**服务端**

### javaScript可以用来做什么

1. 最初，js被用来在浏览器端实现表单的验证。
2. 在浏览器端可以使用js操作dom实现各种各样的效果和动画。
3. 浏览器端的js可以处理数据，实现各种页面功能。

**那么运行在node上的JavaScript可以做什么呢？**

1. 可以制作命令行工具（所谓命令行工具就是通过dos命令来操作的软件）
2. 可以做服务端的开发 （基于node可以开发服务器应用，就像php一样。。。）

### nodejs是什么

1. nodejs是一款软件，就像qq一样，qq提供给我们的功能是交友，聊天。而**node的功能就是去执行JavaScript程序**。
2. 在使用方式上，会有一些区别，使用qq我们通过鼠标的点击就可以操作了。但是node是通过命令行去操作的。

**所以，nodejs其实就是为JavaScript提供运行环境的一款软件（windows版本，linux版本，mac版本）**

## nodejs快速入门

1. **什么是JavaScript？** Javascript是一门运行在浏览器端的脚本语言。

两大方面：

* + 在浏览器端运行的（还有在服务端运行的，比如php）
  + 脚本语言，嵌入到其他的语言中（插入到html中）

1. **JavaScript的运行环境是？** 就是浏览器。
2. **浏览器中的JavaScript 可以做什么？**

在浏览器端，主要是使用js来实现DOM操作。

* + 表单验证
  + 实现一些特性和动画
  + 更好的组织页面的结构（tab选项卡、轮播图）
  + 可以使用ajax来提升用户体验
  + 页游

1. **浏览器中的JavaScript 不可以做什么？** 作为一门编程语言，js和其他语言相比，它不能干什么？ 其他语言，比如c、c++、c#、java、php等 js不能做的事情如下：
   * 文件操作
   * 网络编程
   * os相关操作
   * 数据库操作
   * ...
2. JavaScript只可以运行在浏览器中吗？ 在09年以前，基本上只能在浏览器中运行的。 但是09年之后，情况发生了变化。

Javascript除了在浏览器中运行，它还可以在其他环境运行，比如服务端。

1. 编程语言的能力取决于什么？ 其实，各门语言的功能大同小异，基本都具备下面这些内容：
   * 变量和常量
   * 数据类型
   * 表达式
   * 语句（顺序、分支、循环）
   * 函数
   * 类和对象

### 什么是node.js？

官网：https://nodejs.org/en/

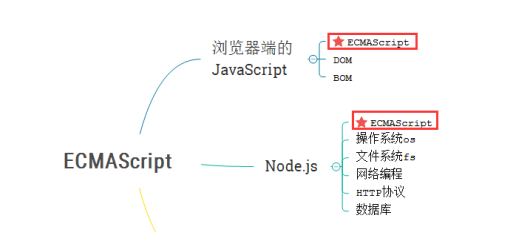
Node.js® is a JavaScript runtime built on Chrome's V8 JavaScript engine. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient. Node.js' package ecosystem, npm, is the largest ecosystem of open source libraries in the world.

中文网站：http://nodejs.cn/

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

记住一点：**node.js其实一个运行环境，和浏览器是同等地位。** 并且，**node.js这个运行环境是基于chrome v8引擎的，**说明在浏览器端写的那些和浏览器环境无关的代码，在node.js中，也是同样支持的。

有很多初学者认为：node.js是一门语言，这是极其错误的。



### nodejs可以做什么？

使用Node.js可以轻松的进行服务器端应用开发，PHP、Python、Ruby能做的事情Node.js几乎都能做，而且在某些方面可以做的更好。

世界上70%以上的网站都是使用php来做的。 世界上100%的网站都使用了js。

具体来说，体现在两个方面：

* webapp（网站）
* 命令行（包括命令行工具和交互式终端程序）

其中，我们的重点是放在webapp上面。 越来越多的网站，都开始大面积的使用node.js

* 阿里（天猫）
* uber
* Yahoo
* Paypal
* Group on

既然已经有了PHP等后台语言，为何还需要Node.js？

* io密集型 --- 异步io操作
* 实时web应用 ---- socket.io

io是指input/output，输入和输出，文件操作。 实时web应用：双十一、12306、游戏、在线客服/咨询

大家需要明白，一个大型的软件，通常是结合各种技术来实现的，并不会使用单一的技术。 比如，阿里用到了php、java、c++、node.js这几种语言，数据库就使用了mysql、oracle、mongodb、redis等数据库。

### [#](http://140.143.201.230:7070/notice/dist/outline/lesson5/1.html" \l "%E5%AD%A6%E4%B9%A0nodejs%E7%9A%84%E5%BF%85%E8%A6%81%E6%80%A7)学习nodejs的必要性

* 高薪资
* 企业的需求
* 更好的理解js语言
* 紧跟潮流

Node.JS超越jQuery成为Github上第二受欢迎的开源项目 GitHub公布10大热门编程语言：Javascript居首

JavaScript = BOM + DOM + ECMAScript

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

Node 是 JavaScript 的服务器运行环境（runtime）。它对 ES6 的支持度更高

### let 和 const 命令

let和const是es6中的两个声明标识符，除此之外还有class在后期会提到

**es5中声明变量可用var 和 function ，function的优先级较高，也就是说，浏览器会优先解析使用function声明的变量**

##### **const**

constant -- 不变的；恒定的

特点：**不能对变量重新赋值；产生块级作用域；不会变量提升；不能在声明时不赋值；**

**不能重复声明同一个变量；暂时性死区；不会挂载到window下**

1. 不能对变量重新赋值
   * 基本类型不能修改值
   * 引用类型不能修改地址
2. 产生块级作用域

**const** a = 'a';

**if** (true) {

**const** a = 'aa';

console.log(a);

}

*// 木有任何问题~*

1. 不会变量提升
2. 不能在声明时不赋值

**var** a; *// √*

**let** a; *// √*

**const** a; *// × Missing initia lizer in const declaration*

1. 不能重复声明同一个变量

**const** str = 'a';

var str = 'b'; *// Identifier 'str' has already been declared*

1. 暂时性死区

str = 'hello';

**const** str = 'world';

*// str is not defined;*

##### Let

特点：**能对变量重新赋值；产生块级作用域；不会变量提升；能在声明时不赋值；**

**不能重复声明同一个变量；暂时性死区；不会挂载到window下**

1. 能对变量重赋值
2. 产生块级作用域

for (let i = 0*; i < 5; i++) {*

setTimeout(() => {

console.log(i)*;*

}, 0)*;*

}

// es5模拟块级作用域

for (var i = 0*; i < 5; i++) {*

(function (i) {

setTimeout(() => {

console.log(i)*;*

}, 0)*;*

})(i)

}

// 都是0-4

1. 不会变量提升

**let** i = 2;

function **fn** () {

console.log(i);

**let** i = 9;

}

**fn**();

*// ReferenceError: i is not defined*

1. 能在声明时不赋值
2. 在同一作用域下不能重复声明同一个变量

let a = 2*;*

{

var a = 3*;*

}

console.log(a)*;*

**for** (**let** i = 0; i < 5; i++) {

**let** i = 10;

console.log(i); *// 5个10*

{

**let** i = 20;

console.log(i); *// 5个20*

}

}

**let** i = 30;

console.log(i); *// 30*

1. 暂时性死区

i = 1;

let i = 3;

// ReferenceError: i is **not** **defined**

1. 不会挂载到window下

**顶级对象**

浏览器 window

Node global

**结构赋值**

右边必须是可遍历解构 数组 字符串 对象 arguments

**1,解构不成功 左多右少 let[a]=[];**

**2,不完全解构 左少右多 let[a,b]=[1,2,3,4];**

允许指定默认值 let [str = 'holle'] = [];

console.log(str); //hello

let [str = 'holle'] = ['hi'];

console.log(str); //hi

值严格等于undefined

let [str = 'holle'] = [undefined];

console.log(str); //hello

foo：模式 b：变量名 ‘’aaa’ ：变量赋的值

let { foo: b, bar } = { foo: "aaa", bar: "bbb" };

console.log(b); //aaa

# Let []={} 报错

Let {}=[] true

# 解构赋值

ES6 允许按照一定模式，从数组和对象中提取值，对变量进行赋值，这被称为解构

### 数组解构赋值

let [a, b, c] = [1, 2, 3];

##### 解构不成功

如果解构不成功，变量的值就等于undefined。

let [foo] = [];

let [bar, foo] = [1];

以上两种情况都属于解构不成功，foo的值都会等于undefined。

##### 不完全解构

不完全解构，即等号左边的模式，只匹配一部分的等号右边的数组。这种情况下，解构依然可以成功。

let [x, y] = [1, 2, 3];

x *// 1*

y *// 2*

let [a, [b], d] = [1, [2, 3], 4];

a *// 1*

b *// 2*

d *// 4*

let [a, b, ...c] = [1];

console.log(a); *// 1*

console.log(b); *// undefined*

console.log(c); *// []*

如果等号的右边不是数组（或者严格地说，**不是可遍历的结构**（比如说类数组），那么将会报错。

*// 报错*

**let** [foo] = 1;

**let** [foo] = false;

**let** [foo] = NaN;

**let** [foo] = undefined;

**let** [foo] = null;

**let** [foo] = {};

let [a, ...b] = [1, 2, 3];

console.log(b) *// [2, 3]*

##### 默认值

解构赋值允许指定默认值。

**let** [foo = true] = [];

foo *// true*

**let** [x, y = 'b'] = ['a']; *// x='a', y='b'*

**let** [x, y = 'b'] = ['a', undefined]; *// x='a', y='b'*

注意，ES6 内部使用严格相等运算符（===），判断一个位置是否有值。所以，只有当一个数组成员**严格等于undefined**，默认值才会生效。

**let** [x = 1] = [undefined];

x *// 1*

**let** [x = 1] = [null];

x *// null*

如果默认值是一个表达式，那么这个表达式是惰性求值的，即只有在用到的时候，才会求值。

**function** **f**() {

console.log('aaa');

}

**let** [x = f()] = [1];

默认值可以引用解构赋值的其他变量，但该变量必须已经声明。

**let** [x = 1, y = x] = []; // x=1; y=1

**let** [x = 1,y = x] = [2]; // x=2; y=2

**let** [x = 1, y = x] = [1, 2]; // x=1; y=2

**let** [x = y, y = 1] = []; // ReferenceError: y is not defined

### 对象的解构赋值

**let** { foo: foo, bar: bar } = { foo: "aaa", bar: "bbb" };

*// 等号左边名为foo的key是模式，名为foo的val是变量*

*// 简写*

**let** { foo, bar } = { foo: "aaa", bar: "bbb" };

foo *// "aaa"*

bar *// "bbb"*

对象的解构与数组有一个重要的不同。数组的元素是按次序排列的，变量的取值由它的位置决定；而对象的属性没有次序，变量必须与属性同名，才能取到正确的值。

**let** { bar, foo } = { foo: "aaa", bar: "bbb" };

foo *// "aaa"*

bar *// "bbb"*

**let** { baz } = { foo: "aaa", bar: "bbb" };

baz *// undefined*

如果变量名与属性名不一致，必须写成下面这样。

let { foo: baz } = { foo: 'aaa', bar: 'bbb' };

baz *// "aaa"*

let obj = { first: 'hello', last: 'world' };

let { first: f, last: l } = obj;

f *// 'hello'*

l *// 'world'*

对象的解构也可以指定默认值。

**var** {x = 3} = {};

x *// 3*

**var** {x, y = 5} = {x: 1};

x *// 1*

y *// 5*

**var** {x: y = 3} = {};

y *// 3*

**var** {x: y = 3} = {x: 5};

y *// 5*

**var** { message: msg = 'Something went wrong' } = {};

msg *// "Something went wrong"*

**默认值生效的条件是，对象的属性值严格等于undefined。**

var {**x** = 3} = {**x**: undefined}*;*

**x** // 3

var {**x** = 3} = {**x**: **null**}*;*

**x** // **null**

如果解构失败，变量的值等于undefined。

**let** {foo} = {bar: 'baz'};

foo // undefined

如果解构模式是嵌套的对象，而且子对象所在的父属性不存在，那么将会报错。

// 报错

**let** {foo: {bar}} = {baz: 'baz'};

*// 重要*

**let {} = []; *// ok***

**let [] = {}; *// no***

*// 解构时等号右边的数据类型必须和左边保持一致*

**...运算符用于操作数组**

https://blog.csdn.net/chauncywu/article/details/73060918

1. 展开运算符
2. 剩余操作符

var str = 'hello';

console.log(str.includes('o', 3)) //true 以下标为3开始找有没有o；

**Includes 相当于indexof**

Indexof 存在返回下标 不存在返回-1；

Includes 存在返回true 反之 false； 有两个属性 2：以哪个下标开始

**startsWith endsWith repeat**

var str=’hello’;

str.startsWith('e', 1) //以下标为1的字符开始查找第一个字符是否为‘e' true

str.endsWith('o') //以哪个字符结束 true

str.endsWith('e',2) //2代表根据‘he’判断是不是以‘e’为结尾 true

str.repeat(2) //’hellohello’ 2代表重复次数 2.9,2.1 都会取2

str.repeat(0.9) //’’ 0.9,0.1，NaN会取0 返回空字符串

str.repeat(-1) //报错

str.repeat('sss') NaN会取0 返回空字符串 (括号里面传的会转为number类型)

str.repeat(‘2’) //’hellohello’ 字符串2通过number转为数字2；

let str = 'wangzhan';

console.log(str.padEnd(10, 'a')) //wangzhanaa

str.**padStart**(10, 'a') //aawangzhan

str.padEnd(2, 'a') //wangzhan

/\*\*

\* **push()** 在数组末尾添加一个元素

\*

\* **unshift()** 在数组的开头添加一个元素

\*

\* **pop()** 在数组的末尾删除一个元素

\*

\* **shift()** 在数组的开头删除一个元素

\*

\* **slice(startIndex,?endIndex=this.index)** 截取数组 不会改变原数组

\*

\* **splice(startIndex,num,newItem)** 截取数组 修改数组 会改变原数组

\*

\* 可以改变原数组

\*

\* **concat(arr1,arr2,...)** 合并数组

\*

\* **every(function(item,index,array){})** // 遍历数组，如果都满足条件返回true，有一项不符合返回false

\*

\* 返回值：Boolean

\*

\* **some(function(item,index,array){})**

\*

\* 功能：遍历数组

\*

\* 返回值：只要有一项符合条件就返回true Boolean

\*

\* **filter(function(item,index,array){})**

\*

\* 功能：过滤数组

\*

\* 返回值：将满足条件的所有项存到新的数组返回

\*

\* **forEach(function(item,index,array)(){},?thisObj)**

\*

\* 功能：循环数组 没有返回值

\*

\* **map(function(item,index,array){})**

\*

\* 功能：循环数组 有返回值

\*

\* **indexOf()**

\*

\* 功能：查找数组内是否包含指定元素的下标

\*

\* 返回值：number -1

\*

\* **join()**

\*

\* 功能：把数组以某个字符拼接字符串

\*

\* **reduce(function(pre,cur){})**

\*

\* 功能： 方法接收一个函数作为累加器，数组中的每个值（从左到右）开始缩减，最终计算为一个值。

\*

\* **reverse()**

\*

\* 功能：翻转数组

\*

\* **sort(function(a,b){})**

\*

\* 功能：排序

\*

\* **Array.isArray() typeof array 返回object**

\*

\* 功能：检测是否是数组

\*

\* 返回值：Boolean

\*

\*

**[...new Set(arr)]**  数值去重 返回值是一个对象

**类数组转数组**

**http://es6.ruanyifeng.com/#docs/array#Array-from**

Array.from

{

let arrayLike = {

'0': 'a',

'1': 'b',

'2': 'c',

length: 3

};

// ES5的写法

var arr1 = [].slice.call(arrayLike);

console.log(arr1)

// ES6的写法

let arr2 = Array.from(arrayLike);

console.log(arr1)

}

**对象里面的方法**

不可通过遍历得到的属性称为不可枚举属性

const obj = {

name: 'zs',

age: 20

};

//用于给对象里追加键值对或者替换键值对

//value数据属性

Object.defineProperty(obj, 'sex', {

value: '女'，

enumerable：true //可枚举 为true遍历可以拿到sex属性 name age sex 为false 是不可枚举

writable: true //可写 为true 可以通过obj.sex = '男'修改

configurable: true //可配置 可删除 可以通过delete obj.name; //{age: 20, sex: "女"}

}) //不可枚举 //name age

console.log(obj) //{name: "zs", age: 20, sex: "女"}

for (const i in obj) {

console.log(i) //name age

}

const flag = Object.getOwnPropertyDescriptor(obj, 'sex');

//获取属性

console.log(flag) //女

**//数据属性 value enumerable writable configurable**

**//访问器属性 set get**

**//两者不能共存**

**// Object.is() 判断两个值是否相等**

console.log(Object.is('2', 2)) // 2==='2' false

console.log(Object.is(0, -0)) //false

console.log(0 === -0) //true

console.log(NaN === NaN) //false

console.log(Object.is(NaN, NaN)) //true

const obj = {

name: 'zs',

age: 12

}

console.log(Object.entries(obj)) // [[name,zs], [age,12]]

console.log(Object.keys(obj)) //['name','gae']

console.log(Object.values(obj)) //['zs','12']

const entries = Object.entries(obj); // **遍历器对象**

for (const i of entries) {

console.log(i) //[name,zs], [age,12]

}

**Object.assign({}, obj1, obj2);**

* 合并对象
* 返回第一个参数 默认修改第一个参数
* 只能合并可枚举属性

**let** obj1 = {

name: 'zs',

age: 12,

sex: 'man'

}

**let** obj2 = {

name: 'ww',

age: 20,

class: '1610'

}

**function** **extend** (obj, ...arg) {

arg.forEach(**function** (file) {

**for** (**let** i **in** file) {

obj[i] = file[i];

}

})

**return** obj;

}

**let** obj = extend({}, obj1, obj2);

console.log(obj);

**let** objs = Object.assign({}, obj1, obj2);

console.log(objs);

**Object.getOwnPropertyDescriptor(obj, 属性名, [配置项]);**

* 获取当前对象的属性的描述对象
* 返回值为对象

**let** obj = {

name: 'zs',

age: 12

}

**let** o = Object.getOwnPropertyDescriptor(obj, 'name');

console.log(o);

*<!--{ value: 'zs', 属性值-->*

*<!-- writable: true, 是否可编辑-->*

*<!-- enumerable: true, 是否可枚举-->*

*<!-- configurable: true 是否可配置（删除）-->*

*<!--}-->*

**Object.defineProperty(obj, 属性名, [配置项]);**

* 定义属性

**let** obj = {

name: 'zs'

}

Object.defineProperty(obj, 'age', {

value: 12,

configurable: true, *// 默认false*

enumerable: true, *// 默认为false*

writable: true *// 默认false*

})

obj.age = 22;

console.log(obj)

console.log(Object.keys(obj)) *// 打印不出来的为不可枚举属性*

Object.defineProperty(obj, 'aa', {

**get** () { *// 当使用该键值对时函数被触发*

console.log(0)

**return** 's'

},

**set**(val) { *// 当设置该变量时函数触发*

console.log(val);

}

})

*// 使用了get和set函数之后 aa被称之为访问器属性，不适用时属性称之为数据属性*

obj.aa = 'ss';

console.log(obj.aa)

**Object.keys(obj);**

**Object.values(obj);**

**Object.entries(obj);**

这三个返回的也是一个遍历器对象，可以使用for of 进行遍历

**[Object.is](http://Object.is" \t "_blank)(v1, v2) 比较两个值是否完全相等**

NaN === NaN *// false NaN和任何数据类型都不想等包括自身*

Object.is(NaN, NaN) *// true*

**copyWithin**

https://www.cnblogs.com/kongxianghai/p/7527177.html

**箭头函数**

**箭头函数只用于匿名函数**

**this指向**

****箭头函数如果在深层嵌套的对象中，this始终取决于最外层的对象的this指向****

****如果箭头函数嵌套在函数里，那么他的this指向取决于外层函数的this指向****

箭头函数无法用call apply bind 改变this指向

**Promise 的含义**

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

**优点**

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，使得控制异步操作更加容易。

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 方法和 reject 方法。

如果异步操作成功，则用 resolve 方法将 Promise 对象的状态，从「未完成」变为「成功」（即从 pending（准备状态） 变为 resolved（成功））；

如果异步操作失败，则用 reject 方法将 Promise 对象的状态，从「未完成」变为「失败」（即从 pending 变为 rejected（失败））。



promise.prototype.then

then里面的第二个参数是捕捉reject的异常的

catch 是then里面的第二个参数的别名 不但能捕捉reject的异常，还能捕捉上一个then链中的语法错误

finally 不管成功与否都会走的一个方法

const res = Promise.resolve('success');

res.then((data) => {

console.log(data)

})

const err = Promise.reject('error');

err.catch((data) => {

console.log(data)

})

{

function p1() {

return new Promise((resolve, reject) => {

resolve('p1')

})

}

function p2() {

return new Promise((resolve, reject) => {

resolve('p2')

})

}

function p3() {

return new Promise((resolve, reject) => {

resolve('p3')

})

}

Promise.all([p1(), p2(), p3()]).then((data) => {

console.log(data) // 都为resolve 打印["p1", "p2", "p3"]

}).catch((data) => {

console.log(data) //假如p2为reject打印 p2

})

**async await**

// async用于声明函数是个异步函数 async function

// async function start() {

// }

// console.log(start())

// await用于等待一个异步函数的执行

// const sleep = time => {

// return new Promise((resolve, reject) => {

// setTimeout(() => {

// resolve('hello11')

// }, time)

// })

// }

// const start = async() => {

// // await 函数调用 await fn() -> 必须返回promise

// const res = await sleep(2000); // await必须等到promise给出状态之后才会执行下面的代码

// console.log(res);

// console.log('end');

// }

// console.log(0)

// start();

**Class**

**Class没有声明提升**

**不能重名**

**必须new调用**

class Fn {

constructor(name) {

console.log(this) //Fn构造函数

this.name = name;

console.log(this.name) //1

}

init() {

console.log(this) //Fn构造函数

console.log(this.name) //1

}

}

new Fn(1).init()

**构造函数继承**

function Person(name, age) {

this.name = name;

console.log(this.name)

}

Person.prototype.sagName = function() {

return this.name

}

let p = new Person('zs');

console.log(p)

function Student() {

Person.apply(this, arguments);

console.log(this.name)

}

**Student.prototype = Object.create(Person.prototype);**

let s = new Student('zz');

console.log(s.sagName())

class Person {

constructor(name, age) {

this.name = name;

this.age = age;

console.log(this)

}

sayName() {

//alert(this.name)

}

}

new Person()

// class 子类 extends 父类

class Student extends Person {

constructor(name, age) {

super(); //必须调用

// super() 加括号是方法 改变this指向

this.name = name;

this.age = age;

super.sayName() //super. 是调用父类原型对象的函数

}

sayAge() {

alert(this.age)

}

}

new Student('zs', 22).sayAge()

**commonjs**

有多人协助或者大量的js文件批量引入一个页面时，很容易产生变量名冲突被覆盖掉或者方法被重写，特别是当有一些依赖关系的时候，页面特别容易出错，这是因为js天生就缺少一种模块管理机制来隔离实现不同功能的js片段。

* 介绍
  + commonjs是一种规范 因为js没有模块的功能所以CommonJS应运而生，它希望js可以在任何地方运行，不只是浏览器中
  + **NodeJS是commonjs规范的实现**
  + **Node程序由许多个模块组成，每个模块就是一个文件。Node模块采用了CommonJS规范**
  + 根据CommonJS规范，**一个单独的文件就是一个模块。每一个模块都是一个单独的作用域**，也就是说，在一个文件定义的变量（还包括函数和类），都是私有的，对其他文件是不可见的。
  + (function (require, exports, module) {
    - 所有变量都包在匿名函数里，所以不会污染全局变量
* **require**:
  + 1. require命令是CommonJS规范之中，用来加载其他模块的命令。
  + 2. require命令的基本功能是，读入并执行一个JavaScript文件，然后返回该模块的**module.exports**对象。如果没有发现指定模块，会报错。
  + 如果指定的模块文件没有发现，Node会尝试为文件名添加.js、.json、.node后缀，再去搜索。
  + 3. 根据参数的不同格式，require命令去不同路径寻找模块文件。
    - 如果参数字符串以“/”开头，则表示加载的是一个位于**绝对路径**的模块文件 c盘 /
    - 如果参数字符串以“./”开头，则表示加载的是一个位于**相对路径**（跟当前执行脚本的位置相比）的模块文件。 ./ ../
    - 如果参数字符串不以“./“或”/“开头，则表示加载的是一个默认提供的核心模块（位于Node的系统安装目录中-> 内置），或者一个位于各级node\_modules目录的已安装模块->第三包（全局安装或局部安装）。
  + 4.require发现参数字符串指向一个目录以后，会自动查看该目录的package.json文件，然后加载main字段指定的入口文件。如果package.json文件没有main字段，则会加载该目录下的index.js文件作为入口文件
* **module**:
  + 1. 每个模块内部，都有一个module对象，代表当前模块。
  + 2.Node内部提供一个Module构建函数。所有模块都是Module的实例。
  + 3.module属性：
    - module.id 模块的识别符，通常是带有绝对路径的模块文件名。
    - module.filename 模块的文件名，带有绝对路径。
    - module.loaded 返回一个布尔值，表示模块是否已经完成加载。
    - module.parent 返回一个对象 / null，表示调用该模块的模块。
    - module.children 返回一个数组，表示该模块要用到的其他模块。
    - **module.exports 表示模块对外输出的值**。
  + 4.CommonJS规定**，每个文件的对外接口是module.exports对象**。这个对象的所有属性和方法，都可以被其他文件导入。当调用require方法引入模块时，其实是加载该模块的module.exports属性。
  + 5**.module.exports 初始值为一个空对象 {}。**
* exports:
  + 为了方便，Node为每个模块提供一个exports变量，指向module.exports。 var e = m.e
* module.exports和exports区别：
  + **module.exports抛出当前内容形式，exports抛出对象**
  + reason -> **当调用require方法引入模块时，实则引入的是当前模块的module.exports属性值，Node为每个模块提供一个exports变量，指向module.exports，如果直接给exports赋值，相当于切断了它与module.exports的联系，此时exports变量将不再指向module.exports属性。所以只能采用向exports变量中追加键值对的方式将其抛出**
* 模块缓存：
  + 模块可以多次加载，但是只会在第一次加载时运行一次，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
* require.main属性指向模块本身。
* 清除模块的缓存：
  + **Object.keys(require.cache).forEach(function (file) {**
    - **delete require.cache[file];**
  + **});**
* CommonJS模块的特点如下
  + 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，但是只会在第一次加载时运行一次，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 require.cache
  + 模块加载的顺序，按照其在代码中出现的顺序。
* 如果想在多个文件分享变量，必须定义为global对象的属性。

**node**

npm n:node p:package m:manager node包管理工具

node模块

·核心模块（内置）：安装node存在硬盘 fs http url require(‘模块名’)

·用户模块

自定义模块

第三方包-->npm社区 [www.npmjs.com](http://www.npmjs.com)

exe/cmd ----->可执行文件

exe 双击、命令执行

cmd 命令行程序 命令执行 （由node编写的）

npm是node.js的包管理器

下载包

npm是nodejs的包管理器==>包（用户模块）可下载、卸载、更新、发布；

下载：npm install 包名

下载特定版本：npm i 包名@版本号

如果不指定版本号，默认下载最稳定的版本

npm-->包管理器

node-modules-->存包

package.json-->包管理文件（npm init [-y]）

name:包名 默认以所在文件夹命名 不能有汉字和大写字母 不能与社区已存在的包重名

version 版本号

修改规则 如果修复了上个版本的bug 改第三位1.0.1

如果新增功能且兼容上个版本 该第二位 1.1.0

如果新增功能但不兼容上个版本 改第一位 2.0.0

description 描述

entry point 入口文件

keywords 搜包

author 作者

dependencies 依赖 生产 项目依赖

devDependencies 开发依赖 开发 开发环境 工具依赖

**下载包、**

npm init 固定包解构 不加-y得自己配置

nmp init -y 直接生成json文件

下载命令

npm i 包

npm i 包 --save/-S

npm i 包@版本号 下载指定版本号的包

以上在package.json生成dependencies项目依赖 （生产）

npm i 包 --save-dev/-D

在package.json生成devDependencies 开发依赖

本地存储包取决与它依赖的什么包

node root 查找本地第三方包路径

**本地卸载命令**

npm uninstall 包

**查看npm的默认配置**

npm config ls/npm config list

查看本地第三方包

npm li/list

当输入npm i 命令时，npm会自动检查本地的p.j文件，如果p.j中设置了依赖（不管是dep还是devDep），npm都会将依赖模块下到本地，反之，如果npm中有A这个包，

**批量下包**

npm i 包 包 包 。。。。 包之间用空格隔开

手动在dependencies加字段 不知道版本号写’\*‘，在终端写npm i

**全局安装包**

npm i 包 -g/--global

npm root -g 查看npm的全局路径

加上-g参数只能保证把包下载到npm root -g 所指向路径中 要想被全局访问到 必须配置 NODE\_PATH 环境变量 ---->环境变量 新建 名：NODE\_PATH 值为绝对路径

**全局卸载命令**

npm uninstall 包 -g

**更改、查看 全局路径**

npm root -g 查看全局路径

缓存路径 npm config get cache

第三方包路径 你拼命config get prefix

更改

先改缓存 再改包路径

npm config set cache 更改的路径

npm config set prefix 更改的路径 更改原地址 ’Http://registy-npm.taobao.org‘

**加载机制**

require 查找机制 从当前工作目录（命令执行位置）逐步向上查找 直到碰到磁盘根目录 如果找不到 则访问NODE\_PATH 环境变量 NODE\_PATH将一串路径（全局路径）返回给系统 系统再顺着路径去查找

**当require 发现参数指向一个目录 实则加载的是这一目录下的某一个js文件 该文件被称为入口文件**

**每个包必须有入口文件**

**配置入口文件**

通过包内的package.json 指定其main字段

通过配置index.js作为入口文件 只是package.json里面没有main字段了 但是还是需要package.json

main 字段优先级比index.js高

内置模块

用户模块 ---->第三方包 自定义模块

**用户模块**：1、自定义

2、第三方

用户模块特点：1、一定是文件夹；

2、必须存于node\_modules;

3、包内必须配置package.json;

4、必须指定入口文件：·package.json-->main-->优先级高

·index

5、入口文件必须提供对外接口（抛出）

**xx=aa&aa==bb 序列化的字符串**

**{xx:oo,aa:bb} 反序列化得对象**

**{“xx”:”oo”,”aa”:”bb”} json格式的对象**

**{“xx”:”oo”,”aa”:”bb”} json格式的字符串**

**fs**

**操作磁盘文件（目录）的内置模块**

引入

require（‘fs’） 返回一个对象

fs中每一个方法都有同步和异步两种形式

**同步方法名以sync结尾**

**异步方法 最后一个参数必须是回调函数 值返回到回调函数里 异步方法里没有返回值**

**同步的返回值是异步回调函数的第二个参数**

**如果异步回调函数只有一个参数 同步没有返回值**

读取**文件**

const fs=require(‘fs’);

Fs.readFile(‘文件路径’，[opt/str]，function(err,data){})

fs.readFile('./index.html', { encoding: 'utf-8', flag: 'r+' }, function(err, data) {

//不写 flag: 'r+' 完全式覆盖

//callback是读取的动作完成之后执行的回调

//console.log(err) //如果读取成功 是null 失败 报错并返回一个对象

//console.log(data) //如果读取成功 是文件内容 失败是undefined

// if (err) {

// // return console.log(err);

// throw err; //失败抛出一个错误

// }

console.log(data) //./index.html的所有内容

})

异步写法

fs.readFileSync(‘./index.html’,’utf-8’)

写入

fs.writeFileSync(‘./index.html’,写入的内容，{flag:’a+’})

{flag:’a+’} 追加内容 不写为 完全覆盖

{flag:’r+’} 从开始开始覆盖 不写为全部覆盖

追加式写入

fs.appendFileSync(文件，追加的内容）

读取**文件夹**

fs.readdir('./node\_modules/', (err, paths) => {

if (err) {

throw err;

}

console.log(paths) //[ 'a.js', 'urls' ] 返回一个数值包含文件夹里的每一项

})

同步

fs.readdirSync(‘./lib’); // 返回值既是结果集/**异步回调函数第二个参数的值**

**fs创建文件夹**

fs.mkdir(文件夹, function (err) {}) // 异步

fs.mkdirSync(文件夹); // 同步

fs.mkdir('./src', function (err) { // null

if (err) {

throw err;

}

});

**删除空文件**  只能使用同步删除

fs.rmdirSync('./src') //src必须是一个空文件

**fs判断 文件 / 文件夹是否存在**

异步:

fs.exists('./style.css', function (exist) {

console.log(exist); // 返回布尔值,true -> 存在 false-> 不存在

});

同步：

console.log(fs.existsSync(‘./main.js’)); // 返回布尔值

**fs查看文件/文件夹信息 (大小，创建时间，类型)**

异步

fs.stat('lib', function (err, st) {

if (err) {

throw err;

}

console.log(st.isDirectory()); // 返回布尔值 是不是文件夹

});

同步

Fs.statSync(‘lib’)

Fs.unlink() 删除已有文件夹 没有这个文件夹 就会报错

**小文件拷贝**

**同步拷贝**

**fs.writeFileSync('./b.js', fs.readFileSync('./a.js’));**

**异步拷贝**

**fs.readFile('./a.js', function (err, data) {**

**if (err) {**

**throw err;**

**}**

**fs.writeFile('./b.js', data, function (err) {**

**if (err) {**

**throw err;**

**}**

**});**

**});**

**大文件拷贝 流**

由于数据是流，这就意味着在完成文件读取之前， 从收到最初几个字节开始，就可以对数据动作。

由于一次性读取再写入的方式不适用与大文件拷贝，容易造成内存爆仓，对于大文件，我们只能读一点儿写一点儿直到拷贝完成

**流操作**

创建一个只读数据流

var rs = fs.createReadStream('./src/1.mp4’);

创建一个只写数据流

var ws = fs.createWriteStream('./src/5.mp4’);

管道流输出

rs.pipe(ws); // 管道

**path**

Node.js path 模块提供了一些用于**处理文件路径**的小工具

require（‘path’）

**格式化路径**

**path.normalize(p)**

该方法可以将不标准的路径转为标准的路径

Eg:

console.log(path.normalize('C://src/////lib///a.js’));

返回C:/src/lib/a.js

提供特定于平台的路径分隔符

path.delimiter

; 对于Windows

: 为POSIX

**path.extname(p)**

**获取文件后缀名。**

如果**一个路径中并不包含'.'或该路径只包含一个'.' 或这个'.'为路径的第一个字符，则此命令返回空字符串**

Eg

console.log(path.extname('index.txt.md’)); // .md

console.log(path.extname('.src’)); // ‘’

console.log(path.extname('.’)); // ‘’

console.log(path.extname(‘’)); // ‘’

console.log(path.extname('src.')); // .

**path.join() 可以传0-多个参数**

使用平台特定的分隔符将所有给定的段连接在一起，然后对结果路径进行规范化。

Eg:

console.log(path.join('src', 'a.js’)); // src/a.js

**返回代表文件夹的部分**

**path.dirname()**

console.log(path.dirname('./src/a.js')) // ./src

**返回路径的最后一部分**

**path.basename(path ,[ext])**

**ext 可选文件扩展名**

console.log(path.basename('C://src/b/a.js', '.js')); //a

console.log(path.basename('C://src/b/a.js')) //a.js

Eg:

**规则，组合成一个文件路径。**

**path.parse(filepath)：path.format()方法的反向操作。**

Eg:

var add = 'C://src/lib/a.js';

var obj = path.parse(add);

{ root: 'C:/',

dir: 'C://src/lib',

base: 'a.js',

ext: '.js',

name: 'a' }

var str = path.format(obj);

// = 'C://src/lib/a.js';

**服务器**

**http协议 超文本传输协议**

超文本有两大类

1. 静态资源 html、css、js、img、MP3.......------->存在于本地磁盘
2. Json数据 -----> 数据库 存json数据的（xxx.json）

传输

是从服务器把超文本传到客服端

**客服端**  client

发起请求的一方

**服务器**

接受请求 处理请求 并作出相应

**状态：服务器给客服端的**

自己的端口号（本地浏览器地址栏） 127.0.0.1 ========localhost

const http = require('http');

const fs = require('fs');

//console.log(fs.readFileSync('./index.html', 'utf-8'))

/**/搭建服务器 response响应 request请求 listen 开启服务**

http.createServer(function(request, response) {

//请求头

response.writeHead(状态码, {

/'Content-Type': 'text/plain;charset=utf-8',

'Content-length': 6 //3代表一个汉字

})

**callback在什么时候执行 客服端发起请求的时候**

**callback的功能:接受请求 处理请求 作出响应**

**callback的执行次数和请求次数有关**

**刷新一下有两个请求 一个是地址栏请求 一个是favicon.ico（图标）请求**

console.log(request.url) //以/开头 路径+？+参数

console.log（request.method）//GET GET 通过地址栏(link、script、img)发起的请求一定是get方式

response.statusCode=200 //设置响应的状态码

response.end(fs.readFileSync('./index.html', 'utf-8'))

}).listen(9090) //通过localhoat:9090 可以再自己浏览器显示index.html的内容

**Post请求** form表单 ajax

request参数是一个对象，表示客户端的HTTP请求；response参数也是一个对象，表示服务器端的HTTP回应。

**response.write() 设置响应内容**

**response.end() 结束响应**

write end 里面只能放字符串 或者 buffer

**以上也等于 response.end(设置的内容)**

**服务器地址组成**

协议+IP+端口+路径+[参数]+#

协议：http https mongodb 默认http

IP：169.254....

端口： 默认80 ->网站 27017-->mongodb

路径： 默认：/ 不显示 所有请求路径都以/开始

参数： ?k=v&k1=v1

<http://localhost:8080/路径>?k=v&k1=v1

**状态码**

200 OK

302 缓存

304 重定向

400 Bad request 错误的请求

403 Forbidden 禁止访问

404 找不到页面 Not Found

500 服务器异常

200-300 没有问题

400-500 客服端有问题

500 前端问题

node-dev 自动重启

**Content-Type： 请求头**

text/plain纯文本

text/html

text/css

text/javascript

image/png

image/jpg

application/json json数据

console.log(**\_\_dirname**) 表示当前改变量所在文件（不包括当前文件）的绝对路径

console.log(**\_\_filename**) 表示当前改变量所在文件（包括当前文件）的绝对路径

**加载静态资源**

const fs = require('fs');

const path = require('path');

const url = require('url');

const http = require('http');

const text = {

'.js': 'text/javascript',

'.html': 'text/html',

'.css': 'text/css'

}

//加载静态资源

// http.createServer((request, response) => {

// if (request.url.includes('favicon.ico')) {

// return response.end()

// }

// let pathname = url.parse(request.url).pathname == '/' ? '/index.html' : url.parse(request.url).pathname;

// //console.log(path.extname(pathname))

// if (fs.existsSync(\_\_dirname + pathname)) {

// response.setHeader('Content-Type', text[path.extname(pathname)] || 'text/plain');

// response.end(fs.readFileSync(\_\_dirname + pathname))

// } else {

// response.writeHead(400, {

// "Content-Type": "text/html;charset=utf-8"

// })

// response.end('no found')

// }

// }).listen('9999')

**前后端交互**

前端：收集数据，传递数据

（将数据以参数的形式传递）

传递方式：get post

get:安全性低 暴露了地址栏 （通过地址栏/头传播）

post：传输数据量大

传递：（ajax---->不跳,form--->跳转页面）

后台：接收数据 存储数据

接收方式：get post

存储数据到数据库 先存到json文件里面

**当post数据量较大时 网络传输过程中会将数据分段，每次发送一段数据，后台的data事件就会触发一次**

有？ get 用url模块

没有？ post 用queryscript模块

//get

// console.log(url.parse(request.url, true).query);

// let str = '';

// //post流 格式 xx.on('data')

// request.on('data', buf => {

// str += buf;

// })

// request.on('end', () => {

// console.log(require('querystring').parse(str))

// })

// 1、设置让服务器支持客户端的跨域访问

// response.setHeader('Access-Control-Allow-Origin', '\*')

// 2、将html放到同源环境下

if (request.url === '/a.html') {

response.end(require('fs').readFileSync('./www/a.html'))

} else {

response.end('hello')

}

**url**

<http://localhost:8080/a.html?name=zs&age=22>

http: 协议（protocol）

localhost 域名（hostname）

8080 端口（port）

**/a.html 路径（pathname）**

?name=zs&age=22 （search）

**name=zs&age=22 参权 （query）**

**path=pathname+search======>request.url**

**解决跨域**

res.setHeader('Access-Control-Allow-Origin', '\*')

**gulp**

**全局安装是为了使用gulp的指令**

**npm i gulp -g**

**css的:**

**gulp.src**

**- 功能: 读取**

**gulp-sass**

**- 功能: 编译为css**

**gulp-autoprefixer**

**- 功能: 添加内核前缀**

**gulp-concat**

**- 功能: 合并**

**gulp-clear-css**

**- 功能: 压缩**

**JS的**

**gulp.src**

**- 读取**

**gulp-babel es6->es5**

**- 功能: 读取**

**- 依赖:**

**babel-core**

**babel-preset-es2015**

**babel-preset-env**

**gulp-concat**

**- 功能:合并**

**gulp-uglify**

**- 压缩**

**gulp.dest**

**- 输出**

**关于html**

**gulp.src**

**- 读取**

**gulp-htmlmin**

**- 压缩**

**gulp.dest**

**- 输出**

**gulp.src 读取文件**

**gulp.dest 写文件**

**gulp.watch 监听文件变化**

**gulp.watch（文件路径，文件改动时要执行的任务）**

**gulp.task 注册任务**

**\*匹配所有文件 eg：src/\*.js ------>找src 下的所有文件**

**！除去某个文件 eg：[‘src/\*.js’,‘！src/a.js’] ------>找出来a.js之外的所有文件**

**\*\*匹配0-多个文件夹 eg：src/\*\*/\*.js------->找src下的所有文件夹中的js文件**

**{} 匹配类型文件 eg：src/{a,b}.js------>找src下的a.js b.js**

**src/\*.{png,jpg,gif}------->找src下所有类型的图片**

1. **安装**
2. **gulpfile.js gulp默认执行文件**
3. **引入gulp require(‘gulp’)**
4. **注册任务 gulp.task（任务名（默认default），callback） 回调函数一定要用return**
5. **执行任务 命令gulp 任务名/gulp**
6. **下载npm i babel-cli -g**
7. **本地下载 babel babel-core babel-preset-es2015 babel-preset-env**
8. **执行编译**

**命令：1.babel a.js -o b.js 把a.js完全复制到b.js**

**2.babel a.js -o b.js --presets=es2015 es6转为es5复制到b.js**

**3.babel src -d lib 把文件夹src下的所有js文件编译到lib文件**

**.babelrc**

**console.time(‘time’)**

**console.timeEnd(‘time’) 计算开始到结束用的时间**

**express**

//web开发框架 -->搭建服务器

//express=路由+中间件

**//1、下载+引入**

const express = require('express')

**//2、创建应用/express的实例化**

const app = express();

**//3、注册路由-->接受请求，处理请求、做出响应**

**//请求--->请求方式+请求路径 app.get app.post**

//<1> 路由构成：http请求方法+路径

//<2>路由作用： **接受请求，处理请求、做出响应**

//<3> 路由后面的callback什么时候执行：当路由匹配时->路由匹配=方式匹配+路径

//路由有从上到下的匹配机制 上面如果完全匹配就不会往下面走 想要往下走加next()

**//next()移交控制权 将控制权交给下一个处理器函数**

**//next('route') 移交控制权 跳过处理器函数，直接将控制权交给下一个同名路由**

//callback可以叫处理器函数/路由句柄

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

res.end('ab?cd')

}) //？相当于正则里面的？ ？0-1次 匹配b 0到1次

//ab+cd + 1到多次 匹配b 1到多次

//ab\*cd 匹配ab开头cd结尾的

app.get('/src', (req, res, next) => {

res.end('hello src')

})

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

res.end('hello /')

})

app.listen(8080)//web开发框架 -->搭建服务器

process.cwd() 当前工作目录

resolve()

1、 不传参->返回当前工作目录

2、 如果在拼接完给定的路径后，还没有生成绝对路径，该方法会再次以当期工作目录进行拼接

3、路径从右往左依次拼接，找到绝对路径就停止拼接，直接返回

express简介：

**Express 是一个自身功能极简，完全是由路由和中间件构成一个的 web 开发框架**

路由：

1、路由（Routing）是由一个 URL（或者叫路径）和一个特定的 HTTP 方法（GET、POST 等）组成的，涉及到应用如何响应客户端对某个网站节点的访问。

2、路由方法源于 HTTP 请求方法，和 express 实例相关联。

3、每一个路由都可以有一个或者多个处理器函数，当匹配到路由时，这些函数将被执行。

4、路由的定义由如下结构组成：app.METHOD(PATH, HANDLER)。其中，app 是一个 express 实例；METHOD 是某个 HTTP 请求方式中的一个；PATH 是服务器端的路径；HANDLER 是当路由匹配到时需要执行的函数。

注：app.all() 是一个特殊的路由方法，没有任何 HTTP 方法与其对应，它的作用是对于一个路径上的所有请求加载中间件。

5、路由路径：路由路径和请求方法一起定义了请求的端点，它可以是字符串、字符串模式或者正则表达式。

// 匹配 acd 和 abcd

app.get('/ab?cd', function(req, res) {

res.send('ab?cd');

});

// 匹配 abcd、abbcd、abbbcd等

app.get('/ab+cd', function(req, res) {

res.send('ab+cd');

});

// 匹配 abcd、abxcd、abRABDOMcd、ab123cd等

app.get('/ab\*cd', function(req, res) {

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

});

// 匹配 /abe 和 /abcde

app.get('/ab(cd)?e', function(req, res) {

res.send('ab(cd)?e');

});

// 匹配任何路径中含有 a 的路径：

app.get(/a/, function(req, res) {

res.send('/a/');

});

// 匹配 butterfly、dragonfly，不匹配 butterflyman、dragonfly man等

app.get(/.\*fly$/, function(req, res) {

res.send('/.\*fly$/');

});

6、路由句柄

可以为请求处理提供多个回调函数，这些回调函数有可能调用 next('route') 方法而略过其他路由回调函数。可以利用该机制为路由定义前提条件，如果在现有路径上继续执行没有意义，则可将控制权交给剩下的路径。

重要：next

如果当前中间件没有终结请求-响应循环，则必须调用 next() 方法将控制权交给下一个中间件，否则请求就会挂起。

next函数主要负责将控制权交给下一个中间件，如果当前中间件没有终结请求，并且next没有被调用，那么请求将被挂起，后边定义的中间件将得不到被执行的机会

next函数主要是用来确保所有注册的中间件被一个接一个的执行，如果我们定义的中间件终结了本次请求，那就不应该再调用next函数，

它负责同一个路由的多个中间件的控制权的传递，并且它会接收一个参数"route"，如果调用next(“route”)，则会跳过当前路由的其它中间件，直接将控制权交给下一个路由

7、路由句柄有多种形式，可以是一个函数、一个函数数组，或者是两者混合

**响应方式：**

res.download() 提示下载文件。

**res.end() 终结响应处理流程。**

**res.json() 发送一个 JSON 格式的响应。**

**res.header('content-type','application/json')**

**设置json格式的响应 header+end**

**res.jsonp() 发送一个支持 JSONP 的 JSON 格式的响应。**

res.redirect() 重定向请求。

res.render() 渲染视图模板。

**res.send() 发送各种类型的响应。 如果为数字则转为状态码 header+end**

**res.sendFile() 里面放绝对路径**

**res.sendStatus() 设置响应状态代码，并将其以字符串形式作为响应体的一部分发送。**

**app.route()**

可使用 app.route() 创建路由路径的链式路由句柄。由于路径在一个地方指定，这样做有助于创建模块化的路由，而且减少了代码冗余和拼写错误。

express.Router

可使用 express.Router 类创建模块化、可挂载的路由句柄。

中间件概念：

中间件（Middleware） 是一个函数，它可以访问请求对象（request object (req)）, 响应对象（response object (res)）, 和 web 应用中处于请求-响应循环流程中的中间件，一般被命名为 next 的变量。

分类：

应用级中间件

路由级中间件

错误处理中间件

内置中间件

第三方中间件

应用级中间件

**express.static 处理静态资源**

应用级中间件绑定到 app 对象 使用 app.use() 和 app.METHOD()

路由级中间件

路由级中间件和应用级中间件一样，只是它绑定的对象为 express.Router()。

错误处理中间件

内置中间件

express通过 Express 内置的 express.static 可以方便地托管静态文件，例如图片、CSS、JavaScript 文件等。

如果你的静态资源存放在多个目录下面，你可以多次调用 express.static 中间件

1、该root参数指定从中提供静态资产的根目录。该函数通过req.url与提供的root目录结合来确定要提供的文件。

2、当找不到文件时，它不是发送404响应，而是调用next（）继续下一个中间件

**第三方：**

**bodyParser中间件用来解析http请求体，是express默认使用的中间件之一**

**bodyParser.urlencoded则是用来解析我们通常的form表单提交的数据**

**当extended为false的时候，键值对中的值就为'String'或'Array'形式**

app.route() 路由链式路由句柄

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

fn

res.end('a1')

})

//应用级中间件

app.use((req, res, next) => {

//next(错误对象) 将控制权交给下一个错误处理中间件;

//res.status(404).end()

const err = new Error('很抱歉 页面跑丢了');

next(err)

})

//错误处理中间件 四个参数 路由中发生异常时执行

app.use((err, req, res, next) => {

console.log(err.message) // err.message 捕捉的异常

res.status(500).end('err')

})

**数据库**

Mongodb 数据库

Robomongo 可视化工具

关系数据库：Mysql 表格的形式

Id user pwd

1 zs 123

2 ls 456

非关系数据库：Mongodb ------》json格式

Mongodb=文档+集合+数据库

文档:相当于mysql的每一行

集合：相当于mysql的表格

数据库：所有表格的集合

db:database 数据库

1. 下载+引入mongodb

node init -y ----->npm i mongodb

const mongodb=require(‘mongodb’)

1. 客服端

const mongoClient=mongodb.MongoClient;

const dbAddress=’mongodb://localhost:27017’;

1. 连接数据库

mongoClient.connect(dbAddress,{useNewUrlParser:true},(err,res)=>{

If(err){ throw err }

1. 选择数据库

const db=res.db(dbname)

1. 选择集合

const collection=db.collection(colname);

1. 操作文档

7、res.close()

})

/collection.find() 是一个对象 toArray()转为一个数组

// collection.find().toArray((err, res) => {

// if (err) {

// throw err

// }

// })

//分页

//skip(0) 默认为0 从下标为0的开始

//limit(2) 2为截取长度

// collection.find().skip(0).limit(2).toArray((err, res) => {

// console.log(res)

// })

//模糊查询

const reg = /z/; //查询属性为str 属性值有z的

//const reg = new RegExp(name, 'i') name为一个变量 i区分大小写

collection.find({ str: { $regex: reg } }).toArray((err, res) => {

console.log(res)

})

**查询**

//find().toArray(callback)

//findOne(function(err,res){}) 查询第一个

**增加**

//insert(废弃)

//1、insertOne({},function(err,res){}) 增加一个

//2、insertMany([{},{}],function(err,res){}) 增加多个

**删除**

//1.deleteOne({},callback)

//2.deleteMany({},callback)

**改**

//1.updateOne({},callback)

//2.updateMany({},callback)

**ejs**

e：代表effective 高效的

**是一套简单的模板语言，利用普通的javascript代码生成html页面**

**下载 npm i express ejs**

//**关于路径**

//如果提供了绝对路径 就去该绝对路径下查找文件

//如果提供相对路径或不提供路径 默认去views下寻找路径 process.cwd()+views 当前工作目录（终端前目录）

//**统一设置路径**

app.set('views', \_\_dirname + '/views');

//**统一设置模板引擎的类型（ejs jade） 加后缀**

app.set('view engine', 'ejs') //第二个参数为后缀名

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

//第二个参数对象的作用：将值传递到所渲染的模板引擎

res.render('user', {

a: data

})

})

//**locals属性**

//locals是res和app下的属性

//locals作用是：将值传递给所渲染的模板引擎 app.locals 通过键值对（共享）

//app.locals 通过键值对 相当于render中的第二个参数对象

//区别：app.locals定义的属性在应用的整个生命周期都有效，res.locals只在当前请求的生命周期有效

**向前端传值**

//1、res.render(path,{})

//2、res.locals

//3、app.locals

**//process.cwd()当前工作目录**

**//process.env 动态设置端口 获取命令行参数**

**//process.argv 只要用于封装命令行工具，获取命令行参数**

**// <%js%>执行js脚本**

**// <%=data%> 转义输出**

**// <%-'data'%> 非转义输出**

**// <%#data%> 注释**

应用生成器

npm i express-generator -g