NPM |
| 模块的基本认识 | ★ |
| CommonJS规范 | ★ |
| 事件订阅与发布机制 | ★ |
| 包和NPM | ★ |

4.1 Node.js简介

Node.js是一个能够在服务器端运行JavaScript的开放源代码、跨平台JavaScript运行环境。Node.js采用Google开发的V8引擎运行js代码，使用事件驱动、非阻塞和异步I/O模型等技术来提高性能，可优化应用程序的传输量和规模。Node.js大部分基本模块都用JavaScript编写。在Node.js出现之前，JS通常作为客户端程序设计语言使用，以JS写出的程序常在用户的浏览器上运行。目前，Node.js已被IBM、Microsoft、Yahoo!、Walmart、Groupon、SAP、LinkedIn、Rakuten、PayPal、Voxer和 GoDaddy等企业采用。

Node.js主要用于编写像Web服务器一样的网络应用，这和PHP和Python是类似的。但是Node.js与其他语言最大的不同之处在于，PHP等语言是阻塞的而Node.js是非阻塞的。Node.js是事件驱动的。开发者可以在不使用线程的情况下开发出一个能够承载高并发的服务器。其他服务器端语言难以开发高并发应用，而且即使开发出来，性能也不尽人意。Node.js正是在这个前提下被创造出来。Node.js把JS的易学易用和Unix网络编程的强大结合到了一起。

Node.js和其它语言相比：

● Node.js不是一种独立的语言，与PHP、JSP、Python的“既是语言，也是平台”不同，Node.js的使用JavaScript进行编程，运行在JavaScript引擎上（V8）。

● 与PHP、JSP等相比（PHP、JSP、.net都需要运行在服务器程序上，Apache、Nginx、IIS。），Node.js跳过了Apache、Nginx、IIS等HTTP服务器，它自己不用建设在任何服务器软件之上。Node.js的许多设计理念与经典架构（LAMP = Linux + Apache + MySQL + PHP）有着很大的不同，可以提供强大的伸缩能力。一会儿我们就将看到，Node没有web容器。

官网：https://nodejs.org/en/

4.2 特点

所谓的特点，就是Node.js是如何解决服务器高性能瓶颈问题的。

4.2.1 单线程

在Java、PHP或者.net等服务器端语言中，会为每一个客户端连接创建一个新的线程。而每个线程需要耗费大约2MB内存。也就是说，理论上，一个8GB内存的服务器可以同时连接的最大用户数为4000个左右。要让Web应用程序支持更多的用户，就需要增加服务器的数量，而Web应用程序的硬件成本当然就上升了。

Node.js不为每个客户连接创建一个新的线程，而仅仅使用一个线程。当有用户连接了，就触发一个内部事件，通过非阻塞I/O、事件驱动机制，让Node.js程序宏观上也是并行的。使用Node.js，一个8GB内存的服务器，可以同时处理超过4万用户的连接。

另外，单线程的好处还有操作系统完全不再有线程创建、销毁的时间开销。

坏处，就是一个用户造成了线程的崩溃，整个服务都崩溃了，其他人也崩溃了。

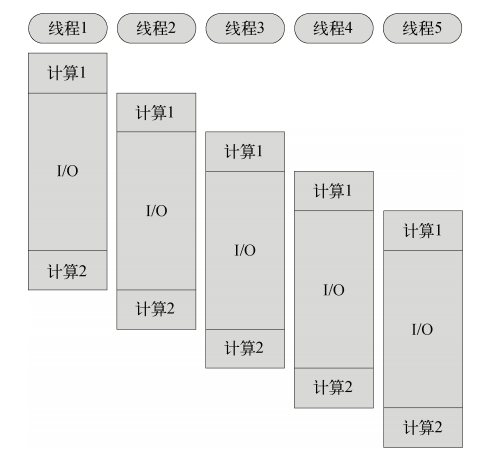


图4-1

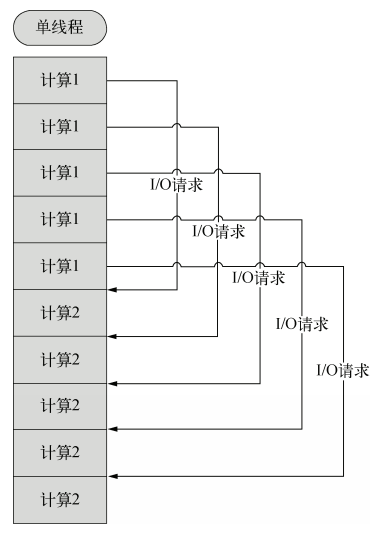


图4-2

总结：

● 进程就是一个一个的工作计划（工厂中的车间）

● 线程是计算机最小的运算单位（工厂中的工人）线程是干活的

● 传统的服务器都是多线程的，每进来一个请求，就创建一个线程去处理请求

● Node.js的服务器单线程的，Node.js处理请求时是单线程，但是在后台拥有一个I/O线程池

4.2.2 非阻塞I/O

I/O操作指的是对磁盘的读写操作。

异步IO的概念和同步IO相对。当一个异步过程调用发出后，调用者不能立刻得到结果。实际处理这个调用的部件在完成后，通过状态、通知和回调来通知调用者。在一个CPU密集型的应用中，有一些需要处理的数据可能放在磁盘上。预先知道这些数据的位置，所以预先发起异步IO读请求。等到真正需要用到这些数据的时候，再等待异步IO完成。使用了异步IO，在发起IO请求到实际使用数据这段时间内，程序还可以继续做其他事情。

例如，当在访问数据库取得数据的时候，需要一段时间。在传统的单线程处理机制中，在执行了访问数据库代码之后，整个线程都将暂停下来，等待数据库返回结果，才能执行后面的代码。也就是说，I/O阻塞了代码的执行，极大地降低了程序的执行效率。

由于Node.js中采用了非阻塞型I/O机制，因此在执行了访问数据库的代码之后，将立即转而执行其后面的代码，把数据库返回结果的处理代码放在回调函数中，从而提高了程序的执行效率。

当某个I/O执行完毕时，将以事件的形式通知执行I/O操作的线程，线程执行这个事件的回调函数。为了处理异步I/O，线程必须有事件循环，不断的检查有没有未处理的事件，依次予以处理。

阻塞模式下，一个线程只能处理一项任务，要想提高吞吐量必须通过多线程。而非阻塞模式下，一个线程永远在执行计算操作，这个线程的CPU核心利用率永远是100%。所以，这是一种特别有哲理的解决方案：与其人多，但是好多人闲着；还不如一个人玩命，往死里干活儿。

4.2.3 事件驱动

在Node.js中，客户端请求建立连接，提交数据等行为，会触发相应的事件。在Node.js中，在一个时刻，只能执行一个事件回调函数，但是在执行一个事件回调函数时，可以转而处理其他事件（比如，又有新用户连接了），然后返回继续执行原事件的回调函数，这种处理机制，称为“事件环”机制。

Node.js底层是C++（V8也是C++写的）。底层代码中，近半数都用于事件队列、回调函数队列的构建。用事件驱动来完成服务器的任务调度。用一个线程，担负起了处理非常多的任务的使命。

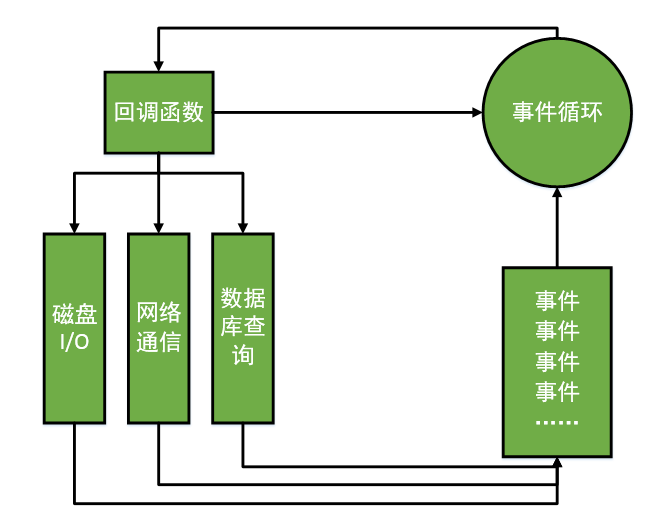


图4-3

单线程，单线程的好处，减少了内存开销，如果某一个事情，进入了，但是被I/O阻塞了，所以这个线程就阻塞了。非阻塞I/O， 不会等I/O语句结束，而会执行后面的语句。

非阻塞就能解决问题了么？比如执行着A的业务，执行过程中，B的I/O回调完成了，此时怎么办？

事件机制，事件环，不管是新用户的请求，还是老用户的I/O完成，都将以事件方式加入事件环，等待调度。

Node.js中所有的I/O都是异步的，回调函数套回调函数。

4.3 适合开发什么？

Node.js适合用来开发什么样的应用程序呢？

善于I/O，不善于计算。因为Node最擅长的就是任务调度，如果你的业务有很多的CPU计算，实际上也相当于这个计算阻塞了这个单线程，就不适合Node.js开发。

当应用程序需要处理大量并发的I/O，而在向客户端发出响应之前，应用程序内部并不需要进行非常复杂的处理的时候，Node.js非常适合。Node.js也非常适合与web socket配合，开发长连接的实时交互应用程序。

比如：

● Web服务API，比如REST

● 实时多人游戏

● 后端的Web服务，例如跨域、服务器端的请求

● 基于Web的应用

● 多客户端的通信，如即时通信

4.4 Node.js安装

4.4.1 官网下载

Node.js的官网为：<https://nodejs.org/en/>

安装包的下载地址：<https://nodejs.org/en/download/>

在LTS（Long Term Support，长期支持版）的选项卡下，根据不同平台系统选择你需要的 Node.js 安装包并下载：如下图

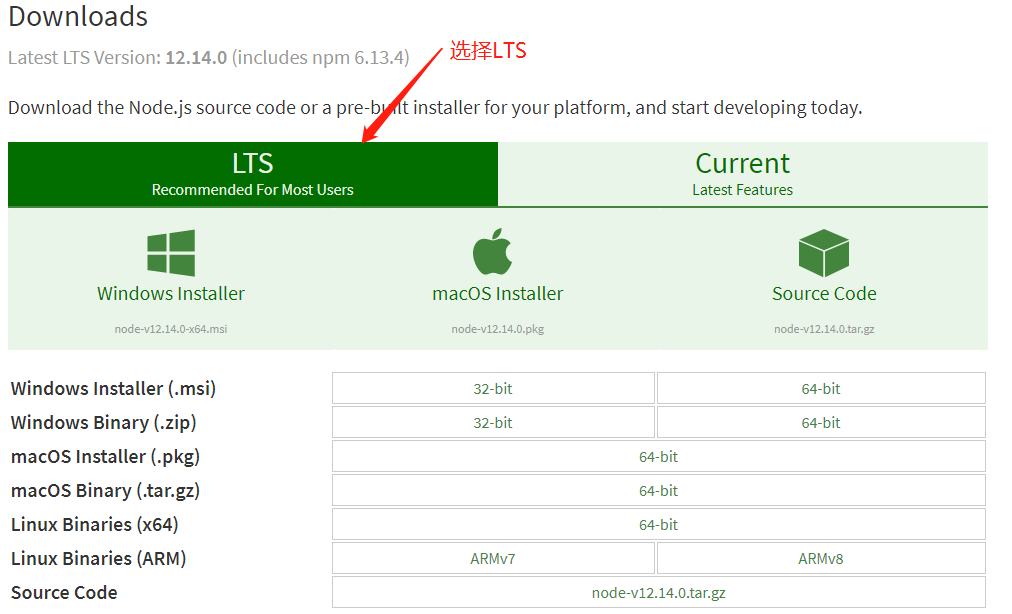


图4-4

4.4.2 安装Node.js

本文实例以 ***v12.14.0*** 版本为例，其他版本类似，安装步骤如下：

步骤一： 双击下载后的安装包 node-v12.14.0-x64.msi，并点击运行，如下图。

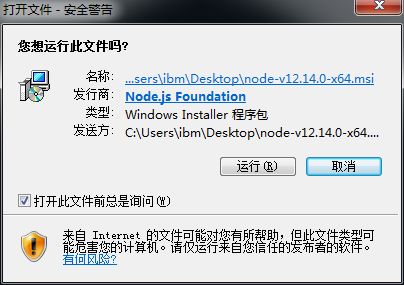


图4-5

步骤二：点击以上的运行，将出现如下界面，点击Next继续：

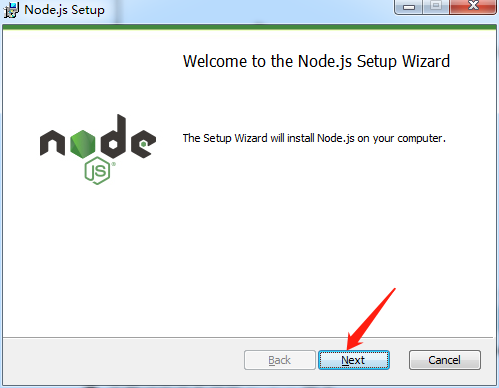


图4-6

步骤三：勾选接受协议选项，点击 Next（下一步） 按钮 :



图4-7

步骤四：Node.js默认安装目录为 "C:\Program Files\nodejs\" , 你可以修改目录，并点击 next（下一步）：

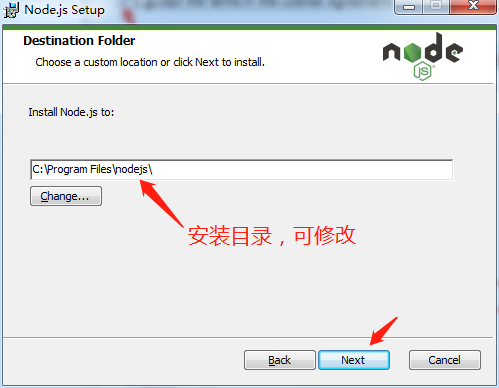


图4-8

步骤五: 默认Node.js的安装模式 , 直接点Next（下一步）

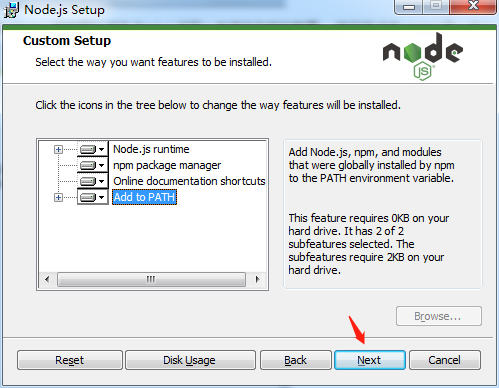


图4-9

这里共有4个大项：

1.安装Node核心模块

2.安装npm

3.安装文档

4.把Node.js和npm及其模块添加到环境变量

这里可以看到1和2就是Node核心程序和npm，安装Node的时候，默认会把Node和npm都安装上了。 另外4是把Node和npm添加到环境变量，如果不选，之后在cmd窗口想运行node、npm等执行程序会报错，系统找不到指定文件，当然也可以再手动配置。

这里建议全选，也就是默认选项，直接 Next

步骤六：点击next，然后继续安装。

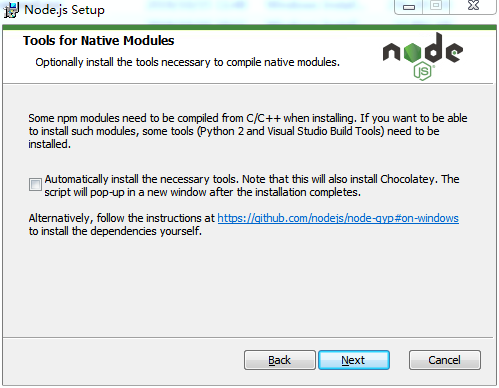


图4-10

步骤七：点击Install开始安装。

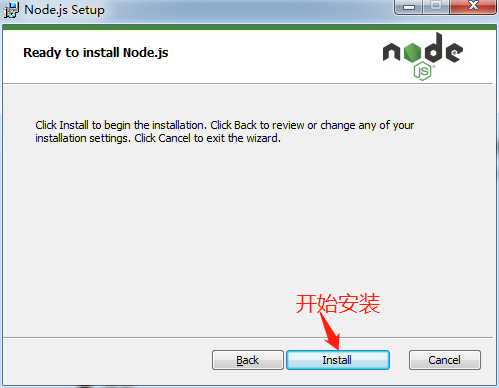


图4-11

安装过程：

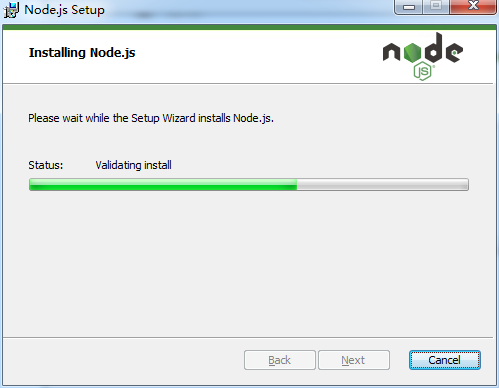


图4-12

步骤八：结束安装Node.js

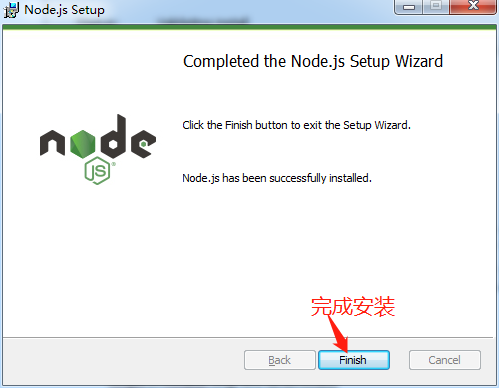


图4-13

4.4.3 CMD命令

cmd命令提示符是在操作系统中，提示进行命令输入的一种工作提示符。在不同的操作系统环境下，命令提示符各不相同。

在windows环境下，命令行程序为cmd.exe，是一个32位的命令行程序，微软Windows系统基于Windows上的命令解释程序，类似于微软的DOS操作系统。输入一些命令，cmd.exe可以执行，比如输入shutdown -s -t 30就会在30秒后关机。打开方法：开始-所有程序-附件 或 开始-寻找-输入：cmd/cmd.exe 回车。它也可以执行BAT文件。

执行node -v 命令检查Node.js的版本号。



图4-14

cd命令，就是change directory的缩写，表示更改当前目录。

**常用的指令：**

cls 清屏

dir 列出当前目录下的所有文件

cd 目录名 进入到指定的目录

md 目录名 创建一个文件夹

rd 目录名 删除一个文件夹（只能删除空文件夹）

cd .. 返回上层目录

4.4.4 环境变量

环境变量（environment variables）一般是指在操作系统中用来指定操作系统运行环境的一些参数，如：临时文件夹位置和系统文件夹位置等。

环境变量是在操作系统中一个具有特定名字的对象，它包含了一个或者多个应用程序所将使用到的信息。例如Windows和DOS操作系统中的path环境变量，当要求系统运行一个程序而没有告诉它程序所在的完整路径时，系统除了在当前目录下面寻找此程序外，还应到path中指定的路径去找。用户通过设置环境变量，来更好的运行进程。

安装完Node.js，环境变量path就会被重写，就是我们node安装的目录。

什么叫做环境变量path？对于刚刚安装的Node.js而言，就是在系统的任何目录下，都能运行c:\program files\nodejs里面的程序。在cmd中，输入node -v就能够查看Node.js的版本。你会发现，我们现在的盘符，不在安装目录下，但是也能够运行，这就是因为有系统环境变量。系统的环境变量已经有了c:\program files\nodejs了，所以，这个文件夹中的node.exe就能够在任何盘符运行。

用cmd命令path查看环境变量path：如下图

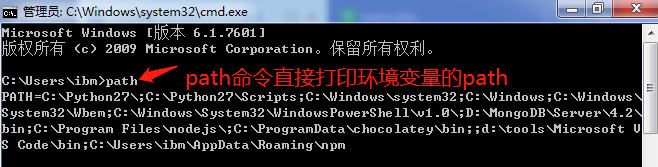


图4-15

编辑环境变量：控制面板->系统和安全->系统->高级系统设置->环境变量->系统变量->path:编辑path的值；

编辑path值的方式是：在原有值得后面添加我们要加入的文件夹路径，并用分号（;）跟之前的值隔离。

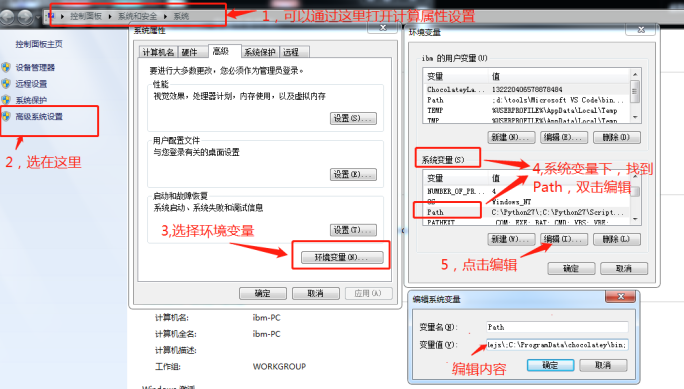


图4-16

4.5 Node.js运行和顶层对象

4.5.1 Node.js的交互环境

在cmd运行node命令，或者运行安装的node.js.exe程序进入Node.js的交互环境，在交互环境中可以执行js语言，如下图

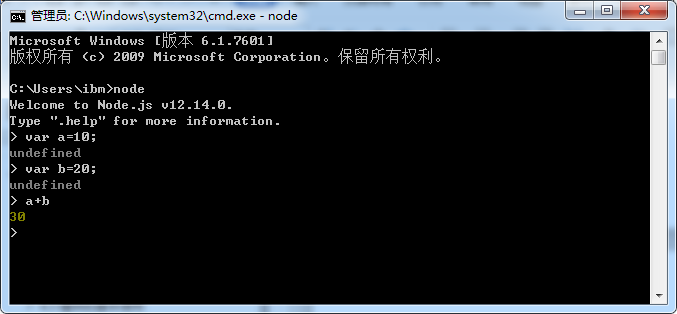


图4-17

### 4.5.2 Node.js运行js文件

例如在d盘下新建test.js文件，文件路径为：d:\test.js。写入简单代码console.log(‘hello Node.js’)。

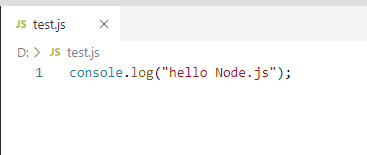


图4-18

在cmd中用node命令来运行该js文件，执行方式：

node 文件路径

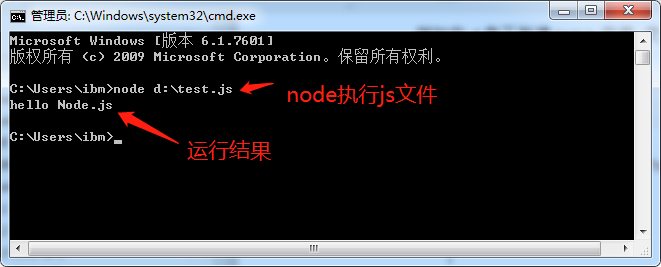


图4-19

也可以不使用完整的路径名，而是通过先进入案例文件夹，然后node 相对地址：

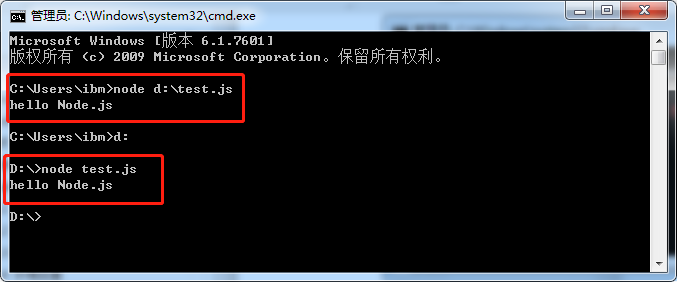


图4-20

### 4.5.3 顶层对象global

JavaScript 中有一个特殊的对象，称为顶层对象（Global Object），它及其所有属性都可以在程序的任何地方访问，即全局变量。

在浏览器 JavaScript 中，通常 window是顶层对象，而 Node.js 中的顶层对象是 global，所有ES5方式的全局变量都是 global 对象的属性。在ES6中，通过let、const等声明的全局变量不在global属性上

### 4.5.4 全局变量

Node.js中有几个新的默认全局变量。

**1、\_\_filename**

\_\_filename 表示当前正在执行的脚本的文件名。它将输出文件所在位置的绝对路径。 如果在模块中，返回的值是模块文件的路径。

**实例**

创建文件 main.js ，代码如下所示：

// 输出全局变量 \_\_filename 的值

console.log( \_\_filename );

执行 main.js 文件，代码如下所示:

$ node main.js

D:\nodejs\main.js

**2、\_\_dirname**

\_\_dirname 表示当前执行脚本所在的目录。

**实例**

创建文件 main.js ，代码如下所示：

// 输出全局变量 \_\_dirname 的值

console.log( \_\_dirname );

执行 main.js 文件，代码如下所示:

$ node main.js

D:\nodejs

4.6 Buffer

在Node、ES2015出现之前，前端工程师只需要进行一些简单的字符串或DOM操作就可以满足业务需要，所以对二进制数据是比较陌生。node出现以后，前端面对的技术场景发生了变化，可以深入到网络传输、文件操作、图片处理等领域，而这些操作都与二进制数据紧密相关。

在Node.js中，Buffer类是随Node内核一起发布的核心库。Buffer库为Node.js带来了一种存储原始数据的方法，可以让 Node.js处理二进制数据，每当需要在Node.js中处理I/O操作中移动的数据时，就有可能使用 Buffer库。原始数据存储在Buffer类的实例中。一个Buffer类似于一个整数数组，但它对应于V8堆内存之外的一块原始内存。

Buffer的结构和数组很像，操作的方法也和数组类似，但数组中不能存储二进制的文件，而buffer就是专门用来存储二进制数据，使用buffer不需要引入模块，直接使用即可。在buffer中存储的都是二进制数据，但是在显示时都是以16进制的形式显示，buffer中每一个元素的范围是从00 – ff即0 – 255。buffer中的一个元素，占用内存的一个字节，Buffer的大小一旦确定，则不能修改，Buffer实际上是对底层内存的直接操作。

### 4.6.1 二进制

二进制是计算技术中广泛采用的一种数制。二进制数据是用0和1两个数码来表示的数。它的基数为2，进位规则是“逢二进一”，借位规则是“借一当二”。

二进制数（binaries）是逢2进位的进位制，0、1是基本算符 ；计算机运算基础采用二进制。电脑的基础是二进制。在早期设计的常用的进制主要是十进制（因为我们有十个手指，所以十进制是比较合理的选择，用手指可以表示十个数字，0的概念直到很久以后才出现，所以是1－10而不是0－9）。电子计算机出现以后，使用电子管来表示十种状态过于复杂，所以所有的电子计算机中只有两种基本的状态，开和关。也就是说，电子管的两种状态决定了以电子管为基础的电子计算机采用二进制来表示数字和数据。常用的进制还有8进制和16进制，在电脑科学中，经常会用到16进制，而十进制的使用非常少，这是因为16进制和二进制有天然的联系：4个二进制位可以表示从0到15的数字，这刚好是1个16进制位可以表示的数据，也就是说，将二进制转换成16进制只要每4位进行转换就可以了。

二进制的“00101000”直接可以转换成16进制的“28”。字节是电脑中的基本存储单位，根据计算机字长的不同，字具有不同的位数，现代电脑的字长一般是32位的，也就是说，一个字的位数是32。字节是8位的数据单元，一个字节可以表示0－255的十进制数据。对于32位字长的现代电脑，一个字等于4个字节，对于早期的16位的电脑，一个字等于2个字节。

十进制转换为二进制技巧

只能举例了，文字说不清的，通常将一个十进制数的整数部分和小数部分分开处理。

1、整数的数制转换

——采用“基数除法”，具体步骤如下：

（1）将给定的十进制整数除以基数2，余数便是等值的二进制的最低位。

（2）将上一步的商再除以基数2，余数便是等值的二进制数的次低位。

（3）重复步骤2，直到最后所得的商等于0为止。各次除得的余数，便是二进制各位的数，最后一次的余数是最高位

二进制与八进制十六进制转换技巧

二进制从最低位开始每三位转换为十进制即为其对应八进制。

高位不足三位，补零。

同理二进制从最低位开始每四位转换为十进制即为其对应十六进制。

高位不足四位，补零。

例如 1001100₂ = 114₈ = 4C₁₆

4.6.2 Buffer初始化

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.from(string[, encoding])：返回一个被 string 的值初始化的新的 Buffer 实例

// 创建一个长度为 10、且用 0 填充的 Buffer。

const buf1 = Buffer.alloc(10);

// 创建一个长度为 10、且用 <Buffer 61 61 61 61 61 61 61 61 61 61> 填充的 Buffer。

const buf2 = Buffer.alloc(10,"a","utf-8")

// 创建一个包含 UTF-8 字节 <Buffer 75 6a 69 75 79 65 2e 63 6f 6d> 的 Buffer。

const buf3 = Buffer.from('ujiuye.com');

4.6.3 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。

const buf = Buffer.from('ujiuye.com', 'UTF-8');

console.log(buf.toString('hex'));// 输出 756a697579652e636f6d

console.log(buf.toString('base64'));// 输出 dWppdXllLmNvbQ==

console.log(buf.toString(utf8));// 输出 ujiuye.com

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。

● latin1 - 一种把 Buffer 编码成一字节编码的字符串的方式。

● binary - latin1 的别名。

● hex - 将每个字节编码为两个十六进制字符。

4.6.4 写入缓冲区

**语法：**

写入 Node 缓冲区的语法如下所示：

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

**参数**

参数描述如下：

● string - 写入缓冲区的字符串。

● offset - 缓冲区开始写入的索引值，默认为 0 。

● length - 写入的字节数，默认为 buffer.length

● encoding - 使用的编码。默认为 'utf8' 。

根据 encoding 的字符编码写入 string 到 buf 中的 offset 位置。 length 参数是写入的字节数。 如果 buf 没有足够的空间保存整个字符串，则只会写入 string 的一部分。超出buf长度的部分解码的字符不会被写入。

**返回值**

返回实际写入的大小。如果 buffer 空间不足， 则只会写入部分字符串。

**实例**

var buf = Buffer.alloc(256);

var len = buf.write("www.offcn.com");

console.log("写入字节数 : "+ len); //写入字节数 : 13

执行以上代码，输出结果为：

$node main.js写入字节数 : 13

4.6.5 从缓冲区读取数据

**语法**

读取 Node 缓冲区数据的语法如下所示：

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

**参数**

参数描述如下：

● encoding - 使用的编码。默认为 'utf8' 。

● start - 指定开始读取的索引位置，默认为 0。

● end - 结束位置，默认为缓冲区的末尾。

**返回值**

解码缓冲区数据并使用指定的编码返回字符串。

**实例**

buf = Buffer.alloc(26);

for (var i = 0 ; i < 26 ; i++) {

buf[i] = i + 97;

}

console.log( buf.toString('ascii')); // 输出: abcdefghijklmnopqrstuvwxyz

console.log( buf.toString('ascii',0,5)); // 输出: abcde

console.log( buf.toString('utf8',0,5)); // 输出: abcde

console.log( buf.toString(undefined,0,5)); // 使用 'utf8' 编码, 并输出: abcde

4.6.6 将Buffer转换为JSON对象

**语法**

将 Node Buffer 转换为 JSON 对象的函数语法格式如下：

buf.toJSON()

当字符串化一个 Buffer 实例时，JSON.stringify() 会隐式地调用该 toJSON()。

**返回值**

返回 JSON 对象。

**实例**

const buf = Buffer.from("12345");

const json = buf.toJSON()

console.log(json);// 输出: { type: 'Buffer', data: [ 49, 50, 51, 52, 53 ] }

4.6.7 缓冲区合并

**语法**

Node.js 缓冲区合并的语法如下所示：

Buffer.concat(list[, totalLength])

**参数**

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。

**返回值**

返回一个有多个成员Buffer合并而成的新 Buffer 对象。

**实例**

var buffer1 = Buffer.from(('中公教育'));

var buffer2 = Buffer.from(('www.offcn.com'));

var buffer3 = Buffer.concat([buffer1,buffer2]);

console.log("buffer3 内容: " + buffer3.toString());

执行以上代码，输出结果为：

buffer3 内容: 中公教育www.offcn.com

4.6.8 拷贝缓冲区

**语法**

Node 缓冲区拷贝语法如下所示：

buf.copy(targetBuffer[, targetStart[, sourceStart[, sourceEnd]]])

**参数**

参数描述如下：

● targetBuffer - 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

**返回值**

没有返回值。

**实例**

var buf1 = Buffer.from('abcdefghijkl');

var buf2 = Buffer.from('offcn');

//将 buf2 插入到 buf1 指定位置上

buf2.copy(buf1, 2);

console.log(buf1.toString());

执行以上代码，输出结果为：

aboffcnhijkl

4.6.9 缓冲区裁剪

Node 缓冲区裁剪语法如下所示：

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

**参数**

参数描述如下：

● start - 数字, 可选, 默认: 0

● end - 数字, 可选, 默认: buffer.length

**返回值**

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。

**实例**

var buffer1 = Buffer.from('offcn');// 剪切缓冲区

var buffer2 = buffer1.slice(0,2);

console.log("buffer2 content: " + buffer2.toString());

执行以上代码，输出结果为：

buffer2 content: of

4.6.10 缓冲区长度

**语法**

Node 缓冲区长度计算语法如下所示：

buf.length;

**返回值**

返回 Buffer 对象所占据的内存长度。

**实例**

var buffer = Buffer.from('www.offcn.com');// 缓冲区长度

console.log("buffer length: " + buffer.length);

执行以上代码，输出结果为：

buffer length: 13

4.7 模块系统

4.7.1 模块概述

在计算机程序的开发过程中，随着程序代码越写越多，在一个文件里代码就会越来越长，越来越不容易维护。

为了编写可维护的代码，我们把很多函数分组，分别放到不同的文件里，这样，每个文件包含的代码就相对较少，很多编程语言都采用这种组织代码的方式。在Node.js中，一个js文件就称之为一个模块（Module）。

使用模块有什么好处？

最大的好处是大大提高了代码的可维护性。其次，编写代码不必从零开始。当一个模块编写完毕，就可以被其他地方引用。我们在编写程序的时候，也经常引用其他模块，包括Node.js内置的模块和来自第三方的模块。

使用模块还可以避免函数名和变量名冲突。相同名字的函数和变量完全可以分别存在不同的模块中，因此，我们自己在编写模块时，不必考虑名字会与其他模块冲突。但是也要注意，尽量不要与内置函数名字冲突。

4.7.2 模块加载机制与Commonjs

ES5中没有原生支持模块化，我们只能通过script标签引入js文件来实现模块化。在node中为了对模块管理，引入了CommonJS规范。

Node.js借鉴CommonJS的Modules规范实现了一套模块系统，所以先来看看CommonJS的模块规范。CommonJS对模块的定义十分简单，主要分为模块引用、模块定义和模块标识3个部分。

4.7.2.1 模块引用

模块引用的示例代码如下：

var fs = require(‘fs’);

在CommonJS规范中，存在require()方法，这个方法接受模块标识，以此引入一个模块的API到当前上下文中。

4.7.2.2 模块定义

在模块中，上下文提供require()方法来引入外部模块。对应引入的功能，上下文提供了exports对象用于导出当前模块的方法或者变量，并且它是唯一导出的出口。在模块中，还存在一个module对象，它代表模块自身，而exports是module的属性。在Node中，一个文件就是一个模块，将方法挂载在exports对象上作为属性即可定义导出的方式，语法：

使用 exports

exports.属性 = 属性值;

exports.方法 = 函数;

使用module.exports

module.exports.属性 = 属性值;

module.exports.方法 = 函数;

module.exports = {};

**● 模块化**

在Node环境中，每一个js文件中的js代码都是独立运行在一个函数中，并且不是全局作用域，所以一个模块的中的变量和函数在其他模块中无法访问。

**● 采用commonJS规范**

我们可以通过 exports 来向外部暴露变量和方法，只需要将需要暴露（挂载）给外部的变量或方法设置为exports的属性即可。

**● 暴露模块语法**

//向外部暴露属性或方法

exports.a = "我是a";

exports.b = "我是b";

exports.fn = function () {

console.log('fn')

};

**● exports与module.exports**

module.exports是真实存在的东西。exports只是module.exports的辅助方法。你的模块最终返回module.exports给调用者，而不是exports。exports所做的事情是收集属性，如果module.exports当前没有任何属性的话，exports会把这些属性赋予module.exports。如果module.exports已经存在一些属性的话，那么exports中所用的东西都会被忽略。

新建foo.js

module.exports = 'ujiuye';

exports.name = function() {

console.log('My name is ujiuye');

};

然后把下面的内容放到另一个文件中，执行它：

var foo = require('./foo.js');

foo.name(); // TypeError:

foo模块完全忽略了exports.name，然后返回了一个字符串'ujiuye'。通过上面的例子，充分说明模块不一定非得是模块实例（module instances）。模块可以是任何合法的JavaScript对象 - boolean，number，date，JSON， string，function，array和其他。模块可以是任何你赋予module.exports的值。如果你没有明确的给module.exports设置任何值，那么exports中的属性会被赋给module.exports中，然后并返回它。

**● 暴露一个类**

module.exports = function(name, age) {

this.name = name;

this.age = age;

this.about = function() {

console.log(' 姓名'+this.name +' 年龄 '+ this.age);

};

};

使用

var Person = require('./ Person.js');

var r = new Person ('优就业', 18);

r.about(); // 姓名优就业年龄18

**● 暴露一个数组**

module.exports = ['web大前端', 'Java开发', 'UI设计'];

使用

var course = require('./course.js');

console.log('我们的课程为' + course [0]); //我们的课程为web大前端

4.7.2.3 模块标识

模块标识其实就是传递给require()方法的参数，它必须是符合小驼峰命名的字符串，或者以.、..开头的相对路径，或者绝对路径。它可以没有文件名后缀.js。模块的定义十分简单，接口也十分简洁。它的意义在于将类聚的方法和变量等限定在私有的作用域中。每个模块具有独立的空间，它们互不干扰，在引用时也显得干净利落。

模块的标识就是模块的名字或路径，我们node通过模块的标识来寻找模块的。对于核心模块（npm中下载的模块），直接使用模块的名字对其进行引入。

var http = require("http");

var fs = require("fs");

对于自定义的文件模块，需要通过文件的路径来对模块进行引入，路径可以是绝对路径，如果是相对路径必须以./或 ../开头。

var router = require("./router");

4.7.3 模块的实现

Node在实现中并非完全按照规范实现，而是对模块规范进行了一定的取舍，同时也增加了少许自身需要的特性。尽管规范中exports、require和module听起来十分简单，但是Node在实现它们的过程中究竟经历了什么，这个过程需要知晓。

在Node中引入模块，需要经历如下主要3个步骤。

● 路径分析

● 文件定位

● 编译执行

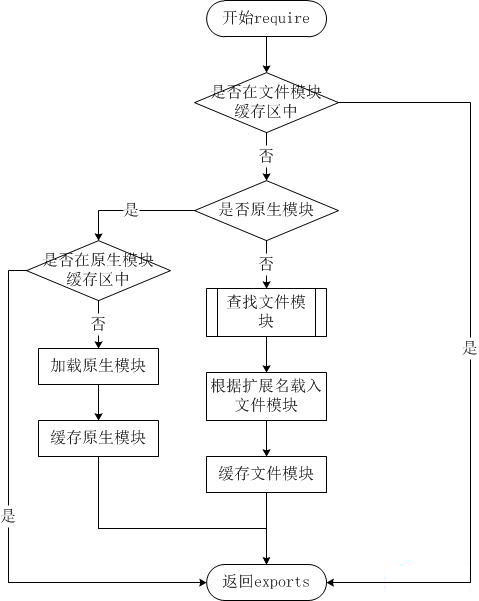


图4-21

在Node中，模块分为两类：一类是Node提供的模块，称为核心模块；另一类是用户编写的模块，称为文件模块。

4.7.3.1 核心模块

核心模块部分在Node源代码的编译过程中，编译进了二进制执行文件。在Node进程启动时，部分核心模块就被直接加载进内存中，所以这部分核心模块引入时，文件定位和编译执行这两个步骤可以省略掉，并且在路径分析中优先判断，所以它的加载速度是最快的。

4.7.3.2 文件模块

文件模块则是在运行时动态加载，需要完整的路径分析、文件定位、编译执行过程，速度比核心模块慢。

4.7.4 路径分析和文件定位

因为标识符有几种形式，对于不同的标识符，模块的查找和定位有不同程度上的差异。

4.7.4.1 核心模块

核心模块的加载仅次于缓冲加载。

4.7.4.2 路径形式的文件模块

首先将其转换为真实的路径，然后以真实路径作为索引，将其编译后放进缓冲中，等待调用。由于是通过确切的文件地址找到的，所以需要花费一定的时间，

4.7.4.3 自定义模块

这是非核心模块，也不是路径形式的标识，它是一种特殊的文件模块，可能是一个文件或者是一个包。这种查找是最费时间的。首先需要知道一下/模块路径/的概念/。

比如你要加载一个包，这个包放在了node\_modules文件夹下，你要引入的话可以不以路径的形式写，可以是只写名称。（也就是引入一个自己npm的包）

console.log(module.paths);

// 会得到下面的数组

[ 'D:\\myitem\\node\\node\\module\\node\_modules',

'D:\\ myitem \\node\\node\\node\_modules',

'D:\\ myitem \\node\\node\_modules',

'D:\\ myitem \\node\_modules',

'D:\\node\_modules' ]

这就是模块路径了，查找机制是： 首先在当前路径下找是否有node\_modules文件夹下的该包，如果没有就查找上一层目录，依次类推，直到根目录下的node\_modules，如果依旧没有找到，那么就报错了。显然这种情况就是导致它速度较慢的主要原因了，(查找的路径越深就越慢)。

**注意：**

node 的模块引入的时候是可以不写扩展名的，node会按照js、json、node的顺序来分析。

另外，如果没有找到对应的文件，确实找到了一个目录，那么将会将其当作是一个包来处理了。

如何在这个包下找到我们需要引入的入口文件呢？

●首先找是否含有package.json（后续会讲），如果有，则分析它的main属性，找到main属性对应的那个文件。

●如果没有package.json或者是main解析失败了，那么就找文件名为index的文件，依次从index.js、index.json、index.node查找。

●如果在该目录下依旧没有找到，那么就查找写一个匹配的目录，如果仍然没有找到，那么就报错了。

4.8 事件订阅机制EventEmitter

大多数 Node.js 核心 API 构建于惯用的异步事件驱动架构，其中某些类型的对象（又称触发器，Emitter）会触发命名事件来调用函数（又称监听器，Listener）。例如，net.Server 会在每次有新连接时触发事件，fs.ReadStream 会在打开文件时触发事件，stream会在数据可读时触发事件。

所有能触发事件的对象都是 EventEmitter 类的实例。这些对象有一个 eventEmitter.on() 函数，用于将一个或多个函数绑定到命名事件上。事件的命名通常是驼峰式的字符串。当EventEmitter对象触发一个事件时，所有绑定在该事件上的函数都会被同步地调用。

4.8.1 events 模块

events 模块只提供了一个对象：events.EventEmitter。EventEmitter 的核心就是事件触发与事件监听器功能的封装。我们可以通过require("events");来访问该模块。示例如下：

var EventEmitter = require('events').EventEmitter;

var event = new EventEmitter();

event.on('some\_event', function() {

console.log('some\_event 事件触发');

//当监听器函数被调用时，this 关键词会被指向监听器所绑定的 EventEmitter 实例。

//console.log( this )

//true

console.log( this === event )

});

setTimeout(function() {

event.emit('some\_event');

}, 1000);

运行这段代码，1 秒后控制台输出了 'some\_event 事件触发'。其原理是 event 对象注册了事件 some\_event 的一个监听器，然后我们通过 setTimeout 在 1000 毫秒以后向 event 对象发送事件 some\_event，此时会调用some\_event 的监听器。

4.8.2 回调函数及参数

EventEmitter 的每个事件由一个事件名和若干个参数组成，事件名是一个字符串，通常表达一定的语义。对于每个事件，EventEmitter 支持 若干个事件监听器。

当事件触发时，注册到这个事件的事件监听器被依次调用，事件参数作为回调函数参数传递。

var events = require('events');

var emitter = new events.EventEmitter();

emitter.on('someEvent', function(arg1, arg2) {

console.log('listener1', arg1, arg2);

});

emitter.on('someEvent', function(arg1, arg2) {

console.log('listener2', arg1, arg2);

});

emitter.emit('someEvent', 'arg1 参数', 'arg2 参数');

4.8.3 EventEmitter的属性方法

提供了多个属性，如 on 和 emit。on 函数用于绑定事件函数，emit 属性用于触发一个事件。接下来我们来具体看下EventEmitter的属性介绍。

**方法**

|  |  |
| --- | --- |
| 序号 | 方法 & 描述 |
| 1 | addListener(event, listener)  为指定事件添加一个监听器到监听器数组的尾部。 |
| 2 | on(event, listener)  为指定事件注册一个监听器，接受一个字符串 event 和一个回调函数。  server.on('connection', function (stream) {  console.log('someone connected!');}); |
| 3 | once(event, listener) 为指定事件注册一个单次监听器，即 监听器最多只会触发一次，触发后立刻解除该监听器。  server.once('connection', function (stream) {  console.log('Ah, we have our first user!');}); |
| 4 | removeListener(event, listener)  移除指定事件的某个监听器，监听器必须是该事件已经注册过的监听器。  它接受两个参数，第一个是事件名称，第二个是回调函数名称。  var callback = function(stream) {  console.log('someone connected!');};  server.on('connection', callback);// ...  server.removeListener('connection', callback); |
| 5 | removeAllListeners([event]) 移除所有事件的所有监听器， 如果指定事件，则移除指定事件的所有监听器。 |
| 6 | setMaxListeners(n) 默认情况下， EventEmitters 如果你添加的监听器超过 10 个就会输出警告信息。 setMaxListeners 函数用于提高监听器的默认限制的数量。 |
| 7 | listeners(event) 返回指定事件的监听器数组。 |
| 8 | emit(event, [arg1], [arg2], [...]) 按监听器的顺序执行执行每个监听器，如果事件有注册监听返回 true，否则返回 false。 |

4.9 包与NPM

4.9.1 包的定义

CommonJS的包规范允许我们将一组相关的模块组合到一起，形成一组完整的工具。CommonJS的包规范由包结构和包描述文件两个部分组成。

**● 包结构**

用于组织包中的各种文件

**● 包描述文件**

描述包的相关信息，以供外部读取分析

但是，完全符合CommonJS规范的包目录应该包含如下这些文件。

● package.json：包描述文件。

● bin：用于存放可执行二进制文件的目录。

● lib：用于存放JavaScript代码的目录。

● doc：用于存放文档的目录。

● test：用于存放单元测试用例的代码。

4.9.2 包描述文件

包描述文件用于表达非代码相关的信息，它是一个JSON格式的文件——package.json，位于包的根目录下，是包的重要组成部分。而NPM的所有行为都与包描述文件的字段息息相关。

在cmd中，使用npm init可以初始化一个package.json文件，用回答问题的方式生成一个新的package.json文件。

{

// 项目的名称

"name": "package",

// 项目的版本号

"version": "1.0.0",

// 项目的描述

"description": "a good demo",

// 项目入口文件

"main": "app.js",

// 项目依赖的模块

"dependencies": {

// 模块名称和版本号，“^2.0.0”表示固定2这个版本号，就是别人在安装你的依赖的时候如果模块版本发生变化，比如变为“2.1.1”就会更新版本，如果变成“3.0.0”则不更新，你也可以固定所有的（^2.^0.^0）

"time-stamp": "^2.0.0"

},

// 项目关键词

"keywords": [

"good"

],

// 作者

"author": "ujiuye",

// 模块协议，默认就可以

"license": "ISC"

// 描述和关键词要写全，这是方便别人搜索的时候可以据此搜到并引用你的模块，上述所有的内容填写都要是英文

}

4.9.3 NPM

在上一节，我们学习了模块的定义。模块就是一些功能的封装，所以一些成熟的、经常使用的功能，都有人封装成为了模块。并且放到了社区中，供人免费下载。

这个伟大的社区，叫做npm。也是一个工具名字node package management

网址为：<https://www.npmjs.com/>

为啥我们需要一个包管理工具呢？因为我们在Node.js上开发时，会用到很多别人写的JavaScript代码。如果我们要使用别人写的某个包，每次都根据名称搜索一下官方网站，下载代码，解压，再使用，非常繁琐。于是一个集中管理的工具应运而生：大家都把自己开发的模块打包后放到npm官网上，如果要使用，直接通过npm安装就可以直接用，不用管代码存在哪，应该从哪下载。

更重要的是，如果我们要使用模块A，而模块A又依赖于模块B，模块B又依赖于模块X和模块Y，npm可以根据依赖关系，把所有依赖的包都下载下来并管理起来。否则，靠我们自己手动管理，肯定既麻烦又容易出错。

同时NPM还随同Node.JS一起安装的包管理工具，能解决Node.JS代码部署上的很多问题，常见的使用场景有以下几种：

允许用户从NPM服务器下载别人编写的第三方包到本地使用。

允许用户从NPM服务器下载并安装别人编写的命令行程序到本地使用。

允许用户将自己编写的包或命令行程序上传到NPM服务器供别人使用。

由于新版的nodejs已经集成了npm，所以之前npm也一并安装好了。同样可以通过输入 "npm -v" 来测试是否成功安装。命令如下，出现版本提示表示安装成功:

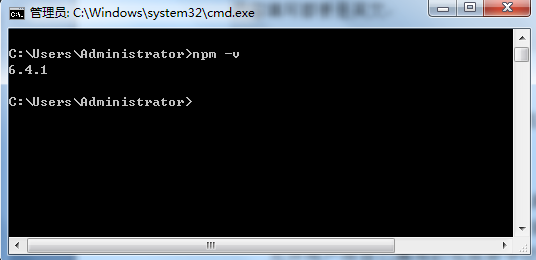


图4-22

4.9.4 npm install命令

如果我们需要使用一个模块，直接去社区搜索，点进去，看API即可。

如果要配置一个模块，那么直接在cmd中使用：

npm install <packageName>

就可以安装。 模块名字全球唯一。

安装的时候，要注意，命令提示符的所在位置。

下面我们来实际配置一个模块，如我们的需求是生成一个日期格式（20171109095211）：

第一步：在社区中搜索和日期或时间戳相关的信息，这里搜索的是date；

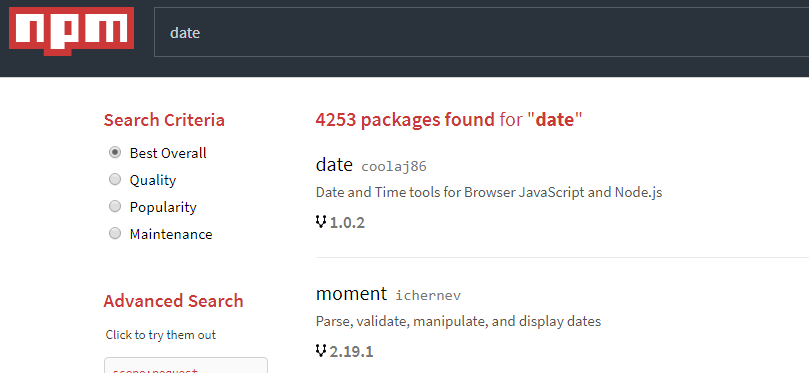


图4-23



图4-24

我们找到使用一个时间戳的模块，点击进去看API

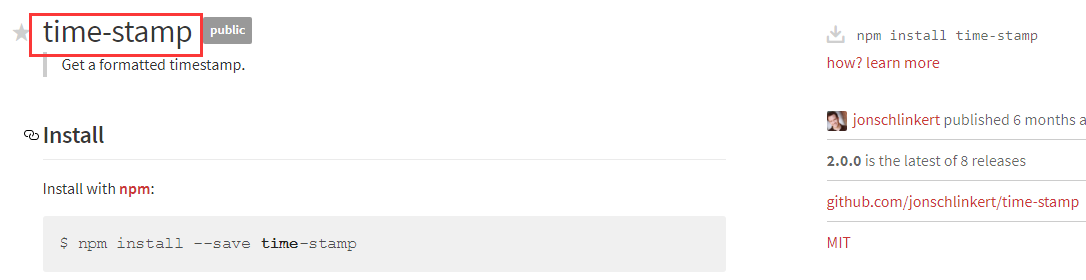


图4-25

第二步：我们再cmd中进入到我们项目的目录，使用：

npm install time-stamp

进行安装

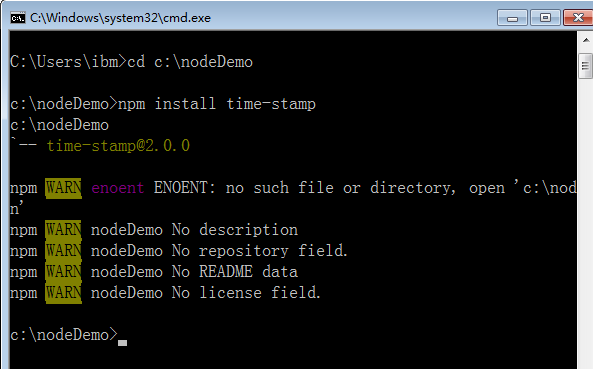


图4-26

安装结束后，我们看项目文件node\_modules文件下多了一个文件夹time-stamp



图4-27

第三步：使用这个包

// 引入安装好的包

var timestamp = require("time-stamp");

// 根据模块的API实现我们想要的功能

console.log(timestamp('YYYYMMDDHHMMss'));

//返回对应的系统当前时间

我们可以看到我们想要的结果就被打印出来了。

1. 我们的依赖包，可能在随时更新，我们永远想保持更新，或者保持某一个版本；

2. 项目越来越大的时候，给别人看的时候，没有必要再次共享我们引用的第三方模块。

4.9.5 全局安装与本地安装

npm 的包安装分为本地安装（local）、全局安装（global）两种，从敲的命令行来看，差别只是有没有-g而已，比如

npm install express # 本地安装

npm install express -g # 全局安装

**本地安装**

1. 将安装包放在 ./node\_modules 下（运行 npm 命令时所在的目录），如果没有 node\_modules 目录，会在当前执行 npm 命令的目录下生成 node\_modules 目录。

2. 可以通过 require() 来引入本地安装的包。

**全局安装**

1. 将安装包放在 /usr/local 下或者你 node 的安装目录。

2. 可以直接在命令行里使用。

4.9.6 安装不同版本

install 命令总是安装模块的最新版本，如果要安装模块的特定版本，可以在模块名后面加上 @ 和版本号。

$ npm install sax@latest

$ npm install sax@0.1.1

$ npm install sax@">=0.1.0 <0.2.0"

install 命令可以使用不同参数，指定所安装的模块属于哪一种性质的依赖关系，即出现在 packages.json 文件的哪一项中。

–save：模块名将被添加到 dependencies，可以简化为参数 - S。

–save-dev：模块名将被添加到 devDependencies，可以简化为参数 - D。

$ npm install sax --save

$ npm install node-tap --save-dev

# 或者

$ npm install sax -S

$ npm install node-tap -D

4.9.7 dependencies 依赖

这个可以说是我们 npm 核心一项内容，依赖管理，这个对象里面的内容就是我们这个项目所依赖的 js 模块包。下面这段代码表示我们依赖了 markdown-it 这个包，版本是 ^8.1.0 ，代表最小依赖版本是 8.1.0 ，如果这个包有更新，那么当我们使用 npm install 命令的时候，npm 会帮我们下载最新的包。当别人引用我们这个包的时候，包内的依赖包也会被下载下来。

"dependencies": {

"markdown-it": "^8.1.0"

}

4.9.8 devDependencies 开发依赖

在我们开发的时候会用到的一些包，只是在开发环境中需要用到，但是在别人引用我们包的时候，不会用到这些内容，放在 devDependencies 的包，在别人引用的时候不会被 npm 下载。

"devDependencies": {

"autoprefixer": "^6.4.0",0",

"babel-preset-es2015": "^6.0.0",

"babel-preset-stage-2": "^6.0.0",

"babel-register": "^6.0.0",

"webpack": "^1.13.2",

"webpack-dev-middleware": "^1.8.3",

"webpack-hot-middleware": "^2.12.2",

"webpack-merge": "^0.14.1",

"highlightjs": "^9.8.0"

}

当你有了一个完整的 package.json 文件的时候，就可以让人一眼看出来，这个模块的基本信息，和这个模块所需要依赖的包。我们可以通过 npm install 就可以很方便的下载好这个模块所需要的包。

npm install 默认会安装 dependencies 字段和 devDependencies 字段中的所有模块，如果使用 --production 参数，可以只安装 dependencies 字段的模块。

$ npm install --production

# 或者

$ NODE\_ENV=production npm install

一旦安装了某个模块，就可以在代码中用 require 命令加载这个模块。

var backbone = require('backbone')

console.log(backbone.VERSION)

4.9.9 npm init

npm init 用来初始化生成一个新的 package.json 文件。它会向用户提问一系列问题，如果你觉得不用修改默认配置，一路回车就可以了。  
如果使用了 -f（代表 force）、-y（代表 yes），则跳过提问阶段，直接生成一个新的 package.json 文件。

$ npm init -y

4.9.10 npm set

npm set 用来设置环境变量

$ npm set init-author-name 'Your name'

$ npm set init-author-email 'Your email'

$ npm set init-author-url 'http://yourdomain.com'

$ npm set init-license 'MIT'

上面命令等于为 npm init 设置了默认值，以后执行 npm init 的时候，package.json 的作者姓名、邮件、主页、许可证字段就会自动写入预设的值。这些信息会存放在用户主目录的 ~/.npmrc 文件，使得用户不用每个项目都输入。如果某个项目有不同的设置，可以针对该项目运行 npm config。

4.9.11 npm info

npm info 命令可以查看每个模块的具体信息。比如，查看 underscore 模块的信息。

$ npm info underscore

上面命令返回一个 JavaScript 对象，包含了 underscore 模块的详细信息。这个对象的每个成员，都可以直接从 info 命令查询。

$ npm info underscore description

$ npm info underscore homepage

$ npm info underscore version

4.9.12 npm search

npm search 命令用于搜索 npm 仓库，它后面可以跟字符串，也可以跟正则表达式。

$ npm search <搜索词>

4.9.13 npm list

npm list 命令以树形结构列出当前项目安装的所有模块，以及它们依赖的模块。

$ npm list

# 加上 global 参数，会列出全局安装的模块

$ npm list -global

# npm list 命令也可以列出单个模块

$ npm list underscore

4.9.14 淘宝镜像cnpm

国内直接使用 npm 的官方镜像是非常慢的，这里推荐使用淘宝NPM镜像。淘宝 NPM 镜像是一个完整 npmjs.org 镜像，你可以用此代替官方版本(只读)，同步频率目前为 10分钟 一次以保证尽量与官方服务同步。

所以我们可以使用淘宝定制的 cnpm (gzip 压缩支持) 命令行工具代替默认的 npm:

**第1种方式：**

npm install -g cnpm --registry=https://registry.npm.taobao.org

此安装方式会安装在npm目录下。如图所示：

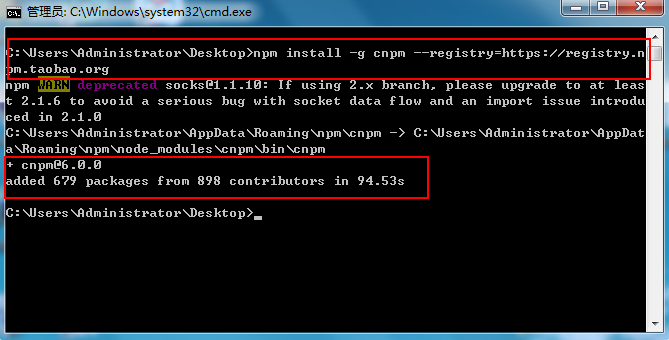


图4-28

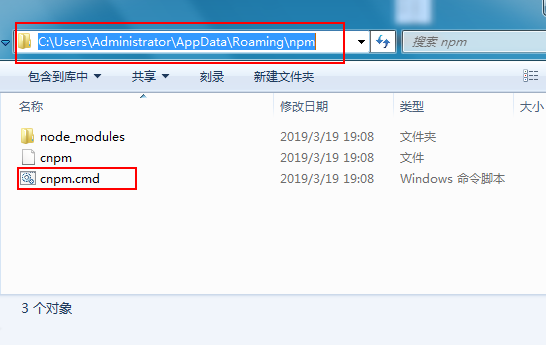


图4-29

这样就可以使用 cnpm 命令来安装模块了：

cnpm install [name]

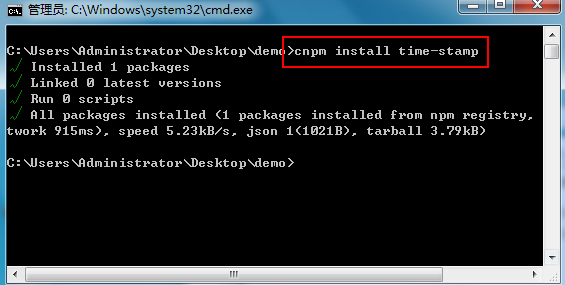


图4-30

**第2种方式：**

npm config set registry https://registry.npm.taobao.org

这种方式会在系统用户的目录下，创建一个.npmrc文件，并写入以下配置registry=https://registry.npm.taobao.org，打开如图：

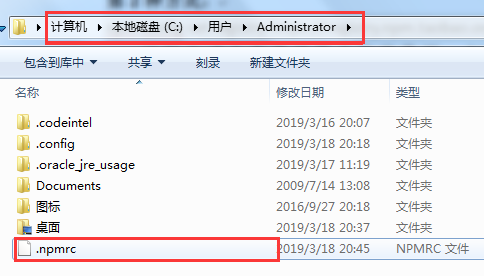


图4-31

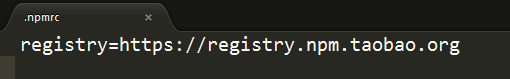


图4-32

4.10 高频面试

● Node.js特性

单线程、异步io，非阻塞，事件驱动

● 事件订阅与发布机制

var events = require('events');

var emitter = new events.EventEmitter();

emitter.on('someEvent', function(arg1, arg2) {

console.log('listener1', arg1, arg2);

});

emitter.on('someEvent', function(arg1, arg2) {

console.log('listener2', arg1, arg2);

});

emitter.emit('someEvent', 'arg1 参数', 'arg2 参数');

● NPM 的包依赖

● npm全局安装和局部安装的区别

第5讲 Node.js 文件系统

|  |  |  |  |
| --- | --- | --- | --- |
| 知识模块 | 课程内容 | 重点 | 课程目标 |
| Node.js  文件系统 | Node.js文件系统简介 | ★ | 掌握：Node.文件系统的基本操作方法，包含了文件、文件夹。  掌握：文件流操作方法。  理解：文件操作方法中的同步、异步的区别 |
| 文件基本操作方法 | ★ |
| 文件夹操作方法 | ★ |
| 文件流操作方法 | ★ |

5.1 简介及fs模块

文件系统（File System）简单来说就是通过Node来操作系统中的文件。使用文件系统，需要先引入fs模块，fs是核心模块，用commonjs规范直接导入后使用。

Node.js 导入文件系统模块(fs)语法如下所示：

var fs = require("fs");

所有文件系统操作都具有同步和异步的形式。

异步的形式总是将完成回调作为其最后一个参数。 传给完成回调的参数取决于具体方法，但第一个参数始终预留用于异常。 如果操作成功完成，则第一个参数将为 null 或 undefined。

const fs = require('fs');

fs.unlink('/tmp/hello', (err) => {

if (err) throw err;

console.log('已成功删除 /tmp/hello');

});

使用同步的操作发生的异常会立即抛出，可以使用 try…catch 处理，也可以允许冒泡。

const fs = require('fs');

try {

fs.unlinkSync('/tmp/hello');

console.log('已成功删除 /tmp/hello');

} catch (err) {

// 处理错误

}

使用异步的方法时无法保证顺序。 因此，以下的操作容易出错，因为 fs.stat() 操作可能在 fs.rename() 操作之前完成：

fs.rename('/tmp/hello', '/tmp/world', (err) => {

if (err) throw err;

console.log('重命名完成');

});

fs.stat('/tmp/world', (err, stats) => {

if (err) throw err;

console.log(`文件属性: ${JSON.stringify(stats)}`);

});

要正确地排序这些操作，则将 fs.stat() 调用移动到 fs.rename() 操作的回调中：

fs.rename('/tmp/hello', '/tmp/world', (err) => {

if (err) throw err;

fs.stat('/tmp/world', (err, stats) => {

if (err) throw err;

console.log(`文件属性: ${JSON.stringify(stats)}`);

});

});

在繁忙的进程中，强烈建议使用这些调用的异步版本。 同步的版本将阻塞整个进程，直到它们完成（停止所有连接）。

5.2 文件操作方法

5.2.1 文件路径

大多数 fs 操作接受的文件路径可以指定为字符串、Buffer、或使用 file: 协议的 URL 对象。字符串形式的路径被解析为标识绝对或相对文件名的 UTF-8 字符序列。**相对路径将相对于 process.cwd() 指定的当前工作目录进行解析。**

5.2.2 打开文件

**语法**

以下为在异步模式下打开文件的语法格式：

fs.open(path, flags[, mode], callback)

**参数**

参数使用说明如下：

● path - 文件的路径。

● flags - 文件打开的行为。具体值详见下文。

● mode - 设置文件模式(权限)，文件创建默认权限为 0666(可读，可写)。

● callback - 回调函数，带有两个参数如：callback(err, fd)。err为错误对象，fd为数字型文件描述符

当 flag 选项采用字符串时，可用以下标志：

● 'a' - 打开文件用于追加。如果文件不存在，则创建该文件。

● 'ax' - 与 'a' 相似，但如果路径已存在则失败。

● 'a+' - 打开文件用于读取和追加。如果文件不存在，则创建该文件。

● 'ax+' - 与 'a+' 相似，但如果路径已存在则失败。

● 'as' - 以同步模式打开文件用于追加。如果文件不存在，则创建该文件。

● 'as+' - 以同步模式打开文件用于读取和追加。如果文件不存在，则创建该文件。

● 'r' - 打开文件用于读取。如果文件不存在，则出现异常。

● 'r+' - 打开文件用于读取和写入。如果文件不存在，则出现异常。

● 'rs+' - 以同步模式打开文件用于读取和写入。指示操作系统绕过本地的文件系统缓存。

这对于在 NFS 挂载上打开文件时非常有用，因为它允许跳过可能过时的本地缓存。 它对 I/O 性能有非常实际的影响，因此除非需要，否则不建议使用此标志。这不会将 fs.open() 或 fsPromises.open() 转换为同步的阻塞调用。 如果需要同步的操作，则应使用 fs.openSync() 之类的。

● 'w' - 打开文件用于写入。如果文件不存在则创建文件，如果文件已存在则截断文件。

● 'wx' - 与 'w' 相似，但如果路径已存在则失败。

● 'w+' - 打开文件用于读取和写入。如果文件不存在则创建文件，如果文件已存在则截断文件。

● 'wx+' - 与 'w+' 相似，但如果路径已存在则失败。

**备注：创建文件只是代表创建文件本身，并不代表能够创建文件所在目录文件夹；**

**实例：**

接下来我们创建 file.js 文件，代码如下所示,当文件不存在时，会创建文件：

var fs = require("fs");

// 异步打开文件

console.log("准备打开文件！");

fs.open('input.txt', 'a', function(err, fd) {

if (err) {

return console.error(err);

}

console.log("文件打开成功！");

});

5.2.3 获取文件信息

**语法**

以下为通过异步模式获取文件信息的语法格式：

fs.stat(path, callback)

**参数**

参数使用说明如下：

● path - 文件路径。

● callback - 回调函数，带有两个参数如：(err, stats), stats 是 fs.Stats 对象。

fs.stat(path)执行后，会将stats类的实例返回给其回调函数。可以通过stats类中的提供方法判断文件的相关属性。例如判断是否为文件：

var fs = require('fs');

fs.stat('/offcn/ujiuye/fs.txt', function (err, stats) {

console.log(stats.isFile());

})

stats类中的方法有：

|  |  |
| --- | --- |
| 方法 | 描述 |
| stats.isFile() | 如果是文件返回 true，否则返回 false。 |
| stats.isDirectory() | 如果是目录返回 true，否则返回 false。 |
| stats.isBlockDevice() | 如果是块设备返回 true，否则返回 false。 |
| stats.isCharacterDevice() | 如果是字符设备返回 true，否则返回 false。 |
| stats.isSymbolicLink() | 如果是软链接返回 true，否则返回 false。 |
| stats.isFIFO() | 如果是FIFO，返回true，否则返回 false。FIFO是UNIX中的一种特殊类型的命令管道。 |
| stats.isSocket() | 如果是 Socket 返回 true，否则返回 false。 |

**实例**

接下来我们创建 file.js 文件，代码如下所示：

var fs = require("fs");

console.log("准备打开文件！");

fs.stat('ujiuye.txt', function (err, stats) {

if (err) {

return console.error(err);

}

console.log(stats);

console.log("读取文件信息成功！");

// 检测文件类型

console.log("是否为文件(isFile) ? " + stats.isFile());

console.log("是否为目录(isDirectory) ? " + stats.isDirectory());

});

5.2.4 写入文件

**语法**

以下为异步模式下写入文件的语法格式：

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

writeFile 直接打开文件默认是 w 模式，所以如果文件存在，该方法写入的内容会覆盖旧的文件内容。

**参数**

参数使用说明如下：

● file - 文件名或文件描述符。

● data - 要写入文件的数据，可以是 String(字符串) 或 Buffer(缓冲) 对象。

● options - 该参数是一个对象，包含 {encoding, mode, flag}。默认编码为 utf8, 模式为 0666 ， flag 为 'w'

● callback - 回调函数，回调函数只包含错误信息参数(err)，在写入失败时返回。

**实例**

接下来我们创建 file.js 文件，代码如下所示：

var fs = require("fs");

console.log("准备写入文件");

fs.writeFile('input.txt', '我是通 过fs.writeFile 写入文件的内容', function(err) {

if (err) {

return console.error(err);

}

console.log("数据写入成功！");

console.log("--------我是分割线-------------")

console.log("读取写入的数据！");

fs.readFile('input.txt', function (err, data) {

if (err) {

return console.error(err);

}

console.log("异步读取文件数据: " + data.toString());

});});

以上代码执行结果如下：

$ node file.js 准备写入文件数据写入成功！--------我是分割线-------------读取写入的数据！异步读取文件数据: 我是通 过

fs.writeFile 写入文件的内容

5.2.5 读取文件

**语法**

以下为异步模式下读取文件的语法格式：

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

该方法使用了文件描述符来读取文件。

**参数**

参数使用说明如下：

● fd - 通过 fs.open() 方法返回的文件描述符。

● buffer - 数据写入的缓冲区。

● offset - 缓冲区写入的写入偏移量。

● length - 要从文件中读取的字节数。

● position - 文件读取的起始位置，如果 position 的值为 null，则会从当前文件指针的位置读取。

● callback - 回调函数，有三个参数err, bytesRead, buffer，err 为错误信息， bytesRead 表示读取的字节数，buffer 为缓冲区对象。

**实例**

input.txt 文件内容为：

接下来我们创建 file.js 文件，代码如下所示：

var fs = require("fs");var buf = new Buffer.alloc(1024);

console.log("准备打开已存在的文件！");

fs.open('input.txt', 'r+', function(err, fd) {

if (err) {

return console.error(err);

}

console.log("文件打开成功！");

console.log("准备读取文件：");

fs.read(fd, buf, 0, buf.length, 0, function(err, bytes){

if (err){

console.log(err);

}

console.log(bytes + " 字节被读取");

// 仅输出读取的字节

if(bytes > 0){

console.log(buf.slice(0, bytes).toString());

}

});});

以上代码执行结果如下：

$ node file.js 准备打开已存在的文件！文件打开成功！准备读取文件：42 字节被读取中公教程官网地址：www.ujiuye.com

5.2.6 关闭文件

**语法**

以下为异步模式下关闭文件的语法格式：

fs.close(fd, callback)

该方法使用了文件描述符来读取文件。

**参数**

参数使用说明如下：

● fd - 通过 fs.open() 方法返回的文件描述符。

● callback - 回调函数，没有参数。

**实例**

input.txt 文件内容为：

中公教程官网地址：www.ujiuye.com

接下来我们创建 file.js 文件，代码如下所示：

var fs = require("fs");var buf = new Buffer.alloc(1024);

console.log("准备打开文件！");

fs.open('input.txt', 'r+', function(err, fd) {

if (err) {

return console.error(err);

}

console.log("文件打开成功！");

console.log("准备读取文件！");

fs.read(fd, buf, 0, buf.length, 0, function(err, bytes){

if (err){

console.log(err);

}

// 仅输出读取的字节

if(bytes > 0){

console.log(buf.slice(0, bytes).toString());

}

// 关闭文件

fs.close(fd, function(err){

if (err){

console.log(err);

}

console.log("文件关闭成功");

});

});});

以上代码执行结果如下：

$ node file.js 准备打开文件！文件打开成功！准备读取文件！中公教程官网地址：www.ujiuye.com文件关闭成功

5.2.7 截取文件

**语法**

以下为异步模式下截取文件的语法格式：

fs.ftruncate(fd, len, callback)

该方法使用了文件描述符来读取文件。

**参数**

参数使用说明如下：

● fd - 通过 fs.open() 方法返回的文件描述符。

● len - 文件内容截取的长度。

● callback - 回调函数，没有参数。

**实例**

input.txt 文件内容为：

site:www.ujiuye.com

接下来我们创建 file.js 文件，代码如下所示：

var fs = require("fs");var buf = new Buffer.alloc(1024);

console.log("准备打开文件！");

fs.open('input.txt', 'r+', function(err, fd) {

if (err) {

return console.error(err);

}

console.log("文件打开成功！");

console.log("截取10字节内的文件内容，超出部分将被去除。");

// 截取文件

fs.ftruncate(fd, 10, function(err){

if (err){

console.log(err);

}

console.log("文件截取成功。");

console.log("读取相同的文件");

fs.read(fd, buf, 0, buf.length, 0, function(err, bytes){

if (err){

console.log(err);

}

// 仅输出读取的字节

if(bytes > 0){

console.log(buf.slice(0, bytes).toString());

}

// 关闭文件

fs.close(fd, function(err){

if (err){

console.log(err);

}

console.log("文件关闭成功！");

});

});

});});

以上代码执行结果如下：

$ node file.js 准备打开文件！文件打开成功！截取10字节内的文件内容，超出部分将被去除。文件截取成功。读取相同的

文件

site:www.r文件关闭成功

5.2.8 删除文件

**语法**

以下为删除文件的语法格式：

fs.unlink(path, callback)

**参数**

参数使用说明如下：

● path - 文件路径。

● callback - 回调函数，没有参数。

**实例**

input.txt 文件内容为：

site:www.ujiuye.com

接下来我们创建 file.js 文件，代码如下所示：

var fs = require("fs");

console.log("准备删除文件！");

fs.unlink('input.txt', function(err) {

if (err) {

return console.error(err);

}

console.log("文件删除成功！");});

以上代码执行结果如下：

$ node file.js 准备删除文件！文件删除成功！

再去查看 input.txt 文件，发现已经不存在了。

5.3 文件夹操作方法

5.3.1 创建目录

**语法**

以下为创建目录的语法格式：

fs.mkdir(path[, options], callback)

**参数**

参数使用说明如下：

● path - 文件路径。

● options 参数可以是：

● recursive - 是否以递归的方式创建目录，默认为 false。

● mode - 设置目录权限，默认为 0777。

● callback - 回调函数，没有参数。

**实例**

接下来我们创建 file.js 文件，代码如下所示：

var fs = require("fs");// tmp 目录必须存在

console.log("创建目录 /tmp/test/");

fs.mkdir("/tmp/test/",function(err){

if (err) {

return console.error(err);

}

console.log("目录创建成功。");});

以上代码执行结果如下：

$ node file.js 创建目录 /tmp/test/目录创建成功。

可以添加 recursive: true 参数，不管创建的目录 /tmp 和 /tmp/a 是否存在：

fs.mkdir('/tmp/a/apple', { recursive: true }, (err) => {

if (err) throw err;});

5.3.2 读取目录

**语法**

以下为读取目录的语法格式：

fs.readdir(path, callback)

**参数**

参数使用说明如下：

● path - 文件路径。

● callback - 回调函数，回调函数带有两个参数err, files，err 为错误信息，files 为 目录下的文件数组列表。

**实例**

接下来我们创建 file.js 文件，代码如下所示：

var fs = require("fs");

console.log("查看 /tmp 目录");

fs.readdir("/tmp/",function(err, files){

if (err) {

return console.error(err);

}

files.forEach( function (file){

console.log( file );

});});

以上代码执行结果如下：

$ node file.js 查看 /tmp 目录

input.out

output.out

test

test.txt

5.3.3 删除目录

**语法**

以下为删除目录的语法格式：

fs.rmdir(path, callback)

**参数**

参数使用说明如下：

● path - 文件路径。

● callback - 回调函数，没有参数。

**实例**

接下来我们创建 file.js 文件，代码如下所示：

var fs = require("fs");// 执行前创建一个空的 /tmp/test 目录

console.log("准备删除目录 /tmp/test");

fs.rmdir("/tmp/test",function(err){

if (err) {

return console.error(err);

}

console.log("读取 /tmp 目录");

fs.readdir("/tmp/",function(err, files){

if (err) {

return console.error(err);

}

files.forEach( function (file){

console.log( file );

});

});});

以上代码执行结果如下：

$ node file.js 准备删除目录 /tmp/test读取 /tmp 目录……

5.4 文件流操作

**Node.js Stream(流)**

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）。

**Node.js，Stream 有四种流类型：**

● Readable - 可读操作。

● 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。

所有的 Stream 对象都是 EventEmitter 的实例。常用的事件有：

● data - 当有数据可读时触发。

● end - 没有更多的数据可读时触发。

● error - 在接收和写入过程中发生错误时触发。

● finish - 所有数据已被写入到底层系统时触发。

5.4.1 从流中读取数据

创建 input.txt 文件，内容如下：

中公教程官网地址：www.ujiuye.com

创建 main.js 文件, 代码如下：

var fs = require("fs");var data = '';

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

// 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");

以上代码执行结果如下：

程序执行完毕中公教程官网地址：www.ujiuye.com

5.4.2 写入流

创建main.js文件, 代码如下：

var fs = require("fs");var data = '中公教程官网地址：www.ujiuye.com';

// 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(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");

以上程序会将 data 变量的数据写入到 output.txt 文件中。代码执行结果如下：

$ node main.js 程序执行完毕写入完成。

查看 output.txt 文件的内容：

$ cat output.txt 中公教程官网地址：www.ujiuye.com

5.4.3 管道流

管道提供了一个输出流到输入流的机制。通常我们用于从一个流中获取数据并将数据传递到另外一个流中。

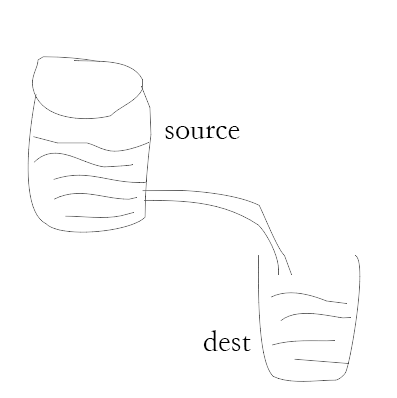


图5-1

如上面的图片所示，我们把文件比作装水的桶，而水就是文件里的内容，我们用一根管子(pipe)连接两个桶使得水从一个桶流入另一个桶，这样就慢慢的实现了大文件的复制过程。

以下实例我们通过读取一个文件内容并将内容写入到另外一个文件中。

设置 input.txt 文件内容如下：

中公教程官网地址：www.ujiuye.com管道流操作实例

创建 main.js 文件, 代码如下：

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");

代码执行结果如下：

$ node main.js 程序执行完毕

查看 output.txt 文件的内容：

$ cat output.txt 中公教程官网地址：www.ujiuye.com管道流操作实例

5.4.4 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。

接下来我们就是用管道和链式来压缩和解压文件。

创建 compress.js 文件, 代码如下：

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。");

代码执行结果如下：

$ node compress.js 文件压缩完成。

执行完以上操作后，我们可以看到当前目录下生成了 input.txt 的压缩文件 input.txt.gz。

接下来，让我们来解压该文件，创建 decompress.js 文件，代码如下：

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。");

代码执行结果如下：

$ node decompress.js 文件解压完成。

5.5 高频面试

第6讲 Node.js web模块

|  |  |  |  |
| --- | --- | --- | --- |
| 知识模块 | 课程内容 | 重点 | 课程目标 |
| Node.js的web模块 | Web服务原理简介 | ★ | 了解：web服务器的原理  掌握：HTTP协议  掌握：核心模块的基本使用  掌握：理解路由原理及设计路由 |
| HTTP协议 | ★ |
| 核心模块 | ★ |
| 路由的认识 | ★ |

6.1 什么是Web服务器？

Web服务器一般指网站服务器，是指驻留于因特网上某种类型计算机的程序，Web服务器的基本功能就是提供Web信息浏览服务。它只需支持HTTP协议、HTML文档格式及URL，与客户端的网络浏览器配合。

大多数 web 服务器都支持服务端的脚本语言（php、python、ruby）等，并通过脚本语言从数据库获取数据，将结果返回给客户端浏览器。

目前最主流的三个Web服务器是Apache、Nginx、IIS。

6.2 Web应用架构



图6-1

Client - 客户端，一般指浏览器，浏览器可以通过 HTTP 协议向服务器请求数据。

Server - 服务端，一般指 Web 服务器，可以接收客户端请求，并向客户端发送响应数据。

Business - 业务层， 通过 Web 服务器处理应用程序，如与数据库交互，逻辑运算，调用外部程序等。

Data - 数据层，一般由数据库组成。

6.3 使用Node创建Web服务器

Node.js 提供了 http 模块，http 模块主要用于搭建 HTTP 服务端和客户端，使用 HTTP 服务器或客户端功能必须调用 http 模块，代码如下：

var http = require('http');

以下是演示一个最基本的 HTTP 服务器架构(使用 8080 端口)，创建 server.js 文件，代码如下所示：

**实例**

var http = require('http');

var fs = require('fs');

var url = require('url');

// 创建服务器

http.createServer( function (request, response) {

// 解析请求，包括文件名

var pathname = url.parse(request.url).pathname;

// 输出请求的文件名

console.log("Request for " + pathname + " received.");

// 从文件系统中读取请求的文件内容

fs.readFile(pathname.substr(1), function (err, data) {

if (err) { console.log(err);

// HTTP 状态码: 404 : NOT FOUND

// Content Type: text/html

response.writeHead(404, {'Content-Type': 'text/html'});

}else{

// HTTP 状态码: 200 : OK // Content Type: text/html

response.writeHead(200, {'Content-Type': 'text/html'});

// 响应文件内容

response.write(data.toString());

}

// 发送响应数据

response.end();

});

}).listen(8080);

// 控制台会输出以下信息

console.log('Server running at http://127.0.0.1:8080/');

接下来我们在该目录下创建一个 index.html 文件，代码如下：

index.html 文件:

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8">

<title>中公教程(runoob.com)</title>

</head>

<body>

<h1>我的第一个标题</h1>

<p>我的第一个段落。</p>

</body>

</html>

执行 server.js 文件：

$ node server.js

Server running at http://127.0.0.1:8080/

接着我们在浏览器中打开地址：<http://127.0.0.1:8080/index.html>就能在浏览器中展示index.html页面。

执行 server.js 文件：

$ node server.js

Server running at http://127.0.0.1:8080/

接着我们在浏览器中打开地址：**http://127.0.0.1:8080/index.html**。

执行 server.js 的控制台输出信息如下：

Server running at <http://127.0.0.1:8080/>

Request for /index.html received. # 客户端请求信息

6.4 HTTP协议

6.4.1 HTTP协议概述

HTTP(Hyper Text Transfer Protocol)<超文本传输协议>的缩写.是用于从WWW服务器传输超文本到本地浏览器的传输协议.HTTP是一个应用层协议,由请求和响应构成,是一个标准的个客户端和服务器模型.它的主要特点如下:

1. 支持客户端/服务器模型

2. 简单快速

客户向服务器请求服务时，只需传送请求方法和路径。请求方法常用的有GET、HEAD、POST。每种方法规定了客户与服务器联系的类型不同。由于HTTP协议简单，使得HTTP服务器的程序规模小，因而通信速度很快

3.灵活

HTTP允许传输任意类型的数据对象.正在传输的类型由Content-Type加以标记.常见的Content-Type的取值如下:

4.无连接

无连接的含义是限制每次连接只处理一个请求.服务器处理完客户端的请求,然后响应,并收到应答之后,就断开连接.这种方式可以节省传输时间.

5.无状态

HTTP协议是无状态协议.无状态是指协议 对于事务处理没有记忆能力.这种方式的一个坏处就是,如果后续的处理需要用到之前的信息,则必须要重传,这样就导致了每次连接传输的数据量增大.好处就是,如果后续的连接不需要之前提供的信息,响应就会比较快.而为了解决HTTP的无状态特性,出现了Cookie和Session技术。

6.4.2 URL说明

HTTP协议是一个基于请求和应答模式的,存在于传输层之上的应用层协议,是一个无状态的协议,通常是基于TCP的连接方式.HTTP的URL是一种特殊类型的URI,包含了用于定位查找某个网络资源的路径,格式如下:  
 <http://host[':'port][abs_path]>

http表示通过http协议来定位网络资源;host表示合法的Internet主机域名或者IP地址;port指定一个端口,如果缺省,默认是80端口.abs\_path表示的是请求的资源的URL,如果什么都没写,则浏览器会帮我们加上/,作为初始的路径地址.

6.4.2 HTTP请求(Request)

HTTP请求信息由3部分组成：

● 请求方法URI协议/版本

● 请求头(Request Header)

● 请求正文

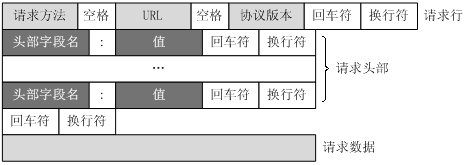


图6-2

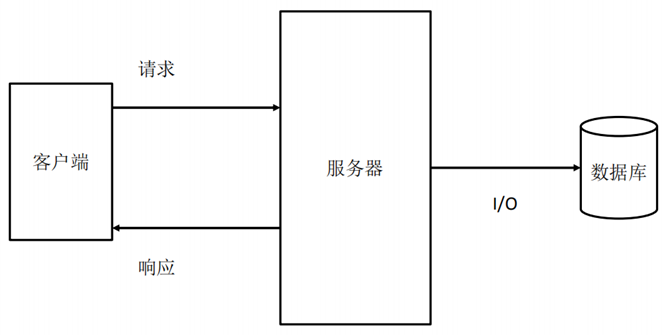


图6-3

6.4.3 HTTP相应(*Response*)

在接收和解释请求信息后，服务器返回一个 HTTP 响应信息。

HTTP 响应由三部分组成：

●状态行

常见状态代码、状态描述、说明：

200 OK // 客户端请求成功

400 Bad Request // 客户端请求有语法错误，不能被服务器所理解

401 Unauthorized // 请求未经授权，这个状态代码必须和WWW-Authenticate 报头域一起使用

403 Forbidden // 服务器收到请求，但是拒绝提供服务

404 Not Found // 请求资源不存在，eg：输入了错误的 URL

500 Internal Server Error // 服务器发生不可预期的错误

503 Server Unavailable // 服务器当前不能处理客户端的请求，一段时间后可能恢复正常

●消息报头：响应报头后述

●响应正文：服务器返回的资源内容

6.4.4 HTTP的工作原理

HTTP协议定义了web客户端如何从web服务器请求Web页面,以及服务器如何把Web页面传送给客户端.HTTP协议采用了请求/响应模型.客户端向服务器发送一个请求报文,请求报文包括请求的方法,url,协议版本,请求头部和请求数据.服务器以一个状态行作为响应,响应的内容包括协议的版本,成功或者错误代码,服务器信息,响应头部和响应数据

以下是HTTP协议的请求/响应步骤

1.客户端连接到we服务器

一个HTTP客户端,通常是浏览器,与Web服务器的HTTP端口(默认是80)建立一个TCP套接字连接.

2.发送HTTP请求

通过TCP套接字,客户端向Web服务器发送一个文本的请求报文,一个请求报文由请求行,请求头部,空行和请求体4个部分构成.

3.服务区接收解释请求并返回HTTP响应

Web解析请求,定位请求资源.服务器将资源复本写到TCP套接字,由客户端获取.一个响应由状态行,响应 头,空行和响应数据4部分组成.

4.释放连接TCP连接

若Connection模式为close,则服务器主动关闭TCP连接,客户端被动关闭TCP连接,释放TCP连接.若Connection为keepalive,则该连接会保持一段时间,该时间内可以持续使用该连接接收请求,做出响应

6.5 第三方模块

6.5.1 path模块

path 模块提供用于处理文件路径和目录路径的实用工具。path 模块的默认操作因 Node.js 应用程序运行所在的操作系统而异。具体来说，当在 Windows 操作系统上运行时，path 模块将假定正在使用 Windows 风格的路径。因此，使用 path.basename() 可能会在 POSIX 和 Windows 上产生不同的结果：

let path = require('path')

//在 POSIX 上:

path.basename('C:\\temp\\myfile.html');

// 返回: 'C:\\temp\\myfile.html'

//在 Windows 上:

path.basename('C:\\temp\\myfile.html');

// 返回: 'myfile.html'

//要在任何操作系统上使用 Windows 文件路径时获得一致的结果，则使用 path.win32：

//在 POSIX 和 Windows 上:

path.win32.basename('C:\\temp\\myfile.html');

// 返回: 'myfile.html'

//要在任何操作系统上使用 POSIX 文件路径时获得一致的结果，则使用 path.posix：

//在 POSIX 和 Windows 上:

path.posix.basename('/tmp/myfile.html');

// 返回: 'myfile.html'

Path提供的其它方法如下:

● path.basename(path[, ext])

返回 path 的最后一部分，类似于 Unix 的 basename 命令。尾部的目录分隔符将被忽略。

● path.delimiter

提供平台特定的路径定界符。

● path.dirname(path)

path.dirname() 方法返回 path 的目录名，类似于 Unix 的 dirname 命令。 尾部的目录分隔符将被忽略。

● path.extname(path)

path.extname() 方法返回 path 的扩展名，从最后一次出现 .（句点）字符到 path 最后一部分的字符串结束。

如果在 path 的最后一部分中没有 . ，或者如果 path 的基本名称（参阅 path.basename()）的第一个字符是 .，则返回空字符串。

● path.isAbsolute(path)

path.isAbsolute() 方法检测 path 是否为绝对路径。如果给定的 path 是零长度字符串，则返回false。

● path.join([...paths])

path.join() 方法使用平台特定的分隔符作为定界符将所有给定的 path 片段连接在一起，然后规范化生成的路径。

零长度的 path 片段会被忽略。 如果连接的路径字符串是零长度的字符串，则返回 '.'，表示当前工作目录。

● path.normalize(path)

path.normalize() 方法规范化给定的 path，解析 '..' 和 '.' 片段。

当找到多个连续的路径段分隔字符时（例如 POSIX 上的 /、Windows 上的 \ 或 /），

则它们将被替换为单个平台特定的路径段分隔符（POSIX 上的 /、Windows 上的 \）。 尾部的分隔符会保留。

如果 path 是零长度的字符串，则返回 '.'，表示当前工作目录。

● path.parse(path)

方法返回一个对象，其属性表示 path 的重要元素。 尾部的目录分隔符将被忽略。

6.5.2 url模块

URL 字符串是结构化的字符串，包含多个含义不同的组成部分。 解析字符串后返回的 URL 对象，每个属性对应字符串的各个组成部分。

url 模块提供了两套 API 来处理 URL：一个是旧版本遗留的 API，一个是实现了 WHATWG标准的新 API。

遗留的 API 还没有被废弃，保留是为了兼容已存在的应用程序。 新的应用程序应使用 WHATWG 的 API。

WHATWG 的 API 与遗留的 API 的区别如下。 在下图中，URL 'http://user:pass@sub.example.com:8080/p/a/t/h?query=string#hash'

上方的是遗留的 url.parse() 返回的对象的属性。 下方的则是 WHATWG 的 URL 对象的属性。

WHATWG 的 origin 属性包括 protocol 和 host，但不包括 username 或 password。

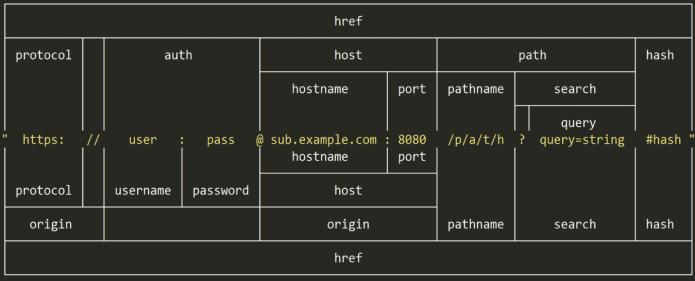


图6-4

示例：

let {URL} = require('url')

let urlString = "https://user:pass@sub.example.com:8080/p/a/t/h?query=string#hash"

let urlObj = new URL(urlString)

console.log( urlObj )

返回结果：

/\*

URL{

href:'https://user:pass@sub.example.com:8080/p/a/t/h?query=string#hash',

origin: 'https://sub.example.com:8080',

protocol: 'https:',

username: 'user',

password: 'pass',

host: 'sub.example.com:8080',

hostname: 'sub.example.com',

port: '8080',

pathname: '/p/a/t/h',

search: '?query=string',

searchParams: URLSearchParams { 'query' => 'string' },

hash: '#hash'

}

\*/

6.5.3 querystring

提供用于解析和格式化 URL 查询字符串的实用工具

querystring.parse(str)

返回的对象不是原型继承自 JavaScript Object。默认情况下，将假定查询字符串中的百分比编码字符使用 UTF-8 编码。

querystring.stringify(obj[, sep[, eq[, options]]])

通过迭代对象的自身属性从给定的 obj 生成 URL 查询字符串。

示例如下：

let querystring = require('querystring')

let obj = { foo: '#bar', baz: ['qux', '中文'] }

console.log('\*\*\*\*stringify\*\*\*\*')

console.log( querystring.stringify( obj ) )

//返回： foo=%23bar&baz=qux&baz=%E4%B8%AD%E6%96%87

let str = 'foo=%23bar&baz=qux&baz=%E4%B8%AD%E6%96%87'

console.log('\*\*\*\*parse\*\*\*\*')

console.log( querystring.parse( str ) )

//返回： { foo: '#bar', baz: ['qux', '中文'] }

6.6 get和post请求

在项目开发的过程中，我们的服务器需要跟用户的浏览器做数据交互，比如表单提交，ajax提交数据等等。这些有客户端向服务器端发起的请求一般都会涉及到get、post两种方式。

这一节，介绍在Node.js的后端如何区分get、post请求，并且如何获取这些请求参数和数据。

6.6.1 区分get、post请求

在Node.js搭建的后端中，当有客户端浏览请访问时，我们可以通过请求对象request的属性method判断访问类型。

6.6.2 获取get请求的内容

由于GET请求时，get的内容是直接被嵌入在URL路径中的，具体内容在URL中？后面的部分，因此你可以手动解析?后面的内容作为GET请求的参数。在node.js 中 url 模块中的 parse 函数提供了这个功能。

const http = require("http");

const url = require("url");

var server = http.createServer(function (request, response) {

    console.log(`${request.method}请求`);

    response.writeHead(200, { "Content-Type": "text/plain;charset=utf-8" });

    var params = url.parse(request.url, true).query;

    response.write(`

        网站名：${params.name};

        网站URL:${params.url}

        `);

    response.end();

}).listen(3000, () => {

    console.log("服务已启动，端口:3000");

})

在浏览器中访问 http://localhost:3000?name=优就业&url=www.ujiuye.com 然后查看返回结果:

网站名：优就业;

网站URL:www.ujiuye.com

6.6.3 获取 POST 请求内容

POST 请求的内容全部的都在请求体中，http.ServerRequest 并没有一个属性内容为请求体，原因是等待请求体传输可能是一件耗时的工作。比如上传文件，而很多时候我们可能并不需要理会请求体的内容，恶意的POST请求会大大消耗服务器的资源，所以 node.js 默认是不会解析请求体的，当你需要的时候，需要手动来做。

实例:

const http = require("http");

const querystring = require("querystring");

const fs = require("fs");

var server = http.createServer(function (request, response) {

    if (request.method == "GET") {

        fs.readFile("./test.html", (err, data) => {

            response.writeHead(200, { "Content-Type": "text/html;charset=utf-8" });

            response.write(data);

            response.end();

        })

    } else if (request.method == "POST") {

        var bodyData = "";

        request.on("data", (chunk) => {

            bodyData += chunk;

        })

        request.on("end", () => {

            bodyData = querystring.parse(bodyData);

            response.writeHead(200, { "Content-Type": "text/plain;charset=utf-8" })

            response.write(`

                网站名称：${bodyData.name}

                网站URL:${bodyData.url}

            `);

            response.end();

        })

    } else {

        console.log(`${request.method}请求`);

response.end();

    }

}).listen(3000, () => {

    console.log("服务已启动，端口:3000");

})

其中test.html代码如下：

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <meta http-equiv="X-UA-Compatible" content="ie=edge">

    <title>Document</title>

</head>

<body>

    <form action="" method="POST">

        <label for="">网站名称:</label><input type="text" name="name"><br/>

        <label for="">网站URL:</label><input type="text" name="url"><br/>

        <input type="submit" value="提交">

    </form>

</body>

</html>

解析代码：当一个http请求来的时候，如果是get请求，读取页面test.html返回给客户端，浏览器加载完test.html页面之后，如下图：

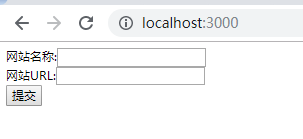


图6-5

在页面中填写表单，并点击提交按钮。表单是post方式提交都nodejs后台，这里的提交也是一次对后台的请求。此时nodejs后台解析post的数据并将其返回客户端；如下

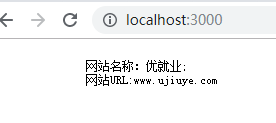


图6-6

6.7 路由与静态资源

6.7.1 认识路由

6.7.1.1 url

url是统一资源定位符，对可以从互联网上得到的资源的位置和访问方法的一种简洁的表示，是互联网上标准资源的地址。互联网上的每个文件都有一个唯一的URL，它包含的信息指出文件的位置以及浏览器应该怎么处理它。

基本URL包含模式（或称协议）、服务器名称（或IP地址）、路径和文件名，如“协议：//授权/路径?查询”。完整的、带有授权部分的普通统一资源标志符语法看上去如下：

协议：//用户名:密码@子域名.域名.顶级域名:端口号/目录/文件名.文件后缀?参数=值#标志。

例如：一个网址为

http://www.a.com/aa

定义：/aa = bb/cc/dd

那么：http://www.a.com/aa就是一个url，那么我们可以得出：网址=url

而当我访问这个url之后，系统读取到了/aa，那么url实际上访问的是http://www.a.com/bb/cc/dd

那么/aa = bb/cc/dd就是一种路由规定，规定了/aa等于bb/cc/dd，当用户输入/aa的时候，系统就会通过/aa获取到bb/cc/dd，进行映射。

为什么我们要设置路由规定？因为如果我们访问的路径过于长的话，那么看起来是不是非常囊肿？

6.7.1.2 定义路由

定义路由，就是假设按照上面的网址，那么访问这样网站的时候，也是可以正确访问。那么就是定义路由成功了。定义路由的规则也可以自己定。

思考，使用Node.js定义以下路由：

127.0.0.1:3000/student/00001

127.0.0.1:3000/teacher/000001

输入url展示学生及老师的信息。

var http = require("http");

var server = http.createServer(function(req,res){

var reqUrl = req.url;

res.writeHead(200,{"Content-type":"text/html;charset=UTF-8"});

if(reqUrl.substr(0,9)=="/student/"){

var studentid = reqUrl.substr(9);

if(/^\d{5}$/.test(studentid)){

res.end("你要查询学生信息，id为："+studentid);

}else{

res.end("学生学号位数不对");

}

}else if(reqUrl.substr(0,9)=="/teacher/"){

var teacherid = reqUrl.substr(9);

if(/^\d{6}$/.test(teacherid)){

res.end("你要查询老师信息，id为："+teacherid);

}else{

res.end("老师工号位数不对");

}

}else{

res.end("请检查URL");

}

});

server.listen(3000,"127.0.0.1",function () {

console.log('server is starting')

console.log('127.0.0.1:3000')

});

6.7.2 资源静态化（静态服务器）

6.7.2.1 传统服务器

Web网络服务，一般是指允许用户通过浏览器访问到互联网中各种资源的服务。Apache程序是目前拥有很高市场占有率的Web服务程序之一，其跨平台和安全性广泛被认可且拥有快速、可靠、简单的API扩展。

像我们平时上网，这些大多数的网站基本上都是apache、tomcat及Nginx这三类服务器部署的。而且这三类服务器提供静态资源服务也是非常方便。

6.7.2.2 Node.js静态资源

node.js提供了http模块，自身就可以用来构建服务器，而且http模块是由C++实现的，性能可靠。所以node.js更有点偏向底层，可以按照我们的想法与设计服务器。同时也反映了node.js设计一个静态服务器都比较麻烦。同时我们需要http、url、fs、path等模块。

代码如下：

var http = require("http");

var url = require("url");

var fs = require("fs");

var path = require("path");

var server = http.createServer(function(req,res){

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

if(pathname=="/"){

pathname="/index.html";

}

var extname = path.extname(pathname);

var mime = getMime(extname);

fs.readFile("./static"+ pathname,function(err,data){

res.writeHead(200,{"Content-type":mime});

res.end(data);

});

});

server.listen(3000,"127.0.0.1",function () {

console.log('server is starting')

console.log('127.0.0.1:3000')

});

function getMime(extname){

switch(extname){

case ".html":

return "text/html";

break;

case ".css":

return "text/css";

break;

case "jpg":

return "image/jpg";

break;

case ".json":

return "application/json";

break;

default :

return 'text/plain';

}

}

**mime类型**

又叫文档类型。MIME(Multipurpose Internet Mail Extensions)多用途互联网邮件扩展类型。是设定某种扩展名的文件用一种应用程序来打开的方式类型，当该扩展名文件被访问的时候，浏览器会自动使用指定应用程序来打开。多用于指定一些客户端自定义的文件名，以及一些媒体文件打开方式。

常见的MIME类型(通用型)：

超文本标记语言文本 .html text/html

xml文档 .xml text/xml

XHTML文档 .xhtml application/xhtml+xml

普通文本 .txt text/plain

RTF文本 .rtf application/rtf

PDF文档 .pdf application/pdf

Microsoft Word文件 .word application/msword

PNG图像 .png image/png

GIF图形 .gif image/gif

JPEG图形 .jpeg,.jpg image/jpeg

6.8 高频面试

第7讲 Express 4.X

|  |  |  |  |
| --- | --- | --- | --- |
| 知识模块 | 课程内容 | 重点 | 课程目标 |
| Express | 安装, 概念, 基础使用 | ☆ | 掌握：express以及生成器安装  掌握：app级别路由和router级别路由配置  掌握：内置，自定义中间件使用  掌握：express搭建web服务器 |
| 路由配置 | ★ |
| 中间件使用 | ★ |
| 综合案例 | ☆ |

7.1 简介

Express 是简洁灵活的 node.js Web应用框架, 提供强大特性帮助你创建各种Web应用。Express 不对 node.js 已有的特性进行二次抽象，我们只在它基础上扩展了Web应用所需的功能。丰富的HTTP工具以及来自Connect框架的中间件随取随用，创建强健、友好的API变得快速又简单。

Express中主要学习：路由、中间件、模板引擎、其中路由和模板引擎的概念与其他语言一致，比如PHP语言中的路由和PHP中的smarty模板引擎，或前端Vue框架中的路由(Vue-router)和模板语法，Express中间件提供了强大的扩展功能，它类似于我们使用的第三方类库或插件，在本次讲义中我们将学习到这些。

7.2 安装

7.2.1 常规安装

Express的安装是基于npm的，常规安装步骤如下：

1.新建一个空白的文件夹，在当前目录下执行npm init 或 npm init -y（注意此时的文件夹名字尽量不要起express，以免产生工程名和要下载的包名冲突问题）产生一个package.json文件。

2.在cmd命令行（或vscode的终端窗口）中切换目录到当前工程文件夹下，执行如下命令:

npm install express --save

注意：npm 5.0+ 版本在默认情况下会将安装的模块添加到 package.json 文件中的 dependencies 列表中。对于较老的 npm 版本，你就必须指定 --save 参数。然后，照旧执行 npm install 命令即可自动安装依赖列表中所列出的所有模块。

7.2.2 常规安装后实例

● 工程目录结构如下：

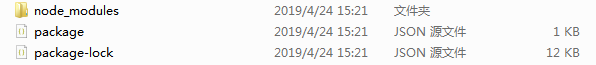


图7-1

● Package.json文件内容如下：



图7-2

在package.json文件中我们需要注意两个配置参数：main和scripts；其中main指明了程序的启动文件，也就是说我们需要在当前工程根目录下手动新建一个index.js的文件；这个index.js的名字是默认的，我们也可以改成其他的名字，比如01\_express\_常规安装.js，但这样一来就需要在工程根目录下新建一个01\_express\_常规安装.js的文件与之对应。

scripts指明了需要执行的脚本代码，执行node程序使用node命令，所以我们可以增加这样的配置信息：

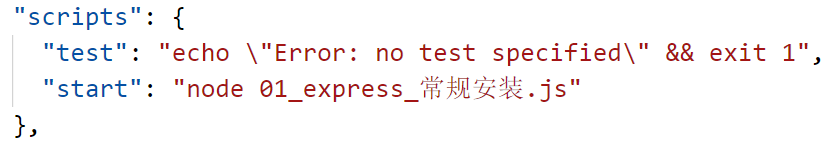


图7-3

这样启动工程的时候就不需要在命令行输入node 01\_express\_常规安装.js这样代码了，我们只需要输入 npm start 就可以了；这里有一个注意事项，在scripts配置项中，start键值，后面的命令可以是下面的形式之一 (index等同于01\_express\_常规安装)：

node ./index.js

node index.js

node ./index

node index

node ./

这种情况的主要原因是main的配置，工程启动会寻找main中的配置文件。

● 01\_express\_常规安装.js，示例代码：

const express = require("express");

var app = express();

app.get("/", (req, res) => {

res.send("helloworld");

});

app.listen(3000, () => {

console.log("应用实例，访问地址为 http://localhost:3000"); .

});

●启动

Cmd命令行（或vscode的终端窗口）切换至当前工程目录后输入: npm start

浏览器中输入：<http://localhost:3000/>

7.2.3 Express生成器安装

通过应用生成器工具 express-generator 可以快速创建一个应用的骨架，包含很多搭建好的目录和配置文件及其中的代码。

express-generator 包含了命令行工具express 。通过如下命令即可安装：

npm install express-generator -g

常用参数如下：

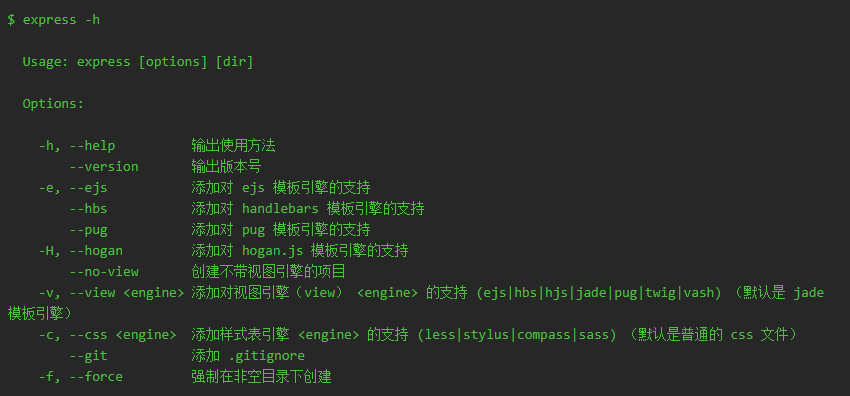


图7-4

通过express命令，可以创建一个工程，命令如下：

express：命令

--view：添加对视图引擎的支持，ejs引擎

exp1 是项目目录工程的名字

express --view=ejs exp1

项目中还需要第三方包，所以需要执行如下命令进到目标工程中，并且安装

cd exp1

npm install

7.2.4 Express生成器生成项目后的示例

● 工程目录结构如下：

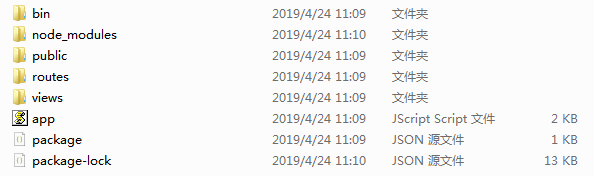


图7-5

● package.json文件内容如下：

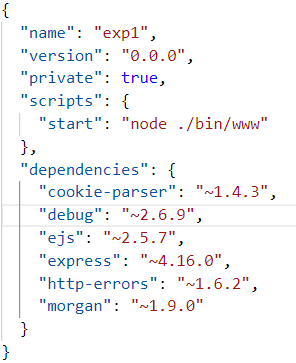


图7-6

通过上个示例信息我们得知了scripts 里面的配置信息，它的启动文件就是bin目录下的www文件，dependencies依赖文件中多出了一些中间件，后面我们会学到这些。

● 启动

命令行输入： npm start

浏览器中输入： <http://localhost:3000/>

注意：Express生成器安装的工程，我们并没有编写任何代码，直接运行即可，因为生成器帮我们默认生成了一些示例代码，这些代码在实际工作的时候需要修改或删除。

7.3 路由配置

7.3.1 路由介绍

如果是第一次在程序中见到“路由”这个这个词，很容易会联想到路由器；从它们的工作原理上来看，确实可以这样去联想；我们在浏览器输入一个网址后经过一系列的路由最终找到我们需要的答案（即endPoints），那么程序中的路由也是这样，一个网站往往不止一个页面，我们通过配置路由让不同的路径对应不同的页面（即endPoints）

7.3.2 app级路由

使用app对象，来监听前端HTTP请求的方法，以及路径，并执行响应的函数，来做出对应的响应

Express中的路由组成包含：Http方法（GET、POST、PUT、DELETE等），路径，中间件方法。方法和路径以及函数里的代码，可以根据具体的业务需求，进行编写。

写作形式常见：

const express = require("express");

const app = express();

app.get("/", (req, res) => { // 这就是一个app级路由的使用, 监听前端以get方法请求/路径, 触发函数执行

res.send("hello world");

})

app.post("/", (req, res) => {

res.send("post方法, 访问/路径");

})

app.put("/", (req, res) => {

res.send("put方法, 访问/路径");

})

app.delete("/", (req, res) => {

res.send("delete方法, 访问/路径");

})

1）get/post/put/delete表示Http方法，需要小写

2）"/" 表示请求路径，"/"一般表示网站根目录或首页

3）(req,res) => { } 是中间件方法，中间件方法的参数可能有err、req、res、next，err参数一般用于错误中间件，next参数用于传递中间件控制权，req和res分别对应request和response对象，关于req和res对象我们在7.6小结详细介绍。

● 路由Http方法：



图7-7

有一个特殊的Http方法all，它能接受上图中的任意http方法。

示例：

app.all("/", (req, res) => {

res.send("all方法, 访问/路径");

})

● 路由path路径：

Path路径可以有两种形式：字符串，正则表达式，动态路由。

1）字符串形式：

下面路由路径匹配 /

app.get("/", (req, res) => {

res.send("访问、根路径");

})

下面路由路径匹配 /about

app.get("/about", (req, res) => {

res.send("访问/about路径")

})

下面路由路径匹配 /random.text

app.get("/random.text", (req, res) => {

res.send("访问/random.text路径");

})

2）正则表达式

只要请求的路径, 包含正则表达式符合部分, 即可命中

下面路由路径匹配只要包含: abcd或acd情况即可

例如请求路径为： xxxabbacdxxx (有acd 符合)

? 匹配前面b字母, 0次或1次

app.get(/ab?cd/, (req, res) => {

res.send("ab?cd");

})

下面路由路径匹配只要包含: abcd、abbcd、abbbcd, abbb(b循环)cd 等

+ 匹配前面b, 1次或多次

app.get(/ab+cd/, (req, res) => {

res.send("ab+cd");

})

下面路由路径匹配 abcd, abxcd, abRANDOMcd, ab123cd等

.点 匹配除“\n”和"\r"之外的任何单个字符

\* 匹配前面.点任意次

app.get(/ab.\*cd/, (req, res) => {

res.send("ab\*cd");

})

下面路由路径匹配 以ab开头的

app.get(/ab.\*/, (req, res) => {

res.send("ab.\*");

})

下面路由路径匹配 以cd结尾的

app.get(/.\*cd$/, (req, res) => {

res.send(".\*cd$");

})

3）动态路由（路由参数）

app.get("/users/:userId/books/:bookId", (req, res) => {

let reqObj = req.params;

res.send(`作者ID: ${reqObj['userId']}, 书籍ID: ${reqObj['bookId']}`);

});

Route path: /users/:userId/books/:bookId

Request URL: http://localhost:3000/users/34/books/8989

req.params: { "userId": "34", "bookId": "8989" }

前端得到响应内容: 作者ID: 34, 书籍ID: 8989

上面代码中的userId和bookId都是动态参数，对应URL里面的34和8989；要获取这些参数值使用req.params

● App.route()方法

前面的学习中我们知道了路由Http方法get和post，如果get和post的地址是一样的，比如我们需要登录，可以这样写：

const express = require("express");

const app = express();

app.get("/login", (req, res) => {

res.send("返回登录页面给访问者");

})

app.post("/login", (req, res) => {

res.send("处理用户提交的登录信息");

})

app.listen(3000, () => {

console.log("应用实例，访问地址为 http://localhost:3000");

});

如果还有其他的Http方法用到一样的path路径，这就显得冗余了，可以使用app.route()方法链式写法：

app.route("/login")

.get((req, res) => {

res.send("返回登录页面给访问者");

})

.post((req, res) => {

res.send("处理用户提交的登录信息");

})

.put((req, res) => {

res.send("更新用户的登录信息");

})

7.3.3 router级别路由

● router路由介绍

上一个小结里面我们介绍了app级别的路由，app级别的路由有一个不好的地方就是不方便做模块化封装，我们在看过express生成器构建的项目后会发现它的app.js文件中并没有写路由代码，路由被设置在一个专门的routes文件夹里面，routes文件夹里面每个文件可以对应一个独立的页面，便于管理。

● express.Router()创建使用

使用express.router()一般需要在单独的文件中，而不是在index.js或app.js中，因为这不便于模块化，就像express生成器创建的工程一样，各个文件夹分工明确，routers里面操作路由，views里面操作视图。

它的用法基本和app路由一致，也是Http方法，path路径和handlers回调中间件方法；只是前面的调用者由app变成和router：

const express = require("express");

const router = express.Router(); // 得到Router级路由对象, 使用方式同app对象

router.get("/news", (req, res) => {

res.send("假如你拿到了消息");

})

module.exports = router; // 此为路由模块, 需要导出路由对象

需要注意的是，我们使用了router级别路由符合了模块化编程思想，那么势必需要模块化处理:module.exports 和 require引入。

const express = require("express");

const RouterFn= require("./05\_route"); // 应用Router级路由函数

const app = express();

app.use(RouterFn); // 使用Router级路由函数中间件, 把前端请求的路由路径和Router级路由对象中的路由路径匹配

app.listen(3000, () => {

console.log("应用实例，访问地址为 http://localhost:3000");

});

这里app.use() 涉及到了自定义中间件概念, 在下一个章节会进行阐述，其他与app路由一致的内容这里就不再赘述了。

7.4 中间件使用

7.4.1 中间件简介

Express是一个自身功能极简，完全是路由和中间件构成一个web开发框架：从本质上来说，一个Express应用就是在调用各种中间件。由此可见，中间件在Express开发中的重要性，因此这里我们就专门来总结一下中间件。

​中间件是可以访问请求对象 （req），响应对象（res）以及next应用程序请求-响应周期中的函数。使用app.use() 来使用/定义中间件

​本质上，中间件就是在请求后，响应前要执行的函数，中间件就是函数使用app.use() 方法，可以给express应用，添加任意中间件功能函数

7.4.2 内置中间件

从Express 4.X开始，express不再依赖Connect，这也意味着express将大多数内置中间件都剥离出去了，比如之前的body-parser，cookie-parser等中间件；现在使用body-parser或cookie-parser的方式是第三方中间件。

Express 4.X的内置中间件是express.static静态资源挂载；

在express 4.16之后又新增了：

express.json 和 express.urlencoded，分别用来解析req传入的json格式、urlencoded格式数据。

● express.static的使用

静态资源指的就是存在于服务器端的文件，例如html/css/js/图片等文件都是静态资源文件，浏览器需要访问，服务器才会返回这些静态资源文件的内容

express.static()封装了路径和读取/响应文件的代码在方法内，我们可以把某个文件夹暴露给浏览器直接进行路径的访问即可。

const express = require("express");

const app = express();

app.use(express.static("./public")); // 从public下索引前端访问的资源路径, (注意不包含public)

app.listen(3000, () => {

console.log("应用实例，访问地址为 http://localhost:3000");

});

● app.use() 虚拟路径使用

可以在第一个参数，设置一个虚拟的路径，前端必须先访问命中此虚拟路径，才可以用接下来的路径匹配后面的路由路径

const express = require("express");

const app = express();

app.use("/www", express.static("./public")); // 前端需要先访问/www, 在访问public下的资源

app.listen(3000, () => {

console.log("应用实例，访问地址为 http://localhost:3000");

});

浏览器需要访问: http://localhost:3000/www/html/06.html

7.4.3 自定义中间件

自定义中间件遵循MiddleWare Function（所谓的中间件其实是一个方法）就可以了，而且可以访问req和res对象，即request和response，代码如下：

app.use((req, res) => { // 可以访问req, 和res即, reqeust和response对象

console.log("中间件函数执行");

res.send("本次响应结束");

});

自定义中间件函数中，还有参数1和参数4(next)，next作用，用于调用下一个中间件函数执行, 可以传递参数到第一个中间件函数的参数1上，并且在过程中仅执行一次res.send() 结束本次响应

app.use((req, res, next) => {

req.nowTime = new Date(); // 可以在请求对象上, 扩展属性

next(Math.floor(Math.random() \* 100)); // 也可以用next调用下个中间件函数执行, 并且传递参数

});

app.use((arg, req, res, next) => {

res.send("第二个中间件函数执行, 时间: %s, 随机数: %d", req.nowTime, arg);

next();

// 注意, 一次请求, 只能对应一次响应, 在next()过程中, 只能有一次res.send()

})

7.4.4 错误处理中间件

错误处理中间件一般写在index.js或app.js最后，即当所有路由都无法匹配的时候进行处理

const express = require("express");

const app = express();

app.get("/news", (req, res) => {

res.send("访问了/news路径");

})

app.get("/api", (req, res) => {

res.send("访问了/api路径");

})

// 错误处理的中间件

// 如果上面都未命中, 则执行这个中间件函数, 所以要写在最后

app.use((req, res, next) => {

res.status(404); // 返回404响应状态码

res.send("你所访问的路径不存在"); // 返回响应正文

})

app.listen(3000, () => {

console.log("应用实例，访问地址为 http://localhost:3000");

});

注意：中间件的执行是自顶向下顺序执行的，所以一定要注意中间件的调用次序

7.5 案例

解析中公教育网站到express项目中，返回给浏览器端展示，static目录中有做好的网站供我们使用，请通过express后台，返回给浏览器端渲染

const express = require("express");

const app = express();

// 1. 使用express.static()内置中间件函数, 暴露静态资源给浏览器直接访问

// app.use(express.static("./static"));

// 前端直接访问http://localhost:3000/index.html 即可访问中公网站

app.listen(3000, () => {

console.log("应用实例，访问地址为 http://localhost:3000");

});

● express.static原理讲解

当然我们这里只模拟现用的几种静态资源的类型，实际上它的源码要比这复杂的多。

const express = require("express");

const app = express();

const fs = require("fs");

const path = require("path");

// 2. 原理讲解

// 根据请求的路径, 读取对应的静态资源, 再用res.send()返回给前端

app.use((req, res) => {

// 问题1: 我们发现请求的有.html / .css / .js / .png / .jpg, 而我们默认返回的内容格式 都是文本

// 解决1: 应该设置响应头, 告知浏览器返回的内容类型(mime-type)

if (path.extname(req.url) === ".png" || path.extname(req.url) === ".jpg") {

res.header("Content-type", "image/jpg");

} else if (path.extname(req.url) === ".css") {

res.header("Content-type", "text/css");

} else {

res.header("Content-type", "text/html");

}

res.send(fs.readFileSync("./static/" + req.url));

})

app.listen(3000, () => {

console.log("应用实例，访问地址为 http://localhost:3000");

});

7.6 高频面试

● 使用express的原因是？

帮助开发者快速构建基于nodejs的web应用，轻量，快速，高效。

* 请说下express 4中app和router的区别 ？

App是express的应用，router是app下面的Router实例；app一般用来挂载中间件或进行全局的set设置，router一般用来配置路由

● 请说下express 4中如何接受get请求和post请求

● 请说下express 4中如何设置静态资源目录

● 请说下express 4中的中间件是做什么的，如何使用

第8讲 Express 模板引擎

|  |  |  |  |
| --- | --- | --- | --- |
| 知识模块 | 课程内容 | 重点 | 课程目标 |
| 模板引擎 | ejs模板引擎 | ★ | 掌握：ejs模板引擎的用法  掌握：art模板引擎的用法  掌握：第三方中间件使用 |
| artTemplate模板引擎 | ☆ |
| 第三方中间使用 | ★ |

8.1 ejs模板引擎

8.1.1 ejs简介

EJS是一套简单的模板语言，在NodeJS后端利用ejs模板引擎, 生成HTML页面。

**特点:**

- 纯JavaScript

- 快速开发

- 执行迅速

- 语法简单:可以直接在HTML中书写JS代码

- 同时支持前后端环境

- 相比pug,更贴近于我们现使用的HTML标签语法

8.1.2 express中集成模板引擎

express中使用ejs比较简单，使用express-generator生成器新建项目的使用使用

--view=ejs, 还需指定app.set(‘view engin’,’ejs’)

如果是原生express项目, 先要手动安装ejs模块, 命令如下

npm i ejs

指定express项目app的模板引擎为ejs

app.set('view engine', 'ejs');

指定模板所在目录ejs\_template, 默认目录名为views

app.set('views', './ejs\_template');

01\_ejs模板引擎\_渲染ejs.js文件内代码如下

const express = require("express");

const app = express();

app.set('view engine', 'ejs'); // 设置ejs作为express应用的模板引擎

app.set('views', './ejs\_template'); // 设置模板引擎的文件夹(默认是views文件夹)

app.get('/', function (req, res) {

res.render('01') // 指定要返回渲染的ejs模板名字, 可以省略.ejs

});

app.listen(3000, () => {

console.log("应用实例，访问地址为 http://localhost:3000");

});

8.1.3 ejs模板文件内标签定义

<%  '脚本' 标签，用于流程控制，无输出。

<%=  输出数据到模板（输出是转义 HTML 标签）

<%#  注释标签，不执行、不输出内容

<%%  输出字符串 '<%'

%>  一般结束标签

02\_ejs模板引擎\_ejs传参.js

app.get('/', function (req, res) {

res.render('02', {

theName: "ejs模板",

arr: ["春天", "夏天", "秋天", "冬天"],

obj: {

age: 19

}

}) // 可以向01.ejs中传递参数对象, 键作为变量名直接使用

});

02.ejs文件使用

<body>

<h1>hello, ejs</h1>

<h2> <%=theName%> </h2>

<ul>

<% for(let i = 0; i < arr.length; i++) { %>

<li><%= arr[i]%></li>

<% } %>

</ul>

<% if(obj.age >= 18) { %>

<p>你已经成年了!!!</p>

<% } else { %>

<p>未成年的小伙子</p>

<% } %>

<%# 我是注释的内容!!! %>

<%%我是个字符串%>

</body>

8.1.4 ejs中的包含（include）

通过include指令将相对于模板路径中的模板片段包含进来。(需要提供’filename’参数)例如，如果存在”./views/users.ejs”和”./view/user/show.ejs”两个模板文件，你可以通过<%- include(‘user/show’);%>代码包含后者。

你可能需要能够输出原始内容的标签(<%-)用于include指令，避免对输出的HTML代码做转义处理。

<ul>

<% users.forEach(function(user){ %>

<%- include(‘user/show’,{user:user}); %>

<% }); %>

</ul>

03\_ejs模板引擎\_include使用.js

app.get('/', function (req, res) {

res.render('03', {

theName: "ejs模板",

arr: ["春天", "夏天", "秋天", "冬天"],

obj: {

age: 19

},

dataArr: [

{

name: "八戒",

hobby: "高家大小姐",

age: 500

},

{

name: "悟空",

hobby: "吃桃子",

age: 1000

},

{

name: "沙师弟",

hobby: "捕鱼",

age: 190

}

]

})

});

03.ejs文件核心代码如下：

<body>

<h1>把03\_v循环使用, 作为复用的组件</h1>

<h2>讲解include的使用</h2>

<% dataArr.map(obj => { %>

<%- include("03\_v.ejs", {userObj: obj}) %>

<% }) %>

</body>

03\_v.ejs模板文件核心代码如下：

<div>

<p>年龄: <%= userObj.name %> </p>

<p>爱好: <%= userObj.hobby %> </p>

<p>年龄: <%= userObj.age %> </p>

</div>

8.2 arttemplate模板引擎

8.2.1 arttemplate简介

art-template 是一个简约、超快的模板引擎。

它采用作用域预声明的技术来优化模板渲染速度，从而获得接近 JavaScript 极限的运行性能，并且同时支持 NodeJS 和浏览器。

它的官网: <https://aui.github.io/art-template/zh-cn/docs/index.html>

8.2.2 express项目中使用arttemplate

需要先下载到当前工程目录中，不光要下载art-template，还要下载支持express项目的扩展包模块，命令如下

npm install --save art-template

npm install --save express-art-template

安装后，需要在express项目应用下，配置如下代码

app.engine('art', require('express-art-template')); // 让app支持引擎为art

app.set('views', path.join(\_\_dirname, 'art\_template')); // 指定模板所在目录

app.set('view engine', 'art'); // 设置模板渲染引擎为art

8.2.3 arttemplate模板语法使用

使用时, 分为标准语法和原始语法, 一般使用标准语法即可, 原始语法与ejs类似

1）输出相关

标准语法：{{ 变量/表达式 }} 输出变量/表达式结果到此处

原始语法：<%= 变量/表达式 %>

2）标签解析相关

标准语法：{{@ 变量 }} 输出变量如果是标签，则直接被浏览器解析

原始语法：<%- value %>

3）条件相关

标准语法：{{if 判断表达式}} ... {{/if}}

{{if 判断表达式}} ... {{else }} ... {{/if}}

原始语法:

<% if (判断表达式) { %> ... <% } %>

<% if (判断表达式) { %> ... <% } else { %> ... <% } %>

4）循环相关

标准语法:

{{each 数组变量}}

{{$index}} {{$value}}

{{/each}}

原始语法:

<% for(leti = 0; i < target.length; i++){ %>

<%= i %> <%= target[i] %>

<% } %>

<body>

<h1>hello, art-template模板</h1>

<!-- 输出语法 -->

{{ user.name }}

{{ user.isMarray ? '已婚' : '未婚'}}

{{ 10 + 8}}

<!-- 不对内容的html标签进行转义处理, 而是会解析此字符串为html标签, 谨慎使用防止注入攻击 -->

{{@ user.theTag }}

<!-- if和else的使用 -->

{{if user.age >= 18 }}

<p>听说你已经成年了</p>

<p>要好好吃饭</p>

{{else}}

<p>还是个小朋友</p>

<p>年轻就要做点刺激的事情</p>

{{/if}}

<!-- 循环使用 -->

{{each user.tags}}

{{$index}} {{$value}}

{{/each}}

</body>

8.2.4 arttemplate中的 include

我们可以封装一个.art文件，封装成复用的标签，在别的.art页面中进行引入使用，这就是include的使用场景

数据准备，06\_art模板引擎\_include使用.js 文件中核心代码

app.get('/', (req, res) => {

res.render('06', {

user: {

dataArr: [{

name: "八戒",

hobby: "高家大小姐",

age: 500

},

{

name: "悟空",

hobby: "吃桃子",

age: 1000

},

{

name: "沙师弟",

hobby: "捕鱼",

age: 190

}

]

}

}); // .art可以省略, 渲染04.art文件到前端

});

复用的.art文件准备，06\_v.art 文件中核心代码

<div>

<!-- $data接收外部传入的参数3数据 -->

<p>姓名: <%= $data.name %> </p>

<p>爱好: <%= $data.hobby %> </p>

<p>年龄: <%= $data.age %> </p>

</div>

.art页面准备，06.art文件内核心代码

<body>

<h1>hello, art-template模板, include使用</h1>

<ul>

{{each user.tags}}

{{$index}} {{$value}}

{{/each}}

{{each user.dataArr}}

{{include './06\_v.art' $value}}

{{/each}}

</ul>

</body>

8.3 第三方中间件使用

在express项目中，我们可以使用一些其他程序员编写的中间件使用，集成到我们的项目中，给项目添加额外的功能，使用时，需要先利用npm下载到当前工程中，调用app.use()方法使用引入的中间件函数，下面就介绍几种常见的中间件模块的使用

8.3.1 nodemon热更新

在编写调试Node.js项目，修改代码后，需要频繁的手动close掉，然后再重新启动，非常繁琐。现在，我们可以使用nodemon这个工具，它的作用是监听代码文件的变动，当代码改变之后，自动重启。

需要全局安装nodemon模块，因为一会儿要使用终端的nodemon命令来替代node命令执行某个node.js的文件运行，命令如下

npm install nodemon -g

安装成功后，在当前express项目下，新建nodemon.json配置文件，配置nodemon相关的设置

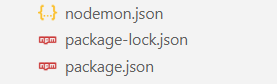


图8-1

nodemon.json配置中内容如下：

restartable：重启的命令，默认是 rs

ignore：忽略的文件后缀名或者文件夹，文件路径的书写用相对于 nodemon.json 所在位置的相对路径

verbose：true 表示输出详细启动与重启信息

execMap：运行服务的后缀名和对应的运行命令，"js": "node --harmony" 表示用 nodemon 代替 node --harmony 运行 js 后缀文件

watch：监控的文件夹路径或者文件路径。

env：运行环境 development 是开发环境，production 是生产环境。

ext：监控指定后缀名的文件，用空格间隔。

{

"restartable": "rs",

"ignore": [

".git",

".svn",

"node\_modules/\*\*/node\_modules"

],

"verbose": true,

"execMap": {

"js": "node --harmony"

},

"watch": [

],

"env": {

"NODE\_ENV": "development"

},

"ext": "js json"

}

用nodemon命令来启动热监听，终端效果如下



图8-2

效果：当你再次编辑此文件后，不用来终端停止，再次开启来让代码生效了，终端可以检测到文件变更后，自动重启，这就是nodemon的作用

8.3.2 cookie-parser cookie管理

负责设置客户端的cookie相关信息，或者获取浏览器端相关的cookie信息

首先需要在当前工程中安装cookie-parser模块，命令如下：

npm install cookie-parser

在express应用中，集成cookie-parser模块

const cookieParser = require("cookie-parser"); // 引入cookieparser中间件函数

app.use(cookieParser()); // 让express使用cookieparser中间件函数, 拥有解析cookie能力

安装设置完成后，先来使用设置客户端cookie的方法，当前端get方法请求路由路径为/时，设置客户端的cookie，并设置过期时间为1天后

app.get("/", (req, res) => { // 访问/, 设置客户端cookie

let now = new Date();

let outTime = now.setDate(now.getDate() + 1); // 过1天的时间毫秒13位

res.cookie("a", "10", {

"expires": (new Date(outTime)) // 设置日期对象

});

res.send("设置cookie的键为a, 值为10, 参数3可以设置过期时间");

});

启动项目后，浏览器访问: http://localhost:3000/，查看浏览器上的cookie内容，键为a， 内容为10，而且过期时间是一天后，如图



图8-3

当然，每次浏览器发起请求时，会默认把cookie带给服务器端，所以再次访问路由路径/getCookie，会发现后台把获取到的键为a的cookie值，通过响应内容返回给前端了（说明后台能够访问到cookie的值进行使用）， 代码如下

app.get("/getCookie", (req, res) => {

res.send(req.cookies.a); // 返回前端发来的cookie中键为a的值

})

8.3.3 express-session session会话管理

session技术就是一种基于后端有别于数据库的临时存储数据的技术，session保存的位置是在服务器端，session一般来说是要配合cookie使用，如果是浏览器禁用了cookie功能，也就只能够使用URL重写来实现session存储的功能

先下载express-session模块到express项目内，命令如下

npm i express-session

在express项目中集成session功能

const session = require('express-session'); // 引用session中间件

app.use(session({ // 使用 session 中间件

secret : 'secret', // 对session id 相关的cookie 进行签名

resave : true,

saveUninitialized: false, // 是否保存未初始化的会话

cookie : {

maxAge : 1000 \* 60 \* 3, // 设置 session 的有效时间，单位毫秒

},

}));

express-session即能设置服务器端的session内容，还可以读取session内容，先来个设置部分吧，设置访问人员的随机身份验证码（随机数6位的一个）

app.get("/setsession", (req, res) => {

let randNum = Math.floor(Math.random() \* (999999 - 100000 + 1) + 100000);

req.session.rand = randNum;

res.send("session设置成功, session值为 " + randNum);

});

也可以在访问时，访问现在建立会话的session数据

app.get("/getsession", (req, res) => {

let sessionStr = req.session.rand;

res.send("后台获取到的session的值为" + sessionStr);

});

8.3.4 body-parser

body-parser是一个HTTP请求体解析的中间件，使用这个模块可以解析JSON、Raw、文本、URL-encoded格式的请求体

安装body-parser到当前express项目中，命令如下：

npm install body-parser

在express中进行配置，设置post请求内容格式为application/x-www-form-urlencoded， 才能正确进行发来的参数，这也就要请求客户端的请求头中的Content-type：必须是application/x-www-form-urlencoded，配置如下：

app.use(bodyParser.urlencoded({ extended: true })); // 创建 application/x-www-form-urlencoded 解析, 可以解析utf8编码数据

如果客户端发送的post请求的参数类型是JSON格式的内容，在请求头中Content-type为application/json的话，证明前端直接发来的数据格式是一个JSON对象，则后端向正常的解析，则需要配置下面这行：

app.use(bodyParser.json()); // 创建 application/json 解析

可以解析客户端使用post方法，传递过来的参数，因为浏览器无法直接使用post方式请求路由路径，所以可以借助第三方请求工具，例如postman，发送post请求，准备好路由接口

app.post("/", (req, res) => {

res.send(req.body); // 把请求的body再用响应内容发回到前端查看

});

8.3.5 http-errors

为express应用，轻松的创建HTTP错误，返回HTTP状态的一个第三方中间件函数

首先，安装此模块到express应用中去

npm install http-errors

当用户访问时，我们直接返回一个错误的状态给前端查看，你可以用在任何的一种错误的逻辑下，返回一些错误提示信息给客户端，举例：

app.get("/", (req, res, next) => { // 可以用next, 直接执行默认的错误提示

next(createError(401, 'Please login to view this page.')); // 没有响应内容, createError默认响应错误提示的消息格式给前端

})

客户端效果查看

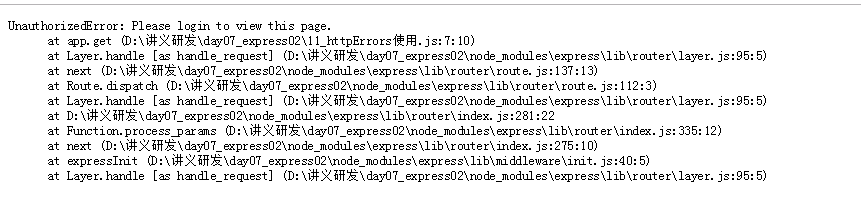


图8-4

8.3.6 serve-favicon

用于服务器端设置favicon视觉提示，其实就是浏览器标签标题栏上的小图标

先下载此模块到express项目中去

npm install serve-favicon

配置代码如下

const favicon = require('serve-favicon'); // 引入图标中间件

const path = require('path');

app.use(favicon(path.join(\_\_dirname, 'public', 'favicon.ico')))

app.get("/", (req, res) => {

res.send("图标成功了吗?");

}),

// 注意, 每个端口号的favicon.ico图标有缓存

// 解决方式1: 清理浏览器的缓存, 再运行查看

// 解决方式2: 换一个express的端口号即可

浏览器访问http://localhost:3000/ 效果如下：



图8-5

8.3.7 svg-captcha

这是一个svg图形验证码插件，把text内容转成svg图形返回使用

首先在express项目中，下载此模块到项目中来

npm install svg-captcha

在具体的接口中进行使用

const svgCaptcha = require("svg-captcha");

app.get("/", (req, res) => {

const captcha = svgCaptcha.create();

res.type("svg");

res.status(200).send(captcha.data);

})

浏览器访问后，效果如下



图8-6

8.3.8 formidable

一个Node.js模块，用于解析表单数据，特别是文件上传，先安装此模块到工程目录中

npm i formidable

准备14\_formidable使用.js，准备引入模块和初始化form对象并且进行配置

const express = require("express");

const app = express();

const fs = require("fs");

const formidable = require("formidable");

const formObj = new formidable.IncomingForm(); // 新建form对象

formObj.encoding = 'UTF-8'; // UTF8编码

formObj.uploadDir = "./tempDir"; // 接收的文件缓存路径

编写接口， 格式化接收到的form表单的信息，fields是接收到的post参数，files是接收到的文件保存在缓存路径中，需要自己利用fs模块，把缓存中的图片文件移动到具体业务文件夹中保存起来

app.post("/uploadFile", (req, res) => {

formObj.parse(req, (err, fields, files) => {

let fileObj = files.image;// 4. 获取真正的文件对象, (image是前后端传递值时的key值名称)

let oldPath = fileObj['path'];// 6. 从缓存里把图片移动到真正的业务文件夹中

let newPath = "./uploadFile/" + fileObj['name'];

fs.rename(oldPath, newPath, (err) => { // 7. 挪动

if (err) {

console.error(err);

res.send("上传失败, 至于为啥:" + err.toString());

} else {

res.send("上传成功");

}

});

});

});

以上后台接口部分就已经准备完毕，请求的方式为post，地址为/uploadFile，图片参数名为image，前端可以使用form表单标签+input的type为file，name设置image进行图片的上传即可，注意表单method为post并且enctype为multipart/form-data（以二进制流形式传递数据给后台解析），提交成功后，后台的uploadFIle文件夹中就会出现前端上传到服务器的图片文件，打开可以预览。



图8-7

8.3.9 multer

Multer是一个node.js中间件，用于处理多部分/表单数据，主要用于上传文件。首先还是需要先下载此模块到工程中去。

npm i multer

准备15\_multer使用.js文件，引入此模块并准备文件上传过来存放的缓存目录为tempDir，并且在当前工程中准备目录tempDir

const express = require("express");

const app = express();

const fs = require("fs");

const multer = require('multer')

const upload = multer({

dest: 'tempDir/' // 设置文件缓存的目录

})

编写接口/uploadFile，用于接收前端提交过来的表单数据，传递图片的参数名为image，提交表单文件数据格式为multipart/form-data，代码如下：

app.post('/uploadFile', upload.single('image'), function (req, res, next) {

// req.file 是 `image` 文件, 前端用image字段传参

// req.body 对象中是表单中提交的文本字段(如果有)

let fileObj = req.file; // 4. 获取真正的文件对象, (image是前后端传递值时的key值名称)

let oldPath = fileObj['path']; // 6. 从缓存里把图片移动到真正的业务文件夹中

let newPath = "./uploadFile/" + fileObj['name'];

fs.rename(oldPath, newPath, (err) => { // 7. 挪动

if (err) {

console.error(err);

res.send("上传失败, 至于为啥:" + err.toString());

} else {

res.send("上传成功");

}

});

})

8.4 高频面试

● 什么是ejs模板引擎.

● 你都用过哪些node第三方的模块, 功能是什么?

第9讲 MySQL

|  |  |  |  |
| --- | --- | --- | --- |
| 知识模块 | 课程内容 | 重点 | 课程目标 |
| Express | 数据库概念介绍 | ☆ | 掌握：数据库的基本概念  掌握：掌握关系和非关系数据库的区别和特性  掌握：数据库三范式  掌握：基本CRUD操作和常用DDL语句 |
| 安装配置 | ☆ |
| 数据库设计 | ★ |
| CRUD | ★ |

9.1 数据库介绍

9.1.1 数据库概念

数据库是“按照数据结构来组织、存储和管理数据的仓库”。它的存储空间很大，可以存放百万条、千万条、上亿条数据。但是数据库并不是随意地将数据进行存放，是有一定的规则的，否则查询的效率会很低。当今世界是一个充满着数据的[互联网](https://baike.baidu.com/item/%E4%BA%92%E8%81%94%E7%BD%91/199186" \t "/var/folders/9m/hyy790l140ggcdnk2gqzjfvw0000gn/T/com.kingsoft.wpsoffice.mac/wps-apple/x/_blank)世界，充斥着大量的数据。即这个互联网世界就是数据世界。数据的来源有很多，比如出行记录、消费记录、浏览的[网页](https://baike.baidu.com/item/%E7%BD%91%E9%A1%B5/99347" \t "/var/folders/9m/hyy790l140ggcdnk2gqzjfvw0000gn/T/com.kingsoft.wpsoffice.mac/wps-apple/x/_blank)、发送的消息等等。除了文本类型的数据，图像、音乐、声音都是数据。

9.1.2 数据库管理系统

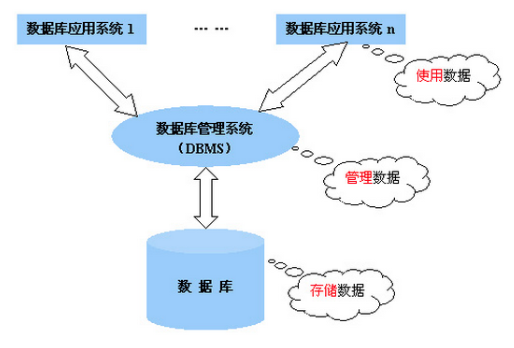


图9-1

要想管理数据库，就需要数据库管理系统（DBMS），用来对数据进行存储、管理等操作的软件。大部分DBMS都提供数据定义语言DDL和数据操作语言DML，供用户定义数据库的模式结构与权限约束，实现对数据的追加、删除等操作；关于DDL和DML稍后会详细介绍。

目前常见的数据库管理软件（DBMS）有：

一、 Oracle

Oracle Database，又名Oracle RDBMS，或简称Oracle。是甲骨文公司的一款关系数据库管理系统，其主要特性为：

处理速度快，非常快。

 安全级别高。支持快闪以及完美的恢复，即使硬件坏了 也可以恢复到故障发前一秒。

几台数据库做负载数据库，可以做到30s以内故障转移。

网格控制，以及 数据仓库方面 也非常强大。

二、 MySQL

MySQL是一个小型关系型数据库管理系统,MySQL被广泛地应用在Internet上的中小型网站中，其特点有：

1. 开放源码；

2. 高度非过程化；

3. 面向集合的操作方式；

4. 以一种语法结构提供多种使用方式；

语言简洁，易学易用。

三、 MS SQL Server

SQLserver数据库是美国微软公司发布的一款RMDBS数据库，也就是关系型数据库系统。SQLserver的优点为：

1. 真正的客户服务器体系结构。

2. 图形化用户界面，更加直观、简单。

3. 丰富的编程接口工具，为用户进行程序设计提供更多选择余地。

4. SQLserver和Windows NT完成集成，可以利用NT的愈多功能。

5. 具有很好的伸缩性，可跨界运行。从膝上型电脑到大型处理器可多台使用。

对web技术的支持，使用户能够容易的将数据库中的数据发布到web上。

9.1.3 数据库分类

在当今的互联网中，最常见的数据库模型主要是两种，即关系型数据库（SQL）和非关系型数据库（NoSQL，Not Only SQL）。

一、关系型数据库（SQL）

关系型数据库，存储的格式可以直观地反映实体间的关系。关系型数据库和常见的表格比较相似，关系型数据库中表与表之间是有很多复杂的关联关系的。常见的关系型数据库有MySQL，Oracle，SqlServer等。虽然关系型数据库有很多，但是大多数都遵循SQL（结构化查询语言，Structured Query Language）标准。 常见的操作有查询，新增，更新，删除，求和，排序等。

关系型数据库对于结构化数据的处理更合适，如学生成绩、地址等，这样的数据一般情况下需要使用结构化的查询，例如join，这样的情况下，关系型数据库就会比NoSQL数据库性能更优，而且精确度更高。由于结构化数据的规模不算太大，数据规模的增长通常也是可预期的，所以针对结构化数据使用关系型数据库更好。

二、非关系型数据库（NoSQL）

NoSQL数据存储不需要固定的表结构，通常也不存在连接操作。在大数据存取上具备关系型数据库无法比拟的性能优势。目前非关系型数据库仍然没有一个统一的标准，下面是非关系型数据库的两个分类：

1、搜索键值存储数据库（key-value）：

键值数据库就类似传统语言中使用的哈希表。可以通过key来添加、查询或者删除数据库，因为使用key主键访问，所以会获得很高的性能及扩展性。Key/value模型对于IT系统来说的优势在于简单、易部署、高并发。典型产品：Memcached、Redis

2、面向文档（Document-Oriented）数据库

文档型数据库可以 看作是键值数据库的升级版，允许之间嵌套键值。而且文档型数据库比键值数据库的查询效率更高。面向文档数据库会将数据以文档形式存储。典型产品：MongoDB。

三、关系和非关系型数据库的区别

**存储方式**

传统的关系型数据库采用表格的储存方式， 数据以行和列的方式进行存储，要读取和查询都十分方便。而非关系型数据库通常以数据集的方式，大量的数据集中存储在一起，类似于键值对、图结构或者文档。

**存储结构**

关系型数据库按照[结构化](https://baike.baidu.com/item/%E7%BB%93%E6%9E%84%E5%8C%96/22244423" \t "/var/folders/9m/hyy790l140ggcdnk2gqzjfvw0000gn/T/com.kingsoft.wpsoffice.mac/wps-apple/x/_blank)的方法存储数据， 每个数据表都必须对各个字段定义好（也就是先定义好表的结构），再根据表的结构存入数据。NoSQL数据库采用的是动态结构，对于数据类型和结构的改变非常的适应，可以根据数据存储的需要灵活的改变数据库的结构。

**存储规范**

关系型数据库为了避免重复、规范化数据以及充分利用好存储空间，把数据按照最小关系表的形式进行存储。而NoSQL数据库的数据存储方式是用平面数据集的方式集中存放。

**扩展方式**

关系型数据库只具备纵向扩展能力。非关系型数据库还可以采用横向的方式来扩展。

**查询方式**

关系型数据库采用结构化查询语言（即SQL）来对数据库进行查询，NoSQL中的文档Id与关系型表中主键的概念类似，NoSQL数据库采用的数据访问模式相对SQL更简单而精确。

**事务性**

关系型数据库强调ACID规则（原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation）、持久性（Durability））。而NoSQL数据库强调BASE原则（基本可用（Basically Available）、软状态（Soft-state）、最终一致性（Eventual Consistency））。

**授权方式**

关系型数据库常见的有Oracle，SQLServer，DB2，Mysql，除了Mysql大多数的关系型数据库如果要使用都需要支付一笔价格高昂的费用，即使是免费的Mysql性能也受到了诸多的限制。而对于NoSQL数据库，比较主流的有redis，HBase，MongoDb，memcache等产品，通常都采用[开源](https://baike.baidu.com/item/%E5%BC%80%E6%BA%90/20720669" \t "/var/folders/9m/hyy790l140ggcdnk2gqzjfvw0000gn/T/com.kingsoft.wpsoffice.mac/wps-apple/x/_blank)的方式，不需要像关系型数据库那样，需要一笔高昂的花费。

9.2 安装配置

9.2.1 下载MySql

浏览器打开MySql官网：<https://www.mysql.com/>



图9-2

选择DOWNLOADS选项卡：

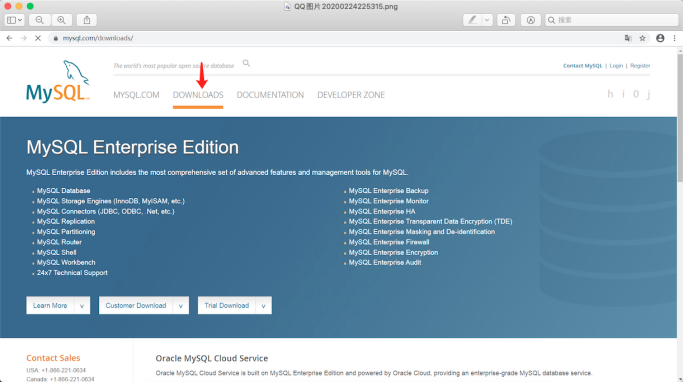


图9-3

在下载页面末端，选择：MySQL Commuinty Server

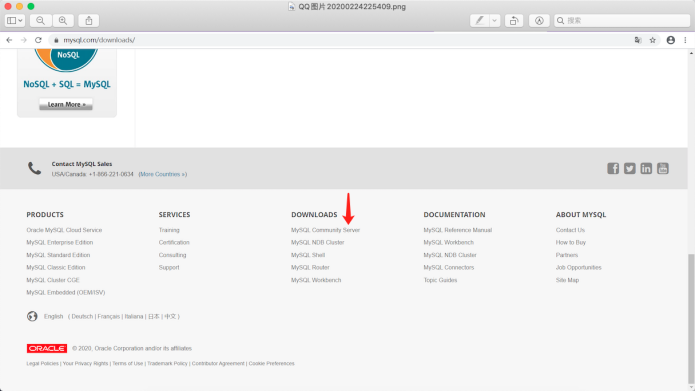


图9-4

弹出下载页面，选择Looking for previous GA versions：

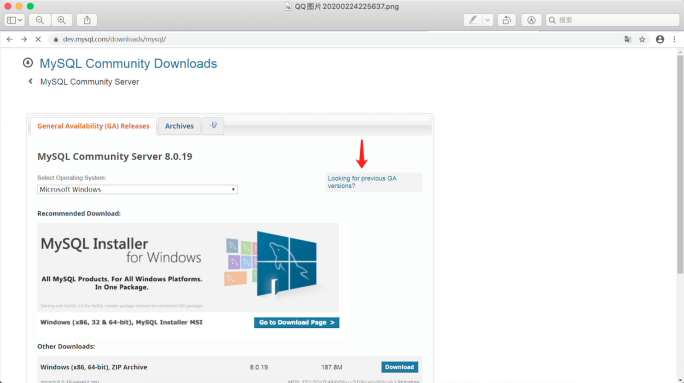


图9-5

选择MySQL版本：5.7.29

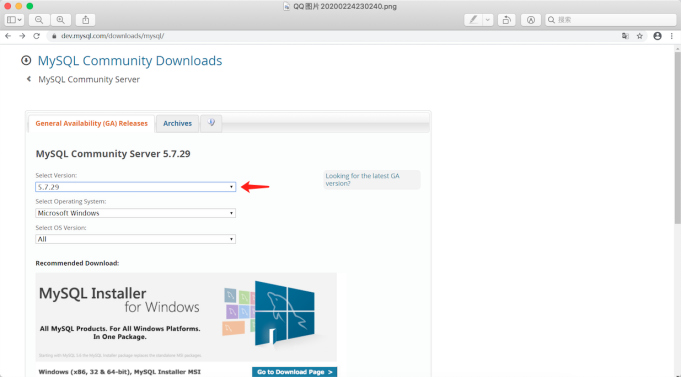


图9-6

根据自己的电脑选择是32位还是64位

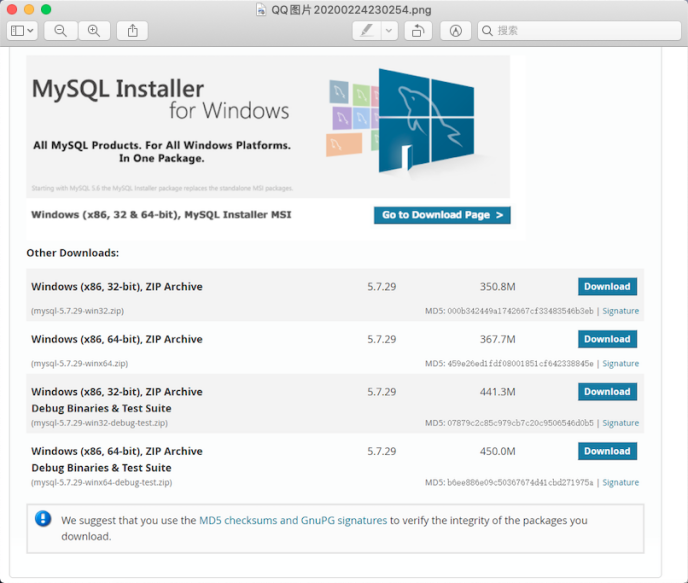


图9-7

不用登陆和注册，点击：No thanks,just start my download



图9-8

注意：打开MySQL官网，下载MySQL可能会很慢。

9.2.2 环境变量配置

下载后的MySQL是zip压缩包，不需要安装，只需要把压缩包解压到对应的目录下即可；建议放到盘符根目录下，尽量不要有中文路径。比如：D:\mysql-5.7.29-winx64

配置环境变量步骤如下：

我的电脑->右键->属性



图9-9

选择高级系统设置

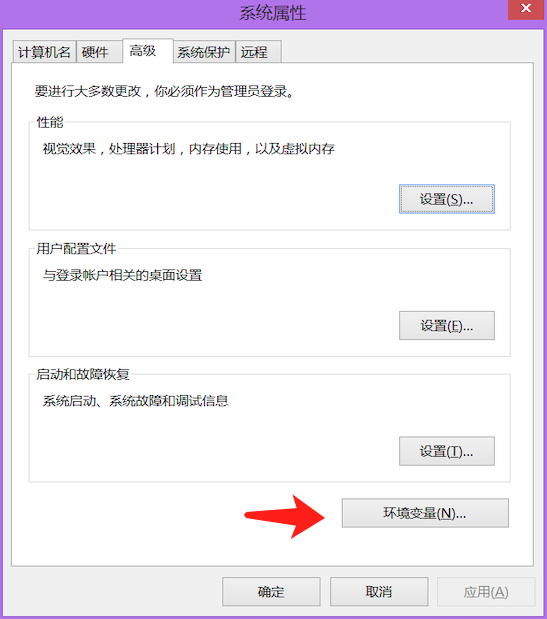


图9-10

选择环境变量



图9-11

选择 系统变量，找到path，点击编辑，在变量值的最后输入 英文分号 分隔，然后输入解压后的mysql目录下的bin文件夹所在目录。

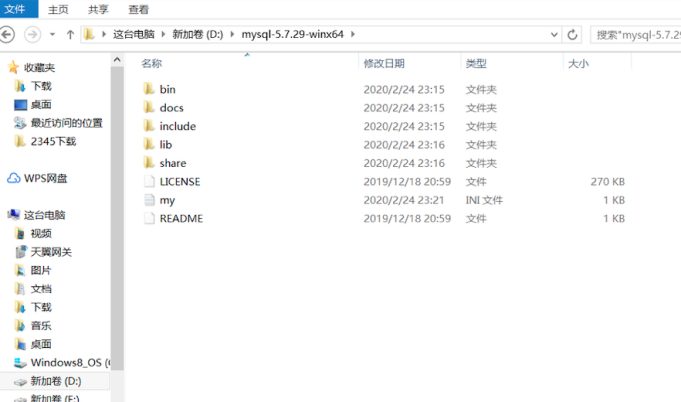


图9-12

在解压后的mysql-5.7.29-winx64的根目录下，新建一个my.ini配置文件，黏贴或输入如下内容：

[mysql]

# 设置mysql客户端默认字符集

default-character-set=utf8

[mysqld]

skip-name-resolve

skip-grant-tables

#设置3306端口

port = 3306

# 设置mysql的安装目录

basedir=D:\mysql-5.7.29-winx64

# 设置mysql数据库的数据的存放目录

datadir=D:\mysql-5.7.29-winx64\data

# 允许最大连接数

max\_connections=200

# 服务端使用的字符集默认为8比特编码的latin1字符集

character-set-server=utf8

# 创建新表时将使用的默认存储引擎

default-storage-engine=INNODB

注意：其中 basedir 和 datadir 需要改成你对应的目录。

9.2.3 安装MySQL

用 管理员方式 打开cmd，进入到 MySQL的bin 路径（ D:\mysql-5.7.29-winx64\bin），之后执行命令开始安装：

1.mysqld --initialize，这个命令执行后没有任何提示

2.net start mysql，如果这个命令执行没有问题，那么执行完这个命令会提示“MySQL 服务正在启动…MySQL 服务已经启动成功。”

执行完毕后在MySQL的根目录下会生成一个data文件夹，后续的mysql数据都会存储到该文件夹下。

**注意：在执行net start mysql的时候如果出现下面情况：**



图9-13

请执行：mysqld --install

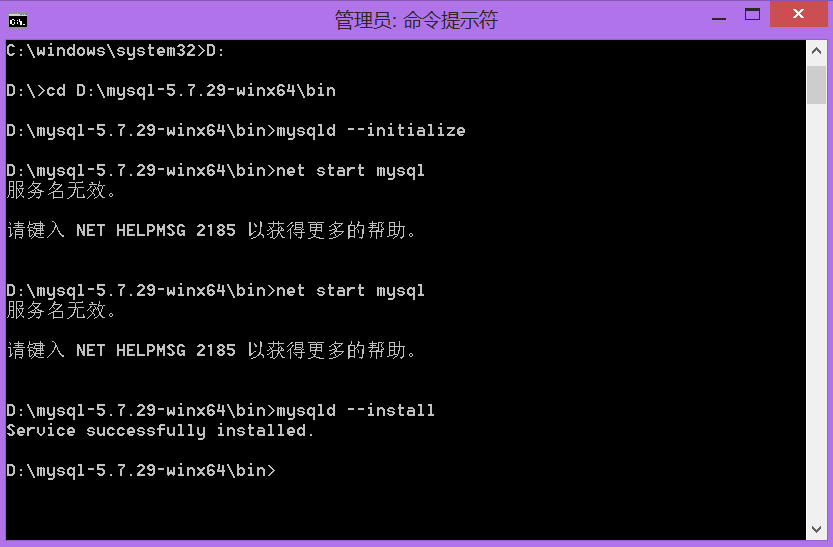


图9-14

然后继续执行：net start mysql

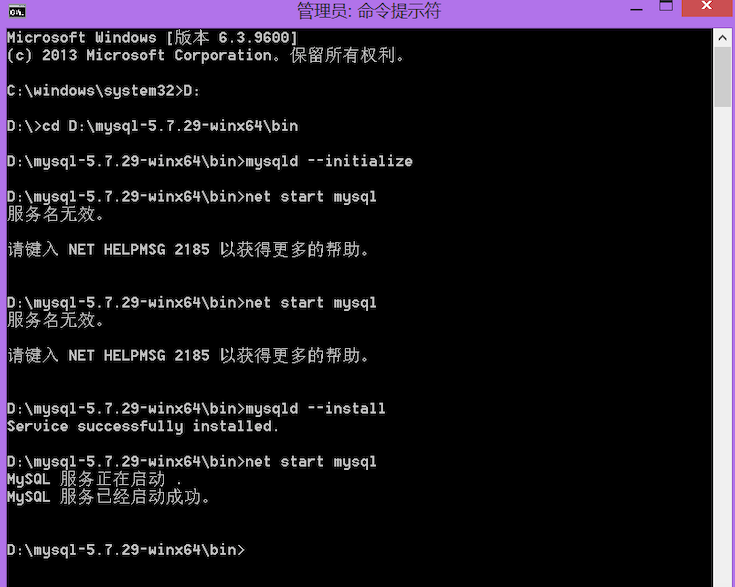


图9-15

9.2.4 配置 MySQL root 密码

使用 mysqld --initialize 命令来初始化 MySQL 会随机生成一个 root 密码，默认密码在 data 文件夹里的一个以 .err 为后缀的文件存储，打开这个文件搜索对应消息，例如：

[Note] A temporary password is generated for root@localhost: caYxCM>!;syt

这里的 caYxCM>!;syt 就是 root 用户的默认密码。

在使用 mysql -u root -p 命令连上 MySQL 后，可以用这个命令修改 root 密码，例如这里我改成 root123：

ALTER USER 'root'@'localhost' IDENTIFIED BY root123;

9.3 数据库设计

目前[关系数据库](https://baike.baidu.com/item/%E5%85%B3%E7%B3%BB%E6%95%B0%E6%8D%AE%E5%BA%93/1237340" \t "/var/folders/9m/hyy790l140ggcdnk2gqzjfvw0000gn/T/com.kingsoft.wpsoffice.mac/wps-apple/x/_blank)有六种范式：第一范式（1NF）、第二范式（2NF）、第三范式（3NF）、巴斯-科德范式（BCNF）、[第四范式](https://baike.baidu.com/item/%E7%AC%AC%E5%9B%9B%E8%8C%83%E5%BC%8F/3193985" \t "/var/folders/9m/hyy790l140ggcdnk2gqzjfvw0000gn/T/com.kingsoft.wpsoffice.mac/wps-apple/x/_blank)(4NF）和[第五范式](https://baike.baidu.com/item/%E7%AC%AC%E4%BA%94%E8%8C%83%E5%BC%8F/5025271" \t "/var/folders/9m/hyy790l140ggcdnk2gqzjfvw0000gn/T/com.kingsoft.wpsoffice.mac/wps-apple/x/_blank)（5NF，又称完美范式）。

通常我们用的最多的就是第一范式（1NF）、第二范式（2NF）、第三范式（3NF），也就是本文要讲的“三大范式”

9.3.1 第一范式（1NF）

第一范式（1NF）：要求数据库表的每一列都是不可分割的原子数据项。



图9-16

在上面的表中，“家庭信息”和“学校信息”列均不满足原子性的要求，不满足第一范式，调整如下：



图9-17

调整后的每一列都是不可再分的，因此满足第一范式（1NF）。

9.3.2 第二范式（2NF）

第二范式（2NF）：在1NF的基础上，非主键属性必须完全依赖于主键（即消除部分依赖）

第二范式需要确保数据库表中的每一列都和主键相关，而不能只与主键的某一部分相关（主要针对联合主键而言）。



图9-18

在上图所示的情况中，同一个订单中可能包含不同的产品，因此主键必须是“订单号”和“产品号”联合组成，但可以发现，产品数量、产品折扣、产品价格与“订单号”和“产品号”都相关，但是订单金额和订单时间仅与“订单号”相关，与“产品号”无关，这样就不满足第二范式的要求，调整如下，需分成两个表：



图9-19

9.3.3 第三范式（3NF）

第三范式（3NF）：在2NF基础上，任何非主键[属性](https://baike.baidu.com/item/%E5%B1%9E%E6%80%A7" \t "/var/folders/9m/hyy790l140ggcdnk2gqzjfvw0000gn/T/com.kingsoft.wpsoffice.mac/wps-apple/x/_blank)不依赖于其它非主键属性（在2NF基础上消除传递依赖）

第三范式需要确保数据表中的每一列数据都和主键直接相关，而不能间接相关。



图9-20

上表中，所有属性都完全依赖于学号，所以满足第二范式，但是“班主任性别”和“班主任年龄”直接依赖的是“班主任姓名”，而不是主键“学号”，所以需做如下调整：



图9-21

9.3.4 反三范式

没有冗余的数据库设计可以做到。但是，没有冗余的数据库未必是最好的数据库，有时为了提高运行效率，就必须降低范式标准，适当保留冗余数据。具体做法是：在概念数据模型设计时遵守第三范式，降低范式标准的工作放到物理数据模型设计时考虑。降低范式就是增加字段，允许冗余，达到以空间换时间的目的。

9.3.5 范式设计和反范式设计的对比

一、范式化

优点：  
 1·可以尽量的减少数据冗余  
 2·数据表更新快体积小  
 3·范式化的更新操作比反范式化更快  
 4·范式化的表通常比反范式化更小

缺点：  
 1·对于查询需要对多个表进行关联，导致性能降低  
 2·更难进行索引优化

二、反范式化

优点：  
 1·可以减少表的关联  
 2·可以更好地进行索引优化

缺点  
 1·存在数据冗余及数据维护异常  
 2·对数据的修改需要更多成本

9.4 CRUD

9.4.1 DDL和DML

关系型数据库的语言分类常用有：DDL和DML。

一、DDL：

DDL（Data Definition Language）数据定义语言-用于定义和管理 SQL 数据库中的所有对象的语言，对数据库中的某些对象(例如，database,table)进行管理。包括的关键字有：create、alter、drop、truncate、comment、grant、revoke等

二、DML：

DML（Data Manipulation Language）数据操作语言-数据库的基本操作，SQL中处理数据等操作统称为数据操纵语言,简而言之就是实现了基本的“增删改查”操作。包括的关键字有：select、update、delete、insert等

三、DML与DDL的区别：

1、DML操作是可以手动控制事务的开启、提交和回滚的。

2、DDL操作是隐性提交的，不能rollback！

9.4.2 DDL语句

一、创建数据库:

create database IF NOT EXISTS ujiuye CHARACTER SET utf8;

二、创建表格:

use ujiuye;

create table IF NOT EXISTS stud(

id int,

name varchar(30),

age int

);

三、更改表结构(设置约束)

desc stud; //查看表结构

alter table stud drop column age;

alter table stud add column age int;

四、删除表、删除数据库

drop table stud;

drop database ujiuye;

9.4.3 DML语句

一、插入(insert)

格式1

INSERT INTO emp(ename,hiredate,sal,deptno)

VALUES('zzx1','2000-01-01','2000',1);

格式2

INSERT INTO emp VALUES('lisa','2003-02-01','3000',2);

不用指定字段名称,但是value值需与表字段严格对应.

格式3

INSERT INTO emp(ename,sal) VALUES('dony',1000);

对于含可空字段、非空但是含有默认值的字段、自增字段，可以不用在INSERT后的字段列表里面出现，VALUES后面只写对应字段名称的VALUE

格式4

在MySQL中，INSERT语句还可以一次性插入多条记录

INSERT INTO tablename(field1,field2,......fieldn)

VALUES

(record1\_value1,record1\_value2,......record1\_valuen),

(record2\_value1,record2\_value2,......record2\_valuen),

......

(recordn\_value1,recordn\_value2,......recordn\_valuen);

二、更新(update)

格式1

UPDATE emp SET sal=4000 WHERE ename='lisa';

格式2

UPDATE t1,t2,...tn SET

t1.field1=expr1,

t2.field2=expr2,

...

tn.fieldn=exprn [WHERE CONDITION];

同时更新多个表中的数据

三、删除(delete)

格式1

DELETE FROM tablename [WHERE CONDITION];

格式2

DELETE t1,t2,...tn FROM t1,t2,...tn [WHERE CONDITION];

注意：不管是单表还是多表，不加where条件将会把表的所有记录删除，所以操作时一定要小心。

四、查询(select)

基本语法:

SELECT \* FROM tablename [WHERE CONDITION];

1.查询不重复的记录，可以用distinct关键字实现：

SELECT DISTINCT deptno FROM emp;

2.条件查询:

SELECT \* FROM emp WHERE deptno=1;

其中，where 后面的条件除“=” 外，还可以使用 >、<、>=、<=、!=等比较运算符；

多个条件之间还可以使用or、and等逻辑运算符进行多条件联合查询

排序和限制:

SELECT \* FROM tablename [WHERE CONDITION] [ORDER BY field1 [DESC|ASC], field2 [DESC|ASC],...fieldn

[DESC|ASC]];

其中，DESC和ASC是排序关键字，DESC表示按照字段进行降序排列（上大-下小），ASC则表示升序排列（上小-下大），如果不写此关键字默认是升序排列。  
 ORDER BY 后面可以跟多个不同的排列字段，并且每个排序字段可以有不同的排序顺序。

如果排序字段的值一样，则值相同的字段按照第二个排序字段进行排序，以此类推。如果只有一个排序字段，则这字段相同的记录将会无序排列。

SELECT ......[LIMIT offset\_start,row\_count];

对于排序后的记录，如果希望只显示一部分，而不是全部，这时，就可以使用LIMIT关键字来实现.

LIMIT 经常和ORDER BY一起配合使用来进行记录的分页显示。

注意：limit属于MySQL扩展SQL92后的语法，在其他数据库上并不能通用。

4.聚合

很多情况下，我们需要进行一些汇总操作，比如统计整个公司的人数或者统计每个部门的人数，这是就要用到SQL的集合操作。

SELECT [field1,field2,......fieldn] fun\_name

FROM tablename

[WHERE where\_contition]

[GROUP BY field1,field2,......fieldn

[WITH ROLLUP]]

[HAVING where\_contition];

对其参数进行以下说明：

fun\_name 表示要做的集合操作，也就是聚合函数，常用的有sum（求和）、count(\*)（记录数）、max（最大值）、min（最小值）。

GROUP BY 关键字表示要进行分类聚合的字段，比如要按照部门分类统计员工数量，部门就应该写在group by后面。

WITH ROLLUP 是可选语法，表明是否对分类聚合后的结果进行再汇总。

HAVING 关键字表示对分类后的结果再进行条件的过滤。

注意：having和where的区别在于having是对聚合后的结果进行条件的过滤，而where是在聚合前就对记录进行过滤，如果逻辑允许，我们尽可能用where先过滤记录，这样因为结果集减小，将对聚合的效率大大提高，最后再根据逻辑看是否用having进行再过滤。

5.表连接

表连接分为内连接和外连接，它们之间的最主要区别是：内连接仅选出两张表中互相匹配的记录，而外连接会选出其他不匹配的记录。我们常用的是内连接。

外连接有分为左连接和右连接，具体定义如下：

左连接：包含所有的左边表中的记录甚至是右边表中没有和它匹配的记录

右连接：包含所有的右边表中的记录甚至是左边表中没有和它匹配的记录

SELECT ename,deptname FROM emp LEFT JOIN dept ON emp.deptno=dept.deptno;

 6.子查询

某些情况，当我们查询的时候，需要的条件是另一个SELECT语句的结果，这个时候，就要用到子查询。  
用于子查询的关键字主要包括in、not in、=、!=、exists、not exists等。

SELECT \* FROM emp WHERE deptno IN(SELECT deptno FROM dept);

注意：子查询和表连接之间的转换主要应用在两个方面：

MySQL 4.1 以前的版本不支持子查询，需要用表连接来实现子查询的功能  
表连接在很多情况下用于优化子查询

7.记录联合

我们经常会碰到这样的应用，将两个表的数据按照一定的查询条件查询出来后，将结果合并到一起显示出来，这个时候，就需要用到union和union all关键字来实现这样的功能，具体语法如下：

SELECT \* FROM t1

UNION|UNION ALL

SELECT \* FROM t2

......

UNION|UNION ALL

SELECT \* FROM tn;

UNION 和 UNION ALL的主要区别是 UNION ALL 是把结果集直接合并在一起，而UNION是将UNION ALL后的结果进行一次DISTINCT，去除重复记录后的结果。

9.5 高频面试

● 关系型数据库和非关系型数据库有哪些区别？

● 常见的关系型和非关系型数据库有哪些？

第10讲 Node操作MySQL

|  |  |  |  |
| --- | --- | --- | --- |
| 知识模块 | 课程内容 | 重点 | 课程目标 |
| Node操作MySQL | Node操作MySQL实现CRUD | ★ | 掌握：Node操作MySQL的操作流程  掌握：掌握MVC和MVP的概念 |
| 封装Db.js文件 | ★ |
| MVC和MVP介绍使用 | ★ |

10.1 Node操作MySQL

10.1.1 安装mysql模块支持

npm install mysql

10.1.2 Node操作MySQL基础案例

var mysql      = require('mysql');

var connection = mysql.createConnection({

  host     : 'localhost',

  user     : 'me',

  password : 'secret',

  database : 'my\_db'

});

connection.connect();

connection.query('SELECT 1 + 1 AS solution', function (error, results, fields) {

if (error) throw error;

console.log('The solution is: ', results[0].solution);

});

connection.end();

上述代码执行过程基本可以分为5个步骤：

1、引入mysql模块

2、配置连接参数

3、发起链接

4、使用query执行CRUD操作

5、关闭连接

注意：在connection 实例上执行的方法：connect(),query(),end()；是按序进行的；它内部维护一个执行队列，保证执行的方法能按照调用次序执行。

10.1.3 connection建立连接

常见连接方式有如下两种：

//创建连接

var connection = mysql.createConnection({

  host     : 'localhost',

  user     : 'me',

  password : 'secret',

  database : 'my\_db'

});

//方式1：

connection.connect();

//方式2（推荐做法）：

connection.connect(function(err) {

if (err) {

console.error('error connecting: ' + err.stack);

return;

}

console.log('connected as id ' + connection.threadId);

});

10.1.4 connection常用连接参数

● host：数据库连接的hostname，默认为localhost

● port：MySQL连接端口，默认为3306

● user：数据库的连接的用户名

● password：数据库连接密码

● database：需要连接的数据库名字

● charset：连接的编码方式

● connectTimeout：连接超时时间，默认为10000毫秒

根据上述连接参数，还可以使用url路径的形式来建立连接，如下：

var connection = mysql.createConnection('mysql://user:pass@host/db?debug=true&charset=BIG5\_CHINESE\_CI&timezone=-0700'

);

10.1.5 connection 终止连接

终止连接的方法有两种：end()和destroy()，用法如下：

connection.end()

connection.end(function(err) {

   // The connection is terminated now

});

或

connection.destroy();

这两个方法的主要区别是end方法在调用过程中如果发生错误，会有一个回调方法执行，而destroy则没有回调方法。

10.1.6 CRUD执行

10.1.6.1 query的使用

mysql模块使用connection.query方法来执行CURD操作，使用形式如下：

方式一：

connection.query(sqlString, callback)

//第一个参数查询语句，第二个参数查询后的回调函数

connection.query('SELECT \* FROM `books` WHERE `author` = "David"', function (error, results, fields) {

  // error will be an Error if one occurred during the query

  // results will contain the results of the query

  // fields will contain information about the returned results fields (if any)

});

方式二（可以防止sql注入攻击）：

connection.query(sqlString, values,callback)

//第一个参数查询语句(带？占位符)，占位符对应的参数，,第二个参数查询后的回调函数

connection.query('SELECT \* FROM `books` WHERE `author` = ?', ['David'], function (error, results, fields) {

  // error will be an Error if one occurred during the query

  // results will contain the results of the query

  // fields will contain information about the returned results fields (if any)

});

多占位符的情况：

connection.query('UPDATE users SET foo = ?, bar = ?, baz = ? WHERE id = ?', ['a', 'b', 'c', userId], function (error, results, fields) {

  if (error) throw error;

  // ...

});

其中，foo对应a，bar对应b，baz对应c，id对应userId

10.1.6.2 新增数据

var usr={name:'zhangsan',password:'pwdzhangsan',mail:'zhangsan@gmail.com'};

connection.query('insert into users set ?', usr, function(err, result) {

if (err) throw err;

console.log('inserted zhangsan');

console.log(result);

console.log('\n');

});

注意：我们在上一章的知识中，学到新增的sql语句为：

insert into table (field1,field2) values (value1,value2)

上面例子中的做法：

insert into table set field1 = value1,field2 = value2

两者的区别如下：

1、values的写法是SQL标准，set的写法是MySQL的扩展（只能用在mysql数据库中使用）

2、从执行效率上来说，使用set语法的语句可以更短，更清晰，以高效率更快写入，并且可以有效避免在编写values子句时混合列顺序导致的拼写错误。

3、在可移植性上面，只在MySQL上支持的set语法明显没有优势，并且在多条语句插入上，做法很鸡肋。

10.1.6.3 更新数据

connection.query('update users set password="ddd" where name="zhangsan"', {password:'ppp'}, function(err, result) {

if (err) throw err;

console.log('updated zhangsan\'s password to ddd');

console.log(result);

console.log('\n');

});

10.1.6.4 删除数据

connection.query('delete from users where name="zhangsan"', {password:'ppp'}, function(err, result) {

if (err) throw err;

console.log('deleted zhangsan');

console.log(result);

console.log('\n');

});

10.1.6.5 查询数据

connection.query('select \* from users', function(err, rows, fields) {

if (err) throw err;

console.log('selected after deleted');

for(var i= 0,usr;usr=rows[i++];){

console.log('user nae='+usr.name + ', password='+usr.password);

}

console.log('\n');

});

10.2 封装Db.js文件

const mysql = require("mysql");

class Db {

constructor({

host = "127.0.0.1",

user = "root",

password = "root",

database = "demo",

port = "3306"

} = {}) {

this.config = {

host,

user,

password,

database,

port

};

this.connection = null;

this.connect();

}

// 连接操作

connect() {

this.connection = mysql.createConnection(this.config);

this.connection.connect(err => {

if (err) {

throw err;

// console.error("error connecting: " + err.stack);

// return;

}

});

}

// 解决async...await异常的封装

to(promise) {

this.close(); //关闭连接

return promise

.then(rst => {

return [null, rst];

})

.catch(err => {

return [err];

});

}

// 封装promise操作

operator(sql, args) {

let p = new Promise((resolve, reject) => {

this.connection.query(sql, args, (err, rows) => {

if (err) {

reject(err);

} else {

esolve(rows);

}

});

});

return p;

}

// 通用CRUD操作方法

async query(sql, ...args) {

return await this.to(this.operator(sql, args));

}

// 关闭连接

close() {

this.connection.end();

}

}

module.exports = Db;

// 测试代码

async function test() {

let db = new Db();

let [err, rows] = await db.query("select \* from ??", "user");

console.log(err, rows);

}

test();

10.3 MVC和MVP

10.3.1 MVC

MVC 模式（Model–view–controller）是软件工程中的一种软件架构模式，它把软件系统分为三个基本部分：模型（Model）、视图（View）和控制器（Controller）。

MVC 模式的目的是实现一种动态的程序设计，简化后续对程序的修改和扩展，并且使程序某一部分的重复利用成为可能。除此之外，MVC 模式通过对复杂度的简化，使程序的结构更加直观。软件系统在分离了自身的基本部分的同时，也赋予了各个基本部分应有的功能。

● 模型（Model）：进行数据管理和数据库设计；

● 控制器（Controller）：负责转发请求，对请求进行处理；

● 视图（View）：图形界面设计。

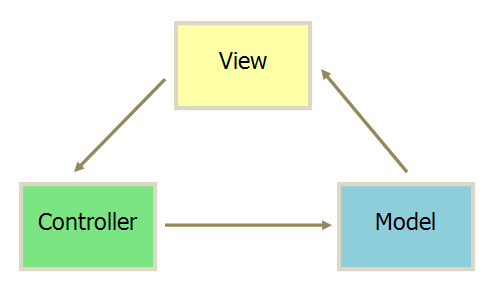


图10-1

1、View 传送指令到 Controller（比如登录或注册按钮的提交）

2、Controller 完成业务逻辑后，要求 Model 改变状态（controller处理登录或注册业务逻辑，然后在model层处理数据）

3、Model 将新的数据发送到 View，用户得到反馈（model层将查询后的结果反馈给用户界面）

备注：所有通信都是单向的。

10.3.2 MVP

MVP(Model–view–presenter ) 是MVC设计模式派生出来的。MVP经常用来创建用户界面。presenter是作为一个“中间人”的角色存在。在MVP中，所有页面显示逻辑都会被推送到presenter处理。

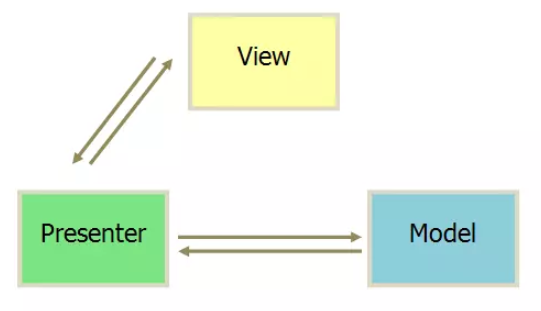


图10-2

● 模型（model）：定义要在用户界面中显示或以其他方式操作的数据的接口。

● 视图（view）：是一个被动界面，它显示数据并将用户命令（例如：表单事件）路由给presenter，以便对该数据进行操作。

● Presenter：对模型和视图进行操作。它从存储库（模型）中检索数据，并格式化数据以便在视图中显示。

10.3.3 MVC和MVP区别

在MVP中View并不直接使用Model，它们之间的通信是通过Presenter (MVC中的Controller)来进行的，所有的交互都发生在Presenter内部，而在MVC中View会直接从Model中读取数据而不是通过 Controller。

在MVC模型里，Model不依赖于View，但是View是依赖于Model的。不仅如此，因为有一些业务逻辑在View里实现了，导致要更改View也是比较困难的，至少那些业务逻辑是无法重用的。

虽然 MVC 中的 View 的确“可以”访问 Model，但是我们不建议在 View 中依赖 Model，而是要求尽可能把所有业务逻辑都放在 Controller 中处理，而 View 只和 Controller 交互，将此进行改进，就有了MVP模式。

在MVP里，Presenter完全把Model和View进行了分离，主要的程序逻辑在Presenter里实现。而且Presenter与具体的View是没有直接关联的，而是通过定义好的接口进行交互，从而使得在变更View时候可以保持Presenter的不变，即重用！

10.4 高频面试

● MVC和MVP的优缺点？

第11讲 模块化和Webpack

|  |  |  |  |
| --- | --- | --- | --- |
| 知识模块 | 课程内容 | 重点 | 课程目标 |
| 模块化与webpack | ES6模块化基本介绍 | ★ | 掌握：ES6的模块化  掌握：webpack基本使用  掌握：webpack常用模块和插件 |
| Webpack的基本使用 | ★ |
| Webpack常用模块和插件 | ★ |

11.1 webpack简介

本质上，webpack 是一个现代 JavaScript 应用程序的静态模块打包器(module bundler)。当 webpack 处理应用程序时，它会递归地构建一个依赖关系图(dependency graph)，其中包含应用程序需要的每个模块，然后将所有这些模块打包成一个或多个 bundle。

webpack作为前端自动构建工具，应用非常广泛，很多框架的脚手架都是基于webpack的，比如vue、angular、reactjs等框架。

11.2 webpack概念

在webpack中，所有的静态资源都可以被处理为一个模块，包括js、图片、css、字体。Webpack的打包过程：

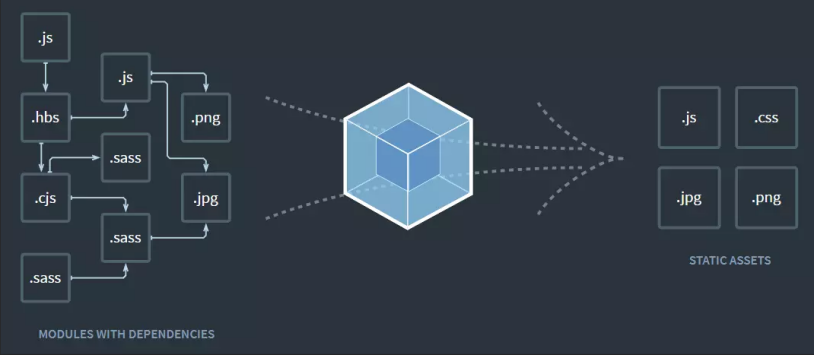


图11-1

webpack中最重要的概念有以下几个：

● Entry, 工程的入口文件配置

● Output, 打包的输出的文件配置

● Chunk, webpack处理和输出的包

● Loaders, 加载器，用于处理各种不同类型的模块，可扩展

● Plugins, 插件，在webpack打包过程中不同时机执行一些任务，比如清除打包目录、复制静态文件、抽取css文件

● Mode, 区分开发环境和生成环境

11.2.1 Entry入口

Entry，顾名思义就是工程的入口文件，Entry的配置写法有三种：

●对象，可配置多入口，可配置chunk名，灵活可扩展，最常用,一个属性就是一个entry chunk：

module.exports = {

entry: {

app: "./src/app.js",

vendors: "./src/vendors.js"

}

};

●字符串, 最简单直接方式，单入口，chunk名默认为main

module.exports = {

entry: "./path/to/my/entry/file.js"

};

●数组, 多入口，将多个入口文件打包为一个chunk，chunk名默认为main

module.exports = {

entry: ["./path/to/my/entry/file.js", "./path/to/my/entry/file1.js"]

};

对象语法会比较繁琐。然而，这是应用程序中定义入口的最可扩展的方式。

对象写法配置告诉我们 webpack 从 app.js 和 vendors.js 开始创建依赖图。这些依赖图是彼此完全分离、互相独立的。这种方式比较常见于，只有一个入口起点（不包括 vendor，vendor一般都是动态加载的第三方模块。动态加载的模块不是入口起点。）的单页应用程序(single page application)中。不过，为了支持提供更佳 vendor 分离能力的 DllPlugin。 官方现在不太建议将第三方模块放到entry.vendors中。

11.2.2 Output出口

Output用于配置打包输出的文件，包括输出文件的文件名、输出路径、静态资源地址，这里列出最常用的4种：

module.exports = {

entry: {

app: "./src/app.js",

search: "./src/search.js"

},

output: {

filename: "js/[name].js",

chunkFilename: "js/[name].js",

path: \_\_dirname + "/dist",

publicPath: "http://cdn.example.com/assets/[hash]/"

}

};

配置项如下：

● filename: 配置输出文件名，可添加路径配置(例子中js/)，可使用占位符，占位符有以下5种：

1）name: chunk名，在该例子中就是app和search

2）hash: 模块标识符的hash值，跟工程内容相关

3）chunkhash: chunk内容的hash值，只和当前chunk内容相关,可用于缓存设置

4）id: 模块标识符

5）query：模块查询参数，取文件名中?后面的内容

● path: 文件的输出路径，必须是绝对地址

● publicPath: 用于设置打包过程中产生的静态文件的最终引用地址，静态文件的最终引用地址为output.publicPath + output.filename，很多时候，你的静态文件放置在CDN上，通过publicPath就可以很方便地设置。如果你的静态引用地址在运行时才能确定，可以在入口文件中设置\_\_webpack\_public\_path\_\_来决定publicPath的值:

\_\_webpack\_public\_path\_\_ = myRuntimePublicPath;

● chunkFilename: 用于设置非entry入口的chunk的输出文件名，非entry入口的chunk一般在动态引入和CommonsChunkPlugin中产生，这是一个Plugin，用于抽取公共代码或者进行代码分割等操作，该plugin已经在webpack4中废除，由webpac4内置的optimization.splitChunks替代。

11.2.3 loader

Loaders可以理解为不同类型模块的处理器，将这些类型的模块处理为浏览器可运行和识别的代码。比如babel-loader将es6以上代码转换为es5代码；sass-loader将sass代码解析为css代码；url-loader和file-loader可以将图片、字体等静态文件解析为base64码或者静态文件地址。Loaders给我们提供了处理模块的入口，在里面可以使用全部的js功能，从而使webpack具有了强大而灵活的能力。

11.2.4 plugins

plugins用来在一些合适的时机执行一些特定的任务，比如代码分割、静态资源处理、环境变量的注入、将所有css的module抽取为单个文件等。webpack自身也是用插件系统构建起来的。

11.2.5 mode

webpack4新增了mode配置。webpack会根据mode值自动帮你做一个不同的优化:

* production（默认值）

在DefinePlugin中将process.env.NODE\_ENV设置为production；默认启用了如下插件：

FlagDependencyUsagePlugin, FlagIncludedChunksPlugin, ModuleConcatenationPlugin, NoEmitOnErrorsPlugin, OccurrenceOrderPlugin, SideEffectsFlagPlugin and UglifyJsPlugin

* evelopment:

在DefinePlugin中将process.env.NODE\_ENV设置为development

默认启用了如下插件：NamedChunksPlugin, NamedModulesPlugin

● none: 什么都不做

11.3 webpack的安装和配置

第一步：新建一个文件夹，命名为wp。

第二步：在cmd命令行中使用cd命令切入到wp文件夹中，输入npm init或npm init -y，初始化package.json文件；需要注意的是如果package.json的name为webpack可能会引起错误，所以name尽量避免输入为webpack。

如图：

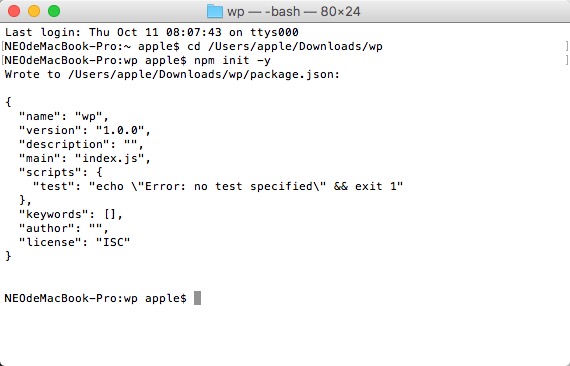


图11-2

第三步：将wp文件夹拖入到sublime等编辑工具中，并在wp根目录下新建一个webpack.config.js的文件。

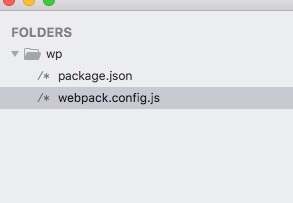


图11-3

webpack.config.js里面应该配置的项目如下：

module.exports = {  
 ssentry: '', // 入口文件

output: {}, // 出口文件

module: {}, // 处理对应模块

plugins: [], // 对应的插件

devServer: {}, // 开发服务器配置

mode: 'development' // 模式配置

}

我们先做一个默认配置：

const path = require('path');

module.exports = {

entry: './src/index.js', // 入口文件

output: {

filename: 'bundle.js', // 打包后的文件名称

path: path.resolve('dist') // 打包后的目录，必须是绝对路径

}

}

及将上述所谓的默认配置代码粘贴到webpack.config.js文件中

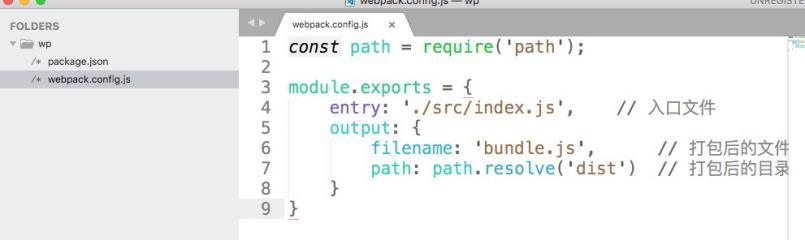


图11-4

解释：  
所谓的入口文件entry和出口文件output指的都是.js文件，当前的配置是单个入口文件和出口文件，后面我们在配置多页面的时候回有多入口和出口文件。

第四步：根据第三步中的配置内容需要注意到两个路径和文件：

entry: './src/index.js',

filename: 'bundle.js'

path: path.resolve('dist')

'./src/index.js'的意思是在当前wp根目录下要新建一个src文件夹，里面，并且在里面新建一个index.js文件。

filename: 'bundle.js'和path: path.resolve('dist')的意思是最终打包完毕后会在wp根目录下自动创建一个叫dist的文件夹，里面包含一个bundle.js文件

打包前目录结构和代码：



图11-5

第五步：接下来我们将根据第四步的内容进行打包，要想使用webpack打包，需要使用npm安装webpack 和 webpack-cli,在cmd命令行中输入：

npm install webpack webpack-cli --save-dev

如图：

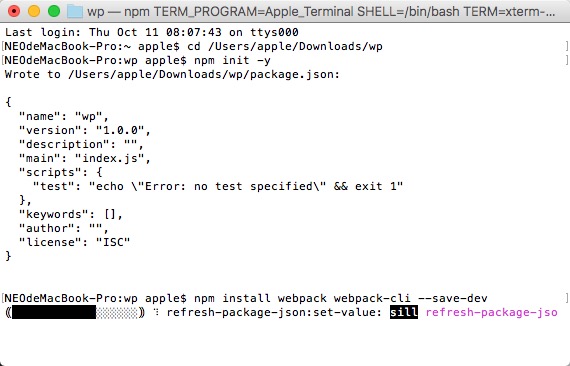


图11-6

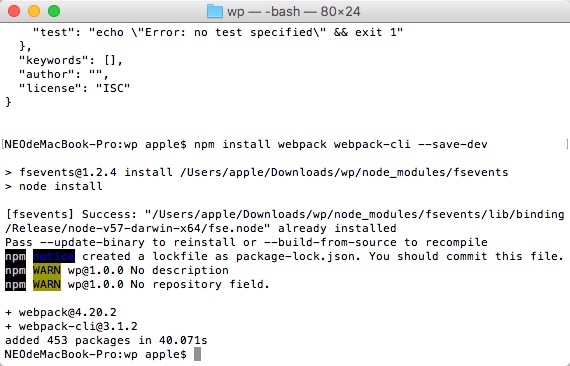


图11-7

安装完毕后修改package.json文件中的scripts，如下：

{

"name": "wp",

"version": "1.0.0",

"description": "",

"main": "index.js",

"scripts": {

"dev" : "webpack"

},

"keywords": [],

"author": "",

"license": "ISC",

"devDependencies": {

"webpack": "^4.20.2",

"webpack-cli": "^3.1.2"

}

}

然后在cmd命令行中输入npm run dev进行打包。

如下图：

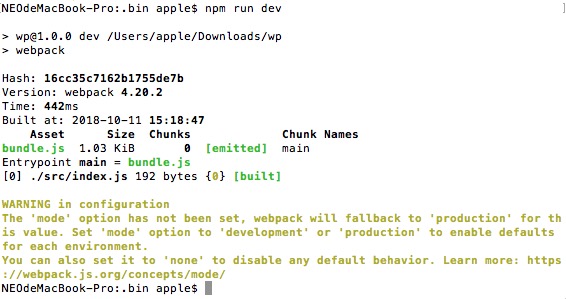


图11-8

打包后目录结构和代码：

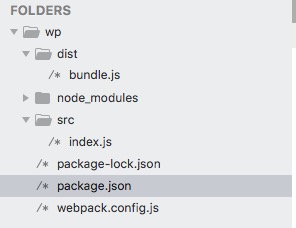


图11-9

其中增加了一个dist文件夹，里面增加了一个bundle.js文件；这就是之前我们说的输出文件。

第六步:在src目录下新建一个index.html文件，里面引入打包后的bundle.js文件作为测试。

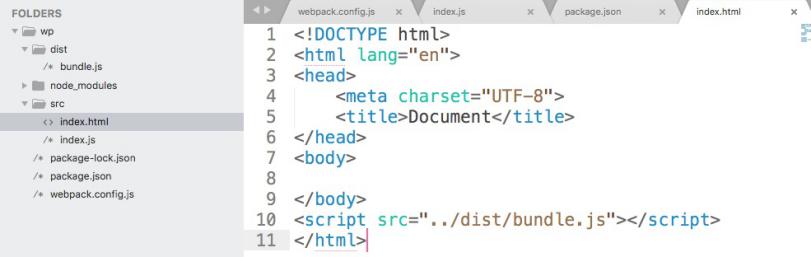


图11-10

运行后的结果如下：

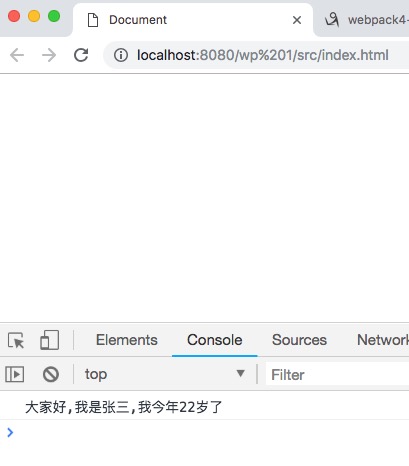


图11-11

至此，基于webpack的项目建立好了，接下来我们准备安装一些常用模块和插件。

11.4 模块和插件配置安装：

11.4.1 配置HTML模板

**简介**

刚才我们建立的index.html文件是在src下面的，但实际打包后的文件是在dist目录下的，我们可以安装一个 html-webpack-plugin 插件,来打包html模板，并将该模板指向对应的入口文件。

**安装**

npm i html-webpack-plugin -D

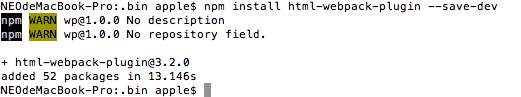


图11-12

**配置**

修改webpack.config.js后如下：

const path = require('path');

const HtmlWebpackPlugin = require('html-webpack-plugin');

module.exports = {

entry: './src/index.js', // 入口文件

output: {

filename: 'bundle.js', // 打包后的文件名称

path: path.resolve('dist') // 打包后的目录，必须是绝对路径

},

plugins: [

// 通过new一下这个类来使用插件

new HtmlWebpackPlugin({

// 在src目录下创建一个index.html页面当做模板来用

template: './src/index.html',

})

]

}

增加了插件引入和插件的配置，仔细对比前面的设置就能知道。

**运行**

运行前修改一下src目录下的index.html代码，取消bundle.js的引用；

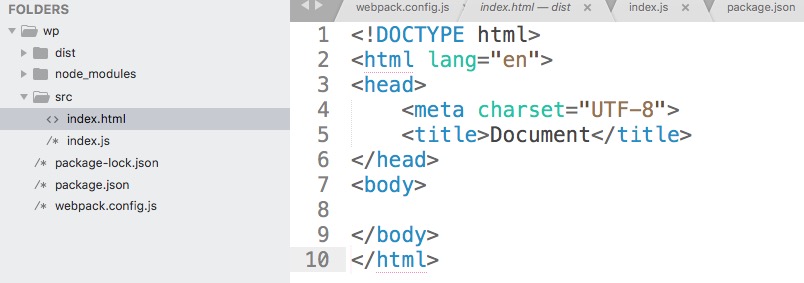


图11-13

再次运行 npm run dev 结果如下：

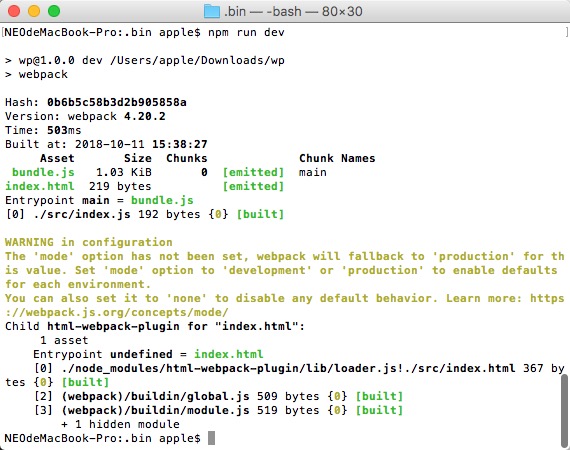


图11-14

在dist目录下自动新增了一个index.html文件，并且里面自动引用了bundle.js，这是因为我们新建的index.html文件是自动指向到index.js入口的；而index.js作为入口文件又最终编译生成为bundle.js。



图11-15

11.4.2 引用CSS文件

**简介**

css文件在打包完成之后也会输出到dist目录下，所以引用css仍然需要模块支持，我们现在讨论的并不是拆分后的css，而是内部样式表；也就是说，css在自动引入到html文件中的时候是以内部样式表的形式出现的。关于拆分css我们后面会讲到。

**安装**

引用css需要安装两个模块:style-loader和css-loader

npm i style-loader css-loader -D

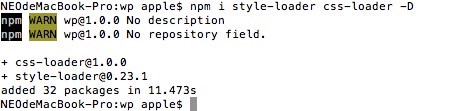


图11-16

其中css-loader是负责解析css的，而style-loader则负责将css以内部样式表的形式嵌入到html中，当使用插件拆分css的时候就不再需要style-loader了，因为那是以外部样式表link到html文件中的。

**配置**

修改webpack.config.js，增加内容如下：

module: {

rules: [

{

test: /\.css$/, // 解析css

use: ['style-loader', 'css-loader'] // 从右向左解析

/\*

也可以这样写，这种方式方便写一些配置参数

use: [

{loader: 'style-loader'},

{loader: 'css-loader'}

]

\*/

}

]

}

修改后的webpack.config.js全部内容如下：

const path = require('path');

const HtmlWebpackPlugin = require('html-webpack-plugin');

module.exports = {

entry: './src/index.js', // 入口文件

output: {

filename: 'bundle.js', // 打包后的文件名称

path: path.resolve('dist') // 打包后的目录，必须是绝对路径

},

plugins: [

// 通过new一下这个类来使用插件

new HtmlWebpackPlugin({

// 在src目录下创建一个index.html页面当做模板来用

template: './src/index.html',

})

],

module: {

rules: [

{

test: /\.css$/, // 解析css

use: ['style-loader', 'css-loader'] // 从右向左解析

/\*

也可以这样写，这种方式方便写一些配置参数

use: [

{loader: 'style-loader'},

{loader: 'css-loader'}

]

\*/

}

]

}

}

**运行**

在运行前我们需要在src目录下新建一个css目录，并新建index.css，目录代码如下：

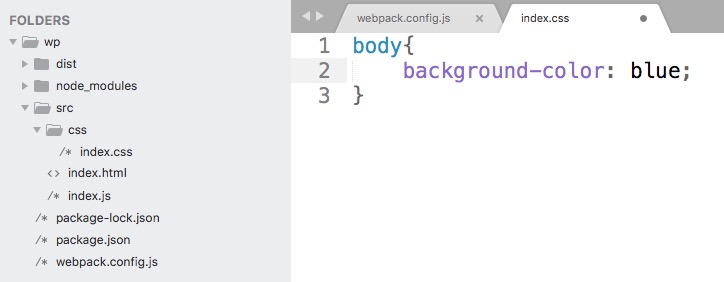


图11-17

修改index.js，增加：

import "./css/index.css";



图11-18

运行 npm run dev后的目录代码如下：

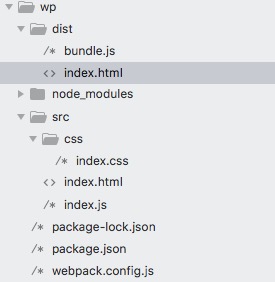


图11-19

目录中并没有增加css文件夹和任何css文件，但是运行dist里面的index.html文件会得到下图效果。

页面效果如下：

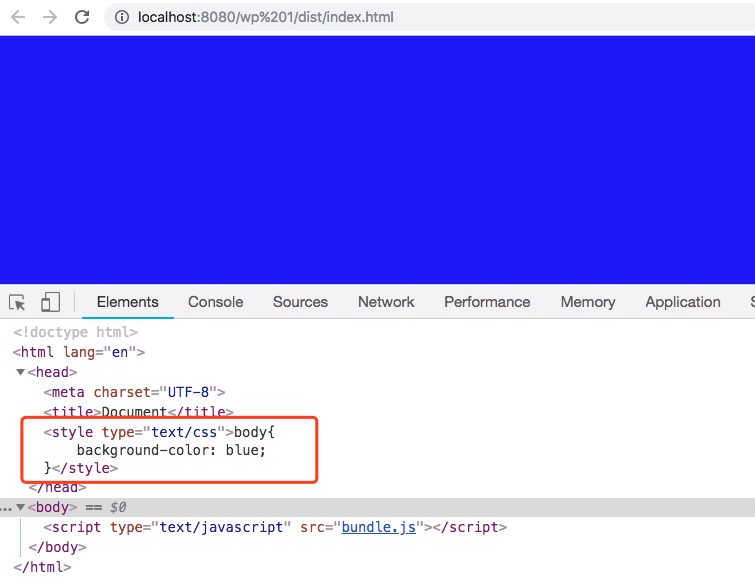


图11-20

这是使用style-loader打包的结果，内部样式表。

11.4.3 拆分CSS文件

**简介**

通过使用extract-text-webpack-plugin插件将刚才的内部样式表拆分为外部样式表

**安装**

npm i extract-text-webpack-plugin@next -D

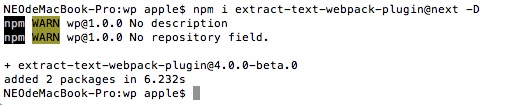


图11-21

**配置**

修改webpack.config.js里面的module：

module: {

rules: [

{

test: /\.css$/,

use: ExtractTextWebpackPlugin.extract({

// 将css用link的方式引入就不再需要style-loader了

use: 'css-loader'

})

}

]

},

新增：

const ExtractTextWebpackPlugin = require('extract-text-webpack-plugin');

新增：

plugins: [

new HtmlWebpackPlugin({

template: './src/index.html',

}),

// 拆分后会把css文件放到dist目录下的css/style.css

new ExtractTextWebpackPlugin('css/style.css')

]

新增new ExtractTextWebpackPlugin('css/style.css') 后，会在dist目录下生产一个css文件夹，且src目录下的css最终会合并到这个style.css文件中，如果不想叫style.css，可以修改。

修改完毕后的webpack.config.js如下：

const path = require('path');

const HtmlWebpackPlugin = require('html-webpack-plugin');

const ExtractTextWebpackPlugin = require('extract-text-webpack-plugin');

module.exports = {

entry: './src/index.js', // 入口文件

output: {  
 filename: 'bundle.js', // 打包后的文件名称

path: path.resolve('dist') // 打包后的目录，必须是绝对路径

},

plugins: [

new HtmlWebpackPlugin({

template: './src/index.html',

}),

// 拆分后会把css文件放到dist目录下的css/style.css

new ExtractTextWebpackPlugin('css/style.css')

],

module: {  
 rules: [  
 {  
 test: /\.css$/,  
 use: ExtractTextWebpackPlugin.extract({

// 将css用link的方式引入就不再需要style-loader了

use: 'css-loader'

})

}

]

}

}

**运行**

运行 npm run dev后的目录代码如下：

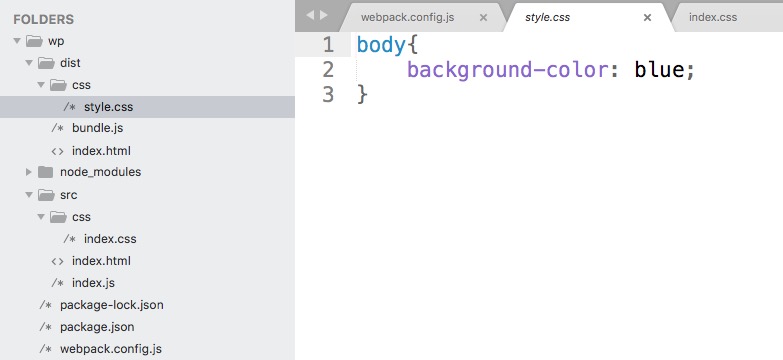


图11-22

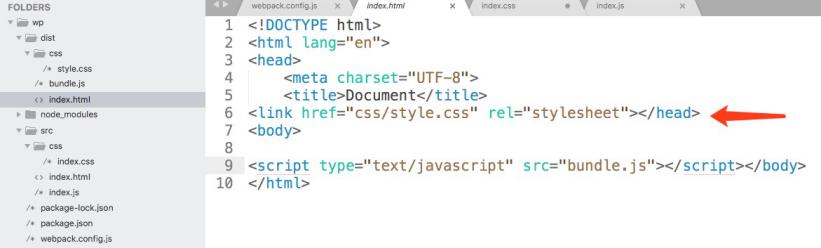


图11-23

11.4.4 引用less文件

**简介**

less作为前端css编译工具广受欢迎，它在前端的使用方式有两种，一种是引入less.js文件来动态编译less代码，这必然是降低了程序的响应时间；第二种就是使用工具或lessc来手动编译转换less代码为css；我们现在要做的事情是在webpack这个自动化工具中配置less，让它变得自动起来。

**安装**

npm install less less-loader --save-dev

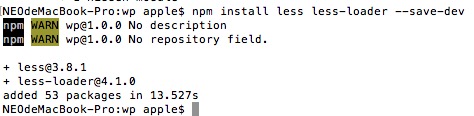


图11-23

**配置**

修改webpack.config.js，增加：

{

test:/\.less$/,

use:ExtractTextPlugin.extract({

fallback:"style-loader",

use:[{

loader:"css-loader"

},{

loader:"less-loader"

}]

})

}

修改完毕后的webpack.config.js如下：

const path = require('path');

const HtmlWebpackPlugin = require('html-webpack-plugin');

const ExtractTextWebpackPlugin = require('extract-text-webpack-plugin');

module.exports = {

entry: './src/index.js', // 入口文件

output: {

filename: 'bundle.js', // 打包后的文件名称

path: path.resolve('dist') // 打包后的目录，必须是绝对路径

},

plugins: [

// 通过new一下这个类来使用插件

new HtmlWebpackPlugin({

// 在src目录下创建一个index.html页面当做模板来用

template: './src/index.html',

}),

new ExtractTextWebpackPlugin('css/style.css')

],

module: {

rules: [

{

test: /\.css$/,

use: ExtractTextWebpackPlugin.extract({

// 将css用link的方式引入就不再需要style-loader了

use: 'css-loader'

})

},

{

test:/\.less$/,

use:ExtractTextWebpackPlugin.extract({

fallback:"style-loader",

use:[{

loader:"css-loader"

},{

loader:"less-loader"

}]

})

}

]

}

}

**运行**

运行前需要做的修改如下：

修改src目录下的index.html文件，新增一个class为less的div

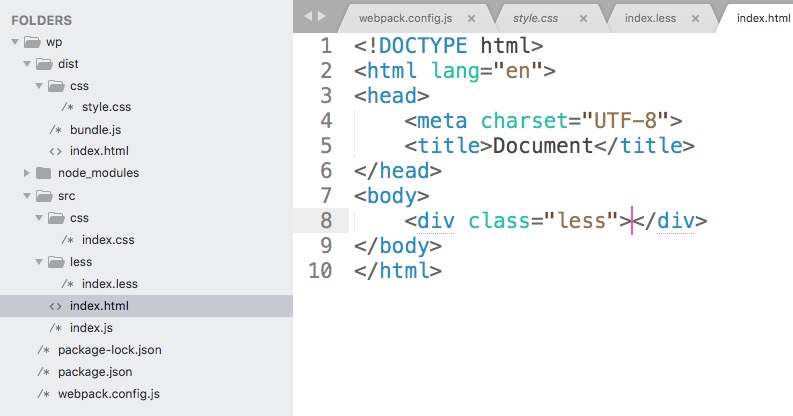


图11-24

在src目录下新建一个less文件夹，并新建index.less文件

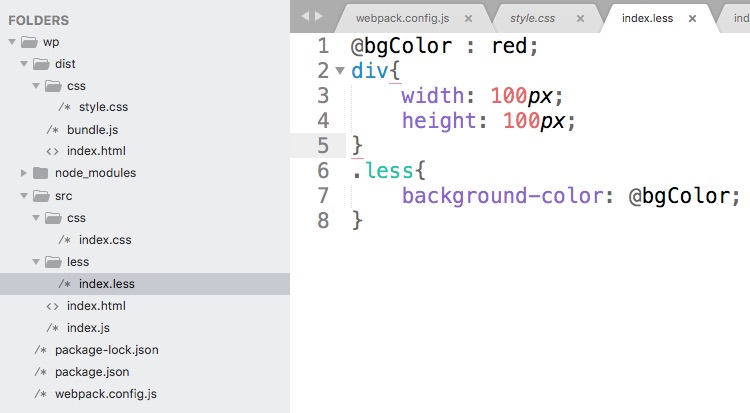


图11-25

在index.js中引入less文件夹下的index.less文件



图11-26

运行npm run dev后的目录代码如下：

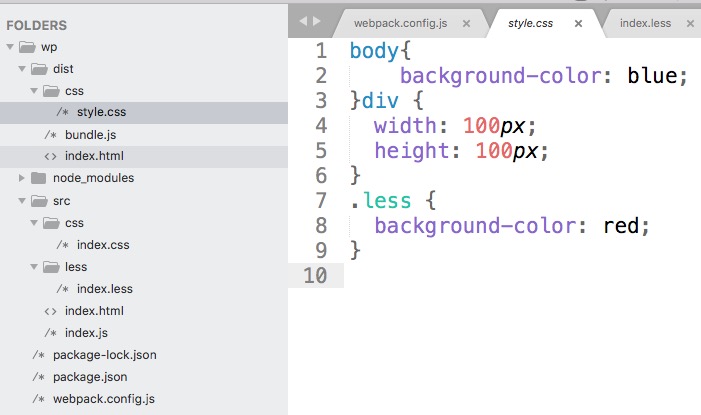


图11-27

这里的style.css是将index.css和index.less的内容合并后的结果，至于如何拆分开，我们在多入口文件的时候再说。

11.4.5 CSS自动添加前缀

**简介**

我们在css或less、sass文件中常常会编写一些CSS3代码，有些时候这些css3代码需要指明私有前缀，这无疑增加了很多工作，可以使用 postcss-loader 插件来帮我们自动增加。

**安装**

npm install --save-dev postcss-loader autoprefixer

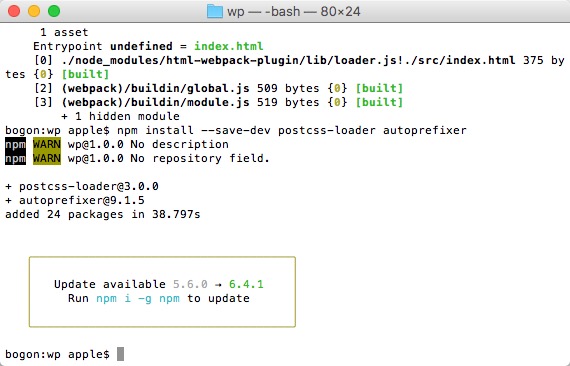


图11-28

**配置**

postcss-loader的配置不同于前面的几个文件，它需要在wp根目录下新建一个postcss.config.js文件，并在里面添加如下代码：

module.exports = {

plugins: [require('autoprefixer')] // 引用该插件即可了

}



图11-29

然后对需要的css或less、sass模块做出修改，比如你需要在css中增加css3前缀的功能，那就在css模块中增加postcss-loader，如下：

{

test: /\.css$/, // 解析css

use: ExtractTextWebpackPlugin.extract({

use:[{

loader:"css-loader"

},{

loader:"postcss-loader"

}],

})

},

less模块和sass模块操作同理。

修改后的webpack.config.js代码如下：

const path = require('path');

const HtmlWebpackPlugin = require('html-webpack-plugin');

const ExtractTextWebpackPlugin = require('extract-text-webpack-plugin');

module.exports = {  
 entry: './src/index.js', // 入口文件

output: {

filename: 'bundle.js', // 打包后的文件名称

path: path.resolve('dist') // 打包后的目录，必须是绝对路径

},

plugins: [

// 通过new一下这个类来使用插件

new HtmlWebpackPlugin({

// 在src目录下创建一个index.html页面当做模板来用

template: './src/index.html',

}),

new ExtractTextWebpackPlugin('css/style.css')

],

module: {

rules: [{

test: /\.css$/,

use: ExtractTextWebpackPlugin.extract({

// 将css用link的方式引入就不再需要style-loader了

// use: 'css-loader'

use:[{

loader:"css-loader"

},{

loader:"postcss-loader"

}],

})

},

{

test:/\.less$/,

use:ExtractTextWebpackPlugin.extract({

fallback:"style-loader",

use:[{

loader:"css-loader"

},{

loader:"less-loader"

}]

})

},

{

test: /\.scss$/, // 解析less

// use: ['style-loader', 'css-loader','sass-loader'] // 从右向左解析

use: ExtractTextWebpackPlugin.extract({

// 将css用link的方式引入就不再需要style-loader了

use:[{

loader:"css-loader"

},{

loader:"sass-loader"

}]

})

},

]

}

}

**运行**

运行前修改src/css/index.css文件，增加了transform css3代码，如下：

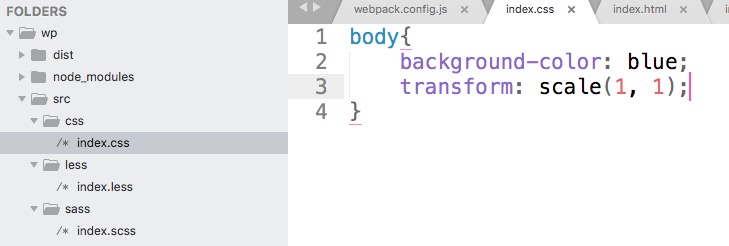


图11-30

运行 npm run dev 后的目录代码如下：

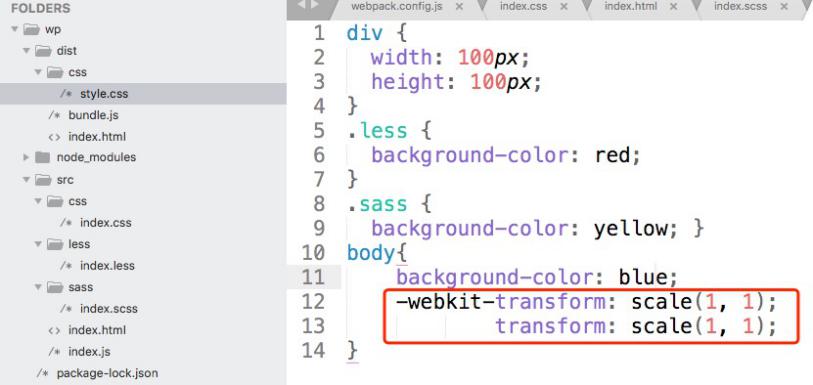


图11-31

刚刚修改的css3代码，自动增加了私有前缀。

11.4.6 CSS中图片的引用处理

**简介**

css中引用图片主要引用file-loader和url-loader，其中file-loader主要解决引用路径问题，url-loader主要解决图片的加载优化问题，可以设置一个阈值，比如将8k以下的图片转换为data-url的形式直接引用到html文件中，从而减少请求次数，达到优化目的。

**安装**

npm install --save-dev file-loader url-loader

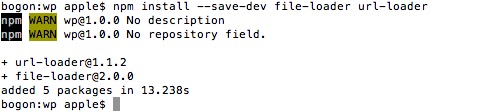


图11-32

**配置**

修改webpack.config.js，增加如下代码：

{

test: /\.(jpe?g|png|gif)$/,

use: [

{

loader: 'url-loader',

options: {

limit: 8192, // 小于8k的图片自动转成base64格式，并且不会存在实体图片

outputPath: 'images/' // 图片打包后存放的目录

}

}

]

},

并增加红框中的内容



图11-33

修改后的webpack.config.js文件如下：

const path = require('path');

const HtmlWebpackPlugin = require('html-webpack-plugin');

const ExtractTextWebpackPlugin = require('extract-text-webpack-plugin');

module.exports = {

entry: './src/index.js', // 入口文件

output: {

filename: 'bundle.js', // 打包后的文件名称

path: path.resolve('dist') // 打包后的目录，必须是绝对路径

},

plugins: [

// 通过new一下这个类来使用插件

new HtmlWebpackPlugin({

// 在src目录下创建一个index.html页面当做模板来用

template: './src/index.html',

}),

new ExtractTextWebpackPlugin('css/style.css')

],

module: {

rules: [

{

test: /\.css$/,

use: ExtractTextWebpackPlugin.extract({

// 将css用link的方式引入就不再需要style-loader了

// use: 'css-loader'

use:[{

loader:"css-loader"

},{

loader:"postcss-loader"

}],

publicPath: '../'

})

},

{

test:/\.less$/,

use:ExtractTextWebpackPlugin.extract({

fallback:"style-loader",

use:[{

loader:"css-loader"

},{

loader:"less-loader"

}]

})

},

{

test: /\.scss$/, // 解析less

// use: ['style-loader', 'css-loader','sass-loader'] // 从右向左解析

use: ExtractTextWebpackPlugin.extract({

// 将css用link的方式引入就不再需要style-loader了

use:[{

loader:"css-loader"

},{

loader:"sass-loader"

}]

})

},

{

test: /\.(jpe?g|png|gif)$/,

use: [

{

loader: 'url-loader',

options: {

limit: 8192, // 小于8k的图片自动转成base64格式，并且不会存在实体图片

outputPath: 'images/' // 图片打包后存放的目录

}

}

]

},

]

}

}

**运行**

运行前在src目录下新建image文件夹，在文件夹当中放一个图片做测试，如图：

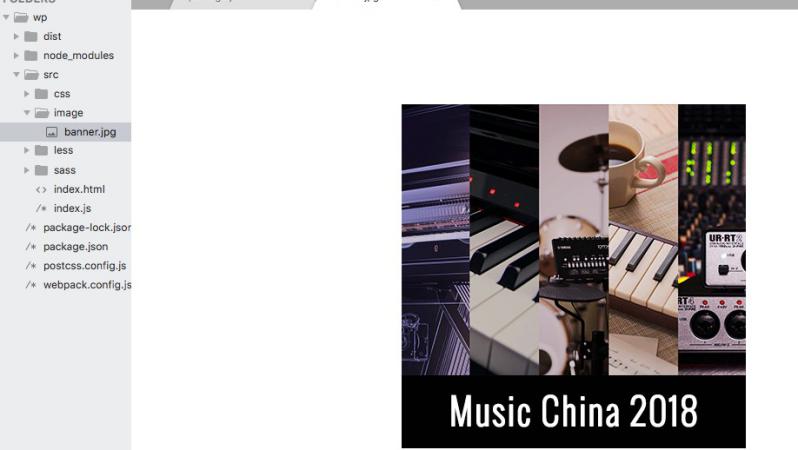


图11-34

修改src目录下的index.html文件：

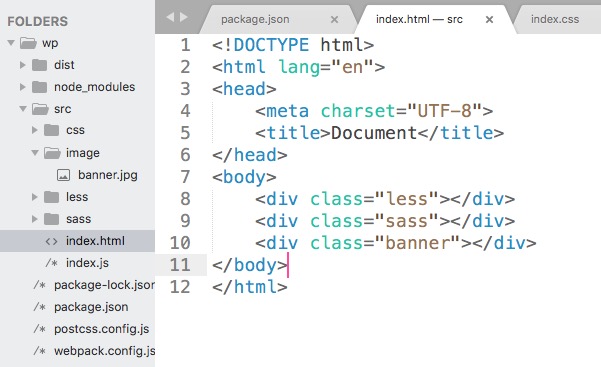


图11-35

修改src/css/index.css文件

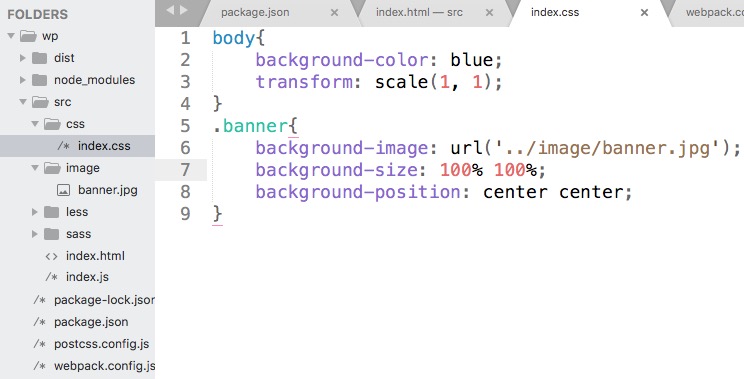


图11-36

运行 npm run dev后的效果为：



图11-37

11.4.7 HTML中图片的引用处理

**简介**

同css中的图片处理，在html文件中使用img标签引用图片同样存在类似问题，需要安装 html-withimg-loader 模块

**安装**

npm install --save-dev html-withimg-loader

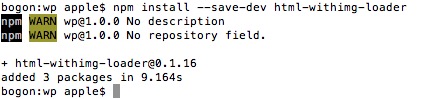


图11-38

**配置**

修改webpack.config.js文件，新增：

{

test: /\.(htm|html)$/,

use: 'html-withimg-loader'

}

**运行**

修改src目录下的 index.html文件新增img标签

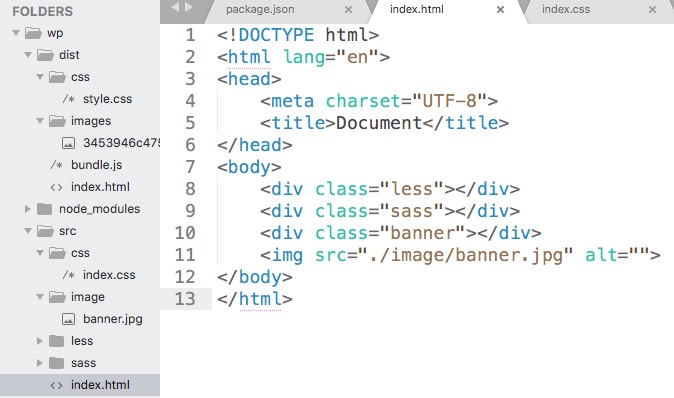


图11-39

运行 npm run dev 后的效果如图：

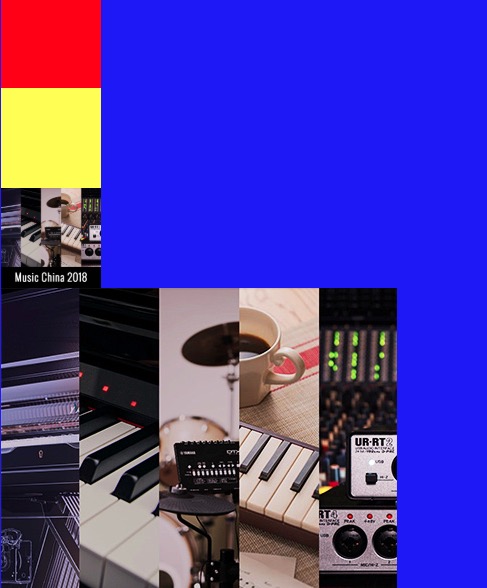


图11-40

11.4.8 iconfont字体引用处理

**简介**

原理同上

**安装**

不需要安装，因为已经安装过file-loader了。

**配置**

修改webpack.config.js文件，新增：

{

test: /\.(eot|ttf|woff|svg)$/,

use: 'file-loader'

},

**运行**

拷贝一个iconfont字体图标库到src/css文件夹下，如图：

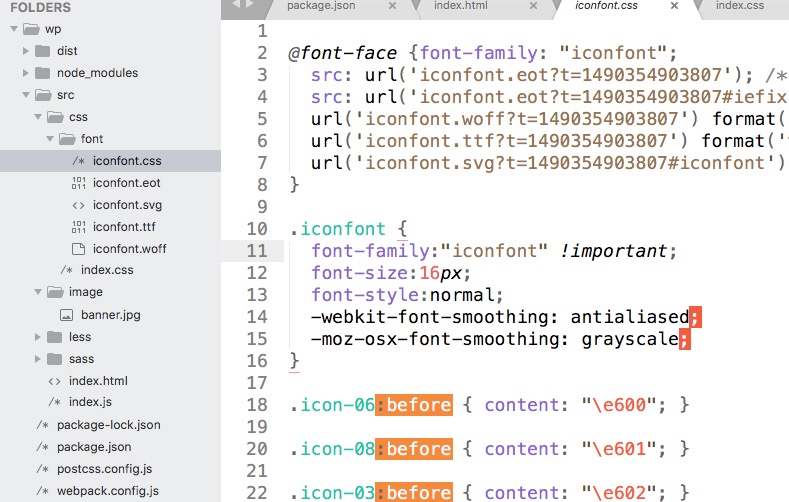


图11-41

在src/index.html文件中新增 span标签如下：

<span class="iconfont">&#xe62c;</span>

在src/index.js文件新增 iconfont.css引入如下：

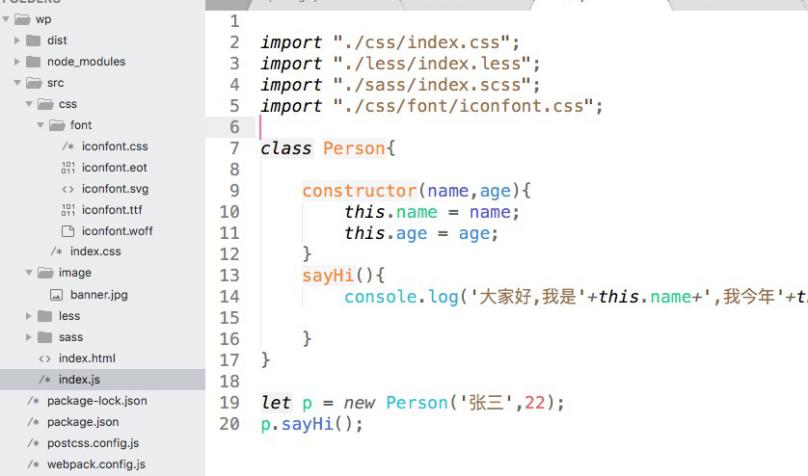


图11-42

运行 npm run dev 后的效果如下：



图11-43

页面中将增加一个麦当劳的M标志，证明成功使用了字体图标。

11.4.9 安装Babel转码ES6

**简介**

ES6因为包含广泛，ES2015,2016,2017等，使用中对ES6的转码是必须的，所以我们采用了babel来转码。

**安装**

npm install --save-dev babel-core babel-loader babel-preset-react babel-preset-env

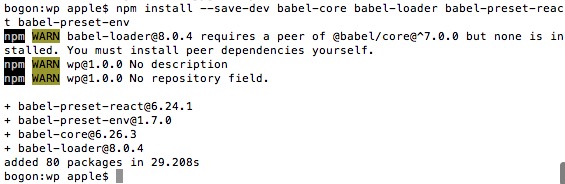


图11-44

这个安装过程可能比较慢，也可能会出现一些关于Parse , Json等的解析错误，出现类似的错误请按照下面两个方法解决：

第一：

运行 ： npm cache clean –force 之后再运行上面的安装命令。

第二：

如果问题没有顺利解决，运行：npm install --registry=https://registry.npm.taobao.org --loglevel=silly

之后再运行安装命令。

**配置**

修改webpack.config.js，新增：

{

test:'/\.(js|jsx)$/',

use: 'bable-loader',

include: /src/, // 只转化src目录下的js

exclude: /node\_modules/ // 排除掉node\_modules，优化打包速度

}

**运行**

运行前，在wp根目录下新建.babelrc文件

type nul>.babelrc

并在里面新增如下代码：

{

"presets": ["env", "stage-0"] // 从右向左解析

}

目录代码如下：



图11-45

运行 npm run dev后观察代码的变化，事实上没有安装babel的时候,webpack也可以解码es6，但考虑到es6代表广泛，所以还是安装了babel解码。

11.4.10 JS文件压缩

在webpack.config.js中新增：

mode: 'development'

表示开发环境，不压缩代码。

mode: 'production'

表示生产环境，压缩代码。



图11-46

11.4.11 CSS文件压缩

**简介**

CSS代码压缩需要安装插件：optimize-css-assets-webpack-plugin

**安装**

npm install optimize-css-assets-webpack-plugin --save-dev

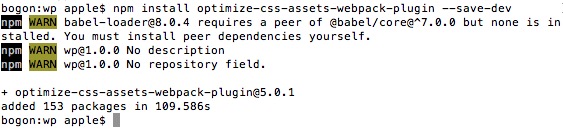


图11-47

**配置**

在webpack.config.js中新增

const OptimizeCssAssetsPlugin = require('optimize-css-assets-webpack-plugin');

和

new OptimizeCssAssetsPlugin({

assetNameRegExp: /\.css$/g,

cssProcessor: require('cssnano'),

cssProcessorOptions: { safe: true, discardComments: { removeAll: true } },

canPrint: true

})

目录代码如图：



图11-48

**运行**

运行 npm run dev 后的目录代码如下：



图11-49

11.4.12 HTML文件压缩

html代码压缩不用再安装额外的插件和模块，修改webpack.config.js中的html插件为：

new HtmlWebpackPlugin({

template: './src/index.html',

//filename: 'index.html',

//chunks: ['index'], // 对应关系,index.js对应的是index.html

minify: {

removeAttributeQuotes:true,

removeComments: true,

collapseWhitespace: true,

removeScriptTypeAttributes:true,

removeStyleLinkTypeAttributes:true

}

}),

目录代码如图：

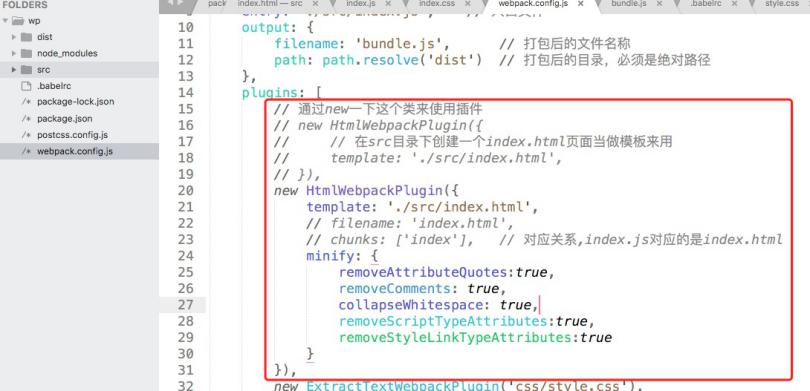


图11-50

运行 npm run dev后可见dist目录下的index.html已经压缩了。

11.4.13 安装JQuery

**简介**

jquery作为前端必需的第三方类库，安装是它是一个示范作用，其他的第三方类库安装可以参考jquery 的安装。

**安装**

npm install jquery --save-dev

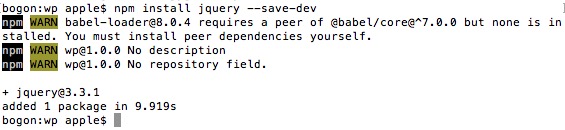


图11-51

**配置**

jquery的引用要用到webpack，所以修改webpack.config.js，新增：

const Webpack = require('webpack');

还要新增：

new Webpack.ProvidePlugin({

$:'jquery'

}),

备注：如果项目还引用了Bootstrap，那jquery的配置还需要修改为如下：

new Webpack.ProvidePlugin({

$:'jquery',

jQuery:'jquery'

}),

修改后的webpack.config.js代码如下：

const path = require('path');

const HtmlWebpackPlugin = require('html-webpack-plugin');

const ExtractTextWebpackPlugin = require('extract-text-webpack-plugin');

const OptimizeCssAssetsPlugin = require('optimize-css-assets-webpack-plugin');

const Webpack = require('webpack');

module.exports = {

mode: 'development',//production

entry: './src/index.js', // 入口文件

output: {

filename: 'bundle.js', // 打包后的文件名称

path: path.resolve('dist') // 打包后的目录，必须是绝对路径

},

plugins: [

// 通过new一下这个类来使用插件

// new HtmlWebpackPlugin({

// // 在src目录下创建一个index.html页面当做模板来用

// template: './src/index.html',

// }),

new HtmlWebpackPlugin({

template: './src/index.html',

//filename: 'index.html',

//chunks: ['index'], // 对应关系,index.js对应的是index.html

minify: {

removeAttributeQuotes:true,

removeComments: true,

collapseWhitespace: true,

removeScriptTypeAttributes:true,

removeStyleLinkTypeAttributes:true

}

}),

new ExtractTextWebpackPlugin('css/style.css'),

new OptimizeCssAssetsPlugin({

assetNameRegExp: /\.css$/g,

cssProcessor: require('cssnano'),

cssProcessorOptions: { safe: true, discardComments: { removeAll: true } },

canPrint: true

}),

new Webpack.ProvidePlugin({

$:'jquery'

})

],

module: {

rules: [

{

test: /\.css$/,

use: ExtractTextWebpackPlugin.extract({

// 将css用link的方式引入就不再需要style-loader了

// use: 'css-loader'

use:[{

loader:"css-loader"

},{

loader:"postcss-loader"

}],

publicPath: '../'

})

},

{

test:/\.less$/,

use:ExtractTextWebpackPlugin.extract({

fallback:"style-loader",

use:[{

loader:"css-loader"

},{

loader:"less-loader"

}]

})

},

{

test: /\.scss$/, // 解析less

// use: ['style-loader', 'css-loader','sass-loader'] // 从右向左解析

use: ExtractTextWebpackPlugin.extract({

// 将css用link的方式引入就不再需要style-loader了

use:[{

loader:"css-loader"

},{

loader:"sass-loader"

}]

})

},

{

test: /\.(jpe?g|png|gif)$/,

use: [

{

loader: 'url-loader',

options: {

limit: 8192, // 小于8k的图片自动转成base64格式，并且不会存在实体图片

outputPath: 'images/' // 图片打包后存放的目录

}

}

]

},

{

test: /\.(htm|html)$/,

use: 'html-withimg-loader'

},

{

test: /\.(eot|ttf|woff|svg)$/,

use: 'file-loader'

},

{

test:'/\.(js|jsx)$/',

use: 'bable-loader',

include: /src/, // 只转化src目录下的js

exclude: /node\_modules/ // 排除掉node\_modules，优化打包速度

}

]

},

}

**运行**

运行前修改src目录下的index.js文件，新增：

$('div').css('border', '10px solid white');

目录代码如下：



图11-52

运行npm run dev后的效果如下：

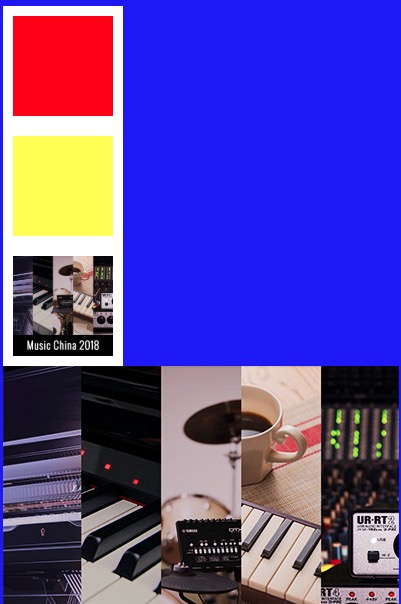


图11-53

11.4.14 热更新

**简介**

每次运行和更新代码都输入npm run dev命令太麻烦，安装热更新来解决该问题，每次改动代码后，只需要保存代码并刷新页面即可，不用再执行命令。

**安装**

npm install webpack-dev-server --save-dev

配置

修改webpack.config.js 新增插件：

new Webpack.HotModuleReplacementPlugin(),//调用webpack的热更新插件

新增热更新配置：

devServer: {

// 设置服务器访问的基本目录

contentBase:path.resolve(\_\_dirname,'dist'), //最好设置成绝对路径

// 设置服务器的ip地址,可以是localhost

host:'localhost',

// 设置端口

port:8090,

// 设置自动打开浏览器

open:true,

// 设置热更新

hot:true

}

修改后的webpack.config.js如下：

const path = require('path');

const HtmlWebpackPlugin = require('html-webpack-plugin');

const ExtractTextWebpackPlugin = require('extract-text-webpack-plugin');

const Webpack = require('webpack');

const OptimizeCssAssetsPlugin = require('optimize-css-assets-webpack-plugin');

const glob = require("glob");

var webpackConfig = {

mode: 'development',//production

// entry: './src/index.js', // 单页面入口文件

// 多页面配置

entry: {

index: './src/index.js',

login: './src/login.js'

},

// 出口文件

output: {

filename: '[name].js',

path: path.resolve('dist')

},

plugins: [

new ExtractTextWebpackPlugin('css/[name].css'),

new Webpack.ProvidePlugin({

$:'jquery'

}),

new Webpack.HotModuleReplacementPlugin(),//调用webpack的热更新插件

new OptimizeCssAssetsPlugin({

assetNameRegExp: /\.css$/g,

cssProcessor: require('cssnano'),

cssProcessorOptions: { safe: true, discardComments: { removeAll: true } },

canPrint: true

})

],

module: {

rules: [

{

test: /\.css$/, // 解析css

// use: ['style-loader', 'css-loader'] // 从右向左解析

use: ExtractTextWebpackPlugin.extract({

// 将css用link的方式引入就不再需要style-loader了

// use: 'css-loader',

use:[{

loader:"css-loader"

},{

loader:"postcss-loader"

}],

publicPath: '../'

})

},

{

test: /\.less$/, // 解析less

// use: ['style-loader', 'css-loader','less-loader'] // 从右向左解析

use: ExtractTextWebpackPlugin.extract({

// 将css用link的方式引入就不再需要style-loader了

use:[{

loader:"css-loader"

},{

loader:"less-loader"

}]

})

},

{

test: /\.scss$/, // 解析less

// use: ['style-loader', 'css-loader','sass-loader'] // 从右向左解析

use: ExtractTextWebpackPlugin.extract({

// 将css用link的方式引入就不再需要style-loader了

use:[{

loader:"css-loader"

},{

loader:"sass-loader"

}]

})

},

{

test: /\.(jpe?g|png|gif)$/,

use: [

{

loader: 'url-loader',

options: {

limit: 8192, // 小于8k的图片自动转成base64格式，并且不会存在实体图片

outputPath: 'images/' // 图片打包后存放的目录

}

}

]

},

{

test: /\.(htm|html)$/,

use: 'html-withimg-loader'

},

{

test: /\.(eot|ttf|woff|svg)$/,

use: 'file-loader'

},

{

test:'/\.(js|jsx)$/',

use: 'bable-loader',

include: /src/, // 只转化src目录下的js

exclude: /node\_modules/ // 排除掉node\_modules，优化打包速度

}

]

},

devServer: {

// 设置服务器访问的基本目录

contentBase:path.resolve(\_\_dirname,'dist'), //最好设置成绝对路径

// 设置服务器的ip地址,可以是localhost

host:'localhost',

// 设置端口

port:8090,

// 设置自动拉起浏览器

open:true,

// 设置热更新

hot:true

}

}

function getView(globPath){

let files = glob.sync(globPath);

let entries = {},

entry, dirname, basename, pathname, extname;

files.forEach(item => {

entry = item;

dirname = path.dirname(entry);//当前目录

extname = path.extname(entry);//后缀

basename = path.basename(entry, extname);//文件名

pathname = path.join(dirname, basename);//文件路径

if (extname === '.html') {

entries[pathname] = './' + entry;

} else if (extname === '.js') {

entries[basename] = entry;

}

});

return entries;

}

let pages = Object.keys(getView('./src/\*html'));

pages.forEach((pathname) => {

let htmlname = pathname.split('src/')[1];

let conf = {

filename: `${htmlname}.html`,

template: `${pathname}.html`,

// hash: true,

chunks:[htmlname],

minify: {

removeAttributeQuotes:true,

removeComments: true,

collapseWhitespace: true,

removeScriptTypeAttributes:true,

removeStyleLinkTypeAttributes:true

}

}

webpackConfig.plugins.push(new HtmlWebpackPlugin(conf));

});

module.exports = webpackConfig;

修改wp根目录下的package.json文件：

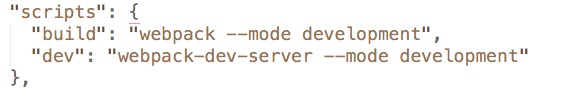


图11-54

修改后的package.json如下：

{

"name": "wp",

"version": "1.0.0",

"description": "",

"main": "index.js",

"scripts": {

"build": "webpack --mode development",

"dev": "webpack-dev-server --mode development"

},

"keywords": [],

"author": "",

"license": "ISC",

"devDependencies": {

"autoprefixer": "^9.1.5",

"babel-core": "^6.26.3",

"babel-loader": "^8.0.4",

"babel-preset-env": "^1.7.0",

"babel-preset-stage-0": "^6.24.1",

"css-loader": "^1.0.0",

"extract-text-webpack-plugin": "^4.0.0-beta.0",

"file-loader": "^2.0.0",

"html-webpack-plugin": "^3.2.0",

"html-withimg-loader": "^0.1.16",

"jquery": "^3.3.1",

"less-loader": "^4.1.0",

"node-sass": "^4.9.3",

"optimize-css-assets-webpack-plugin": "^5.0.1",

"postcss-loader": "^3.0.0",

"sass-loader": "^7.1.0",

"style-loader": "^0.23.1",

"url-loader": "^1.1.2",

"webpack": "^4.20.2",

"webpack-cli": "^3.1.2",

"webpack-dev-server": "^3.1.9"

}

}

**运行**

运行 npm run dev 后的cmd命令行效果如下：

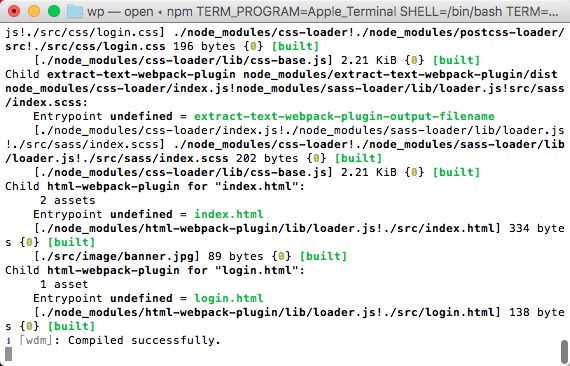


图11-55

11.5 模块化简介及萌芽时代

11.5.1 简介

前端开发中，最初只是在script标签中嵌入几十上百行代码就能实现一些基本的交互效果，后来js得到重视，应用也广泛起来，jQuery，Ajax，Node.Js，MVC，MVVM等的助力也使得前端开发得到重视，也使得前端项目越来越复杂，然而，JavaScript却没有为组织代码提供任何明显帮助，甚至没有类的概念，更不用说模块（module）了，那么什么是模块呢？

一个模块就是实现特定功能的文件，有了模块，我们就可以更方便地使用别人的代码，想要什么功能，就加载什么模块。模块开发需要遵循一定的规范，否则项目中的代码将会变得难以维护。

11.5.2 模块萌芽时代

● 全局function模式

模块就是实现特定功能的一组方法。

只要把不同的函数（以及记录状态的变量）简单地放在一起，就算是一个模块。

function m1(){

　　//...

}

function m2(){

　　//...

}

上面的函数m1()和m2()，组成一个模块。使用的时候，直接调用就行了。

这种做法的缺点很明显："污染"了全局变量，无法保证不与其他模块发生变量名冲突，而且模块成员之间看不出直接关系。

● namespace模式

有module1.js和module2.js两个模块

module1.js

let myModule = {

data: 'ujiuye.com',

foo() {

console.log(`foo() ${this.data}`)

},

bar() {

console.log(`bar() ${this.data}`)

}

}

module2.js

let myModule2 = {

data: 'offcn.com',

foo() {

console.log(`foo() ${this.data}`)

},

bar() {

console.log(`bar() ${this.data}`)

}

}

<script type="text/javascript">

myModule.foo()

myModule.bar()

myModule2.foo()

myModule2.bar()

myModule.data = 'other data' //能直接修改模块内部的数据

myModule.foo()

</script>

说明：

namespace模式: 简单对象封装

作用: 减少了全局变量

问题: 不安全

● IIFE模式

使用"立即执行函数"（Immediately-Invoked Function Expression，IIFE），可以达到不暴露私有成员的目的。

module3.js

(function (window) {

//数据

let data = 'atguigu.com'

//操作数据的函数

function foo() { //用于暴露私有函数

console.log(`foo() ${data}`)

}

function bar() {//用于暴露私有函数

console.log(`bar() ${data}`)

otherFun() //内部调用

}

function otherFun() { //内部私有的函数

console.log('otherFun()')

}

//暴露行为

window.myModule = {foo, bar}

})(window)

test3.html

<script type="text/javascript" src="module3.js"></script>

<script type="text/javascript">

myModule.foo()

myModule.bar()

//myModule.otherFun() //myModule.otherFun is not a function

console.log(myModule.data) //undefined 不能访问模块内部数据

myModule.data = 'xxxx' //不是修改的模块内部的data

myModule.foo() //没有改变

</script>

说明:

IIFE模式: 匿名函数自调用(闭包)

IIFE : immediately-invoked function expression(立即调用函数表达式)

作用: 数据是私有的, 外部只能通过暴露的方法操作

问题: 如果当前这个模块依赖另一个模块怎么办?

● IIFE模式增强

引入jquery到项目中

module4.js

(function (window, $) {

//数据

let data = 'ujiuye.com'

//操作数据的函数

function foo() { //用于暴露有函数

console.log(`foo() ${data}`)

$('body').css('background', 'blue')

}

function bar() {//用于暴露有函数

console.log(`bar() ${data}`)

otherFun() //内部调用

}

function otherFun() { //内部私有的函数

console.log('otherFun()')

}

//暴露行为

window.myModule = {foo, bar}

})(window, jQuery)

test4.html

<script type="text/javascript" src="jquery.js"></script>

<script type="text/javascript" src="module4.js"></script>

<script type="text/javascript">

myModule.foo()

</script>

说明

IIFE模式增强 : 引入依赖

这就是现代模块实现的基础

● 页面加载多个js的问题

页面:

<script type="text/javascript" src="m1.js"></script>

<script type="text/javascript" src="m2.js"></script>

<script type="text/javascript" src="m3.js"></script>

<script type="text/javascript" src="m4.js"></script>

说明：

一个页面需要引入多个js文件

问题:请求过多、依赖模糊、难以维护这些问题可以通过现代模块化编码和项目构建来解决

11.6 Commonjs规范

Commonjs规范的出发点：JS没有模块系统、标准库较少、缺乏包管理工具；为了让JS可以在任何地方运行，以达到Java、C#、PHP这些后台语言具备开发大型应用的能力。

在CommonJs规范中：一个文件就是一个模块，拥有单独的作用域，普通方式定义的变量、函数、对象都属于该模块内，通过require来加载模块，通过exports和modul.exports来暴露模块中的内容。所有代码都运行在模块作用域，不会污染全局作用域；模块可以多次加载，但只会在第一次加载的时候运行一次，然后运行结果就被缓存了，以后再加载，就直接读取缓存结果；模块的加载顺序，按照代码的出现顺序是同步加载的。

11.6.1 后端演示

● 创建项目结构

|-modules

|-module1.js

|-module2.js

|-module3.js

|-app.js

|-package.json

{

"name": "commonJS-demo",

"version": "1.0.0"

}

● 模块化编码

module1.js

module.exports = {

foo() {

console.log('moudle1 foo()')

}

}

module2.js

module.exports = function () {

console.log('module2()')

}

module3.js

exports.foo = function () {

console.log('module3 foo()')

}

exports.bar = function () {

console.log('module3 bar()')

}

app.js

1. 定义暴露模块:

module.exports = value;

exports.xxx = value;

2. 引入模块:

var module = require(模块名或模块路径);

//引用模块

let module1 = require('./modules/module1')

let module2 = require('./modules/module2')

let module3 = require('./modules/module3')

let fs = require('fs')

//使用模块

module1.foo()

module2()

module3.foo()

module3.bar()

fs.readFile('app.js', function (error, data) {

console.log(data.toString())

})

● 通过node运行app.js

命令: node app.js

11.6.2 浏览器端使用Commonjs规范

browserify是一个编译工具,通过它可以在浏览器环境下像nodejs一样使用遵循commonjs规范的模块化编程。我们可以使用browserify来组织代码,也可以使用第三方模块,不需要会nodejs,只需要用到node来编译,用到npm来安装包。browserify模块化的用法和node是一样的,所以npm上那些原本仅仅用于node环境的包,在浏览器环境里也一样能用。现在npm上有越来越多的包,在设计的时候就是想好既能在node环境下用,也能在浏览器环境下用的，甚至还有很多包就是给浏览器环境使用的。npm是为所有的javascript服务的,无论前端后端。

● 创建项目结构

|-js

|-dist //打包生成文件的目录

|-src //源码所在的目录

|-module1.js

|-module2.js

|-module3.js

|-app.js //应用主源文件

|-index.html

|-package.json

{

"name": "browserify-demo",

"version": "1.0.0"

}

● 下载browserify

全局: npm install browserify -g

局部: npm install browserify --save-dev

● 定义模块代码

module1.js

module.exports = {

foo() {

console.log('moudle1 foo()')

}

}

module2.js

module.exports = function () {

console.log('module2()')

}

module3.js

exports.foo = function () {

console.log('module3 foo()')

}

exports.bar = function () {

console.log('module3 bar()')

}

app.js (应用的主js)

//引用模块

let module1 = require('./module1')

let module2 = require('./module2')

let module3 = require('./module3')

//使用模块

module1.foo()

module2()

module3.foo()

module3.bar()

● 打包处理js:

browserify js/src/app.js -o js/dist/bundle.js

● 页面使用引入:

<script type="text/javascript" src="js/dist/bundle.js"></script>

11.7 AMD规范

11.7.1 概述

RequireJS是一个工具库，主要用于浏览器端的模块管理。它可以让客户端的代码分成一个个模块，实现异步或动态加载，从而提高代码的性能和可维护性。它的模块管理遵守AMD规范（Asynchronous Module Definition）。

RequireJS的原理是，通过define方法，将代码定义为模块；通过require方法，实现代码的模块加载。

首先，将require.js引入网页，然后就能在网页中进行模块化编程了。

<script data-main="scripts/main" src="scripts/require.js"></script>

上述代码的data-main属性不可省略，用于指定主代码所在的脚本文件，在上例中为scripts子目录下的main.js文件。用户自定义的代码就放在这个main.js文件中。

11.7.2 定义模块

define方法用于定义模块，RequireJS要求每个模块放在一个单独的文件里。

按照是否依赖其他模块，可以分成两种情况。

1.定义独立模块，即所定义的模块不依赖其他模块；

2.非独立模块，即所定义的模块依赖于其他模块。

● 定义独立模块

如果被定义的模块是一个独立模块，不需要依赖任何其他模块，可以直接用define方法生成。

define({

m1: function() {},

m2: function() {},

});

上面代码生成了一个拥有m1、m2两个方法的模块。

另一种等价的写法是，把对象写成一个函数，该函数的返回值就是输出的模块。

define(function () {

return {

m1: function() {},

m2: function() {},

};

});

后一种写法更灵活一些，可以在函数体内写一些模块初始化代码。值得指出的是，define定义的模块可以返回任何值，不限于对象。

● 非独立模块

如果被定义的模块需要依赖其他模块，则define方法必须采用下面的格式。

define(['m1', 'm2'], function(m1, m2) {

...

});

define方法的第一个参数是一个数组，它的成员是当前模块所依赖的模块。比如，[‘m1’, ‘m2’]表示我们定义的这个新模块依赖于m1模块和m2模块，只有先加载这两个模块，新模块才能正常运行。一般情况下，module1模块和module2模块指的是，当前目录下的m1.js文件和m2.js文件，等同于写成[’./m1’, ‘./m2’]。

define方法的第二个参数是一个函数，当前面数组的所有成员加载成功后，它将被调用。它的参数与数组的成员一一对应，比如function(m1, m2)就表示，这个函数的第一个参数m1对应m1模块，第二个参数m2对应m2模块。这个函数必须返回一个对象，供其他模块调用。

define(['m1', 'm2'], function(m1, m2) {

return {

method: function() {

m1.methodA();

m2.methodB();

}

};

});

上面代码表示新模块返回一个对象，该对象的method方法就是外部调用的接口，method方法内部调用了m1模块的methodA方法和m2模块的methodB方法。

11.8 CMD规范

11.8.1 概述

在 Sea.js 中，所有 JavaScript 模块都遵循 CMD（Common Module Definition） 模块定义规范。该规范明确了模块的基本书写格式和基本交互规则。

在 CMD 规范中，一个模块就是一个文件。

11.8.2 define(factory);

define 是一个全局函数，用来定义模块。

define 接受 factory 参数，factory 可以是一个函数，也可以是一个对象或字符串。

factory 为对象、字符串时，表示模块的接口就是该对象、字符串。比如可以如下定义一个 JSON 数据模块：

define({ "foo": "bar" });

也可以通过字符串定义模板模块：

define('I am a template. My name is {{name}}.');

factory 为函数时，表示是模块的构造方法。执行该构造方法，可以得到模块向外提供的接口。factory 方法在执行时，默认会传入三个参数：require、exports 和 module：

define(function(require, exports, module) {

// 模块代码

});

● define define(id?, deps?, factory)

define 也可以接受两个以上参数。字符串 id 表示模块标识，数组 deps 是模块依赖。比如：

define('hello', ['jquery'], function(require, exports, module) {

// 模块代码

});

id 和 deps 参数可以省略。省略时，可以通过构建工具自动生成。

注意：带 id 和 deps 参数的 define 用法不属于 CMD 规范，而属于 Modules/Transport 规范。

● define.cmd Object

一个空对象，可用来判定当前页面是否有 CMD 模块加载器：

if (typeof define === "function" && define.cmd) {

// 有 Sea.js 等 CMD 模块加载器存在

}

11.8.3 require Function

require 是 factory 函数的第一个参数。

● require require(id)

require 是一个方法，接受 模块标识 作为唯一参数，用来获取其他模块提供的接口。

define(function(require, exports) {

// 获取模块 a 的接口

var a = require('./a');

// 调用模块 a 的方法

a.doSomething();

});

注意：在开发时，require 的书写需要遵循一些 简单约定。

● require.async require.async(id, callback?)

require.async 方法用来在模块内部异步加载模块，并在加载完成后执行指定回调。callback 参数可选。

define(function(require, exports, module) {

// 异步加载一个模块，在加载完成时，执行回调

require.async('./b', function(b) {

b.doSomething();

});

// 异步加载多个模块，在加载完成时，执行回调

require.async(['./c', './d'], function(c, d) {

c.doSomething();

d.doSomething();

});

});

注意：require 是同步往下执行，require.async 则是异步回调执行。require.async 一般用来加载可延迟异步加载的模块。

11.8.4 exports Object

exports 是一个对象，用来向外提供模块接口。

define(function(require, exports) {

// 对外提供 foo 属性

exports.foo = 'bar';

// 对外提供 doSomething 方法

exports.doSomething = function() {};

});

除了给 exports 对象增加成员，还可以使用 return 直接向外提供接口。

define(function(require) {

// 通过 return 直接提供接口

return {

foo: 'bar',

doSomething: function() {}

};

});

11.9 ES6模块化规范

在es6中模块功能主要由两个命令构成：export和import。export命令用于规定模块的对外接口，import命令用于输入其他模块提供的功能。

11.9.1 export

一个模块就是一个独立的文件。该文件内部的所有变量，外部无法获取。如果你希望外部能够读取模块内部的某个变量，就必须使用export关键字输出该变量。下面是一个 JS 文件，里面使用export命令输出变量。

// profile.js

export var firstName = 'Michael';

export var lastName = 'Jackson';

export var year = 1958;

上面代码是profile.js文件，保存了用户信息。ES6 将其视为一个模块，里面用export命令对外部输出了三个变量。

export的写法，除了像上面这样，还有另外一种。

// profile.js

var firstName = 'Michael';

var lastName = 'Jackson';

var year = 1958;

export {firstName, lastName, year};

上面代码在export命令后面，使用大括号指定所要输出的一组变量。它与前一种写法（直接放置在var语句前）是等价的，但是应该优先考虑使用这种写法。因为这样就可以在脚本尾部，一眼看清楚输出了哪些变量。

export命令除了输出变量，还可以输出函数或类（class）。

export function multiply(x, y) {

return x \* y;

};

上面代码对外输出一个函数multiply。

需要特别注意的是，export命令规定的是对外的接口，必须与模块内部的变量建立一一对应关系。

// 报错

export 1;

// 报错

var m = 1;

export m;

上面两种写法都会报错，因为没有提供对外的接口。第一种写法直接输出 1，第二种写法通过变量m，还是直接输出 1。1只是一个值，不是接口。正确的写法是下面这样。

// 写法一

export var m = 1;

// 写法二

var m = 1;

export {m};

// 写法三

var n = 1;

export {n as m};

上面三种写法都是正确的，规定了对外的接口m。其他脚本可以通过这个接口，取到值1。它们的实质是，在接口名与模块内部变量之间，建立了一一对应的关系。

同样的，function和class的输出，也必须遵守这样的写法。

// 报错

function f() {}

export f;

// 正确

export function f() {};

// 正确

function f() {}

export {f};

另外，export语句输出的接口，与其对应的值是动态绑定关系，即通过该接口，可以取到模块内部实时的值。

export var foo = 'bar';

setTimeout(() => foo = 'baz', 500);

上面代码输出变量foo，值为bar，500 毫秒之后变成baz。

最后，export命令可以出现在模块的任何位置，只要处于模块顶层就可以。如果处于块级作用域内，就会报错，下一节的import命令也是如此。这是因为处于条件代码块之中，就没法做静态优化了，违背了 ES6 模块的设计初衷。

function foo() {

export default 'bar' // SyntaxError

}

foo()

上面代码中，export语句放在函数之中，结果报错。

11.9.2 import

使用export命令定义了模块的对外接口以后，其他 JS 文件就可以通过import命令加载这个模块。

// main.js

import {firstName, lastName, year} from './profile.js';

function setName(element) {

element.textContent = firstName + ' ' + lastName;

}

上面代码的import命令，用于加载profile.js文件，并从中输入变量。import命令接受一对大括号，里面指定要从其他模块导入的变量名。大括号里面的变量名，必须与被导入模块（profile.js）对外接口的名称相同。

如果想为输入的变量重新取一个名字，import命令要使用as关键字，将输入的变量重命名。

import { lastName as surname } from './profile.js';

import命令输入的变量都是只读的，因为它的本质是输入接口。也就是说，不允许在加载模块的脚本里面，改写接口。

import {a} from './xxx.js'

a = {}; // Syntax Error : 'a' is read-only;

上面代码中，脚本加载了变量a，对其重新赋值就会报错，因为a是一个只读的接口。但是，如果a是一个对象，改写a的属性是允许的。

import {a} from './xxx.js'

a.foo = 'hello'; // 合法操作

上面代码中，a的属性可以成功改写，并且其他模块也可以读到改写后的值。不过，这种写法很难查错，建议凡是输入的变量，都当作完全只读，轻易不要改变它的属性。

import后面的from指定模块文件的位置，可以是相对路径，也可以是绝对路径，.js后缀可以省略。如果只是模块名，不带有路径，那么必须有配置文件，告诉 JavaScript 引擎该模块的位置。

import {myMethod} from 'util';

上面代码中，util是模块文件名，由于不带有路径，必须通过配置，告诉引擎怎么取到这个模块。

注意，import命令具有提升效果，会提升到整个模块的头部，首先执行。

foo();

import { foo } from 'my\_module';

上面的代码不会报错，因为import的执行早于foo的调用。这种行为的本质是，import命令是编译阶段执行的，在代码运行之前。

由于import是静态执行，所以不能使用表达式和变量，这些只有在运行时才能得到结果的语法结构。

// 报错

import { 'f' + 'oo' } from 'my\_module';

// 报错

let module = 'my\_module';

import { foo } from module;

// 报错

if (x === 1) {

import { foo } from 'module1';

} else {

import { foo } from 'module2';

}

上面三种写法都会报错，因为它们用到了表达式、变量和if结构。在静态分析阶段，这些语法都是没法得到值的。

最后，import语句会执行所加载的模块，因此可以有下面的写法。

import 'lodash';

上面代码仅仅执行lodash模块，但是不输入任何值。

如果多次重复执行同一句import语句，那么只会执行一次，而不会执行多次。

import 'lodash';

import 'lodash';

上面代码加载了两次lodash，但是只会执行一次。

import { foo } from 'my\_module';

import { bar } from 'my\_module';

// 等同于

import { foo, bar } from 'my\_module';

上面代码中，虽然foo和bar在两个语句中加载，但是它们对应的是同一个my\_module实例。也就是说，import语句是 Singleton 模式。

目前阶段，通过 Babel 转码，CommonJS 模块的require命令和 ES6 模块的import命令，可以写在同一个模块里面，但是最好不要这样做。因为import在静态解析阶段执行，所以它是一个模块之中最早执行的。下面的代码可能不会得到预期结果。

require('core-js/modules/es6.symbol');

require('core-js/modules/es6.promise');

import React from 'React';

11.10 高频面试

● webpack的作用是什么，谈谈你对它的理解？

前端为什么需要WebPack？

现在的前端网页功能丰富，特别是SPA（single page web application 单页应用）技术流行后，JavaScript的复杂度增加和需要一大堆依赖包，还需要解决SCSS，Less……新增样式的扩展写法的编译工作。所以现代化的前端已经完全依赖于WebPack的辅助了。

现在最流行的三个前端框架，可以说和webpack已经紧密相连，框架官方都推出了和自身框架依赖的webpack构建工具。

React.js+WebPack

Vue.js+WebPack

AngluarJS+WebPack

从此可以看出，无论你前端走那条线，你都要有很强的Webpack知识，才能祝你成为这个框架领域的大牛。

* 什么是WebPack？

WebPack可以看做是模块打包机：它做的事情是，分析你的项目结构，找到JavaScript模块以及其它的一些浏览器不能直接运行的拓展语言（Sass，TypeScript等），并将其转换和打包为合适的格式供浏览器使用。在3.0出现后，Webpack还肩负起了优化项目的责任。

这段话有三个重点：

打包：可以把多个Javascript文件打包成一个文件，减少服务器压力和下载带宽。

转换：把拓展语言转换成为普通的JavaScript，让浏览器顺利运行。

优化：前端变的越来越复杂后，性能也会遇到问题，而WebPack也开始肩负起了优化和提升性能的责任。

● 常用的loader和plugin有哪些？

● webpack与gulp和grunt的区别是什么？

gulp强调的是前端开发的工作流程，我们可以通过配置一系列的task，定义task处理的事务（例如文件压缩合并、雪碧图、启动server、版本控制等），然后定义执行顺序，来让gulp执行这些task，从而构建项目的整个前端开发流程。

webpack是一个前端模块化方案，更侧重模块打包，我们可以把开发中的所有资源（图片、js文件、css文件等）都看成模块，通过loader（加载器）和plugins（插件）对资源进行处理，打包成符合生产环境部署的前端资源。

● 谈谈Webpack的优缺点

● 为什么要进行模块化

目前前端的开发形势就是模块化和组件化；从软件工程学分析来说就是有了更好的可维护性、可复用性等好处；但是前端的主要语言js在es6之前却没有模块化功能，之前有使用require.js和sea.js但是推出es6的模块化之后，es6的模块化使用形式基本统一了。

●es6模块化规范的注意事项

要弄清楚default 默认导出的好处和作用，import \* as XX from ‘XX’的使用场景

● Commonjs模块化规范的注意事项

在没有彻底弄清楚exports和module.exports的区别之前，请使用module.exports

第12讲 WebSocket及KOA2

|  |  |  |  |
| --- | --- | --- | --- |
| 知识模块 | 课程内容 | 重点 | 课程目标 |
| WebSocket和  KOA2 | WebScoket简介 | ★ | 掌握：socket.io的使用  了解：常见web通讯原理  了解：koa2框架的使用  掌握：koa2和express的区别 |
| websocket和webservice的概念区别 | ☆ |
| Socket.io如何实现通讯 | ★ |
| KOA简介 | ★ |
| Koa2脚手架及koa2在项目中的使用 | ☆ |
| Koa2和express的区别 | ★ |

12.1 websocket与服务器和客户端通信设计

12.1.1 http通讯

http(HTTP ,HyperText Transfer Protocol)是互联网上应用最为广泛的一种网络协议，是用于从www服务器传输超文本到本地浏览器的传送协议；

超文本传输协议，是一个基于请求与响应，无状态的，应用层的协议，常基于TCP/IP协议传输数据，互联网上应用最为广泛的一种网络协议,所有的WWW文件都必须遵守这个标准。设计HTTP的初衷是为了提供一种发布和接收HTML页面的方法。

HTTP无法轻松实现实时应用

http协议是无状态的，服务器只会响应来自客户端的请求，但是它与客户端之间不具备持续连接

我们可以非常轻松的捕获浏览器上发生的事件(比如点击)，这个事件可以轻松产生与服务器的数据交互(ajax)。但是，反过来却是不可能的：服务器端发生了一个事件，服务器无法将这个事件的信息实时主动通知它的客户端。只有在客户端查询服务器的当前状态的时候，所发生事件的信息才会从服务器传递到客户端。

http通信传输：

1. 建立TCP连接，进行三次握手

2. Web浏览器向Web服务器发送请求命令

3. Web浏览器发送请求头信息

4. Web服务器应答

5. Web服务器发送应答头信息

6. Web服务器向浏览器发送数据

7. Web服务器关闭TCP连接

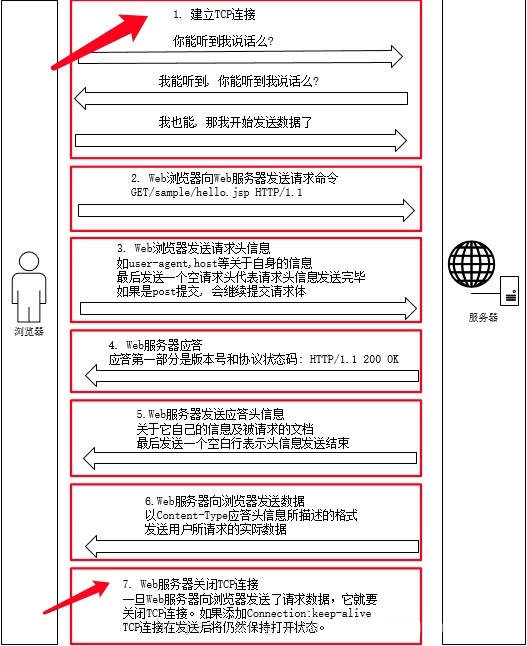


图12-1

在webSocket出现之搭建聊天室使用长轮询和长连接的技术：

● 长轮询：客户端每隔很短的时间，都会对服务器发出请求，查看是否有新的消息，只要轮询速度足够快，例如1秒，就能给人造成交互是实时进行的印象。这种做法是无奈之举，实际上对服务器、客户端双方都造成了大量的性能浪费。

● 长连接：客户端只请求一次，但是服务器会将连接保持，不会返回结果（想象一下我们没有写res.end()时，浏览器一直转小菊花）。服务器有了新数据，就将数据发回来，又有了新数据，就将数据发回来，而一直保持挂起状态。这种做法的也造成了大量的性能浪费。

从上面两种方式中，可以看出是再不断的建立http连接，然后等待服务器处理，可以体现出了http的特点：被动性。即：请求只能由客户端发起，服务器不能主动联系客户端。

12.1.2 webSocket 协议

**WebScoket简介：**

WebSocket协议在2008年诞生，2011年成为国际标准。协议是基于TCP的一种新的网络协议。WebSocket是HTML5开始提供的一种浏览器与服务器间进行全双工通讯的网络技术。依靠这种技术可以实现客户端和服务器端的长连接，双向实时通信。一开始的握手需要借助HTTP请求完成。服务器也能主动通知客户端。

WebSocket的原理非常的简单：利用HTTP请求产生握手，HTTP头部中含有WebSocket协议的请求，所以握手之后，二者转用TCP协议进行交流（QQ的协议）。现在的浏览器和服务器之间，就是QQ和QQ服务器的关系了。

所以WebSocket协议，需要浏览器支持，更需要服务器支持。

下面是HTTP请求响应客户端服务器交互与webSocket请求响应客户端服务器交互的区别：（图示说明）



图12-2

**WebScoket应用场景：**

WebSocket广泛应用于社交聊天、直播、弹幕、多玩家游戏、协同编辑、股票基金实时报价、体育实况更新、视频会议/聊天、基于位置的应用、在线教育、智能家居等需要高实时的场景。

**WebSocket特点：**

它的最大特点就是，服务器可以主动向客户端推送信息，客户端也可以主动向服务器发送信息，是真正的双向平等对话，属于[服务器推送技术](https://en.wikipedia.org/wiki/Push_technology" \t "https://www.cnblogs.com/pengc/p/_blank)的一种。

（1）建立在 TCP 协议之上，服务器端的实现比较容易。

（2）与 HTTP 协议有着良好的兼容性。默认端口也是80和443，并且握手阶段采用 HTTP 协议，因此握手时不容易屏蔽，能通过各种 HTTP 代理服务器。

（3）数据格式比较轻量，性能开销小，通信高效。

（4）可以发送文本，也可以发送二进制数据。

（5）没有同源限制，客户端可以与任意服务器通信。

（6）协议标识符是ws（如果加密，则为wss），服务器网址就是 URL。

12.1.3 webSocket API

**1、webSocket 基本语法**

WebSocket对象提供了一组API，用于创建和管理webSocket连接，以及通过连接发送和接收数据

创建WebSocket对象：

Url:表示要连接的URL，这个URL应该为响应webSocket的地址

Protocols：可以是一个单个的协议名字字符串或者包含多个协议名字字符串的数组

const ws = new WebSocket(url,[protocols])

方法：

close([code],[reason]) 关闭webSocket连接或停止正在进行的连接请求

code:一个数字值表示关闭连接的状态号，表示连接被关闭的原因

reason:一个可读的字符串，表示连接被关闭的原因

send(data) 通过webSocket连接向服务器发送数据

data：一个数字值表示关闭连接的状态号，表示连接被关闭的原因

属性：

onclose：用于监听连接关闭事件监听器，当webSocket对象的readyState状态变为close时会触发该事件，会接收一个close event对象；

onerror：当错误发生时用于监听error事件的事件监听器。会接收一个error event对象

onmessage:一个用于消息事件的事件监听器，这一事件当有消息到达的时候该事件会触发，会接收一个message event对象

onopen:一个用于连接打开事件的事件监听器。当readyState的值变为open的时候会触发该事件。会接收一个open event对象

readyState:连接的当前状态：0 连接还没开启 1连接已开启并准备好进行通信 2连接正在关闭的过程中 3 连接已经关闭，或者连接无法建立

下面我们来使用一下webSocket创建一个连接：

在使用之前要记得，安装 npm install websocket --save；

/\* 服务器端 \*/

var WebSocketServer = require('websocket').server;

var http = require('http');

//创建服务器

var server = http.createServer(function(request, response) {

console.log((new Date()) + ' Received request for ' + request.url);

response.writeHead(404);

response.end();

});

//监听端口号

server.listen(8080, function() {

console.log((new Date()) + ' Server is listening on port 8080');

});

//创建webSocket服务器

wsServer = new WebSocketServer({

httpServer: server

})

//建立连接

wsServer.on('request', function(request) {

//当前的连接

var connection = request.accept(null, request.origin);

console.log((new Date()) + '已经建立连接');

setInterval(function() {

connection.sendUTF("服务端发送信息" + (new Date()));

}, 1000)

//监听当前连接，发送message的时候

connection.on('message', function(message) {

if (message.type === 'utf8') {

console.log('Received Message: ' + message.utf8Data);

connection.sendUTF(message.utf8Data);

}

else if (message.type === 'binary') {

console.log('Received Binary Message of ' + message.binaryData.length + ' bytes');

connection.sendBytes(message.binaryData);

}

});

//监听当前连接，当close关闭

connection.on('close', function(reasonCode, description) {

console.log((new Date()) + ' Peer ' + connection.remoteAddress + ' 断开连接.');

});

});

/\*客户端\*/

// 创建一个socket实例：

var socket = new WebSocket("ws://localhost:8080")

// 打开socket

socket.onopen = (event) => {

// 发送一个初始化消息

socket.send('Hello Server!')

// 服务器有响应数据触发

socket.onmessage = (event) => {

console.log('有新消息', event)

}

//出错时触发， 并且会关闭连接。 这时可以根据错误信息进行按需处理

socket.onerror = (event) => {

console.log('error')

}

// 监听Socket的关闭

socket.onclose = (event) => {

console.log('Client notified socket has closed', event)

}

// 关闭Socket

socket.close(1000, 'closing normally')

}

**2、ws和wss**

我们在上面提到过，创建一个socket实例时可以选填ws和wss来进行通信协议的确定。他们两个其实很像HTTP和HTTPS之间的关系。其中ws表示纯文本通信，而wss表示使用加密信道通信（TCP+TLS）。那为啥不直接使用HTTP而要自定义通信协议呢？这就要从WebSocket的目的说起来，WebSocket的主要功能就是为了给浏览器中的应用与服务器端提供优化的，双向的通信机制，但这不代表WebScoket只能局限于此，它当然还能够用于其他的场景，这就需要他可以通过非HTTP协议来进行数据交换，因此WebSocket也就采用了自定义URI模式，以确保就算没有HTTP，也能进行数据交换。

ws协议：普通请求，占用与HTTP相同的80端口

wss协议：基于SSL的安全传输，占用与TLS相同的443端口

12.1.4 socket.io

Socket.io是一个跨浏览器支持webScoket的实时通讯的JS；它屏蔽了所有底层细节，让顶层调用非常简单。并且还为不支持WebSocket协议的浏览器，提供了长轮询的透明模拟机制。

socket.io的基本应用：

1.先要npm下载这个库

npm install socket.io --save

2.写原生的JS，搭建一个服务器，server创建好之后，创建一个io对象

const app = require('express')();

//读取静态文件

app.use(express.static("./public/"));

let server = app.listen(3000);

const io = require("socket.io")(server);

//监听连接事件

io.on("connection", function(socket) {

console.log("1个客户端连接了");

})

写完这句话之后，你就会发现，http://127.0.0.1:3000/socket.io/socket.io.js 就是一个js文件的地址了;

3.现在需要制作一个index页面，这个页面中，必须引用秘密js文件。调用io函数，取得socket对象。

<body>

<h1>我是index页面，我引用了秘密script文件</h1>

<script type="text/javascript" src="/socket.io/socket.io.js"></script>

<script type="text/javascript">

let socket = io();

</script>

</body>

4.此时，浏览器上读物静态页面。

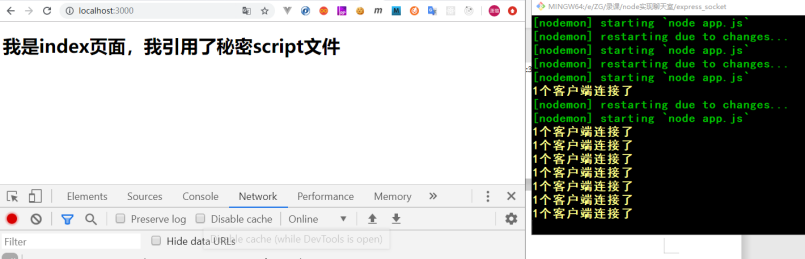


图12-3

至此，服务器和客户端都有socket对象了。服务器的socket对象.

5.服务端

const io = require("socket.io")(server)

//监听连接事件

io.on("connection", function(socket) {

console.log("1个客户端连接了");

socket.on("tiwen", function(msg) {

console.log("本服务器得到了一个提问" + msg);

socket.emit("font\_end", "向前端发送值");

});

})

每一个连接上来的用户，都有一个socket。 由于我们的emit语句，是socket.emit()发出的，所以指的是向这个客户端发出语句。

6.客户端

<script type="text/javascript">

let socket = io();

socket.emit("tiwen", "我是前台传过去的值")

socket.on("font\_end", data => {

console.log("客户端得到了服务端的值：", data)

})

</script>

socket.io常用基础语法：

1.emit和on 语法：

emit 和 on 是最重要的两个api，分别对应 发送 和 监听 事件。

● socket.emit(eventName[, ...args])：发射（触发）一个事件

● socket.on(eventName, callback)：监听一个 emit 发射的事件

我们可以非常自由的在服务端定义并发送一个事件emit，然后在客户端监听 on，反过来也一样。

发送的内容格式也非常自由，既可以是基本数据类型 Number，String，Boolean 等，也可以是Object，Array 类型，甚至还可以是函数。而用回调函数的方式则可以进行更便携的交互。

2.broadcast 广播

broadcast 默认是向所有的socket连接进行广播，但是不包括发送者自身，如果自己也打算接收消息的话，需要给自己单独发送。

io.on('connection', socket => {

const data="后台发送来的值"

socket.broadcast.emit('userin',data); //广播向所有socket连接

socket.emit('userin',data); //给自己也发一份

});

3.namespace 命名空间

隔离作用域，或者划分业务模块，namespace 是个有效的法子。namespace 相当于建立新的频道，可以在一个 socket.io 服务上面隔离不同的连接，事件和中间件。

默认的连接也是有namespace的，那就是 /；

方法一：直接在链接后面加子域名，这种其实用的还是同一个 sokcet 服务进程，可以看成是软隔离吧。

/\*\*\* 客户端 \*\*/

//默认的namespace

const socket = io('http://127.0.0.1:3001');

// mypath

const socket = io('http://127.0.0.1:3001/mypath', { forceNew: true });

/\*\*\* 服务端 \*\*/

//默认的namespace

io.on('connection', socket => {

});

// mypath

io.of('/mypath').on('connection', socket => {

});

方法二：path 参数，这种就是实打实的重新起了一个 socket 服务了。

/\*\*\* 客户端 \*\*/

const socket = io('http://localhost', {

path: '/mypath'

});

/\*\*\* 服务端 \*\*/

// 另外重新起socket服务

const io = require('socket.io')({

path: '/mypath'

});

4.Rooms

每一个socket连接都会有一个独一无二的标志，那就是 socket.id，我们就是通过id来区分不同连接的。除此之外，socket.id 本身也是房间 room 的标志，通俗讲，每个socket 连接自身都拥有一间房 room。那么我们就可以给这个 room 发送消息，还有如果你加入了房间，就能接受到房间里的广播信息。当然你可以自定义 room ，让socket连接加入或离开。还有如果 socket 断开连接，也就是 disconnect 后，它会被自动移出room。

而这就是实现 单独聊天 和 群组聊天 的基础，来看一下对应的api。

● socket.join(rooms[, callback])：加入房间

● socket.leave(room[, callback]) ：离开房间

● socket.to(room)： 给房间发送消息

// 自定义room

io.on('connection', socket =>{

socket.join('some room')); // 加入房间

socket.leave('some room'); // 离开房间

});

// 向房间里的所有客户端发送消息

io.to('some room').emit('some event');

// 默认房间（每一个id一个room）

socket.on('say to someone', (id, msg) => {

socket.broadcast.to(id).emit('my message', msg);

});

12.2 Koa2框架

Koa -- 基于 Node.js 平台的下一代 web 开发框架

Koa 是一个新的 web 框架，由 Express 幕后的原班人马打造， 致力于成为 web 应用和 API 开发领域中的一个更小、更富有表现力、更健壮的基石。 通过利用 async 函数，Koa 帮你丢弃回调函数，并有力地增强错误处理。 Koa 并没有捆绑任何中间件， 而是提供了一套优雅的方法，帮助您快速而愉快地编写服务端应用程序。

12.2.2 安装

koa 依赖 node v7.6.0 或 ES2015及更高版本和 async 方法支持。我们可以使用自己喜欢的版本管理器快速安装支持的 node 版本：

安装koa2

npm install koa --save

12.2.3 应用程序

Koa 应用程序是一个包含一组中间件函数的对象，它是按照类似堆栈的方式组织和执行的。 Koa 类似于你可能遇到过的许多其他中间件系统，例如 Ruby 的 Rack ，Connect 等，然而，一个关键的设计点是在其低级中间件层中提供高级“语法糖”。 这提高了互操作性，稳健性，并使书写中间件更加愉快。

这包括诸如内容协商，缓存清理，代理支持和重定向等常见任务的方法。 尽管提供了相当多的有用的方法 Koa 仍保持了一个很小的体积，因为没有捆绑中间件。

● 简单使用

// 引入

const Koa = require("koa");

let app = new Koa();

//配置中间件（可以先当做路由）

app.use(ctx=>{

ctx.body="hello koa"

})

//监听端口号

app.listen(3000)运行这个脚本。

12.2.4 上下文(Context)

Koa Context 将 node 的 request 和 response 对象封装到单个对象中，为编写 Web 应用程序和 API 提供了许多有用的方法。 这些操作在 HTTP 服务器开发中频繁使用，它们被添加到此级别而不是更高级别的框架，这将强制中间件重新实现此通用功能。

每个请求都将创建一个 Context，并在中间件中作为接收器引用，或者 ctx 标识符，如以下代码片段所示：

app.use(async ctx => {

ctx; // 这是 Context

ctx.request; // 这是 koa Request

ctx.response; // 这是 koa Response

});

为方便起见许多上下文的访问器和方法直接委托给它们的 ctx.request或 ctx.response ，不然的话它们是相同的。 例如 ctx.type 和 ctx.length 委托给 response 对象，ctx.path 和 ctx.method 委托给 request。

剖析ctx，ctx包含request、response、req、res

requset 　　　ctx的请求对象

response 　　 ctx的响应对象

req 　　　　　Node的请求对象

res 　　　　　Node的响应对象

注意：绕过Koa的response是不被处理的，避免使用Node的属性和方法

12.2.5 中间件

**什么是koa的中间件**

通俗的讲： 中间件就是匹配路由之前或者匹配路由完成做的一系列的操作，我们就可以把它叫做中间件

在express中间件是一个函数，它可以访问请求对象，响应对象和web应用中处理请求-响应循环流程中的中间件，一般被命名为next的变量，在koa中中间件和express有点类似。

中间件就是匹配路由（匹配任何路由或者特定的路由，其作用比如打印日志，查看权限）之前或者匹配路由完成之后所得一系列操作，功能有：

1.执行任何代码

2.修改请求和和响应对象

3.终结请求-响应循环

4.调用堆栈中的下一个中间件

通过next来实现

**Koa中间定义**

使用app.use()注册的函数。每次客户端的请求，koa都会调用

1.应用级中间件:匹配路由之前所做的一系列操作

//中间件(中间件要放在use(route)之前)：

//例如：一个匹配任何路由，打印时间的中间件

app.use(async (ctx,next)=>{

console.log(new Date());

await next()//当前路由匹配完成之后继续向下匹配，如果不写await next(),就会终止路由

})

2.路由级中间件：由于这个中间件只对这一个路由有作用，而不是对整个应用的路由都有作用，所以叫做路由级中间件；

如果get、post回调函数中，没有next()参数，那么就匹配上第一个路由，就不会往下匹配了。如果想往下匹配的话，那么需要些next()

router.get('/login', async (ctx, next) => {

console.log("login");

await next()//不写的话就不会继续向下匹配，也就是拿不到ctx.body

})

router.get('/login', async (ctx, next) => {

ctx.body = '中间件'

})

3.错误处理中间件

app.use(async (ctx,next)=>{

if(ctx.status==404){

ctx.status=404;

ctx.body="404页面"

}

})

4.第三方中间件

5.中间件的执行顺序

app.use(async(ctx, next) => {

console.log(1)

await next();

console.log(2)

})

app.use(async(ctx, next) => {

console.log(3)

await next();

console.log(4)

})

app.use(async(ctx, next) => {

console.log(5)

await next();

console.log(6)

})

打印结果是：1，3，5，6，4，2

显而易见：在匹配路由之前，要执行next（）之前的代码，匹配之后，要执行next()之后的代码，类似于进入洋葱和从洋葱中心离开；

koa2 的中间件是洋葱模型。基于async/await 可以更好的处理异步操作。

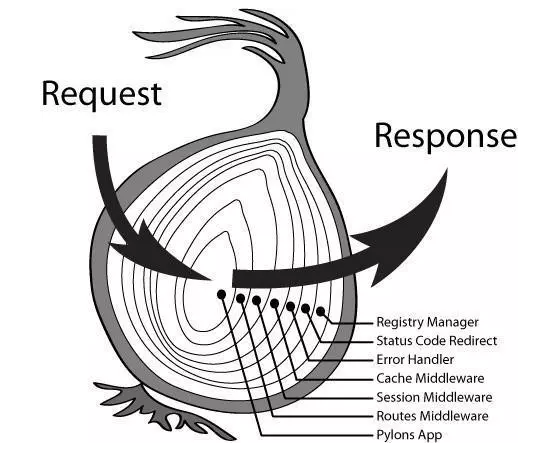


图12-4

PS:执行流程，好比水分进出洋葱一样，洋葱的每层相当于中间件，水分输入相当于请求，水分输出相当于相应。整个过程中，执行指针需要遍历两次中间件（除了最后一个中间件）。

12.2.6 路由

● 原生路由

网站一般都有多个页面。通过ctx.request.path可以获取用户请求的路径，由此实现简单的路由。

const Koa = require('koa');

const app = new Koa();

const main = ctx => {

if (ctx.request.path !== '/') {

ctx.response.type = 'html';

ctx.response.body = '<a href="/">我是首页</a>';

} else {

ctx.response.body = 'Hello koa';

}

};

app.use(main);

app.listen(3000);

运行这个 demo。

访问 http://127.0.0.1:3000/ujiuye ，可以看到一个链接，点击后就跳到首页。

● koa-route 模块

原生路由用起来不太方便，我们可以使用封装好的koa-route模块

安装 koa-router

npm install koa-router --save

1.基本使用

const Koa = require('koa');

// 直接调用的方式

const router = require('koa-router')();

// 或 单独创建router的实例

const Router = require('koa-router');

const router = new Router();

router.get('/', async ctx => {

ctx.body = 'Hello Router';

})

// 启动路由

app.use(router.routes()).use(router.allowedMethods())

// 以上为官方推荐方式，allowedMethods用在routes之后，作用是根据ctx.status设置response header.

app.listen(3000, err => {

if (err) throw err;

console.log('runing...');

});

2.get请求

在koa2中get传值通过request接收，但是接收的方法有两种：query和querystring

● query：返回的是格式化好的参数对象

● querystring：返回的是请求字符串

router.get("/", ctx => {

console.log(ctx.query) //{id:"123"} 获取对象

console.log(ctx.querystring)// id=123 获取字符串

console.log(ctx.request.query) //{id:"123"} 获取对象

console.log(ctx.request.querystring)// id=123 获取字符串

ctx.body = "首页"

})

动态路由

router.get("/:id", async ctx => {

console.log(ctx.params)

ctx.body = "首页"

})

12.2.7 其他第三方中间件

1. post提交数据及koa-bodyparser 中间件的使用

下载 npm install koa-bodypaser --save

1）引入 const bodyparser = require("koa-bodyparser")

2）使用 app.use(bodyparser())

3）取值 router.post("/reg",async ctx=>{

console.log(ctx.request.body)

})

2.静态资源

如果网站提供静态资源（图片、字体、样式表、脚本......），为它们一个个写路由就很麻烦，也没必要。koa-static模块封装了这部分的请求。

下载 npm install koa-static --save

1）引入 const fileStatic = require("koa-static");

2）使用 app.use(fileStatic("./public"))

3. ejs模板使用

下载 npm install koa-views ejs --save

1）引入 const views = require('koa-views');

2）使用 app.use(views('views', { extension: 'ejs' }));

3）渲染 router.get("/",async ctx=>{

//一定要加await 否则就等不到返回值

await ctx.render("index")

})

4.koa中使用cookie和session

1) cookie

不需要下载cookie直接设置：

ctx.cookies.set(name,value,[options])

取值：

ctx.cookies.get(name)

注意：在koa中的cookie值不可以有中文，我们可以将中文作为buffer

解决cookie中的中文问题：

Buffer.from("张三").toString("base64") =》转为buffer

Buffer.from(str, "base64").toString() =》转为字符串

2) session

需要使用第三方中间件

下载 npm install koa-session --save

1）引入 const session = require("koa-session");

2）配置 app.keys = ['some secret hurr']; //cookie的签名

const options = {

key: "koa:sess",//cookie key(default is koa:sess)

maxAge: 10000,//cookie的过期时间 【需要修改】

httpOnly: true,//cookie是否只有服务器端可以访问

signed: true,//签名默认

rolling: false,//在每次请求时强行设置cookie，这将重置cookie过期时间（默认：false）

renew: false //在过期时间快到的时候，重新设置 【需要修改】

}

3)使用 app.use(session(options, app))

4)在路由中使用：

设置：ctx.session.attr = value

获取：ctx.session.attr

12.3 高频面试

● websocket是什么，有什么优点

● koa与express的区别

第13讲 规范及权限

|  |  |  |  |
| --- | --- | --- | --- |
| 知识模块 | 课程内容 | 重点 | 课程目标 |
| 规范权限 | RESTFul API 设计 | ★ | 掌握：RESTFul API 设计  掌握：登录拦截  掌握：token的设计  掌握：权限系统设计  掌握：接口联调和postman工具使用 |
| 登录拦截 | ★ |
| token令牌设计 | ★ |
| 权限系统设计 | ★ |
| 接口联调postman使用 | ★ |

13.1 RESTFul API 设计ss

13.1.1 概述

RESTful架构，是目前最流行的一种互联网软件架构。它结构清晰、符合标准、易于理解、扩展方便，所以正得到越来越多网站的采用。

REST的名称"表现层状态转化"中，省略了主语。"表现层"其实指的是"资源"（Resources）的"表现层"。

所谓"资源"，就是网络上的一个实体，或者说是网络上的一个具体信息。它可以是一段文本、一张图片、一首歌曲、一种服务，总之就是一个具体的实在。你可以用一个URI（统一资源定位符）指向它，每种资源对应一个特定的URI。要获取这个资源，访问它的URI就可以，因此URI就成了每一个资源的地址或独一无二的识别符。

所谓"上网"，就是与互联网上一系列的"资源"互动，调用它的URI。综合上面的解释，我们总结一下什么是RESTful架构：

● 每一个URI代表一种资源；

● 客户端和服务器之间，传递这种资源的某种表现层；

● 客户端通过四个HTTP动词，对服务器端资源进行操作，实现"表现层状态转化"。

13.1.2 设计方式

● HEAD（SELECT）只获取某个资源的头部信息

● GET（SELECT）获取资源

● POST（CREATE）创建资源

● PUT（UPDATE）更新资源，客户端需要提供新建资源的所有属性

● DELETE（DELETE）删除资源

13.1.3 使用方式

● GET http://www.ujiuye.com/api/user # 获取列表

● POST http://www.ujiuye.com/api/user # 创建用户

● PUT http://www.ujiuye.com/api/user/{id} # 修改用户信息

● DELETE http://www.ujiuye.com/api/user/{id} # 删除用户信息

13.2 登录拦截

13.2.1 什么是登录拦截

在我们后端数据管理类的项目当中，所有的操作都是操作数据库相关的内容，如果被一些没有登录的人随意操作，可能会造成数据的丢失或者被篡改，所以想使用后台数据管理类的项目，必须要求用户登录，哪怕你知道具体的url路径，但是在未登录状态你也是无法直接访问和查看的，这就需要登录拦截的出场了

13.2.2 项目工程创建

这里我们使用express-generator模块，初始化一个基础项目，然后再额外配置下nodemon开启实时代码更新，方便我们敲代码

express --view=ejs project3

cd project3

npm install

到此项目的初始化工程就完毕了，接下来配置nodemon相关的东西，在当前工程下载nodemon和配置即可

npm install nodemon

修改项目的配置node ./bin/www 换成用nodemon来启动即可，以后正常启动npm start即可开启这个项目就可以编码保存后，自动更新后台项目代码，无需再停止后重新启动了。



图13-1

13.2.3 设计登录拦截中间件

在后端node.js的代码中，分为静态资源返回，和接口API的请求两大类，我们可以对不同的资源进行分别的权限拦截，例如非登录用户只可以访问登录页面和登录的接口，而登录用户才可以访问除此之外的全部资源。

1）登录页面准备

在public下新建login文件夹，准备login.html登录页面和index.html首页页面

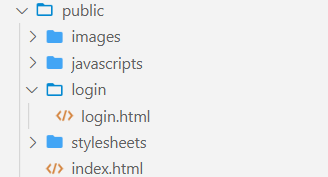


图13-2

配置app.js服务器入口文件，设置登录拦截中间件，现在只能访问login.html登录页面, 而public/index.html首页是无法访问的，app.js文件核心代码如下:

app.use(cookieParser());

// 指定可以被访问的登录页面

app.use(express.static(path.join(\_\_dirname, 'public/login')));

// 拦截中间件: 如果没有这个就进入到index.html首页了

// 拦截所有的请求, 除了上面的登录页 (如果想拦截某一部分， 把它放在对应拦截路由之上即可)

app.use((req, res, next) => {

res.send("请您先进行登录");

})

// 正常资源

app.use(express.static(path.join(\_\_dirname, 'public')));

app.use('/', indexRouter);

当用户访问http://localhost:3000/login.html 可以正常访问，但是访问http://localhost:3000/index.html首页时，就被自定义中间件函数拦截了，需要您先登录才可以去访问（当然这里只是做下效果，讲下中间件函数的拦截效果，具体登录功能往下看）

13.2.4 登录功能设计

在我们印象当中，登录功能应该就是在浏览器的页面上，输入用户名，输入密码，还可能需要输入验证码，然后点击登录按钮进行登录，那么这个过程到底都发生了什么事情呢，接下来听我细细道来

前端主要提供账号名和md5加密（也可能是其他加密方式）后的密码给服务器端，为了防止撞库md5，需要对密码的组成进行前端的正则验证，密码的复杂度加高即可

后端接收到用户名和密码与数据库存储的用户名和密码进行比对，如果成功，保存状态字符串（随机字符串）到session上保存，会给客户端设置一个sessionID的cookie标识这个session的结果，在后面的请求中，都会携带这个sessionID的cookie 给后端，后端就可以验证之前登录的那个用户了。

我们后向前设计，首先准备数据库member和表user，字段id，account，password，per\_status（身份唯一标识，账号，密码，身份字符串），可以准备2个数据，如图所示：

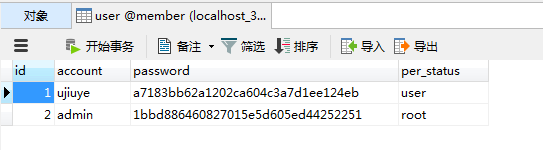


图13-3

routes/users.js跟用户相关的接口都在这里，先加入登录功能，因为需要连接Mysql数据库和操作，需要先下载mysql和express-myconnection模块。

npm install mysql express-myconnection

封装数据库连接类和查询方法utils/Db.js，代码如下：

var mysql = require('mysql'); // node-mysql module

var myConnection = require('express-myconnection') // express连接mysql的中间件

const {getModel} = require("../model");

class Db {

constructor() {

this.connection = myConnection(mysql, { // mysql数据库相关配置

host: 'localhost',

user: 'root',

password: '',

port: 3306,

database: 'member'

}, 'single')

}

// 查询方法 (只有一个结果，而且是And结果)

selectOneAnd(req, tableName, obj) {

return new Promise((resolve, reject) => {

// 把对象的key:value拼接成 key=value的字符串

let sqlStr = '';

for (let prop in obj) {

sqlStr += prop + " = " + (typeof obj[prop] === 'number' ? obj[prop] : `\'${obj[prop]}\'`) + " AND "

}

sqlStr = sqlStr.substr(0, sqlStr.length -5);

req.getConnection(function (err, connection) {

connection.query(`SELECT \* FROM ${tableName} WHERE ${sqlStr}`, [], function (err, results) {

if (err) reject(err);

if (results.length > 0) {

resolve(getModel(results[0], 'user'));

} else {

resolve(null);

}

});

});

})

}

}

module.exports = new Db();

新增model/user.js,设置抽离出哪些字段使用

// user表

// 提前设置好只要哪些字段(跟数据库相同)

module.exports = {

id: "",

account: ""

}

新增model/index.js定义入口文件，传入表时自动找到对应模块js文件并且只抽离出想要的提前准备好的字段， 代码如下：

exports.getModel = (objArg, tableName) => { // 把表格的字段，从传入的对象中提取出来

const objModel = require("./" + tableName);

for (let prop in objModel) {

objModel[prop] = objArg[prop];

}

return objModel;

}

封装统一返回接口字符串内容的utils/Result.js，代码如下：

// 参数正确， 获取成功

exports.Success = (data = [], msg = '获取成功') => {

return {

msg,

code: 200,

data

}

}

// 参数正确， 但是权限不够

exports.Guest = (data = [], msg = '权限非法/账号密码错误') => {

return {

msg,

code: 403,

data

}

}

// 参数错误，请检查传递的参数

exports.Error = (data = [], msg = '参数等发生错误') => {

return {

msg,

code: 400,

data

}

}

app.js代码新增修改如下：

const session = require('express-session'); // 下载并引入express-session模块

app.use(session({ // 使用 session 中间件

secret : 'secret',

resave : true,

saveUninitialized: false,

cookie : {

maxAge : 1000 \* 60 \* 1, // 登录时效30分钟

}

}));

// 登录拦截中间件: 如果没有这个就进入到index.html首页了

// 拦截所有的请求, 除了上面的登录页 (如果想拦截某一部分， 把它放在对应拦截路由之上即可)

app.use((req, res, next) => {

if (req.session.user !== undefined) {

next(); // 放行.

} else {

res.send("您还未登录");

}

})

// 把用户相关的接口提到登录拦截器之前

app.use('/users', usersRouter);

在路由/users对应的路由文件users.js中，实现post方式/login接口，代码如下：

router.post("/login", async (req, res) => { // 登录接口

const resultObj = await Db.selectOneAnd(req, "user", { // 传入用户名+密码得到查询出来的对象

account: req.body['user'],

password: req.body['pass']

})

if (resultObj != null) {

req.session.user = getSessionStr(); // 产出8位的session保存一下

res.send(Success(resultObj, "登录成功"));

} else {

res.send(Guest([], "账号密码错误"));

}

})

前端的login.html代码如下：

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<meta http-equiv="X-UA-Compatible" content="ie=edge">

<title>欢迎使用\_优购物电商平台数据管理系统</title>

<script src="https://cdn.bootcss.com/axios/0.19.0-beta.1/axios.min.js"></script>

<script src="https://cdn.bootcss.com/blueimp-md5/2.12.0/js/md5.min.js"></script>

</head>

<body>

<p>登录</p>

<span>用户名:</span>

<input type="text" id="userName">

<span>密码:</span>

<input type="password" id="passWord">

<button onclick="btn()">登录</button>

<script>

function btn() { // 登录事件

axios({

method: "post",

url: "/users/login",

data: {

user: document.getElementById("userName").value,

pass: md5(document.getElementById("passWord").value)

}

}).then(res => {

if (res['data']['code'] === 200) {

window.location.href = "/index.html"; // 直接跳转到首页

}

})

}

</script>

</body>

</html>

这样就可以测试此接口了，在postman或者前端浏览器上进行使用即可。效果输入用户名和密码即可进行登录，测试用户名ujiuye，密码ujiuye即可。效果如下

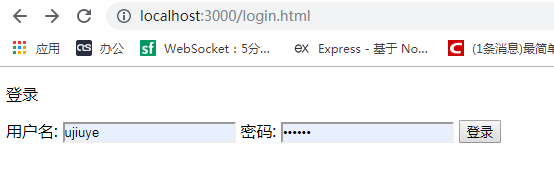


图13-4

点击登录调用登录接口后，前端进行页面跳转，跳转index.html中，效果如下：

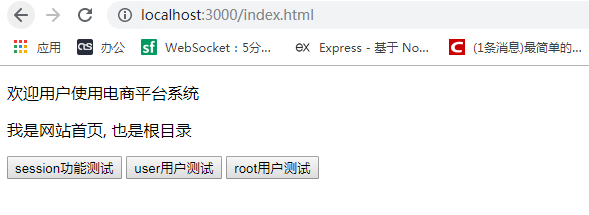


图13-5

可以测试用户登录后，在后端session中保存的鉴别身份的字符串，但是后台一旦刷新，那么保存的session也就不存在了，又变成了未登录状态，而且在后端, 设置了session的过期时间，在登录拦截中间件上，判断session是否存在， 如果登录状态过期 / 未登录状态，保证权限逻辑在后端服务器上，确保安全。

但是session要求协议，域名，端口号一致才可以使用，但是如果前端和后端不在同一个服务器上，那么session是无法使用的，这时需要token令牌来进行验证（它是一个具体的字段，在前后端之间传递，忽略使用session时，协议/域名/端口号的限制和刷新不存在的情况）

13.3 token令牌设计

原理实际上就是登录成功以后, 生成一串字符串, 这个字符串可以按照一定的规则生成, 这串字符串我们就可以叫它token 或者有的叫jwt, JSON Web Token的简称

以后客户端和服务器通信的时候, 特别是需要验证用户身份的时候, 都需要携带这个token给服务器端, 一般通过Body体或者请求头Authorization, 来把这个字段传给服务器端，而且token字符串无需保存在数据库中，不然多服务器多数据库又陷入到了session的问题中。

13.3.1 token的组成部分

Header（头部）

头部里一般放的是当前加密的算法和, 字符串的作用

Payload（负载）

放的是一些校验的信息, 例如

iss (issuer)：签发人

exp (expiration time)：过期时间

sub (subject)：主题

aud (audience)：受众

nbf (Not Before)：生效时间

iat (Issued At)：签发时间

jti (JWT ID)：编号

还可以自定义一些字段, 都可以使用

Signature（签名）

利用RSA加密, 私钥必须存在服务器端, 不能暴露给客户端, 用私钥对前2个对象加密后, 把三部分字符串拼接在一起, 返回给前端使用

而且RSA加密后的字符串, 可以发给后台时, 后台用RSA的公钥解密, 可以查看一些数据, 比如过期时间之类的.

准备RSA类文件utils/RSA.js代码如下：

const NodeRSA = require('node-rsa');

// RSA的私钥

rsaS = `-----BEGIN PRIVATE KEY-----

MIICdwIBADANBgkqhkiG9w0BAQEFAASCAmEwggJdAgEAAoGBALKHXqNqdWzLIOHH

lV6P3leXSoaDZLLaU7nMMywi4Sv7rQaq7Bh0ZkNWW+IYquSWbNFhDUplEE0tvCsL

Y2cyO/X8GuH/O9f2Lw1kOX3KSE0j9amtTyN1WF65FxYLeWEinAFcSG6B2OhIO9N/

Moe4qmX2VRQNv/tzTPjcuc0I+qiDAgMBAAECgYAUdTq9Ezf9jZFpDgvp2CasduPN

YkUp0acChYnfpueSuLZtKyz+6GfHk5/X2ufKZgp2O97gBZDGp1UGwblUXAdOyd6T

a7I4Hy8MaTK6myHVcWjSBRakdVuP7xh0zc4zVbhUVpRpR9NSjp1SOtaQXCN1QZ7E

vZQj+KfzsEb9H1jqwQJBANwWLjZsZmTrpBA34KU3a9jN3Hi8J3uwDto0elMHevKy

dzhQ52fYcE925yB6dLF4bXXIuOMAEP/nXyCn2kRhatMCQQDPqSqbAiwvUrJWA0G5

z3rZ6BOiUIdCIKqXFTK4jAovyZAtWYBGousfbdWGkyaAn1hjgSm6TDykwj1ZQqVx

992RAkEAoqa5zHrcSaVkgJ/rI+8LgxTQ5WGYhG6e62BxLr+Oe6KtPS99dV+d0Px8

36Hgb/AiZTPMf+biqV0jDo0+X97nBQJAVxHwOlRb4G/FNMliyp0TPsDPUrR/Mhvu

SzDpNnYJY2YZXNp1Lajr+x6bdDTI5Q9rCTXEIEgM+Ww0OvP12OZ78QJBAMsUaPuB

jbLsCXEcj0J0LF7r2dg6oJ0XPd4DUp8h6PISTvOidgMd0CbpazaT7ymtezT7ZadP

v7eP1ZqRgQ1EsIE=

-----END PRIVATE KEY-----

`;

// 对应的公钥

publicKey = `-----BEGIN PUBLIC KEY-----

MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCyh16janVsyyDhx5Vej95Xl0qG

g2Sy2lO5zDMsIuEr+60GquwYdGZDVlviGKrklmzRYQ1KZRBNLbwrC2NnMjv1/Brh

/zvX9i8NZDl9ykhNI/WprU8jdVheuRcWC3lhIpwBXEhugdjoSDvTfzKHuKpl9lUU

Db/7c0z43LnNCPqogwIDAQAB

-----END PUBLIC KEY-----`

class RSA {

static base64(str) { //base64加密方法

if (typeof str !== 'string') {

str = JSON.stringify(str);

}

return Buffer.from(str).toString("base64");

}

static base64decode(str) { // base64解密

return Buffer.from(str, "base64").toString('utf-8');

}

static getRSAKey() {

return new NodeRSA(rsaS); // 生成秘钥工具

}

}

module.exports = RSA;

在工具类utils/index.js中，封装一些常用的方法，包括生成token，检验token等代码如下：

// 工具类

const RSA = require("./RSA");

const key = RSA.getRSAKey();

const Db = require("./Db");

// 生成session字符串

exports.getSessionStr = () => {

return Math.floor(Math.random() \* (99999999 - 10000000 + 1) + 10000000);

}

// 生成token字符串

exports.getToken = (theId, perStatus) => {

let header = { // token头部信息

"sec": "RSA", // 密码算法RSA

"type": "JWT" // 统一jwt类型

};

let payload = { // 定义数据信息, 可以自己定义, 但是不要放机密信息

"nbf": new Date().getTime(), // 生效时间当前系统时间毫秒数

"id": theId, // 用户id

"perStatus": perStatus, // 把用户的身份权限放入，（user字符串/root字符串）

"lastTime": 60 \* 1000 \* 10, // 过期时间10分钟, 对应的毫秒数

}

const sign = key.encrypt(RSA.base64(header) + "." +

RSA.base64(payload), 'base64'); // 对前两部分生成RSA签名字符串

const token = RSA.base64(header) + "." +

RSA.base64(payload) + "." + sign; // 生成token

return token;

}

// 检验jwt的token是否过期， 和是否合法

exports.checkToken = async (theToken = "", req) => {

let arr = theToken.split(".");

let jwtObj = JSON.parse(RSA.base64decode(arr[1]));

// 解密拿出过期时间, 判断是否过期

if (new Date().getTime() - jwtObj['nbf'] < jwtObj['lastTime']) {

return true;

} else {

return false;

}

}

更新/login接口里的代码如下：

router.post("/login", async (req, res) => { // 登录接口

const resultObj = await Db.selectOneAnd(req, "user", {

account: req.body['user'],

password: req.body['pass']

})

if (resultObj != null) {

resultObj['token'] = getToken(req.body['id']);

res.send(Success(resultObj, "登录成功"));

} else {

res.send(Guest([], "账号密码错误"));

}

})

增加检验token字段的功能方法，config/utils.js的代码新增方法：

// 检验token是否过期

exports.checkToken = (theToken = "") => {

let arr = theToken.split(".");

let jwtObj = JSON.parse(RSA.base64decode(arr[1]));

// 解密拿出过期时间, 判断是否过期

if (new Date().getTime() - jwtObj['nbf'] < jwtObj['lastTime']) {

return true; // 没过期

} else {

return false; // 前端可以封装网络请求, 判断状态, 统一作出处理

}

}

app.js修改session的权限，换成token验证的规则，注意把静态资源放到检验的上面，只负责检验接口部分，静态网页由前端判断即可。

const {

checkToken

} = require("./config/utils");

// 正常资源

app.use(express.static(path.join(\_\_dirname, 'public')));

app.use((req, res, next) => {

if (!req.headers.authorization) {

res.send(Error([], "请设置请求头"));

} else {

if (!checkToken(req.headers.authorization)) { // 过期

res.send(Guest([], "登录状态过期"));

} else {

next();

}

}

})

修改登录页面login.html保存token到sessionStorage的auth字段中，代码如下：

if (dataObj['code'] === 200) { // 登录成功后

sessionStorage['auth'] = dataObj['data']['token'];

window.location.href = "/index.html";

}

13.4 权限设计

在后台数据管理类项目中, 每个登录的用户应该只能访问这个用户拥有权限的资源, 比如管理员能够获取到所有用户的信息, 和管理普通的使用用户, 则普通用户呢, 则不能进入和访问管理用户的接口功能, 只能进行一些普通用户应有的权限操作, 而且前端的页面也需要根据后端返回的权限判别字段, 分别显示对应的页面. 所以前后端都应该进行权限系统的设计和统一。

最重要的一点是，“root”用户需要在数据库提前准备好，包括权限字段的开启，普通用户可以由超级管理员root进行创建，所以数据表中需要有一个字段来代表用户该有的权限，例如“root”字符串代表超级管理员，“user”代表普通用户，当然数字是可以随便定义的，只要前后台沟通好。实际用什么字符串实际上是个标记, 用什么都可以, 只要你和前端沟通好就可以。

1. 这里我们没有注册界面和功能，所以先提前在数据库中准备root用户账户名admin，密码md5加密后的字符串1bbd886460827015e5d605ed44252251（是8个1进行md5加密后的字符串），user普通用户的账户名ujiuye，密码a7183bb62a1202ca604c3a7d1ee124eb（是ujiuye进行md5加密后的字符串）

2. 前端在登录后，会得到一个token字符串，字符串里面包含用户的登录时间，以及用户的ID，和身份状态字符串（root/user）

3. 前端把token的值可以存在浏览器存储中, 每次请求的时候在请求头/body体中，都携带此参数（下面代码案例是在登录后调用接口时，在请求头中带上此token字符串），请求接口时带上，验证前端发起的请求是否有权限访问对应的接口。

这里为了测试, 准备2个按钮, 分别测试用户不同的权限

index.html代码主要功能代码如下:

function getList() {

axios({

method: "post",

url: "/getGoodsList",

headers: {

authorization: sessionStorage['auth']

}

}).then(res => {

console.log(res);

})

}

function getUserList() {

axios({

method: "post",

url: "/getUserList",

headers: {

authorization: sessionStorage['auth']

}

}).then(res => {

console.log(res);

})

}

在utils/index.js，定义代码进行token字符串校验，包括把东西提取出来，看看token是否被篡改过，以及登录状态是否过期，还有此用户权限是否能够访问此接口，代码如下：

// 检验jwt的token是否过期， 和是否合法

exports.checkToken = async (theToken = "", req) => {

let arr = theToken.split(".");

let jwtObj = JSON.parse(RSA.base64decode(arr[1]));

// 解密拿出过期时间, 判断是否过期

if (new Date().getTime() - jwtObj['nbf'] < jwtObj['lastTime']) {

//用id和per\_status验证一下

const res = await Db.selectOneAnd(req, "user", {

id: jwtObj['id'],

per\_status: jwtObj['perStatus']

})

if (routerAuthFn(req.path, res['per\_status'])) { // 没过期，而且路由有权限访问

return true;

} else {

return false;

}

} else {

return false; // 前端可以封装网络请求, 判断状态, 统一作出处理

}

}

// 路由权限验证

const routerAuthFn = (routerPath, perStatus) => {

// 每个路由， 对应可以有那些权限用户访问

let obj = {

"/getGoodsList": ['user'],

"/getUserList": ['root']

}

if (obj[routerPath] === undefined) return false; // 路由不存在返回false

return obj[routerPath].indexOf(perStatus) > -1; // true就是有权限

}

修改app.js中登录拦截部分代码，调用checkToken检验的方法。代码如下：

app.use(async (req, res, next) => {

if (!req.headers.authorization) {

res.send(Error([], "请设置请求头，到这里session验证其实已经通过了"));

} else {

if (!await checkToken(req.headers.authorization, req)) { // 过期

res.send(Guest([], "登录状态过期/权限身份错误，非法请求"));

} else {

next();

}

}

})

登录ujiuye账户后，调用user接口和root用户接口分别测试，查看后台权限验证功能及token功能。效果如下：

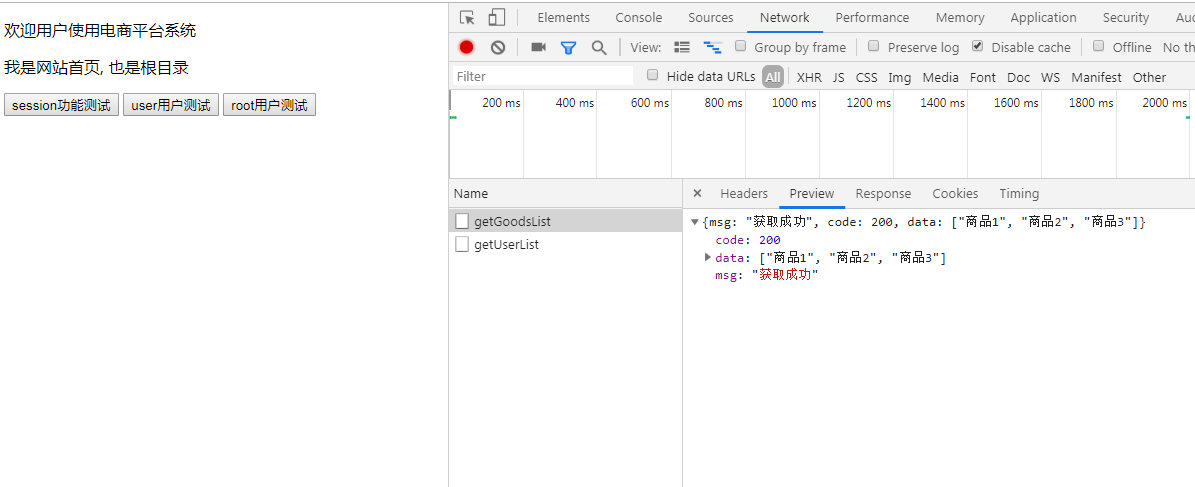


图13-6

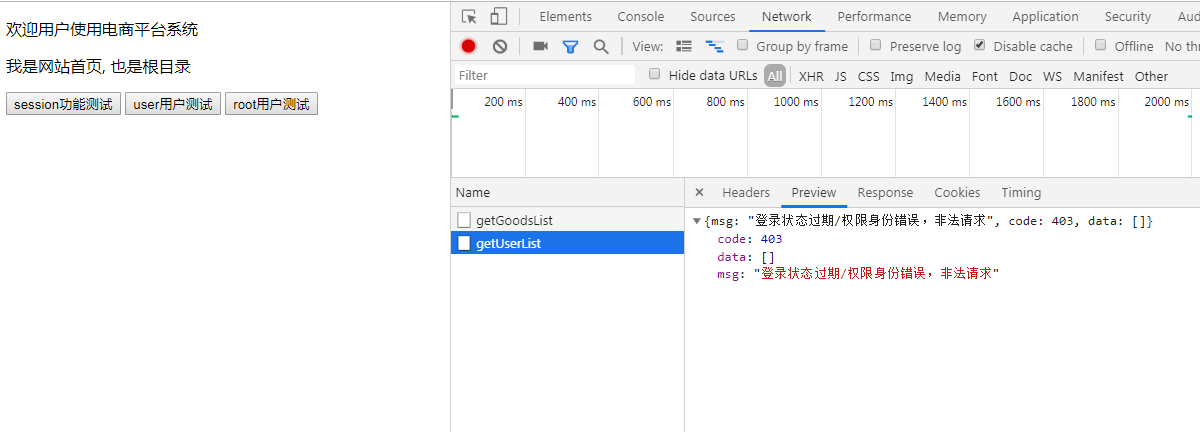


图13-7

13.5 接口测试工具Postman

开发者在开发或者调试网络程序或者是网页B/S模式的程序的时候是需要一些方法来跟踪网络请求的，开发者可以使用一些网络的监视工具比如著名的Firebug等网页调试工具。Postman网页调试工具不仅可以调试简单的css、html、脚本等简单的网页基本信息，它还可以发送几乎所有类型的HTTP请求！使用非常方便。

下图是Postman工具的主界面。

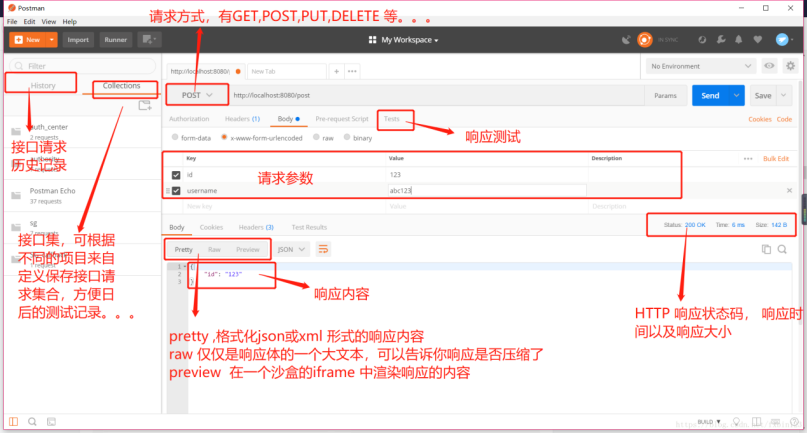


图13-8

13.5.1 接口测试流程

使用express框架快速搭建http服务器，上面登录接口api为例。

1. GET 请求

GET请求：点击Params，输入参数及value，可输入多个，即时显示在URL链接上，

所以，GET请求的请求头与请求参数如在接口文档中无特别声明时，可以不填。

http://localhost:4000/news?id=3

2. POST请求

Post请求中，包括请求URL，请求参数，但是没有设置请求头，在我的使用过程中，请求头是根据请求参数的形式自动生成的。

请求头中的Content-Type与请求参数的格式之间是有关联关系，比如：

Body：设置POST请求的参数。

● form-data： HTTP请求中的multipart/form-data，它会将表单的数据处理为一条消息，以标签为单元，用分隔符分开。

● x-wwww-form-urlencode：HTTP请求中的application/x-www-from-urlencoded，会将表单内的数据转换为键值对。

● raw：可以发送任意格式的接口数据，可以text、json、xml、html等。

● binary：HTTP请求中的相Content-Type:application/octet-stream，只可以发送二进制数据。通常用于文件的上传。

13.6 高频面试

● websocket是什么，有什么优点

● 为什么要进行前后端分离

第14讲 模板及部署

|  |  |  |  |
| --- | --- | --- | --- |
| 知识模块 | 课程内容 | 重点 | 课程目标 |
| 项目部署 | 接口联调和数据渲染 | ★ | 掌握：接口联调和Ajax+art使用  掌握：后台管理系统模板套用  掌握：后台接口的CRUD的使用  掌握：项目部署到线上云服务器 |
| 后台管理系统模板套用 | ★ |
| 后台管理的CRUD使用 | ★ |
| 项目部署 | ★ |

14.1 接口的联调和数据渲染

这是一个前后端分离的项目，前端主要由HTML+CSS+JavaScript组成的前端静态网站，数据也都是固定数据，而真正企业中的项目，数据应该来自由后台语言编写的代码从数据库中查询出，并且按照一定JSON格式响应回给前端。那么前端一般使用Ajax请求后台地址和URL，有时还需要传递一些参数给后台，让后台决定把哪部分数据响应回给前端。而前端把拿到的数据利用JSON解析到前端现有的标签上，这样就实现了一个前后端交互的流程，也实现了接口联调和数据渲染的功能。

所以前端请求+展示数据, 后端接参+返回数据

开发分为三种方式:

1. 前端开发, 暂无后端, 这种情况可以使用前端编写一些数据, 来模拟, 或者使用mock.js进行Ajax拦截产出模拟数据

2. 前后端同步开发, 有接口的直接调用, 一边写前端一边调用接口联调

3. 暂无前端, 只有后端, 就需要后端工程师使用类似请求接口的工具, 例如postman等来调试自己编写的接口, 接收参数, 返回内容是否符合预期

14.1.1 各种方式接口的编写和使用

● get请求

前端get请求不传参的使用，获取到后台返回的接口数据，因为前后端分离项目，不在同一个服务器上开发，所以后端可以暂时开启跨域资源共享的响应头设置。

app.all("\*", (req, res, next) => {

res.header("Access-Control-Allow-Origin", "\*");

next();

})

前端代码使用Ajax请求回来数据，进行JSON解析，把数据渲染到网页上，生成html标签部分，挂载到真实的DOM标签上，在网页显示。

$.ajax({

url: "/api/bookList",

success(res) {

console.log(res);

var html = template('getNoArg', {

dataArr: res.data

});

document.getElementById('getDiv').innerHTML = html;

}

})

后端部分代码，require应当替换成你对应的数据库查询的操作语句，把读取出来的数据按照提前组织的结果返回到前端使用，注意向前端返回的数据格式尽量统一，所以这里定义了对象格式，code代表请求的状态，msg是提示文字，data是具体返回的数据。

const bookArr = require("./data/bookCategory.json"); // 本应该是从数据库调出的

app.get('/api/bookList', function (req, res) {

res.send({

code: 1000,

msg: "获取分类数据成功",

data: bookArr

});

});

●post请求

get请求的传参是在url的后面进行拼接，而post请求的参数是要在body体中进行传参，而且传参的格式也分不同的情况，这里使用jQuery里Ajax请求时默认的内容格式：application/x-www-form-urlencoded(键值对的形式)

通过第一页用户点击的分类信息，传递到第二页的分类文字和分类ID，通过分类ID去后台请求到对应分类书籍的信息回来，展示到网页上供用户查看。

前端代码如下

先从上一页跳转过来，在url上get方式传递过来的参数中，提取对应的信息保存到变量和页面上。

let searchArr = location.search.replace("?", "").split("&");

let typeStr = decodeURIComponent(searchArr[0].split("=")[1]); // 分类名字

let typeId = searchArr[1].split("=")[1] \* 1; // 分类ID

调用分类详情数据接口，根据传过去的分类ID，得到此分类下的分类新闻数据，并且把分类代码如下。

$.ajax({

url: "/api/getList",

type: "post",

data: {

typeId

},

success(res) {

console.log(res);

var html = template('postArg', {

dataArr: res.data

});

document.getElementById('postDiv').innerHTML = html;

}

})

14.2 后台管理系统模板套用

在之前的阶段, 我们已经有了纯前端的适配移动端和PC端的后台管理系统的网站, 我们只需要在之前的网站的模板即HTML+CSS和JS的基础上, 加入Ajax请求等, 进行动态数据获取和数据的动态铺设

我们只需要在对应的前端页面上, 对应的接口中, 调用后台给我们提供的接口,利用Ajax获取数据, 然后铺设到页面上, 用户在页面操作的时候, 把对应参数传递给后台提供的接口地址中, 后台会同步更新到数据库中, 所有的操作基本可以分为查询, 增加, 删除, 修改操作.即我们下面要说的CRUD操作.

14.3 后台管理的CRUD使用

之前前端主要做的是数据的展示，和一些交互的事件，但是数据都是保存在前端的，没有做到真正的数据持久化存储，所以我们第三阶段学了Mysql数据库，做数据的持久化存储

第一步，设计需要的模块，划分出来需要多少张表，以及每张表里需要的字段等，可以提前准备一些数据。

第二步，利用nodejs+对应的第三方模块，连接我们创建的Mysql数据库，然后建立代码的连接访问。

第三步，编写对应的CRUD的接口方法，当前页可以封装一些统一的类进行调用，在非封装的地方，写具体的不同的业务逻辑代码，确认接收的参数，和要执行的操作，和要返回的值给前端进行使用。

第四步，告诉前端请求的地址，和要传递的参数，以及返回值的含义等。

这里后台准备了几个接口，演示一下，比如请求一级分类的数据接口为/firstCategory，具体代码如下：

router.post("/firstCategory", async (req, res) => {

const dataArr = await Db.selectMany(req, "category");

// 按照rowid分数组-1就是大数组下标, 数据结构是二维数组,数组里是对象(包括标题和firstId)

let smallArr = [];

for (let i = 0; i < dataArr.length; i++) {

let obj = dataArr[i];

if (smallArr[obj['rowid'] - 1] === undefined) {

smallArr[obj['rowid'] - 1] = [];

}

smallArr[obj['rowid'] - 1].push(obj); // 把每个分类对象放进去

}

res.send(Success(smallArr));

});

看下postman请求的效果接口如下

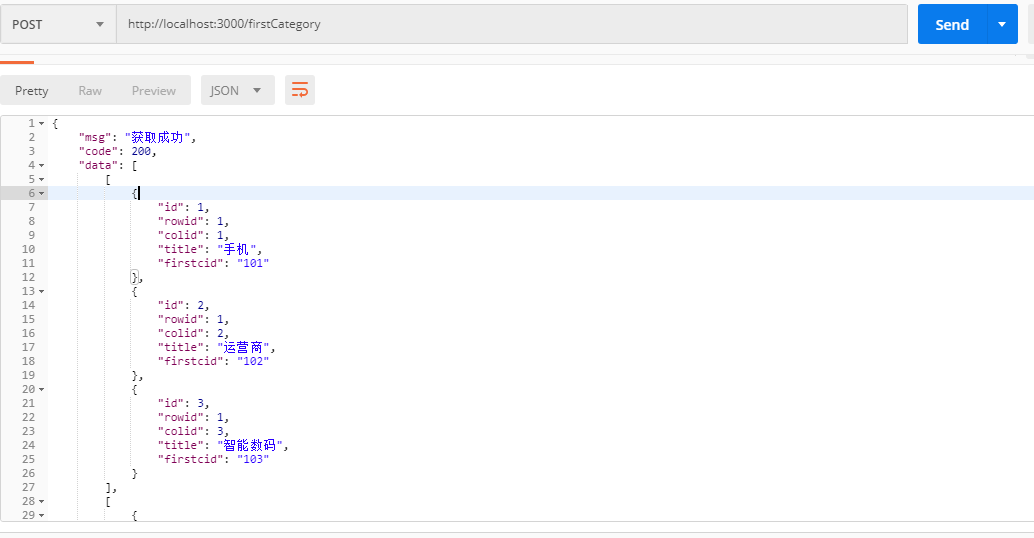


图13-1

如果是获取二级和三级分类数据，接口是/secondthiredCategory 需要进行传参firstcid值为，代码如下

router.post("/secondthiredCategory", async (req, res) => {

const dataArr = await Db.selectMany(req, "category\_1", {

firstcid: req['body']['firstcid']

});

let bigArr = [];

dataArr.map(nowObj => { // 遍历数据库查询出的每个数据

let secondObj = bigArr.find(obj => { // 从大数组中, 查找二级分类(管理下属三级的)的对象

if (obj['secondcid'] === nowObj['secondcid']) {

return obj;

}

})

if (!secondObj) { // 如果没找到二级分类管理对象, 则建立, 扩展属性, 把当前对象插入到bigArr数组中

let nsObj = {};

nsObj['thiredArr'] = []; // 保存下属三级分类对象的数组

nsObj['secondcid'] = nowObj['secondcid']; // 保存当前二级分类ID

if (nowObj['thiredcid'] == 0) { // thiredcid为0的证明是二级分类的对象, 单独提出来信息不要放到数组中

nsObj['ismark'] = nowObj['ismark'];

nsObj['priority'] = nowObj['priority'];

nsObj['title'] = nowObj['title'];

} else {

nsObj['thiredArr'].push(nowObj); // 是三级分类对象, 放到thiredArr管理数组中

}

bigArr.push(nsObj); // 把二级分类对象插入到bigArr数组中

} else {

secondObj['thiredArr'].push(nowObj); // 如果二级分类对象已经在bigArr中, 则查找到此对象后, 调用thiredArr数组把三级分类对象插入进去

}

})

// 数据结构, 按照二级分类组成对象, 下面是三级分类数组里的对象

res.send(Success(bigArr));

})

效果如下所示

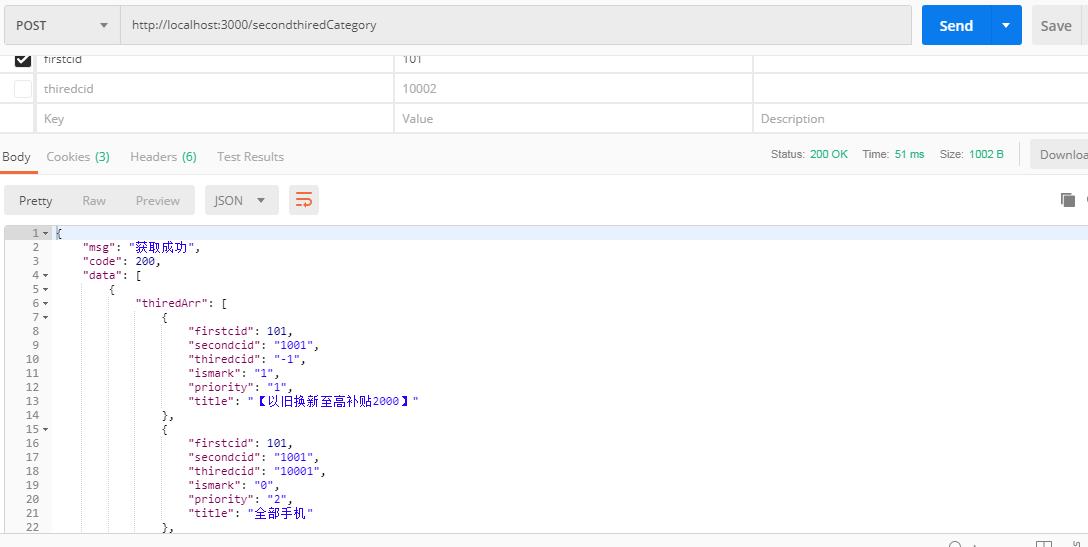


图13-2

14.4 项目到服务器端部署

当我们把Node项目的功能全部完成以后，就可以部署到远程服务器端。最终目的是能确保nodejs成功在云主机运行起来，并能通过自己的IP地址或域名访问得到。以下是需要做的整个流程：

● 购买服务器

● 宝塔面板管理linux服务器

● 安装pm2进程管理工具

14.4.1 购买服务器

服务器提供商有很多选择，其实服务器都是差不多的，根据性价比买即可。个人购买出了系统选择其他一般按默认选择即可。系统选择linux的CentOS系统，Ubuntu也是linux系统，但CentOS更新，在安装软件时候比Ubuntu更加便捷。

百度云一般都会有活动时候，购买的时候性价比超高，2个月的价钱能买1年使用。链接地址如下：

https://cloud.baidu.com/campaign/Annualceremony-2020/index.html?\_=1582809926536&

track=cp:npinzhuan|pf:pc|pp:npinzhuan-biaoti|pu:wenzineirong|ci:2020ndsd|kw:2190962



图13-3

14.4.2 登录宝塔面板

购买并且支付后，默认会安装宝塔面板，一个网页用于操作Linux操作系统，如果没有宝塔面板则自行百度安装即可。宝塔面板官网如下：<https://www.bt.cn/>

每人申请的云服务器都有一个唯一的IP地址，这是云服务器的好处。虚拟主机（和别人共享一个IP地址）

安装完毕以后http://自己的IP:8888，访问宝塔面板管理系统上的软件。系统安装完毕以后，访问这个IP安装宝塔面板。

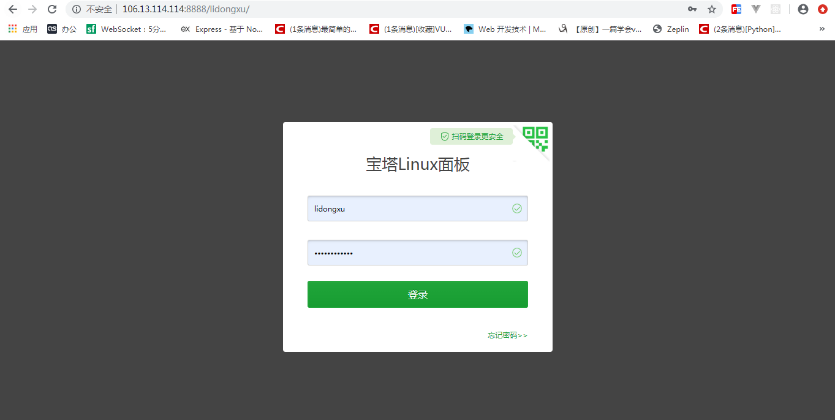


图13-4

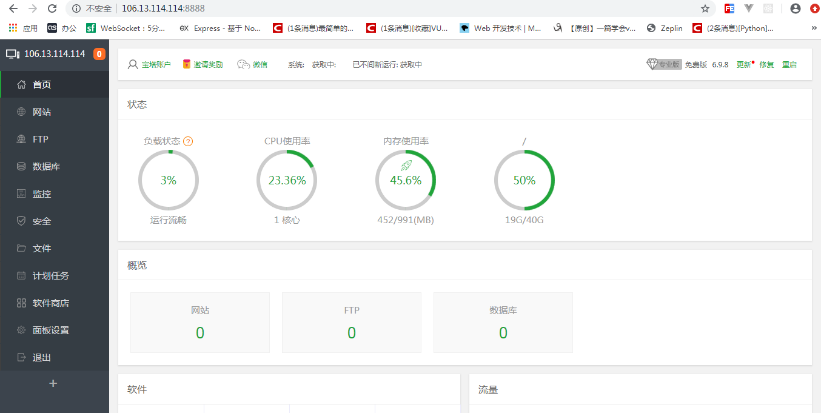


图13-5

**宝塔面板简介：**

● 全面的操作

宝塔面板是一款服务器管理软件，支持windows和linux系统，可以通过Web端轻松管理服务器，提升运维效率。例如：创建管理网站、FTP、数据库，拥有可视化文件管理器，可视化软件管理器，可视化CPU、内存、流量监控图表，计划任务等功能。

● 丰富的功能

宝塔面板拥有极速方便的一键配置与管理，可一键配置服务器环境（LAMP/LNMP/Tomcat/Node.js），一键部署SSL，异地备份；提供SSH开启关闭服务，SSH端口更改，禁ping，防火墙端口放行以及操作日志查看；CPU、内存、磁盘IO、网络IO数据监测，可设置记录保存天数以及任意查看某天数据；计划任务可按周期添加执行，支持SHELL脚本，提供网站、数据库备份以及日志切割，且支持一键备份到又拍云存储空间，或者其他云存储空间里；通过web界面就可以轻松管理安装所用的服务器软件，还有实用的扩展插件；集成方便高效的文件管理器，支持上传、下载、打包、解压以及文件编辑查看。

● 宝塔特色

为了方便用户建立网站，宝塔面板上的一键部署源码插件，可一键部署：Discuz,Wordpress,Ecshop,Thinkphp,Z-blog,Dedecms等程序。还有极其方便的一键迁移，两台服务器安装宝塔Linux面板5.2版本，可实现一键迁移服务器网站、FTP、数据库。

14.4.3 安装项目环境

我们所需要的环境是：Nodejs+Mysql，NodeJs环境我们使用 pm2 工具管理Nodejs环境，所以所需安装为：PM2 + mysql



图13-6



图13-7

安装完宝塔面板进入文件目录

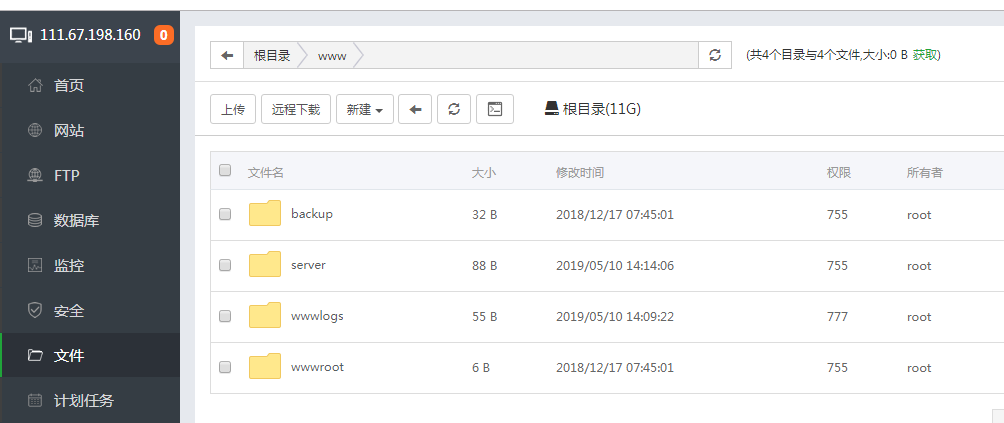


图13-8

目录简介：

● Backup：备份目录

● Server: 安装软件的目录 /pm2 mongodb apache nginx...

● Wwwlogs: 网站日志

● Wwwroot: 自己的网站程序放在此目录下

新建一个空目录存储自己的新项目，如果再有项目，重复此操作。

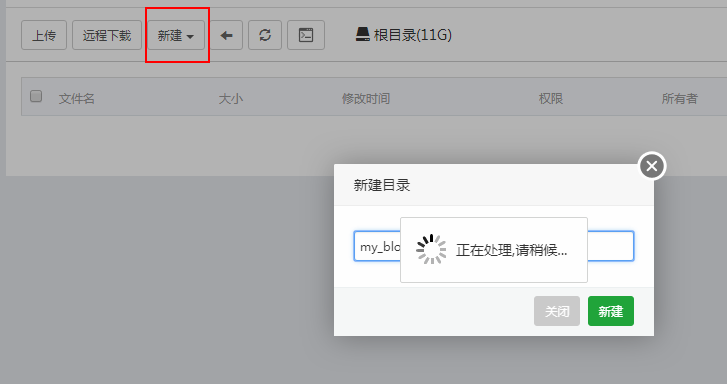


图13-9

上传文件（程序压缩包，必须是zip格式才能解压缩，因为是linux系统）

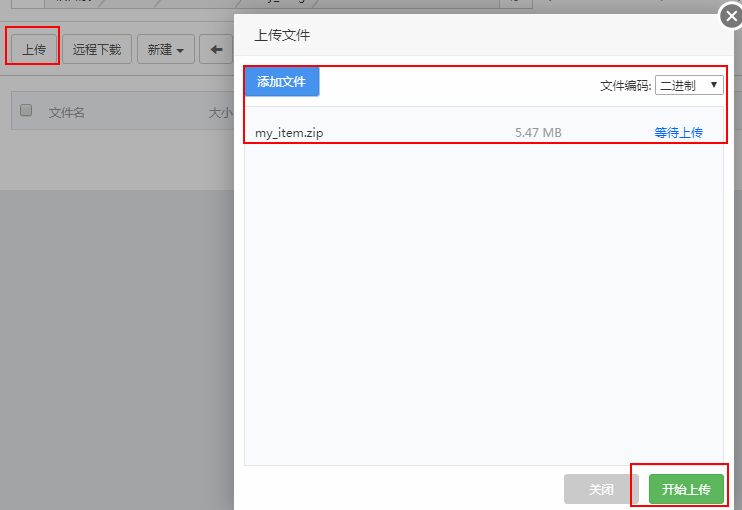


图13-10

解压到本目录下

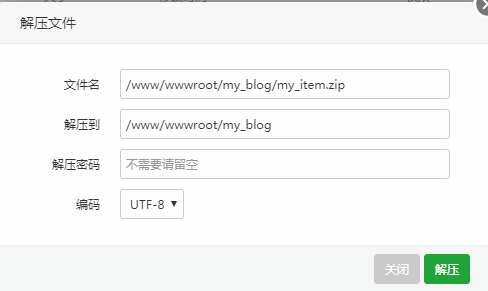


图13-11



图13-12

启动项目并添加。

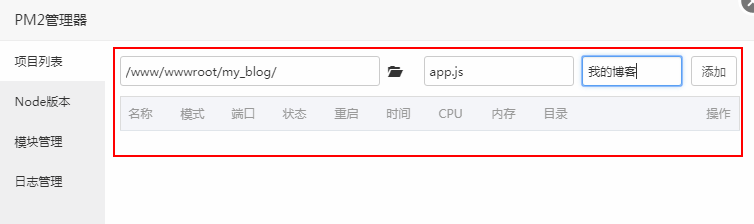


图13-13

然后就可以在浏览器中访问自己的项目了。

14.5 高频面试

● koa与express的区别

● 常见的web优化有哪些