# nodejs

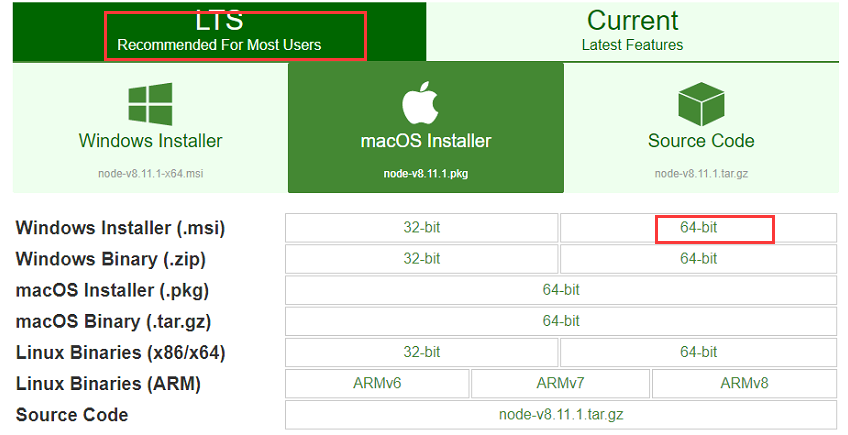
## 介绍

### 搭建开发环境

#### 安装nodejs

官网：nodejs.org

在官网下载长期稳定版本的nodejs



点击进行安装

安装后打开cmd窗口，执行“node -v”命令，查看nodejs版本，能查到则安装成功

#### webstorm编辑器

##### 安装

* 下载安装webstorm
* 破解：

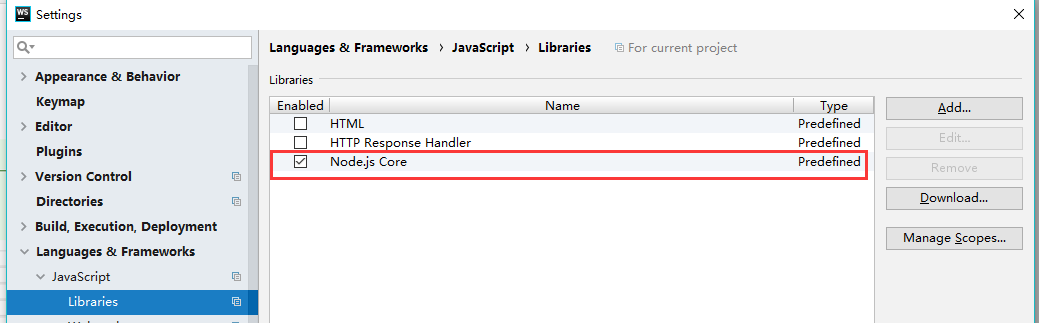
注册时，在打开的License Activation窗口中选择“License server”，在输入框输入下面的网址：

[http://im.js.cn:8888](http://im.js.cn:8888/)

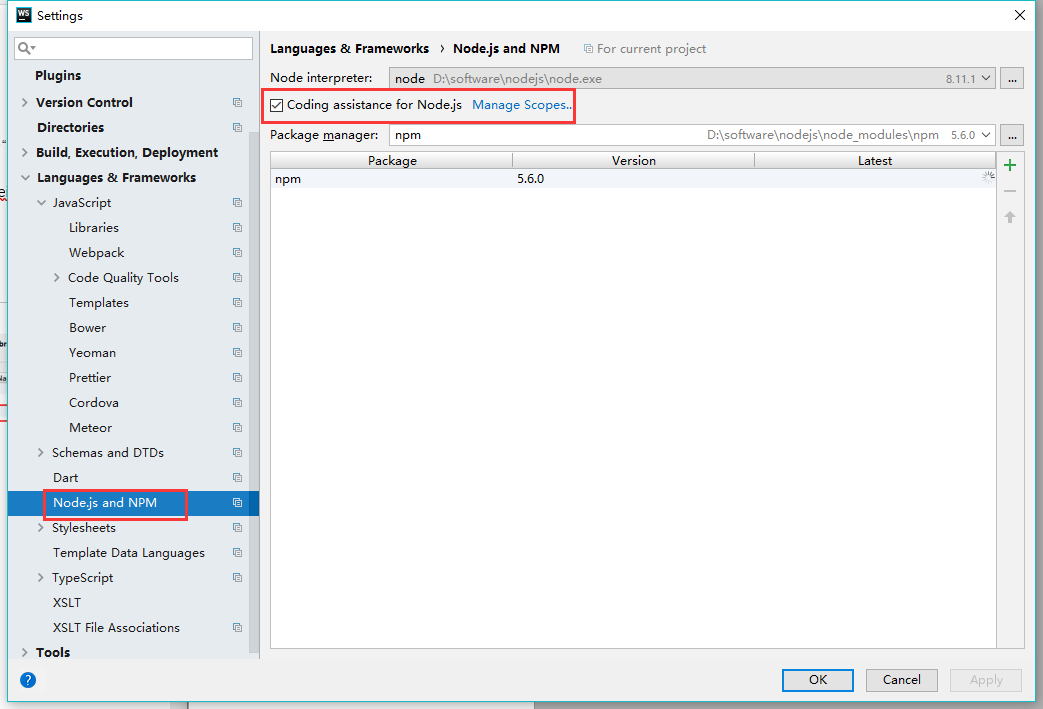
点击：Activate即可。

##### 智能提示

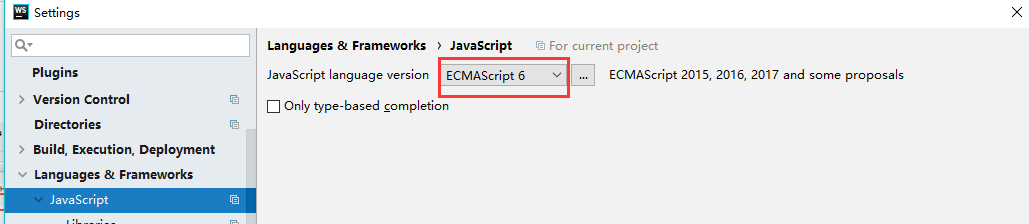
* + librarys



* + node.js and npm

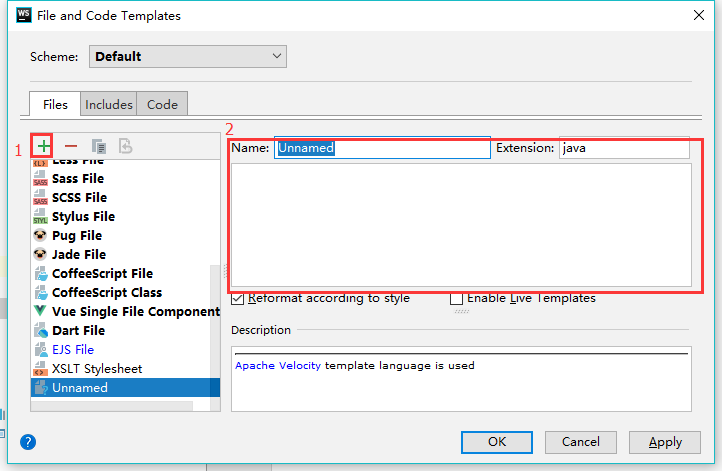


* + javascript language version



##### 新建模板

File 🡪 New 🡪 Edit File Templates...



###### EJS

与html4文件使用相同的模板

|  |
| --- |
| <!DOCTYPE **HTML** PUBLIC **"-//W3C//DTD HTML 4.01 Transitional//EN"  "http://www.w3.org/TR/html4/loose.dtd"**> <**html**> <**head**>  <**title**>$Title$</**title**> </**head**> <**body**> $END$ </**body**> </**html**> |

## js语法

### 数据类型

#### 弱类型

js是弱类型语言，所有类型变量均使用var进行定义

#### 基本类型

##### String

* 双引号与单引号

js中双引号与单引号没有区别

* 保存在栈中

##### Number

* js只有1种数字类型，可以是整形或浮点数

##### Boolean

##### Undefined

* 表示没有这个变量

##### Null

* 表示值为空

##### Array

* 创建数组的3种方式：
  + 方式1

|  |
| --- |
| var cars=new Array(); cars[0]="Saab"; cars[1]="Volvo"; cars[2]="BMW"; |

* + 方式2

|  |
| --- |
| var cars=new Array("Saab","Volvo","BMW"); |

* + 方式3

|  |
| --- |
| var cars=["Saab","Volvo","BMW"]; |

* 数组也是对象类型

##### Object

* 格式

对象由花括号分隔。在括号内部，对象的属性以名称和值对的形式 (name : value) 来定义。属性由逗号分隔

与json字符串的格式相同

|  |
| --- |
| var person={firstname:"John", lastname:"Doe", id:5566}; |

* 特征
* 对象内的每个key都是1个属性
* 对象可以在定义变量后，通过寻址添加或修改对象中的属性
* 除基本类型的String、Number、Boolean、Undefined、Null外，其他的均为对象类型，Array和function也是对象类型的
* 对象中属性具有唯一性（这里的属性包括方法），如果有两个重复的属性，则以最后赋值为准。

|  |
| --- |
| **var** c = {name:**"1"**,name:**"2"**}; console.log(c); *//2* |

* 寻址方式

2种寻址方式：

|  |
| --- |
| name=person.lastname; name=person["lastname"]; |

#### 特殊类型Buffer

#### 获取变量类型

##### typedef操作符

|  |
| --- |
| typeof "John"                 // 返回 string  typeof 3.14                   // 返回 number typeof NaN                    // 返回 number typeof false                  // 返回 boolean typeof [1,2,3,4]              // 返回 object typeof {name:'John', age:34}  // 返回 object typeof new Date()             // 返回 object typeof function () {}         // 返回 function typeof myCar                  // 返回 undefined (如果 myCar 没有声明) typeof null                   // 返回 object |

如果对象是 JavaScript Array 或 JavaScript Date ，我们就无法通过 **typeof** 来判断他们的类型，因为都是 返回 Object。

##### constructor属性

**constructor** 属性返回所有 JavaScript 变量的构造函数。

|  |
| --- |
| "John".constructor                 // 返回函数 String()  { [native code] } (3.14).constructor                 // 返回函数 Number()  { [native code] } false.constructor                  // 返回函数 Boolean() { [native code] } [1,2,3,4].constructor              // 返回函数 Array()   { [native code] } {name:'John', age:34}.constructor  // 返回函数 Object()  { [native code] } new Date().constructor             // 返回函数 Date()    { [native code] } function () {}.constructor         // 返回函数 Function(){ [native code] } |

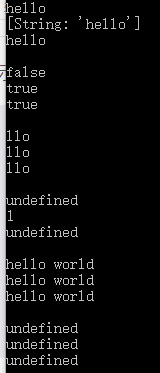
#### 深度理解

##### String与new String区别

* 程序

|  |
| --- |
| **var** s0 = **"hello"**; **var** s1 = **new** String(s0); **var** s2 = String(s0);  *//(1)* console.log(s0); console.log(s1); console.log(s2); console.log();  *//(2)* console.log(s0 === s1); console.log(s0 === s1.toString()); console.log(s1 == s1); console.log();  **var** s10 = s0.substring(2); **var** s11 = s0.substring(2); **var** s12 = s0.substring(2);  *//(3)* console.log(s10); console.log(s11); console.log(s12); console.log();  s0.val = 1; s1.val = 1; s2.val = 1;  *//(4)* console.log(s0.val); console.log(s1.val); console.log(s2.val); console.log();  s0 = s0+**" world"**; s1 = s1+**" world"**; s2 = s2+**" world"**;  *//(5)* console.log(s0); console.log(s1); console.log(s2); console.log();  s0.val = 1; s1.val = 1; s2.val = 1;  *//(6)* console.log(s0.val); console.log(s1.val); console.log(s2.val); |

* 运行结果



* 分析
  + 第1次打印

s0与s2打印结果均为“hello”，他俩的类型均为基本类型中的String类型，而s1则是对象类型；但3者的值均为“hello”

* + 第2次打印

===比较的是值和类型

==比较的仅仅是值

s0与s1值相同类型不同，所以第一行结果为false

s1.toString()是将s1的值以值类型的形式返回，此时该返回值与s0的值与类型均相同，所以第2行打印true

s0与s1的值都是“hello”，只有类型不同，所有用==比较值时，结果为true

* + 第3次打印

无论String是基本类型还是对象类型，当调用方法（如subString）时，都会被包装为对象类型，进行方法调用

* + 第4次打印

对象类型时可以设置属性的，但是基本类型不可以，所以只有s1能打印设置的属性值

* + 第5次打印

无论基本类型还是对象类型，拼接字符串时，都会转成基本类型的字符串

* + 第6次打印

s0、s1、s2此时都进行了字符串拼接，已经转为基本类型的String，所以都不能设置属性了。

##### null和undefined区别

###### 历史原因

这两种类型是因为历史原因保留下来的，其实这两种类型并没有太多区别

###### 区别

* null

用于对象；表示没有对象，即该处不应该有值

如：

函数的实参应该是1个对象，但是不想传入这个对象了，则传入null

* undefined

用于变量，属性和方法；表示缺省值，即该处应该有1个值，但是还没有定义

如：

定义1个变量但是没有初始化

###### 测试对象是否存在

* 错误写法

|  |
| --- |
| if (myObj !== null && typeof myObj !== "undefined") |

* 正确写法

|  |
| --- |
| if (typeof myObj !== "undefined" && myObj !== null) |

##### 浮点数

JavaScript 中的所有数据都是以 64 位浮点型数据(float) 来存储。

所有的编程语言，包括 JavaScript，对浮点型数据的精确度都很难确定：

|  |
| --- |
| **var** x = 0.1; **var** y = 0.2; **var** z = x + y console.log(z); *//0.30000000000000004* **if** (z == 0.3) *// 返回 false* |

##### 数组、对象的索引

###### 数组索引

许多程序语言都允许使用名字来作为数组的索引。

使用名字来作为索引的数组称为关联数组(或哈希)。

JavaScript 不支持使用名字来索引数组，只允许使用数字索引。

|  |
| --- |
| var person = []; person[0] = "John"; person[1] = "Doe"; person[2] = 46; var x = person.length;         // person.length 返回 3 var y = person[0];             // person[0] 返回 "John" |

###### 对象索引

在 JavaScript 中,对象使用 名字作为索引。

如果你使用名字作为索引，当访问数组时，JavaScript 会把数组重新定义为标准对象。

执行这样操作后，数组的方法及属性将不能再使用，否则会产生错误:

|  |
| --- |
| var person = []; person["firstName"] = "John"; person["lastName"] = "Doe"; person["age"] = 46; var x = person.length;         // person.length 返回 0 var y = person[0];             // person[0] 返回 undefined |

### 字面量

* 字面量是给变量赋值时的右值

|  |
| --- |
| **var** var0 = **"123"**; |

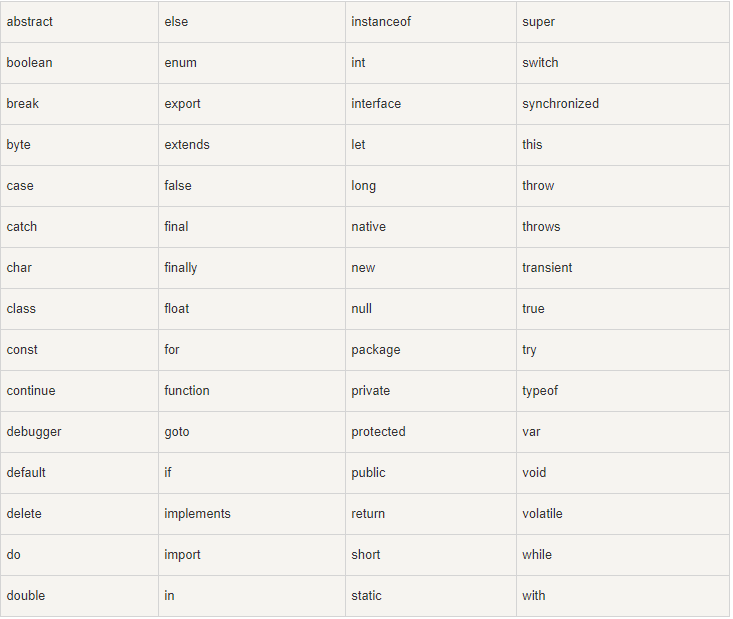
其中，“123”就是字面量

* 注意

字面量不是常量，常量使用const修饰，具有真正的变量名，而字面量只是1个值，不属于任何1个变量

### 关键字

#### 现用关键字



#### 保留关键字

为了向将来Javascript的新版本过渡，严格模式新增了一些保留关键字：

* implements
* interface
* let
* package
* private
* protected
* public
* static
* yield

#### 常用关键字

##### var

* 用于定义变量
* 作用域

var定义的变量的作用域为封闭函数

函数外定义的变量可以在函数内使用，函数内定义的变量不能再函数外使用

块（如if）内定义的变量可以在块外使用，这点与let不同

* 例

|  |
| --- |
| **function** f(){  **if**(**true**){  **var** x = 2;  console.log(x);*//2* }  console.log(x);*//2* } console.log(x);*//not defined* |

##### let

* 用于定义变量
* 作用域

let定义的变量的作用域为同级块（如if）和下级块，上级块内是无法使用的；这点与let不同

* 例

|  |
| --- |
| **function** f1(){  **if**(**true**){  **let** x = 2;  console.log(x);*//2* }  console.log(x);*//not defined* } |

##### this

###### this指向

函数中的this指向取决于函数被调用时，是以何种方式进行调用的

this的指向较为复杂，分为如下几种情况：

* 函数外

函数外，this指向exports和module.exports

|  |
| --- |
| exports.num = 1; console.log(exports); console.log(**this**); |

结果：



* 函数内
  + 构造函数内

构造函数中，this指向这个实例

只要调用时，使用new给方法创建了1个实例，则该实例方法中的this就指向这个实例

|  |
| --- |
| **function** f() {  **this**.num = 1;  console.log(**this**.num); } **var** fun = **new** f(); console.log(fun.num); |

结果：



* + 函数作为对象方法调用
    - 什么叫作为对象方法调用

|  |
| --- |
| *f*(); |

这种直接使用函数名来调用这个函数，叫做作为函数调用

|  |
| --- |
| ***a***.*f*(); |

函数属于某个对象的方法，调用时需要通过对象获取这个方法后再执行，叫做作为对象的方法调用

两者区别在于：上面的函数是直接就能执行的，下面的需要先从对象中取出在执行。

* + - this指向

作为对象方法对用的函数，函数内的this对象始终指向这个对象

|  |
| --- |
| **var** v = {fun:**function** () {  console.log(**this**);  }}; v.fun(); |

结果：



* + 函数作为函数调用
* 什么叫作为函数调用

|  |
| --- |
| *f*(); |

这种直接使用函数名来调用这个函数，叫做作为函数调用

|  |
| --- |
| ***a***.*f*(); |

函数属于某个对象的方法，调用时需要通过对象获取这个方法后再执行，叫做作为对象的方法调用

两者区别在于：上面的函数是直接就能执行的，下面的需要先从对象中取出在执行。

* this指向

作为函数调用时，函数内的this对象始终指向global，严格模式下this的值为undefined

|  |
| --- |
| **function** *f*() {  **this**.**x** = **'a'**;  ***console***.log(**this**.**x**); }; **function** *f1*() {  *f*();  ***console***.log(**this**.**x**); } *f1*(); ***console***.log(global.x); |

###### 练习

* 题目

|  |
| --- |
| **name0** = **"Global object"**; **var *object0*** = {  **name0** : **"My Object0"**,  getNameFunc : **function**(){  **return function**(){  **return this**.**name0**;  };  } }; **var *object1*** = {  **name** : **"My Object1"**,   getNameFunc : **function**(){  **var** that = **this**;  **return function**(){  **return** that.**name**;  };  } }; ***console***.log(***object0***.getNameFunc()());*//Global object* ***console***.log(***object1***.getNameFunc()());*//"My Object1* |

* 分析

我们可以将最后调用函数部分代码拆分为两部分：

|  |
| --- |
| **var *fun*** = ***object0***.getNameFunc(); ***fun***(); |

拆分后可知：

外层函数getNameFunc是作为对象的方法进行调用的

内层函数是作为函数调用的

所以外层函数中的this指向object对象，而内层函数中的this指向global

如果想在内层函数中获取object对象，可以在外层函数中将this对象赋给1个变量，然后在内层函数中使用该变量。

### 变量

#### 重定义

##### 变量、函数重定义共同特征

js中允许变量、函数的重定义，但是程序执行时，只会找使用处之前最后一次定义的函数或变量进行使用。

|  |
| --- |
| **function** f() {  **return "a"**; } **function** f() {  **return "b"**; } console.log(f()); |

##### 变量重定义独有特征

变量重定义后如果没有进行初始化，则会保留重定义之前的值。

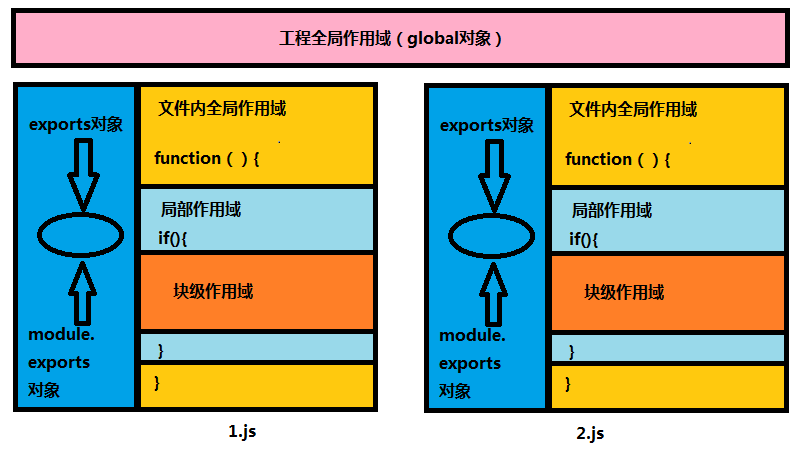
* 代码

|  |
| --- |
| **var** i = 1; **var** i; console.log(i); |

* 结果

1

#### 作用域



##### 块作用域

* 块作用域就是“if(条件){语句}”中的大括号就是1个块作用域
* 块作用域只对使用let定义的变量的作用域有限制作用，对使用var定义的变量的作用域没有限制作用

|  |
| --- |
| **for**(**var** i=0; i<10; i++){  **var** j = 0; } console.log(i); *//10* console.log(j); *//0* |

##### 局部作用域

参见[var](#_var)和[let](#_let)

##### 全局作用域

###### 文件内全局作用域

在函数外使用var、let、const定义的变量的作用域是文件内全局作用域

该变量只能在本文件内当作全局变量使用

###### 工程全局作用域(global对象)

* 该作用域就是global对象，当前模块及其导入的子模块共享同一个global对象
* global对象中的属性可以直接当作全局变量使用
* 也就是在a模块中将1个变量添加到了global对象中，b模块中导入了a模块，则在b模块中可以将该变量直接当做全局变量使用

1.js：

|  |
| --- |
| x=**'a'**; |

2.js：

|  |
| --- |
| **var** module\_a = require(**"./1.js"**); console.log(x); |

结果：



* 哪些方法定义的变量会放在global对象中？
  + 未使用var、let、const关键字定义的变量会默认放在global对象中

|  |
| --- |
| x = **'a'**; console.log(global.x);*//a* |

* + 函数内使用this.key=value时，改变量定义在global对象中

|  |
| --- |
| **function** f() {  **this**.x = **'a'**; } |

函数内this指向global对象，所以相当于“global.x = ‘a’”

* + global.key=value

#### 生命周期

##### 局部变量

var和let变量都是从定义这个变量开始

var变量在退出函数后删除

let变量在退出所在块级后删除

##### 全局变量

定义变量开始

程序终止时变量销毁

#### 向未声明的变量赋值

无论在何处，向未声明的变量赋值时，会立刻声明1个全局的变量

|  |
| --- |
| carname="Volvo"; |

但是这个变量不能再该语句之前使用，这样用该变量不会进行变量提升，参见下面的变量提升

该变量被定义在global对象中

#### 变量提升

* 对于使用var定义的全局变量，会自动将定义提升到文件最上面进行定义，但是自动提升时定义的变量初始值均为undefined，需要执行到真正对该变量初始化或进行赋值的地方才会改变。也就是在定义该变量之前就可以使用该变量了，只不过初始值为undefined

例：

下面第一段代码等效于第二段代码：

|  |
| --- |
| console.log(m); *//undefined* **var** m = **'m'**; console.log(m); *//'m'* |

|  |
| --- |
| **var** m; console.log(m); *//undefined* m = **'m'**; console.log(m); *//'m'* |

* 使用var定义的局部变量也会进行变量提升，只不过会自动提升到函数的最开始进行定义。

例：

下面两段代码等效：

|  |
| --- |
| **function** f() {  **if**(i == **"a"**){  console.log(**"i == a"**);  }  **var** i = **'a'**; } |

|  |
| --- |
| **function** f() {  **var** i;  **if**(i == **"a"**){  console.log(**"i == a"**);  }  i = **'a'**; } |

* 在[向未声明的变量赋值](#_向未声明的变量赋值)中讲到，向未声明的变量进行赋值时，会自动生成1个全局变量，但是该全局变量只有执行到该语句时才会被定义，该变量不会被变量提升。

|  |
| --- |
| *//该处不可以使用变量i* i = **'a'**; *//该处可以使用变量i* |

* 使用let定义的变量不会进行变量提升

|  |
| --- |
| *//该处不可以使用变量i* **let** i = **'a'**; *//该处可以使用变量i* |

### 函数

#### 函数定义的方法

##### 函数声明

* 格式

|  |
| --- |
| **function** 函数名(参数列表) {  *//function body* } |

##### 函数表达式（匿名函数）

* 格式

|  |
| --- |
| **var** var0 = **function** (参数列表) {  *//函数体* } |

* 说明

常用于函数定义后直接赋值给其他变量时

##### 函数构造器

* 格式

|  |
| --- |
| var myFunction = new Function("a", "b", "return a \* b"); var x = myFunction(4, 3); |

* 说明

不常用，一般使用匿名函数即可

##### 箭头函数

* 格式

|  |
| --- |
| **var** fun = (参数列表) =>{  *//函数体* } |

当只有1个参数时，可以简写成：

|  |
| --- |
| **var** fun = 参数 =>{  *//函数体* } |

#### 函数提升

##### 什么叫函数提升

只有使用[函数声明](#_函数声明)方法定义的函数，才可以进行函数提升。

使用[函数声明](#_函数声明)方法定义的函数，等效于将匿名函数赋值给使用var定义的该函数名的变量

|  |
| --- |
| **function** *f*() {  **x** = 1; } *//等效于* **var** *f* = **function** () {  **x** = 1; } |

此时，f也是1个var变量，所以f变量也会被提升，但是f指向的是1个函数，这样的提升叫做函数提升。

提升前：

|  |
| --- |
| ***console***.log(**'hello'**); **function** *f*() {  **x** = 1; } |

提升后：

|  |
| --- |
| **var *f***;*//提升到全局或局部作用域头* ***console***.log(**'hello'**); ***f*** = **function** () {  **x** = 1; } |

##### 函数提升作用域与初始值

函数提升与[变量提升](#_变量提升)差不多，只是这里关于初始值和作用域的问题，与变量提升稍有不同。

* 变量提升中的作用域与初始值

对于变量提升，无论定义与提升是否在同一个作用域，在未执行到定义那条语句之前，该变量一定是undefined。

|  |
| --- |
| ***console***.log(***i***);*//undefined* **var *i*** = **'a'**; ***console***.log(***i***);*//a* |

但是对于函数提升，定义和提升所在的作用域将决定提升后函数变量的初始值

* 函数提升中的作用域与初始值

因为使用函数声明方式定义的函数可以理解为使用var变量指向了1个匿名函数，所以函数提升只会提升到全局作用域或局部作用域的头部。

此时则出现2种情况：

* + - * 定义与提升在同一个作用域

如：

|  |
| --- |
| *//提升位置* **function** *f*() {}*//定义位置* |

* + - * 定义与提升不在同一个作用域

函数定义包包裹在块级作用域中

如：

|  |
| --- |
| *//提升位置* {  **function** *f*() {}*//定义位置* } |

* + 定义与提升在同一个作用域

此时，与变量提升不同，函数被提升到作用域头部时，就已经具有了定义的初始值

|  |
| --- |
| *//变量提升初始值为undefined* ***console***.log(***x***);*//undefined* **var *x*** = **'a'**; ***console***.log(***x***);*//a  //定义与提升在同一个作用域时， // 函数提升初始值为定义时指定的函数 f*();*//hello* **function** *f*() {  ***console***.log(**'hello'**); } |

* + 定义与提升不在同一个作用域时

此时与变量提升相同，必须执行到定义的那条语句，被提升的函数变量才具有初始值。

|  |
| --- |
| *f*();*//f is not a function* {  **function** *f*() {  ***console***.log(**"hello"**);  } } *f*();*//hello* |

#### 调用函数

对于使用[函数表达式（匿名函数）](#_函数表达式（匿名函数）)方法定义的函数，可以通过如下方法直接对该函数进行调用，而不用手动调用。

|  |
| --- |
| (**function** (*//形参) {* console.log(**"1"**); })(*//实参);* |

* 红色括号

用于说明这是1个函数表达式

* 蓝色括号

用于进行自调用，加上该括号函数被定义后直接进行调用

例：

|  |
| --- |
| (**function** (a) {  console.log(a); })(**"10"**); |

#### arguments对象

* 用在函数中，是该函数的内置对象，包含该函数的参数的值与信息
* 例

|  |
| --- |
| x = sumAll(1, 123, 500, 115, 44, 88);  **function** sumAll() {  **var** i, sum = 0;  **for** (i = 0; i < arguments.length; i++) {  sum += arguments[i];  }  **return** sum; } |

#### 参数传递

##### 值类型参数传递

当参数为值类型时，参数传递的是值，而不会将这个变量传递进去，所以在函数内修改形参时，不会改变实参

|  |
| --- |
| **function** f(i) {  i=10; } **var** x = 0; f(x); console.log(x); |

##### 对象类型参数传递

当参数为对象类型时，参数传递的是对象（理解为指针），而不是将这个对象的值拷贝后传递进去，所以在函数内修改形参时，实参也会发生改变

|  |
| --- |
| **function** f(i) {  i[0]=10; } **var** x = [0,1]; f(x); console.log(x); |

### 运算符

#### 多元运算符

* 概述

多元运算符就是三目运算符的增强版

* 格式

|  |
| --- |
| 条件1？条件2？值1：值2：值3 |

* 分析

计算多元运算符时，将第1个？和最后1个：作为分界线，将表达式分割为3部分后，当作三目运算符进行计算；当需要进入到中间部分仍然含有多元运算符的部分时，则继续按照上述规则进行计算，直到得到结果为止

* 例

|  |
| --- |
| **var** b = 2;  **var** j = b>1?b>3?1:2:3; |

分析：

将上述表达式分割为：

b>1? b>3?1:2 :3;

2大于1，所以继续分析中间部分表达式

b>3?1:2

这是1个三目运算符，结果为2

### 语句

#### for循环

##### for循环写法

###### for

普通for循环

###### for/in

增强for循环

|  |
| --- |
| **var** aArray = [**'a'**,123,{a:**'1'**,b:**'2'**}]; aArray.name = 1;  **for**(**let** index **in** aArray){  console.log(aArray[index]); } |

* 可以用来遍历数组和对象，for/in遍历时得到的是key，然后通过key将value取出即可。对于数组，每个元素的下标就是他的key。

###### for/of

ES6新语法

|  |
| --- |
| **var** aArray = [**'a'**,123,{a:**'1'**,b:**'2'**}]; aArray.name = 1;  **for**(**var** value **of** aArray){  console.log(value); } |

* 只能用来遍历数组，可以直接得到value，遍历对象时程序执行错误。

###### for/in与for/of区别

1. 推荐在循环对象属性的时候，使用for...in,在遍历数组的时候的时候使用for...of。
2. for...in循环出的是key，for...of循环出的是value
3. 注意，for...of是ES6新引入的特性。修复了ES5引入的for...in的不足
4. for...of不能循环普通的对象，需要通过和Object.keys()搭配使用

#### try/catch/throw

##### try/catch

|  |
| --- |
| **function** message() {  **try** {  *//函数调用出错* console.loog(**"Welcome guest!"**);  } **catch**(err) {  txt+=**"错误描述："** + err.message + **"\n\n"**;  console.log(txt);  } } |

##### throw

抛出自定义异常，用于被catch捕获

|  |
| --- |
| **var** txt=**""**; **function** message() {  **try** {  **if**(txt == **""**){  **throw "值为空"**;  }  } **catch**(err) {  txt+=**"错误描述："** + err + **"\n\n"**;  console.log(txt);  } } |

#### switch

switch 语句会使用恒等计算符(===)进行比较:

|  |
| --- |
| i= 10; **switch**(i){  **case '10'**:  console.log(**"=="**);  **break**;  **case** 10:  console.log(**"==="**);   **break**; } *//输出结果为“===”* |

### 正则表达式

### 严格模式

#### 介绍

* 在文件或函数的头部使用“use strict”表达式来声明使用严格模式
* “use strict”只能用在文件或函数的开头
* 用在文件开头的声明对本文件内所有语法有效；用在函数开头的声明只对该函数内的语法有效

#### 作用

* 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，包括IE 10在内的主流浏览器，都已经支持它，许多大项目已经开始全面拥抱它。
* 另一方面，同样的代码，在"严格模式"中，可能会有不一样的运行结果；一些在"正常模式"下可以运行的语句，在"严格模式"下将不能运行。

#### 限制

##### 不允许使用未声明变量

###### 全局声明严格模式

|  |
| --- |
| **"use strict"** y = 1; *//报错（y变量未定义）* **function** f(){  x = 3; *//报错（x变量未定义）* } f(); |

###### 局部声明全局模式

|  |
| --- |
| y = 1; *//不报错* **function** f(){  **"use strict"** x = 3; *//报错（x变量未定义）* } f(); |

##### 不允许删除变量、函数

|  |
| --- |
| **"use strict" var** y = 1; **function** f(){  **var** x = 3; } **delete** y; *//报错（不能删除变量）* **delete** f; *//报错（不能删除函数）* |

##### 不允许形参重名

注意：变量和函数还是可以重定义的，这个限制只针对于形参。

|  |
| --- |
| **"use strict"**; **function** x(p1, p1) {}; *//报错（形参重名）* |

##### 不允许使用8进制

|  |
| --- |
| "use strict"; var x = 010;             // 报错 |

##### 不允许使用转义字符

|  |
| --- |
| "use strict"; var x = \010;            // 报错 |

##### 不允许对只读属性赋值

|  |
| --- |
| "use strict"; var obj = {}; Object.defineProperty(obj, "x", {value:0, writable:false}); obj.x = 3.14;            // 报错 |

##### 不允许对1个使用getter方法读取的属性赋值

|  |
| --- |
| "use strict"; var obj = {get x() {return 0} }; obj.x = 3.14;            // 报错 |

##### 不允许删除1个不允许删除的属性

|  |
| --- |
| "use strict"; delete Object.prototype; // 报错 |

##### 不允许使用特定字符串作为变量、函数名

特定字符串包括：

eval、arguments、[保留关键字](#_保留关键字)

|  |
| --- |
| "use strict"; var eval = 3.14;         // 报错 |

##### 不允许使用以下语句

|  |
| --- |
| "use strict"; with (Math){x = cos(2)}; // 报错 |

##### 在作用域 eval() 创建的变量不能被调用

|  |
| --- |
| "use strict"; eval ("var x = 2"); alert (x);               // 报错 |

##### 禁止this关键字指向全局对象。

|  |
| --- |
| function f(){      return !this;  }  // 返回false，因为"this"指向全局对象，"!this"就是false  function f(){      "use strict";      return !this;  }  // 返回true，因为严格模式下，this的值为undefined，所以"!this"为true。 |

因此，使用构造函数时，如果忘了加new，this不再指向全局对象，而是报错。

|  |
| --- |
| function f(){      "use strict";      this.a = 1;  };  f();// 报错，this未定义 |

### 模块

#### 定义

node中1个js文件就是1个模块

#### 导入、暴露

在a模块中，将需要暴露的对象设置到a模块的module.exports对象中，在b模块中使用a模块中的对象时，使用require导入a模块并赋给1个变量（如module\_a），然后使用module\_a.xxx来使用a模块中的对象

a．js



b．js



#### 内置对象

##### exports、module.exports

* module对象的exports属性是用来暴露对象的，但是这个写法过长，所以引入了exports对象，其实这两个对象指向的是同一个对象，作用也相同，只不过exports是module.exports的简写；但是真正导出时使用的是module.exports对象
* 每个模块都有自己的exports对象
* 当使用require引入模块时，require的返回值就是被引入模块的exports对象
* 向exports对象中设置属性的方法
  + exports.key = value
  + module. exports.key = value
  + 在函数外使用this.key=value

##### global

参见[工程全局作用域(global对象)](#_工程全局作用域(global对象))

### 闭包

#### 内嵌函数

* js中，所有函数能访问其上一级的作用域，所以最外层函数能访问全局变量
* js中，允许在函数内定义1个函数并在外层函数内对其进行调用
* 内嵌函数的作用域限制
  + 内层函数就像外层函数的1个局部变量一样，无法在外层函数以外对其进行调用
  + 函数可以访问与它同级或比他大的作用域内的变量，但是不可以直接访问比他小的作用域的变量
* 例

|  |
| --- |
| **function** add() {  **var** counter = 0;  **function** plus() {counter += 1;}  plus();  **return** counter; } |

#### 闭包

##### 定义

将内部函数作为对象，赋值给外部函数以外的变量，此时即使外层函数执行完毕，但是由于有外部函数以外的变量指向该内部函数，则该内部函数不会在内存中被回收，内层函数依赖外层函数，所以外层函数及其局部变量也不会被回收。

这样的内部函数叫做闭包

##### 作用

* 可以读取外部函数的局部变量
* 让外部函数的局部变量（不包括外部函数中使用let定义的块级作用域的变量）始终保持在内存中（相当于C中在函数内使用static）

即使外层函数执行完毕，但是由于有外部函数以外的变量指向该内部函数，则该内部函数不会在内存中被回收，内层函数依赖外层函数，所以外层函数及其局部变量也不会被回收。

##### 例

|  |
| --- |
| **function** f1(){  **var** n=999;  nAdd=**function**(){n+=1}  **function** f2(){  alert(n);  }  **return** f2; } **var** result=f1(); result(); *// 999* nAdd(); result(); *// 1000* |

程序中，f2和nAdd均为闭包

##### 注意点

* 闭包只会让外部函数的局部变量保持在内存中，不包括外层函数中使用let定义的块级作用域的变量
  + let定义的局部变量

|  |
| --- |
| **var *a*** = []; (**function** () {  let i;  **for** (i = 0; i < 10; i++) {  ***a***[i] = **function** () {  ***console***.log(i);  };  } })(); ***a***[6]();*//10* |

* let定义的块级变量

|  |
| --- |
| **var *a*** = []; (**function** () {  **for** (let i = 0; i < 10; i++) {  ***a***[i] = **function** () {  ***console***.log(i);  };  } })(); ***a***[6]();*//6* |

* var定义的局部变量

|  |
| --- |
| **var *a*** = []; (**function** () {  **for** (**var** i = 0; i < 10; i++) {  ***a***[i] = **function** () {  ***console***.log(i);  };  } })(); ***a***[6]();*//10* |

* 由于闭包会使得函数中的变量都被保存在内存中，内存消耗很大，所以不能滥用闭包
* 闭包会在父函数外部，改变父函数内部变量的值。所以，如果你把父函数当作对象（object）使用，把闭包当作它的公用方法（Public Method），把内部变量当作它的私有属性（private value），这时一定要小心，不要随便改变父函数内部变量的值。

### 内置对象

#### JSON

##### 介绍

JSON对象时js中的内置对象，用来在字符串与JSON对象之间互相转换

注意：JSON字符串中引号必须使用双引号

##### 方法

###### parse

格式：parse(str)

功能：将字符串转换为JSON对象

参数： str：待转换字符串

返回值：JSON对象

例：

|  |
| --- |
| ***console***.log(***JSON***.parse(**'{"key":"value"}'**)); |

结果：



###### stringify

格式：

## ES6语法

### 转码

### let和const命令

#### let

##### 介绍

参见[let](#_let)

##### 函数调用外部变量

当函数调用外部变量,分为两种情况:

* 调用时外部变量仍然存在内存中

|  |
| --- |
| **var *a*** = []; **for** (**var *i*** = 0; ***i*** < 10; ***i***++) {  ***a***[***i***] = **function** () {  ***console***.log(***i***);  }; } ***a***[6](); *// 10* |

上述代码中，i是全局变量,退出for循环后，i仍然存在内存中，所以当调用a数组中的函数使用变量i时使用的是全局变量

* 调用时外部变量已经销毁

|  |
| --- |
| **var *a*** = []; **for** (let ***i*** = 0; ***i*** < 10; ***i***++) {  ***a***[***i***] = **function** () {  ***console***.log(***i***);  }; } ***a***[6](); *// 6* |

上述代码中，i是for作用域内的变量,退出for循环后，变量i已经销毁，所以当执行for循环时，会将变量i直接替换为i的值；例如：

当i=6时，赋给a[6]的函数

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

实则变成

***console***.log(***6***);

所以当调用a[6]();时，执行的是“***console***.log(***6***);”，而非“***console***.log(***i***);”

* 在闭包中这么用时，因为局部变量得到了保持，所以相当于这里的全局变量，但是let定义的for作用域中的变量不会被保持，所以相当于上面第2中情况，代码参见[注意点](#_注意点)

##### for循环作用域

for循环有3个作用域，由大到小介绍：

* for循环以外的作用域

不用多说，就是for语句整体所在的作用域

* 设置循环部分作用域

该作用域就是for循环中“（初始值；条件；迭代）”部分，该部分的作用域大于for循环体内的作用域，所以for循环体内可以使用设置循环部分作用域中的变量（也就是迭代变量。

当循环体内没有与设置部分作用域内同名的变量时，循环体内可以正常使用设置部分作用域内的变量

但是在for循环体内定义了与设置循环部分同名的变量时，不会产生覆盖，这是两个单独的变量。此时，循环体内只能使用循环体内定义的该同名变量，不能使用设置部分作用域内的该同名变量。

* 例

|  |
| --- |
| **for**(let ***i***=0; ***i***<10; ***i***++){  ***i*** = 8; / console.log(i); }*//无限循环打印8* **for**(let ***i***=0; ***i***<10; ***i***++){  let ***i*** = 8;  ***console***.log(***i***); }*//打印10遍8* |

##### 暂时性死区

在代码块内，使用let、const命令声明变量之前，该变量都是不可用的（即使该变量在上一层作用域中已经存在）。这在语法上，称为“暂时性死区”（TDZ）

|  |
| --- |
| **if** (**true**) {  *// TDZ开始* **tmp** = **'abc'**; *// ReferenceError* ***console***.log(**tmp**); *// ReferenceError* let ***tmp***; *// TDZ结束* ***console***.log(***tmp***); *// undefined* ***tmp*** = 123;  ***console***.log(***tmp***); *// 123* } |

##### 不允许重复声明

* 在var的使用上，允许同一作用域中的变量重定义，后定义的变量会替换掉先定义的变量

|  |
| --- |
| **var *i*** = **'a'**; ***console***.log(***i***);*//a* **var *i*** = **'b'**; ***console***.log(***i***);*//b* |

* 在let使用中，不允许在同一作用域中的变量重定义。

|  |
| --- |
| **function** *f*() {  let a = 1;  let a = 2; *//报错* } *f*(); **function** *f1*() {  let a = 1;  **var** a = 2; *//报错* } *f1*(); **function** *f2*() {  **var** a = 1;  let a = 2; *//报错* } *f2*(); |

* 但是let允许在不同的作用域中声明同名变量

|  |
| --- |
| **function** *f*() {  let i = 1;  let i = 2;*//报错* } *f*(); **function** *f1*() {  let i = 1;  {  let i = 2;*//不报错* } } *f1*(); |

#### const

##### 介绍

const用来声明1个常量，试图修改常量则会报错

##### 作用域

const作用域与let相同

##### 提升

const与let相同，不会进行提升

##### 暂时性死区

const与let相同，必须在定义后使用，该作用于内定义之前同样存在暂时性死区

##### 不可重复定义

const与let相同，同样不可重复定义

##### const对象

###### const对象

当把对象声明为常量时，只是该指针不可以修改，但是指针指向的内存中的内容还是可以修改的

|  |
| --- |
| **const *foo*** = {}; *// 为 foo 添加一个属性，可以成功* ***foo***.**prop** = 123; ***foo***.prop *// 123 // 将 foo 指向另一个对象，就会报错* ***foo*** = {}; *// TypeError: "foo" is read-only* |

###### 冻结对象

如果真的想将对象冻结，应该使用Object.freeze方法将const常量指向的对象冻结

|  |
| --- |
| **const *foo*** = ***Object***.freeze({}); *// 常规模式时，下面一行不起作用； // 严格模式时，该行会报错* ***foo***.**prop** = 123; |

除了将对象本身冻结，对象的属性也应该冻结。下面是一个将对象彻底冻结的函数。

|  |
| --- |
| *//冻结方法* **var *constantize*** = (obj) =>{  ***Object***.freeze(obj);  ***Object***.keys(obj).forEach( (key, ***i***) => {  **if** ( **typeof** obj[key] === **'object'** ) {  ***constantize***( obj[key] );  } }); }; *//使用constantize方法* **const *obj0*** = {**name**:**"123"**}; ***constantize***(***obj0***); ***obj0***.**key** = 1; *//对象冻结，无法添加* ***obj0***.**name** = **'a'**;*//属性冻结，无法修改* ***console***.log(***obj0***);*//{ name: '123' }* |

### 变量的解构赋值

#### 数组的解构赋值

##### 基本语法

* 变量的解构赋值就是“模式匹配”，只要等号两边的模式相同，左边的变量就会被赋予对应的值。
* 解构后如果左边变量的值存在undefined，则解构不成功
* 等号右边的值转为对象后或本身必须具备[Iterator](#_Iterator)接口
* Set结构和Generator函数本身具备你iterator接口，所以也可以用数组解构
* 例

|  |
| --- |
| let [***foo***, [[***bar***], ***baz***]] = [1, [[2], 3]];*//解构成功* ***foo*** *// 1* ***bar*** *// 2* ***baz*** *// 3* let [ , , ***third***] = [**"foo"**, **"bar"**, **"baz"**];*//不完全解析* ***third*** *// "baz"* let [***x***, , ***y***] = [1, 2, 3];*//不完全解析* ***x*** *// 1* ***y*** *// 3* let [***head***, ...***tail***] = [1, 2, 3, 4];*//解构成功* ***head*** *// 1* ***tail*** *// [2, 3, 4]* let [***x***, ***y***, ...z] = [**'a'**];*//解构不成功* ***x*** *// "a"* ***y*** *// undefined* ***z*** *// []* |

##### 默认值

* 解构赋值允许在等号左边直接指定变量的值来为变量赋予默认值
* 进行解构时，只有等号右边的值为undefined或没有指定值时，对应的变量才会使用默认值
* 如果默认值为表达式，则只有当该变量需要赋予默认值时，才会调用该函数
* 例

|  |
| --- |
| let [***x*** = **'a'**,***y*** = **'b'**,***z*** = **'c'**] = [**'d'**,**undefined**]; ***x****//d* ***y****//b* ***z****//c* **function** *f*() {  ***console***.log(**'f()'**); } let [***u*** = *f*(), ***v*** = *f*()] = [1]; ***u****//u指向的函数不执行* ***v****//v指向的函数执行* |

#### 对象的解构赋值

##### 基本语法

* 对象的解构与数组概念差不多，但是数组解构是通过下标为变量赋予对应的值，而对象解构是通过属性名为变量赋予对应的值，所以对象不用关心属性的顺序，而数组顺序必须一一对应

|  |
| --- |
| let { **bar**, **foo** } = { **foo**: **"aaa"**, **bar**: **"bbb"** }; **foo** *// "aaa"* **bar** *// "bbb"* let { ***baz*** } = { **foo**: **"aaa"**, **bar**: **"bbb"** }; ***baz*** *// undefined* |

* 上面这种形式只是当变量名与属性名相同时的简写方式，其完整写法等号左边的变量也应该是key:value的形式

简写：

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

完整：

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

进行解构时，会重复使用等号右边的对象，依次为等号左边使用逗号分隔的每个变量进行解构赋值。

单次解构时，会使用等号左边键值对中的key在等号右边的对象中寻找对应的属性值，然后将属性值赋值给等号左边键值对中的value变量

* 对变量名与属性名不同的变量进行解构赋值时必须使用完整写法

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

* 解构嵌套解构对象

|  |
| --- |
| **const *node*** = {  **loc**: {  **start**: {  **line**: 1,  **column**: 5  }  } };  let { **loc**, loc: { **start** }, loc: { start: { **line** }} } = ***node***; **line** *// 1* **loc** *// Object {start: Object}* **start** *// Object {line: 1, column: 5}* |

等号右边有2个逗号，所以需要解构3次，剩下自己分析

##### 默认值

与数组的解构赋值中默认值的原理一摸一样

### 扩展

#### 字符串扩展

##### 编码

js中字符串使用Unicode编码,在\u后面跟上超过0xFFFF的数值,对于超出这个范围的字符则会解读为2个字符.

对于中文等在此范围内无法表示的,需要使用2个Unicode编码表示1个中文字符

|  |
| --- |
| "\u0061"  // "a"  "\u20BB7"  // " 7"  "\uD842\uDFB7"  // "𠮷" |

如果1个超出范围的Unicode编码想解析为1个字符,则需要使用{}

|  |
| --- |
| "\u{20BB7}"  // "𠮷" |

##### API

###### at()

字符串的:

charAt()方法，每次读取字符为2个字节

at()方法，能根据字符返回正确的字符

|  |
| --- |
| **'abc'**.charAt(0) *// "a"* **'𠮷'**.charAt(0) *// "\uD842"* **'abc'**.at(0) *// "a"* **'𠮷'**.at(0) *// "𠮷"* |

###### normalize()

有些字符可能有多种编码表示方式，他们在视觉和语义上是等价的，但是js不能识别

引入normalize函数将字符的不同表示法同一为同样的样式，成为Uinicode正规化

|  |
| --- |
| **'\u01D1'**===**'\u004F\u030C'** *//false* **'\u01D1'**.normalize() === **'\u004F\u030C'**.normalize()*//true* |

###### includes()

格式： var includes(str)

var includes(str,index)

功能：返回布尔值，表示是否找到了参数字符串

参数： str：要查找的字符串

index：从第index个字符开始查找（0代表第个字符，开始查找位置包含第index个字符）

返回值：是否查找到

例：

|  |
| --- |
| **var *s*** = **"hello"**; ***console***.log(***s***.includes(**"e"**));*//true* ***console***.log(***s***.includes(**"e"**, 2));*//false* |

###### startsWith()

格式： var startsWith(str)

var startsWith(str,index)

功能：返回布尔值，表示参数字符串是否在原字符串的头部

参数： str：要查找的字符串

index：从第index个字符开始查找（0代表第个字符，开始查找位置包含第index个字符）

返回值：是否在原字符串的头部

例：

|  |
| --- |
| **var *s*** = **"hello"**; ***console***.log(***s***.startsWith(**"he"**));*//true* ***console***.log(***s***.startsWith(**"e"**, 1));*//true* |

###### endsWith()

格式： var endsWith(str)

var endsWith(str,index)

功能：返回布尔值，表示参数字符串是否在原字符串的尾部

参数： str：要查找的字符串

index：在原字符串中的前index个字符之中进行查找

返回值：是否已指定字符串结尾

例：

|  |
| --- |
| **var *s*** = **"hello"**; ***console***.log(***s***.endsWith(**"lo"**));*//true* ***console***.log(***s***.endsWith(**"e"**, 2));*//true* |

###### repeat()

格式：var repeat(n)

功能：将原字符串重复n次返回

参数： n：重复n次，如果是小数则取整数部分

返回值：重复之后的字符串

例：

|  |
| --- |
| **var *s*** = **"ha"**; ***console***.log(***s***.repeat(2));*//haha* ***console***.log(***s***.repeat(3.2));*//hahaha* |

###### padStart()

格式： var padStart(len)

var padStart(len,str)

功能：字符串头部自动补全

参数： len：补全到多少个字符

str：使用哪个字符串不全，没有该参数默认使用空格

返回值：补全后的字符串

例：

|  |
| --- |
| *//重复使用给定字符串补全到目标长度* **'x'**.padStart(5, **'ab'**) *// 'ababx'* **'x'**.padStart(4, **'ab'**) *// 'abax'* **'abc'**.padStart(10, **'0123456789'**)*// '0123456abc' //已有长度大于等于目标长度，返回源字符串* **'xxx'**.padStart(2, **'ab'**) *// 'xxx'* **'x'**.padStart(4) *// ' x'* |

###### padEnd()

用法同padStart()

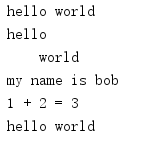
##### 模板字符串

###### 语法

* 模板字符串使用反引号(` `)标识，ESC下面的按键，而非单引号
* 模板字符串可以当作普通字符串使用
* 可以用来定义多行字符串，字符串中的换行、空格等均会保留到字符串中
* 模板字符串可以使用“${}”将变量、表达式、函数嵌入到字符串中
* 例

|  |
| --- |
| ***console***.log(**`hello world`**);  ***console***.log(**`hello  world`**);  let ***name*** = **'bob'**; ***console***.log(**`my name is** ${***name***}**`**);  let ***x***=1,***y***=2; ***console***.log(**`**${***x***} **+** ${***y***} **=** ${***x*** + ***y***}**`**);  **function** *f*() {  **return 'world'**; } ***console***.log(**`hello** ${*f*()}**`**); |

结果：



#### 函数

##### rest参数

* rest参数就是js中的可变参数，可变参数传入函数后将放在该参数的数组中

|  |
| --- |
| **function** *f*(...a) {  ***console***.log(a); } *f*(1,2,3); |

结果：



* 每个函数只能有1个可变参数，而且只能作为最后一个参数

##### 严格模式

* ES6中，只要函数参数使用了默认值、解构赋值、或者扩展运算符，那么函数内部就不能显式设定为严格模式，否则会报错

|  |
| --- |
| *// 报错* **function** *doSomething*(a, b = a) {  **'use strict'**;  *// code* }  *// 报错* **const** *doSomething* = **function** ({a, b}) {  **'use strict'**;  *// code* };  *// 报错* **const *doSomething*** = (...a) => {  **'use strict'**;  *// code* };  **const *obj*** = {  *// 报错* doSomething({a, b}) {  **'use strict'**;  *// code* } }; |

* 规避限制的2种方法
  + 全局严格模式

|  |
| --- |
| **'use strict'**; **function** *doSomething*(a, b = a) {  *// code* } |

* + 将函数包裹在1个无参的函数中

|  |
| --- |
| **const *doSomething*** = (**function** () {  **'use strict'**;  **return function**(value = 42) {  **return** value;  }; }()); |

##### call、apply、bind

###### call、apply

下面3种调用foo方法的方式是等效的

|  |
| --- |
| foo.call(**this**, arg1,arg2,arg3) foo.apply(**this**, arguments) **this**.foo(arg1, arg2, arg3) |

###### bind

创建1个新的函数并绑定到指定的对象中，并将该函数返回。

|  |
| --- |
| **var** *bar* = **function**(){  ***console***.log(**this**.**x**); } **var *foo*** = {  **x**:3 } *bar*(); *// undefined* **var *func*** = *bar*.bind(***foo***); ***func***(); *// 3* |

##### 箭头函数

###### 语法

* 箭头函数基本格式

|  |
| --- |
| **var *f*** = (*/\*参数列表\*/*) =>{*/\*函数体\*/*}; |

* 参数

当参数列表中只有1个参数时，可以省略圆括号

|  |
| --- |
| **var *f*** = **v** => v; |

当参数列表中有0个或多个参数时，必须使用圆括号

|  |
| --- |
| **var *f0*** = () => 1; **var *f1*** = (v,u) => v+u; |

* 函数体

当函数体只有1行语句时，可以不用使用{}

|  |
| --- |
| **var *f1*** = (v,u) => v+u; |

当函数体内有多行语句时，必须使用{}

|  |
| --- |
| **var *f*** = (u,v) =>{**u**+=2;**return** u+v}; |

* 返回值

当函数体没有使用{}来标识代码块时，会将函数体内执行的结果直接返回

当函数体使用了{}来表示代码块时，必须使用return进行返回。

###### 注意事项

* 箭头函数体内不会生成新的this对象，始终会使用定义时上下文中的this

|  |
| --- |
| **function** *foo*() {  *//foo函数中的this指向不明  setTimeout*(() => {  *//箭头函数中的this始终指向foo函数中的this  //该处如果是普通函数则会指向global* **console**.log(**'id:'**, **this**.**id**); }, 100); } **id** = 21; *foo*.call({ **id**: 42 });*//42* *foo*();*//21* |

* 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
* 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
* 不可以使用yield命令，因此箭头函数不能用作 Generator 函数。

##### 感悟

###### 函数内的函数实例

* 函数内定义的函数，如果内部函数被引用，每执行1次外部函数，都会创建1个内部函数的实例，而不是覆盖或直接使用

|  |
| --- |
| **var *f0\_arr*** = []; **var *index*** = 0; **function** *f*() {  **function** *f0*(){  **var** x = 0;  }  *f0*.**num** = ***index***;  ***f0\_arr***[***index***++] = *f0*;  ***console***.log(***f0\_arr***);  *setTimeout*(*f*,1000); } *f*(); |

结果：



* 函数内定义的函数，即使内部函数被引用，如果是在外层函数内重复定义了2次该函数，也不会保留2个实例，先定义的将被覆盖

|  |
| --- |
| **var *f0\_arr*** = []; **function** *f*() {  **function** *f0*(){}  *f0*.**num** = 0;  ***f0\_arr***[0] = *f0*;   **function** *f0*(){}  *f0*.**num** = 1;  ***f0\_arr***[1] = *f0*;   ***console***.log(***f0\_arr***); } *f*(); |

结果：



#### 数组

##### 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。

|  |
| --- |
| **function** *add*(x, y) {  **return** x + y; } **const *numbers*** = [4, 38]; *add*(...***numbers***) *// 42* **function** *sum*(...num) {  *//code* } *sum*(...***numbers***); |

### Set、Map

#### Set

* Set 本身是一个构造函数，用来生成 Set 数据结构，它类似于数组，但是成员的值都是唯一的，没有重复的值

|  |
| --- |
| **const *s*** = **new *Set***([1,1,2,3]); [4,4,5,6].forEach(**x** =>***s***.add(x)); ***console***.log(***s***); |

结果：



#### Map

* Object 结构提供了“字符串—值”的对应，Map 结构提供了“值—值”的对应

|  |
| --- |
| **const *m*** = **new *Map***([[**'a'**,**'a'**],[**'b'**,**'b'**]]); ***m***.set(**'c'**,**'c'**); ***console***.log(***m***.get(**'c'**)); |

### promise

#### 基本用法

|  |
| --- |
| **let** fsModule = *require*(**'fs'**);  */\* \* 1. 创建promise对象，向传入promise对象传入回调函数， \* 在该回调函数中执行异步操作 \* 2. 当异步执行成功或失败时，自行调用通过参数传入 \* 的成功的回调函数resolve和失败的会到函数reject \* 3. promise对象的回调函数在promise对象创建后会立即执行  \*/* **let** promiseObj = **new *Promise***((resolve,reject)=>{  fsModule.stat(**'app.js'**,(err,stats)=>{  **if**(err)  reject(err);  **else** resolve(stats);  }) }); */\* \* 1. then方法用来指定promise对象的回调函数的resolve和 \* reject参数 \* 2. then方法的第1个参数用来指定resolve参数，该参数是必选的 \* 第2个参数用来指定reject参数，该参数是可选的 \* 3. 当promise对象的异步操作完成后调用resolve或reject参数方法时 \* then方法参数指定的回调函数才会被调用  \*/* promiseObj.then((data)=>{  ***console***.log(**`是否为文件：**${data.isFile()}**`**);  ***console***.log(**`是否为目录：**${data.isDirectory()}**`**); },(err)=>{  ***console***.log(err); }); |

#### resolve参数方法

##### 调用与指定顺序

当在promise的回调函数的异步操作中调用resolve参数方法时：

* 如果resolve参数方法已经指定，则立即调用
* 如果resolve参数方法还没有指定，则等待对resolve参数方法进行指定后立即调用

例：

|  |
| --- |
| **const *p1*** = **new *Promise***(**function** (resolve, reject) {  *setTimeout*(() => {  ***console***.log(**'p1 timeout'**);  resolve(**'p1 resolve'**);  }, 1000) });  *setTimeout*(()=>{  ***console***.log(**'timer timeout'**);  ***p1***.then(result => ***console***.log(result))  .catch(error => ***console***.log(error)) },3000);  *// p1 timeout // timer timeout // p1 resolve* |

上例中，1s后p1的异步操作中对resolve参数方法进行调用，但是此时resolve参数方法还没有指定，所以只打印出“p1 timeout”，3s之后执行了3s定时器的会到函数，所以打印出“timer timeout”，并同时指定了p1的resolve参数方法，指定后resolve参数方法立即执行了，并打印出“p1 resolve”

此时就产生1个效果：p1的resolve参数方法并没有在原定的时间点执行

##### 多次指定

当多次指定resolve参数方法时，每个指定的resolve的回调函数都会被调用，并且按指定顺序调用

|  |
| --- |
| **const *p1*** = **new *Promise***(**function** (resolve, reject) {  *setTimeout*(() => resolve(**'p1 resolve'**), 3000) });  ***p1***.then(result => ***console***.log(**'111'**)); ***p1***.then(result => ***console***.log(**'222'**));  *//111 //222* |

##### 什么时候执行

* 在promise中立即调用resolve时，then中指定的resolved状态的回调函数不会立即执行，而是在本轮事件轮训结束时执行
* promise的参数函数中的代码立即执行
* setTimeout(fn,0)是在下一轮时间轮训开始时执行

|  |
| --- |
| setTimeout(()=>{  console.log('timeout'); },0);  *let* promise = *new* Promise(*function*(resolve, reject) {  console.log('Promise');  resolve(); });  promise.then(*function*() {  console.log('resolved.'); });  console.log('Hi!');  *// Promise // Hi! // resolved //timeout* |

##### resolve参数方法的参数

resolve参数方法可以携带参数，当传入的参数类型不同时，处理方式不同

###### promise对象

resolve参数方法的参数为promise对象时， 当调用该resolve参数方法时：

* 不会执行该resolve参数方法绑定的回调函数
* 将当前promise身上绑定的resolve和reject的回调函数，重新绑定到作为参数的promise身上
* 将当前promise变为resolved状态

如：

|  |
| --- |
| **const *p1*** = **new *Promise***(**function** (resolve, reject) {  *setTimeout*(() => resolve(**'p1 resolve'**), 3000) }); **const *p2*** = **new *Promise***(**function** (resolve, reject) {  *setTimeout*(() => {  resolve(***p1***)  ***console***.log(**'p2 timeout'**);  }, 1000) }); ***p2***.then(result => ***console***.log(result)); |

上述代码中，p1和p2两个promise同时启动；1s后，p2的异步操作中的回调函数首先得到执行，但由于resolve参数方法传入的参数是p1对象，所以绑定在resolve参数方法身上的回调函数并没有执行，而是将p2身上绑定的resolve的回调函数重新绑定到了p1身上；2s后，p1的异步操作中的回调函数得到执行，并调用resolve参数方法，此时resolve参数方法身上绑定的回调函数，是1s时通过“resolve(p1)”语句，从p2身上转移到p1身上的，原本绑定在p2身上的resolve的回调函数，也就是p2的then方法指定的回调函数

所以该程序的打印结果是：

1s时，打印“p2 timeout”

3s时，打印“p1 resolve”

###### thenable对象

1. thenable对象

thenable对象，就是具有the方法的对象，如：

|  |
| --- |
| let thenable = {  then: function(resolve, reject) {  resolve(42);  }  }; |

1. 处理方式

resolve参数方法的参数是thenable对象时：

* 调用该对象的then方法，并将当前promise的resolve和reject参数方法传入该对象的then方法，
* 当前promise保持pending状态
* 其他的什么也不做（如果该对象中的then方法调用了resolve或reject方法，另当别论）

|  |
| --- |
| *let* thenable = {  then:()=>{} }; *let* p1 = *new* Promise((resolve,reject)=>{  resolve(thenable);  *//等效于* thenable.then(resolve,reject); }); |

###### 其他

resolve参数方法的参数是其他对象或根本就不是对象（如：初始值）或没有参数时：

* 直接调用使用then绑定的resolved状态的回调函数，并将该参数传入
* 当前promise状态变为resolve

#### then（）

##### then方法作用

then方法的第一个参数用来指定resolved状态的回调函数

第二个参数（可选）用来指定rejected状态的回调函数

##### then与promise参数函数中resolve的区别

then方法注册的resolved状态的回调函数不完全等于resolve参数方法，他俩的关系是：

resolve参数方法中，调用了注册的回调函数，除此之外，还做了其他的事情，比如：检查resolve参数方法传入的参数是否含有promise对象，如果没有，则调用注册的回调函数，否则将不调用注册的回调函数，而是将绑定在它所属的promise身上的通过then方法绑定的回调函数，重新绑定到作为参数传入resolve参数方法的promise身上。参见：[promise对象作为resolve参数方法的参数](#_promise对象作为resolve函数的参数)

##### then方法返回值

* then方法的返回值是1个新的promise实例

对于返回的这个新promise实例，就是将then方法指定的2个回调函数中被调用的函数的返回值使用 [promise.resolve](#resolve)方法转换得到的promise对象；由于[then与promise参数函数中resolve的区别](#_then与promise参数函数中resolve的区别)中的特性，导致了出现了[then的链式写法](#_then的链式写法)中的2种情况。

##### then的链式写法

由于then方法返回的是1个新的promise对象，因此then方法的调用可以采用链式写法，但是这种写法分为两种情况：

1：前1个then指定的resolved状态的回调函数返回的不是promise对象

2：前1个then指定的resolved状态的回调函数返回的是promise对象

###### 返回的不是promise对象

当前1个then指定的resolved状态的回调函数返回的不是promise对象时，后1个then指定的resolved状态的回调函数将会被立即调用，并将返回值作为参数传入

|  |
| --- |
| **const *p1*** = **new *Promise***(**function** (resolve, reject) {  *setTimeout*(() => resolve(**'p1 resolve'**), 1000) }); ***p1***.then((i) => {  ***console***.log(i);  **return '1'** }).then((i) => {  ***console***.log(i); }); |

上述代码中，第1个then指定的resolved状态的回调函数中返回的是字符串，不是promise对象，所以第2个then指定的resolved状态的回调函数会被立即调用

###### 返回的是promise对象

当前1个then指定的resolved状态的回调函数返回的是promise对象时，后1个then指定的resolved状态的回调函数，需要等待返回的promise对象的异步操作执行成功，才会被调用

|  |
| --- |
| **const *p1*** = **new *Promise***(**function** (resolve, reject) {  *setTimeout*(() => resolve(**'p1 resolve'**), 1000) }); ***p1***.then((i) => {  ***console***.log(i);  **return new *Promise***(**function** (resolve, reject) {  *setTimeout*(() => resolve(**'p2 resolve'**), 1000)}) }).then((i) => {  ***console***.log(i); }) |

上述代码中，第1个then指定的resolved状态的回调函数中返回的是promise对象，所以第2个then指定的resolved状态的回调函数，需要等待返回的promise对象的异步操作执行成功，才会被调用

打印结果：

1s后打印：p1 resolve

2s后打印：p2 resolve

#### catch（）

* catch是then指定reject回调函数的别名，cath可以用来捕获异常，以下三种情况可以被当做异常被catch捕获：
* promise的参数函数中发生错误

|  |
| --- |
| *const* promise = *new* Promise(*function* (resolve, reject) {  *throw new* Error('test'); }); promise.then(*function* (value) {  console.log(value) })  .catch(*function* (error) {  console.log(error)  }); *// ERROR:test* |

* resolve参数方法中发生错误

|  |
| --- |
| *const* promise = *new* Promise(*function* (resolve, reject) {  resolve(); }); promise.then(*function* (value) {  *throw new* Error('test'); })  .catch(*function* (error) {  console.log(error)  }); *// ERROR:test* |

* promise变成rejected状态

|  |
| --- |
| *const* promise = *new* Promise(*function* (resolve, reject) {  reject(*new* Error('TEST')); }); promise.then(*function* (value) {  console.log(value); })  .catch(*function* (error) {  console.log(error)  }); *// ERROR:TEST* |

* 但是如果promise的状态已经变为resolved状态，在抛出异常，catch则不能捕获这个异常了

|  |
| --- |
| *const* promise = *new* Promise(*function*(resolve, reject) {  resolve('ok');  *throw new* Error('test'); }); promise  .then(*function*(value) { console.log(value) })  .catch(*function*(error) { console.log(error) }); *// ok* |

* promise对象的错误具有冒泡性质，会一直向后传递，直到被捕获为止

也就是说，无论链式使用了多少个promise，最后放1个catch，就能捕获所有的错误;但是如果错误在前面已经被捕获一次，后面就捕获不到这个错误了

* + 例1：

所有错误在最后捕获

|  |
| --- |
| *const* promise = *new* Promise(*function* (resolve, reject) {  reject(*new* Error('TEST')); }); promise.then(*function* (value) {  console.log(value); })  .catch(*function* (error) {  console.log(error)  }); *// ERROR:TEST* |

这里1共有2个promise对象，1个是创建的promise，另一个是then返回的promise，在创建的promise中产生错误后，会向后传递，直到遇到cath被捕获

* + 例2

错误被前面的reject捕获后不会继续向后传递

|  |
| --- |
| *const* promise = *new* Promise(*function* (resolve, reject) {  reject(*new* Error('TEST')); }); promise.then(*function* (value) {  console.log(value); },*function* (error) {  console.log('reject'); })  .catch(*function* (error) {  console.log(error)  }); *// reject* |

创建的promise中产生错误后，直接被他的then方法指定的reject方法捕获，而不继续向后传递被catch捕获

* promise中产生的错误，如果没有被捕获，不会向外传递，影响外层代码

#### finally（）

finally用于指定，无论状态如何，最后一定要执行的代码；finally指定的回调函数没有参数

#### all（）

* 该方法用于将多个promise实例包装成1个新的promise实例

all方法接受具有iterator接口的对象，比如：数组

数组中的元素如果不是promise实例，会自动先使用promise.resolve方法将该实例转换为promise实例

* 状态

以下面代码为例：

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

p的状态由p1、p2、p3的状态决定：

* p会遍历并执行p1、p2、p3，最后检查p1、p2、p3的状态

当p1、p2、p3的状态全部变为resolved状态时，p的状态变为resolved状态，此时p1、p2、p3的resolved状态的回到函数的返回值组成1个数组，传递给p的resolve回调函数

* p会遍历并执行p1、p2、p3，最后检查p1、p2、p3的状态

当p1、p2、p3中任何1个的状态变为rejected状态，p的状态就变为rejected，此时将第1个变为rejected状态的promise中的错误传递给p的reject回调函数

注意：即使前面的promise已经变为rejected状态，后面的promise依然会执行

* catch

如果数组中变为rejected的promise具有自己的catch或reject方法，则不会被p的catch捕获

原理：具有自己的catch方法的promise发生错误后执行自己的catch后返回1个新的promise实例，该实例的状态为resolved，所以p得状态就是resolve，p的resolved状态的回调函数会被执行，而不是catch

#### race（）

该方法与all除了状态改变时不同，其他均一致

* 状态

以下面代码为例：

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

p的状态由最先改变状态的promise的状态决定

当某个promise的状态已经确定时，p的状态就会变为这个promise的状态，并将这个promise的错误或 resolved状态的回调函数的返回值作为参数传递给p的resolve方法

#### resolve（）

##### 作用

promise.resolve()的作用就是将传入的对象转换为promise对象

##### 原理

promise.resolve()的原理就是：创建1个直接调用resolve参数方法的promise，并将promise.resolve()的参数传入这个新创建的promise的resolve参数方法。

|  |
| --- |
| Promise.resolve('foo')  // 等价于  new Promise(resolve => resolve('foo')) |

##### resolve()的参数

resolve()的参数分为以下4种情况：

* + promise对象
  + thenable对象
  + 其他
  + 没有参数

传入这4种参数时的作用，参见[resolve参数方法的参数](#resolve参数方法的参数)

#### reject()

reject（）的作用是创建1个直接调用其reject参数方法的promise对象，并将promise.reject()的参数原封不动的传给该promise的rejected状态的回调函数（这点与resolve（）不同，不会判断传入参数的类型）

|  |
| --- |
| const thenable = {  then(resolve, reject) {  reject('出错了');  }  };  Promise.reject(thenable)  .catch(e => {  console.log(e === thenable)  })  // true |

上例中，传入的是thenable对象，该对象被原封不动的传给了rejected状态的回调函数。

### async、await

#### 基本用法

* 当某个函数中使用了promise执行异步操作，并且想把该函数内的操作变为同步的，可以使用async修饰这个函数，并且在async函数中使用await等待promise的异步操作执行完成。
* await修饰的promise的返回值就是该promise执行resolve时的参数
* await只能用在async函数中
* async方法返回的是promise对象，并且是立即返回的，不需要等待函数内的异步执行结束；该promise是一个空的promise（这点与[resolve（）](#resolve（）)不同），
  + 当函数内可以执行到的语句（包括promise）全部执行成功后，返回的promise的状态变为resolved状态，并将该函数的返回值作为参数调用该promise的resolve参数方法。
  + 当函数内任何语句执行出错后，则该函数内的语句立即终止执行，后面的语句不会被执行，并将返回的promise变为rejected状态，并将错误作为参数调用返回的promise的reject参数方法
* async函数只能将函数内部变为同步的，但函数内与函数外还是异步的

### Iterator和for…of

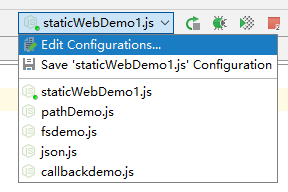
#### Iterator

## WEB

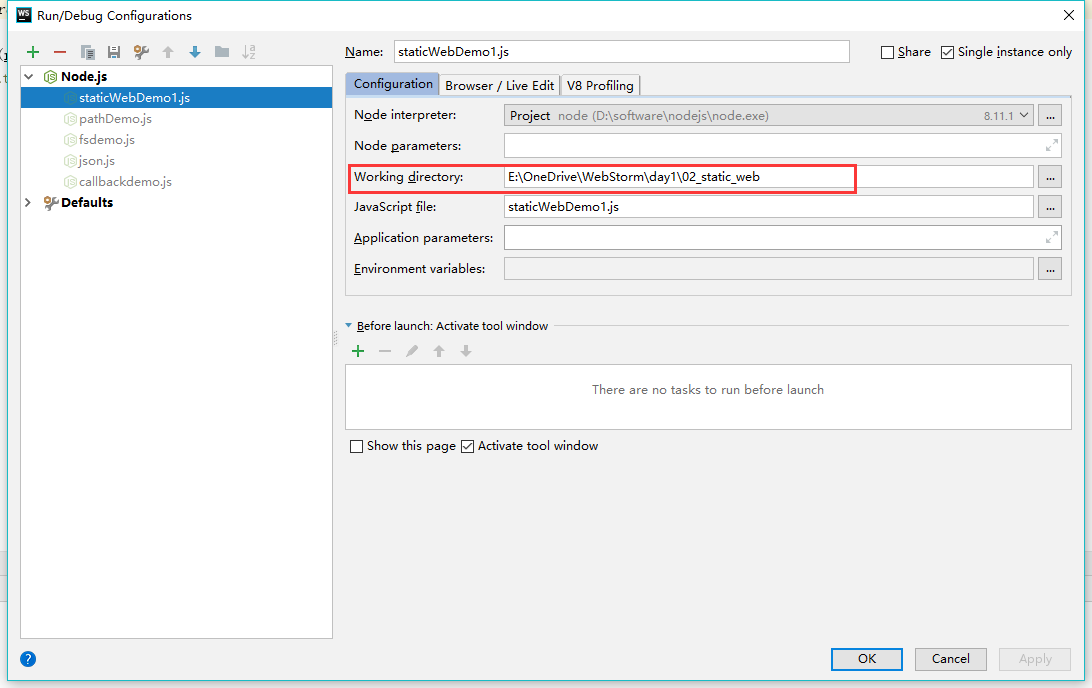
### 工程路径

#### WebStorm

在软件右上角如图所示位置可以设置当前工程的配置



在配置页面下图所示位置可以设置当前工程路径



该路径是项目的根路径，代码中所有涉及到的相对路径都是相对工程路径来说的

### 模块

#### 介绍

* 模块分为核心模块、第三方模块和自定义模块
* 将公共的功能抽离成1个单独的js文件作为1个模块；默认情况下模块中的方法和属性无法直接被外部访问，需要在模块内使用exports暴露属性和方法

在需要使用该模块的文件中，使用require引入这个模块。

##### require

###### nodejs中使用require引入模块

|  |
| --- |
| **var** url = *require*(**'模块'**); |

###### 对于require的参数：

* 可以省略“.js”后缀

|  |
| --- |
| **var** tools = *require*(**'./tools.js'**); *//等效于* **var** tools = *require*(**'./tools'**); |

* 对于核心模块，无需使用路径，直接引入模块名即可

|  |
| --- |
| **var url** = *require*(**'url'**); |

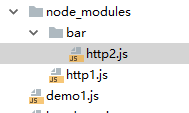
* 对于自定义模块
  + 使用相对路径引入模块

使用相对路径时，会在相对于[工程路径](#_工程路径)的相对路径下寻找指定的模块

|  |
| --- |
| **var** tools = *require*(**'./tools'**); |

* + 自动在当前文件同级的“node\_modules”目录下查找

当引入的文件在与当前文件同级的“node\_modules”目录下时，可以直接引入模块名或“node\_modules”目录下的“目录+模块名”



|  |
| --- |
| **var** http1 = *require*(**'http1'**); **var** http2 = *require*(**'bar/http2'**); |

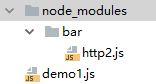
当node\_modules目录下的模块与核心模块重名时，优先引入核心模块

* + 根据package.json文件查找

参见[package.json](#_package.json)

###### package.json

* 在demo1.js文件的同级目录下有1个“node\_modules”目录，该目录下有1个bar目录，bar目录下有1个http2.js文件



* 在bar目录下打开cmd窗口，执行如下命令

|  |
| --- |
| npm init --yes |

在bar目录下会自动生成package.json文件

* 在demo1.js文件中通过如下方式引入http2.js文件

|  |
| --- |
| **var** http2 = *require*(**'bar'**); |

通过该方式，即可成功引入http2模块

其中：require的参数是“node\_modules”下的bar目录名称

* package.json文件解析

package.json文件内容如下：

|  |
| --- |
| {  "name": "bar",  "version": "1.0.0",  "description": "",  "main": "http2.js",  "scripts": {  "test": "echo \"Error: no test specified\" && exit 1"  },  "keywords": [],  "author": "",  "license": "ISC"  } |

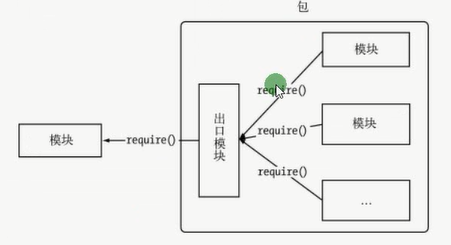
* + main属性

当前目录下有多个js文件时，通过该属性设置引入该目录名代表的模块时，将哪个模块提供出去

* + name属性

require的参数是由当前目录名决定的，与name属性无关，但一般name属性就是当前目录名

#### 包与模块



1个功能由多个模块组成时，将这些模块放在同一个目录下，最终由1个模块向外提供引用，这个目录就是包

package.json方式引入模块其实引入的就是包

#### 下载、使用第三方模块

通过npm命令下载

如：

下载silly-datetime模块

在项目根目录下执行如下命令

|  |
| --- |
| npm i silly-datetime –save |

则在该目录下生成node\_modules目录，将第三方模块包放在该目录下，并将新下载的模块添加到package.json中



然后就可以通过[package.json](#_package.json)的方式引入第三方模块，此时无需自己生成package.json文件，直接引入目录名代表的模块即可

|  |
| --- |
| **var** silly\_datetime = *require*(**'silly-datetime'**); |

### NPM

#### NPM介绍

我们可以通过npm下载各种包，这些包从<http://www.npmjs.com>上下载

#### 命令

以jquery为例

##### 安装模块

在想要安装的目录下执行

|  |
| --- |
| npm install jquery |

install 可以简写为 i

##### 卸载模块

* 命令卸载

|  |
| --- |
| npm uninstall jquery |

* 手动卸载

删除安装位置下“node\_modules”目录下的模块的目录即可

##### 查看已安装模块

|  |
| --- |
| npm list |

##### 查看模块版本

|  |
| --- |
| npm info jquery |

##### 指定版本安装

|  |
| --- |
| npm install jquery@1.8.0 |

#### package.json

##### 介绍

package.json用来描述包的描述、版本、依赖关系等信息的，有了package.json文件，只需要给别人发送出口文件和package,json，对方就可以通过package.json获得包的依赖文件并运行。

##### 生成

package.json的生成有2种方式：

* 手动生成

在包下执行如下命令

|  |
| --- |
| npm init |

这种方式生成时，需要手动执行该文件中的信息

* 自动生成

在包下执行命令

|  |
| --- |
| npm init --yes |

--yes代表强制生成

使用该方式时，文件中的信息为默认的

##### 作用

比如：

当前包下有出口文件main.js，node\_modules目录下有silly\_datetime模块，并且生成了package.json。此时将node\_modules目录删掉，程序将无法运行，此时在package.json目录下执行“npm install”命令，即可安装node\_modules下的依赖包，程序就可以正常运行了

##### 将手动下载的模块添加到package.json

当package.json已经生成了，现在要下载新的模块，新的模块要添加到package.json，则在执行安装命令时，后面加上“--save”或“—save-dev”

|  |
| --- |
| npm install md5-node --save  或  npm install md5-node --save-dev |

两者在使用上没有区别

#### 工具

##### 默认安装路径

|  |
| --- |
| npm config set prefix "D:\Program Files\npm\_global\_modules\node\_modules" |

设置完默认安装路径后需要手动将该路径设置到环境变量中

##### supervisor工具

###### 介绍

使用node命令运行node程序时，每次修改代码都需要手动重启才能生效，使用sipervisor不需要手动重启就能生效新的代码

###### 使用

1. 安装

在cmd窗口执行如下命令进行安装

|  |
| --- |
| npm -g install supervisor |

1. 使用

* cmd窗口使用

使用supervisor命令代替node命令执行js文件

|  |
| --- |
| supervisor xxx.js |

* WS中使用

在WebStorm中选择要Run--Edit Configurations 下选中要运行的JS文件，在右侧选设置Node interpreter为C:\Users\Daobing\AppData\Roaming\npm\node-supervisor.cmd

### 静态WEB服务器

#### 静态页面展示

##### 思想

* 使用http模块创建服务，监听http请求；使用url模块对请求的url进行解析，得到真正的请求路径（有的请求的url会携带get参数），并过滤掉“/favicon.ico”请求
* 将请求路径作为文件路径，在本地读取对应的文件，将文件内容返回
* 对于找不到的文件，跳转到404页面并返回404错误
* 针对不同的文件做简要处理，返回不同的请求头

如：

html文件的“Content-Type”请求头应为：“text/html;charset=utf-8”

css文件的“Content-Type”请求头应为：“text/css;charset=utf-8”

js文件的“Content-Type”请求头应为：“text/javascript;charset=utf-8”

##### 代码

|  |
| --- |
| **var httpModule** = *require*(**'http'**); **var urlModule** = *require*(**'url'**); **var** fsMoudle = *require*(**'fs'**); **var** pathModule = *require*(**'path'**);  **httpModule**.*createServer*((req,res)=>{  *//解析请求头* **let** urlParse = **urlModule**.*parse*(req.**url**,**true**);  *//过滤请求* **if**(urlParse.**pathname** != **'/favicon.ico'**){  *//默认访问index.html* **let** pathname = (urlParse.**pathname** == **'/'**)? (urlParse.**pathname**+**'index.html'**):urlParse.**pathname**;  *//得到本地文件路径* pathname = **'static'**+ pathname;  *//获取该文件对应的请求头中的部分字段* **let** mime = *getMime*(pathname);  *//读取文件* fsMoudle.readFile(pathname,(err,data)=>{  *//读取文件出错* **if**(err){  *//读取404页面内容* fsMoudle.readFile(**'static/404.html'**,(error,data404)=>{  *//返回44页面内容* res.writeHead(404,{**'Content-Type'**:**'text/html;charset=utf-8'**});  res.write(data404);  res.end();  })  }  *//成功读取文件* **else**{  *//返回请求文件内容*  *//根据不同文件设置不同请求头* res.writeHead(200,{**'Content-Type'**:mime+**';charset=utf-8'**});  res.write(data);  res.end();  }  })  } }).listen(80);  **function** *getMime*(path) {  **let** mime = pathModule.extname(path);  **switch** (mime){  **case '.html'**:  **return 'text/html'**;  **case '.css'**:  **return 'text/css'**;  **case '.js'**:  **return 'text/javascript'**;  **default**:  **return 'text/html'**;  } } |

##### getMime方法改进

上面代码中只能处理3种后缀名文件对应的请求头，还有很多文件会对应不同的请求头，如：zip、jpg文件；

我们可以将所有后缀名对应的请求头以json对象的形式放在1个文件中，然后在getMime方法中获取该文件的内容并转换为json对象，在该对象中查找某个后缀名对应的请求头。

|  |
| --- |
| **function** *getMime*(path) {  *//读取文件* **let** data = fsMoudle.readFileSync(**"mime.json"**);  *//将文件内容转换为json对象* **let** json = ***JSON***.parse(data);  *//截取参数的后缀名* **let** mime = pathModule.extname(path);  *//返回该后缀名对应的请求头  //没有该后缀名默认返回'text/html'* **return** json[mime] || **'text/html'**; } |

#### 静态文件托管

静态文件托管就是将[静态页面展示](#_静态页面展示)中的代码封装到1个模块中，专门进行静态文件的展示

* staticRouter.js

|  |
| --- |
| **var urlModule** = *require*(**'url'**); **var** fsMoudle = *require*(**'fs'**); **var** pathModule = *require*(**'path'**);  exports.staticRouter = (req,res,staticPath)=>{  *//解析请求头* **let** urlParse = **urlModule**.*parse*(req.**url**,**true**);  *//过滤请求* **if**(urlParse.**pathname** != **'/favicon.ico'**){  *//默认访问index.html* **let** pathname = (urlParse.**pathname** == **'/'**)? (urlParse.**pathname**+**'index.html'**):urlParse.**pathname**;  *//得到本地文件路径* pathname = staticPath+ pathname;  *//获取该文件对应的请求头中的部分字段* **let** mime = *getMime*(pathname);  *//读取文件* fsMoudle.readFile(pathname,(err,data)=>{  *//读取文件出错* **if**(err){  *//读取404页面内容* fsMoudle.readFile(staticPath+**'/404.html'**,(error,data404)=>{  *//返回44页面内容* res.writeHead(404,{**'Content-Type'**:**'text/html;charset=utf-8'**});  res.write(data404);  res.end();  })  }  *//成功读取文件* **else**{  *//返回请求文件内容  //根据不同文件设置不同请求头* res.writeHead(200,{**'Content-Type'**:mime+**';charset=utf-8'**});  res.write(data);  res.end();  }  })  } }  **function** *getMime*(path) {  *//读取文件* **let** data = fsMoudle.readFileSync(**"mime.json"**);  *//将文件内容转换为json对象* **let** json = ***JSON***.parse(data);  *//截取参数的后缀名* **let** mime = pathModule.extname(path);  *//返回该后缀名对应的请求头  //没有该后缀名默认返回'text/html'* **return** json[mime] || **'text/html'**; } |

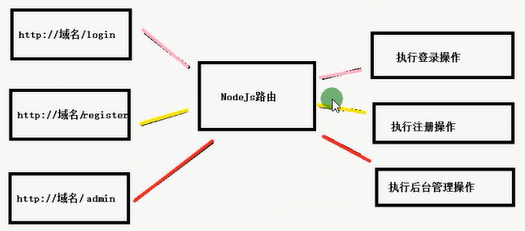
* staticWeb.js

|  |
| --- |
| **var httpModule** = *require*(**'http'**); **var** staticRouterModule = *require*(**'./modules/staticRouter'**);  **httpModule**.*createServer*((req,res)=>{  staticRouterModule.staticRouter(req,res,**'static'**) }).listen(80); |

#### 路由

##### 概念

路由就是根据不同的url，将请求分发到不同的方法中



##### 例

|  |
| --- |
| **httpModule**.*createServer*((req,res)=>{  *//获取请求路径* **let** urlParse = **urlModule**.*parse*(req.**url**,**true**);  **let** pathname = urlParse.**pathname**;  *//根据不同的请求路径执行不同的方法* **switch**(pathname){  **case "/login"**:  res.end(**'login'**);  **break**;  **case "/dologin"**:  res.end(**'dologin'**);  **break**;  **default**:  res.end(**'index'**);  **break**;  } }).listen(80); |

##### 路由封装

###### 模块化（初级）

1. 介绍

模块化方法封装路由的思想就是，保证请求路径与要调用的方法同名，此时就可以将所有请求路径的请求方法封装到1个对象中，并将该对象封装到1个模块中，在主模块中通过请求路径字符串获取对象中的方法并执行即可

1. 例

* moduleRouter.js

|  |
| --- |
| module.exports.**requestFunc** = {  login: (req, res) => {  res.end(**'login'**);  },  regist: (req, res) => {  res.end(**'regist'**);  },  page404: (req, res) => {  res.end(**'page404'**);  } } |

* staticWebDemo.js

|  |
| --- |
| **httpModule**.*createServer*((req, res) => {  *//解析url* **let** urlParse = **urlModule**.*parse*(req.**url**, **true**);  *//获取请求路径并去掉最开始的'/'* **let** pathname = urlParse.**pathname**.substring(1);  res.writeHead(200, {**"Content-Type"**: **"text/html;charset=utf-8"**})  **try** {  *//在路由模块中找到对应的请求方法并执行* moduleRouterModule.**requestFunc**[pathname](req, res);  } **catch** (err) {  *//如果找不到则跳转到404页面* moduleRouterModule.**requestFunc**[**'page404'**](req, res);  } }).listen(80); |

###### 仿EXPRESS路由

1. express路由介绍



将路由模块引入之后，将该路由模块对象作为createServer方法的参数传入来创建http服务，然后使用app对象的get、post等方法为某个url注册1个请求方法，当接受到请求之后，就会根据请求方式和url调用之前注册的对应的方法来响应请求。

1. 思路

以下说明均为路由模块中的实现思路：

* app对象能作为参数传入createServer方法，证明app对象一定是1个参数为（req，res）的方法，该方法功能后面说明
* 创建1个对象requestCallback，格式如下：

|  |
| --- |
| **var** requestCallback = {**GET**:{},**POST**:{}}; |

该对象用来保存不同请求方式下，不同请求路径的请求方法。

* 为app方法添加几个属性方法，如：get、post等；get方法用于将注册的url和对应的请求方法保存到 requestCallback的GET子对象中，post方法用于将注册的url和对应的请求方法保存到 requestCallback的POST子对象中
* 实现app方法，该方法的功能就是根据请求的方式和请求路径，在requestCallback中找到对应的请求方法进行调用。

1. 例

* expressRouter.js

|  |
| --- |
| **let urlModule** = *require*(**'url'**); *//requestCallback对象用于存储注册过的不同请求方式下不同url对应的 // 请求方法* **let** requestCallback = {**GET**:{},**POST**:{}}; *//需要暴露的方法，该方法作为createServer函数的参数，每次请求都会 // 调用该方法；在该方法中根据请求方式及url调用注册的请求方法* **function** *app*(req,res) {  **let** method = req.**method**;  **let** urlPath = **urlModule**.*parse*(req.**url**).**pathname**;  **try**{  requestCallback[method][urlPath](req,res);  }**catch**(err){  res.writeHead(200, {**"Content-Type"**:  **"text/html;charset=utf-8"**});  res.end(**"无法找到路由"**);  } } *app*.get = (urlPath,callback)=>{  requestCallback[**'GET'**][urlPath] = callback; }; *app*.post = (urlPath,callback)=>{  requestCallback[**'POST'**][urlPath] = callback; }; module.exports = *app*; |

* webDemo.js

|  |
| --- |
| **let httpModule** = *require*(**'http'**); **let** ejsModule = *require*(**'ejs'**); **let urlModule** = *require*(**'url'**); **let** *app* = *require*(**'./modules/expressRouter'**);  **httpModule**.*createServer*(*app*).listen(80);  *app*.get(**'/login'**,(req,res)=>{  res.writeHead(200, {**"Content-Type"**: **"text/html;charset=utf-8"**});  ejsModule.renderFile(**'./ejs/login.ejs'**,{},(err,data)=>{  res.end(data);  }) }); *app*.get(**'/dologin'**,(req,res)=>{  res.end(**'GET:dologin'**); }); *app*.post(**'/dologin'**,(req,res)=>{  res.end(**'POST:dologin'**); }); |

#### get、post请求

##### 介绍

get请求和post请求获取请求参数的方法时不同的，所以需要在程序中对请求进行区分并使用对应的方法获取请求参数。

##### 思路

* 在request对象中有1个method属性，标识了本次请求的请求方式
* 程序中通过requst对象的method属性来判断是哪种方式的请求
* 然后在不同方式的请求中分别使用对应的方法获取请求参数

##### 获取请求参数

###### get方式

通过url模块的parse方法解析request对象的url，得到的对象中有1个query属性以对象的方式存储了get请求的请求参数

|  |
| --- |
| **let** urlParse = **urlModule**.*parse*(req.**url**,**true**);  ***console***.log(**'GET:'**+urlParse.**query**); |

###### post方式

通过request对象的事件驱动得到请求参数

|  |
| --- |
| **let** postParam = **''**; req.on(**'data'**,(data)=>{  postParam += data; }); req.on(**'end'**,()=>{  ***console***.log(**'POST:'**+postParam); }) |

##### 例

* login.ejs

|  |
| --- |
| <!DOCTYPE **HTML** PUBLIC **"-//W3C//DTD HTML 4.01 Transitional//EN"  "http://www.w3.org/TR/html4/loose.dtd"**> <**html**> <**head**>  <**title**>登陆</**title**> </**head**> <**body**> <**form action="/dologin" method="post"**>  账号：<**input type="text" name="username"**><**br**>  密码：<**input type="password" name="password"**><**br**>  <**button type="submit"** >登陆</**button**> </**form**> </**body**> </**html**> |

* staticWebDemo.js

|  |
| --- |
| **httpModule**.*createServer*((req,res)=>{  *//获取请求路径* **let** urlParse = **urlModule**.*parse*(req.**url**,**true**);  **let** pathname = urlParse.**pathname**;  res.writeHead(200,{**"Content-Type"**:**"text/html;charset=utf-8"**})  *//根据不同的请求路径执行不同的方法* **switch**(pathname){  **case "/login"**:  ejsModule.renderFile(**"./ejs/login.ejs"**,{},(err,data)=>{  res.end(data);  })  **break**;  **case "/dologin"**:  **if**(req.**method** == **"GET"**){ *//GET请求* res.end(**'GET:'**+urlParse.**query**);  }**else if**(req.**method** == **"POST"**){ *//POST请求* **let** postParam = **''**;  req.on(**'data'**,(data)=>{  postParam += data;  });  req.on(**'end'**,()=>{  res.write(**'POST:'**+postParam);  res.end(**'<script>alert("登陆成功");</script>'**);  })  }  **break**;  **default**:  res.end(**'index'**);  **break**;  } }).listen(80); |

#### 后台编写前台的js代码

##### 介绍

在页面上，可以使用js代码执行一些动作，如alert方法；但是在后台无法世家使用前台的方法，所以需要使用下面的方法在后台编写前台使用的js代码

##### 方法

向前台输出如下格式的字符串：

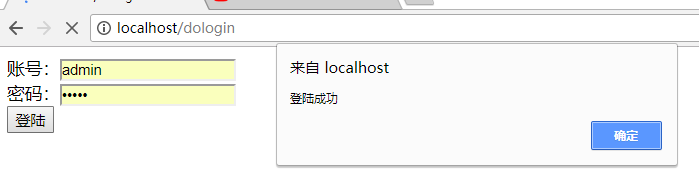
|  |
| --- |
| **'<script>前端js代码</script>'** |

##### 例

|  |
| --- |
| res.end(**'<script>alert("登陆成功");</script>'**); |

结果：

发送请求后弹出登陆成功的提示框



### 非阻塞IO、事件驱动

#### 非阻塞IO

* 阻塞IO

C中常用的同步编程就是阻塞IO

以fs模块中的readFileSync方法为例，这是1个同步读取文件的方法，该方法在文件读取过程中，线程处于阻塞状态，只要文件读取成功，进程恢复，并将读取到的内容通过该函数的返回值返回。

这样的编程方法称为阻塞IO

* 非阻塞IO

以fs模块中的readFile方法为例，这是1个异步读取文件的方法，该方法调用后，启动1个新的线程去读取文件，同时注册1个回调函数，当前线程不会进入阻塞状态，会继续向下执行。在1个不确定的时刻，文件读取成功后，会调用之前注册的回调函数将数据返回。

这种编程方法称为非阻塞IO

实现非阻塞IO编程的方式常用的有2种：回调函数、事件驱动

#### 回调函数

|  |
| --- |
| **function** *f*(num,func) {  **if**(num > 0)  func(num); }  *f*(1,(i)=>{  ***console***.log(i); }) |

#### 事件驱动

参见[events](#_events)模块

#### Demo

##### 打印目录结构

|  |
| --- |
| **var** fs = *require*(**'fs'**)  */\* \* 通过递归实现遍历目录结构 \* 参数： \* dir：当前要遍历的目录 \* finish：当遍历完子目录时，需要返回上一层目录继续遍历上一层目录 \* 的下一项；该参数就是遍历上一层目录下一项的回调函数  \*/* **function** *readDir*(dir, finish = **null**) {  */\*  \* 1. 每遍历1个新目录时，会调用readDir函数来遍历该目录的子项  \* 每次调用时，都会产生1个fs.readdir的回调函数的实例  \* 2. 所以：每个目录都会对应1个fs.readdir的回调函数的实例，  \* 每个实例中保存着对应目录下的所有子项（files参数中）；  \* 和该实例的next方法  \*/* fs.readdir(dir, **function** (err, files) {  */\*  \* 1. 异步编程中不可以在for循环中使用回调函数，否则将产生  \* 调用顺序的不确定性；可以使用递归来代替for循环功能  \* 2. next函数用来遍历当前目录下的所有子项  \*/* (**function** next(i) {  *//是否结束递归的判断条件* **if** (i < files.**length**) {  **var** pathname = dir+**'/'**+files[i];  *printFile*(pathname);   fs.stat(pathname, **function** (err, stats) {  **if** (stats.isDirectory()) {  */\*  \* 1. 如果当前项是目录，则递归遍历该目录的  \* 子项，遍历结束后通过回调函数继续遍历  \* 当前目录的下一项  \* 2. 下面readDir方法的回调函数中的next方  \* 法是当前目录对应的fs.readdir方法的回  \* 调函数的实例中的next方法，而不是当前  \* 项对应的fs.readdir方法的回调函数的实  \* 例中的next方法；  \* 所以调用该回调函数时，会回到当前目录  \* 继续遍历，而不会继续在子目录下遍历或  \* 出现异常  \*/  readDir*(pathname, **function** () {  *next*(i + 1);  });  } **else** {  *//如果当前项是文件，则继续遍历本目录下的下  // 一项  next*(i+1);  }  });  } **else** {  */\*  \* 1. 如果当前目录下的子项全部遍历结束，则调用回调  \* 继续遍历上一层目录的下一项  \* 2. 判断是否存在回调，比如当前已经在最顶层目录，  \* 则不存在上一层目录，也就无需继续遍历上一层目  \* 录的下一项  \*/* finish && finish();  }  })(0);*//通过函数自调用开始遍历本层目录的第一项* }); }  **function** *printFile*(file) {  **var** fileArr = file.split(**'/'**);  ***console***.log(**'\t'**.repeat(fileArr.length - 2) + fileArr[fileArr.length-1]); }  *readDir*(**"node\_modules"**); |

## EJS模板

### 介绍

ejs文件就是1个模板，ejs文件中使用的就是HTML的语法，另外配合3种脚本元素获取传入模板的数据以及使用js语句，来达到动态显示数据的效果

### 3种脚本元素

* <% %>

该标签用于在html页面上使用JS语句

* <%= %>

该标签用于获取变量的值并输出到页面

* <%- %>

如果变量是字符串，并且字符串中包含html标签时，浏览器会将整个字符串进行输出，并不会对字符串中的标签进行解析

此时需要使用该标签，浏览器就可以对字符串中的标签解析之后在输出了

### 引入外部模板

#### 介绍

当多个页面的某部分完全一致时，可以将这部分页面抽离出去，写在1个单独的页面中，在使用这部分页面的页面中使用如下方法引入该页面

|  |
| --- |
| <%**include 要引入的文件相对于当前文件的路径**%> |

#### 例

##### 分离前

* index.ejs

|  |
| --- |
| <!DOCTYPE **HTML** PUBLIC **"-//W3C//DTD HTML 4.01 Transitional//EN"  "http://www.w3.org/TR/html4/loose.dtd"**> <**html**> <**head**>  <**title**>$Title$</**title**> </**head**> <**body**>  <**h1**>你好 <%=msg%></**h1**>  </**body**> </**html**> |

##### 分离后

* header.ejs

|  |
| --- |
| <**h1**>你好 <%=msg%></**h1**> |

* index.ejs

|  |
| --- |
| <!DOCTYPE **HTML** PUBLIC **"-//W3C//DTD HTML 4.01 Transitional//EN"  "http://www.w3.org/TR/html4/loose.dtd"**> <**html**> <**head**>  <**title**>$Title$</**title**> </**head**> <**body**>  <%**include public/header.ejs**%>  </**body**> </**html**> |

* 目录结构如下



# MongoDB

## 介绍

### NoSQL

#### 介绍

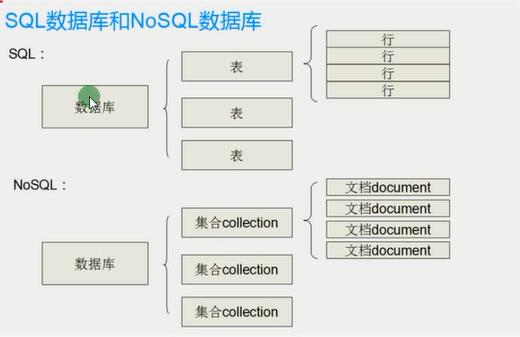
MongoDB属于非关系型数据库（NoSQL），未必遵循SQL标准

#### 适用场景

* 对数据库高并发读写的要求
* 对海量数据的高效存储和访问的需求
* 对数据库的高可扩展性和高可用性的需求

#### 与传统数据库对比

* NoSQL没有行、列的概念，用JSON来存储数据
* 集合相当于SQL的表，文档相当于SQL的行



### MongoDB

#### 介绍

#### 安装

##### 下载

[www.mongodb.com](http://www.mongodb.com)

3.6.3版本未安装成功

##### 安装

* 下一步下一步安装
* 配置环境变量

将bin目录放到环境变量中

在cmd窗口输入“mongo”，未提示不是系统命令则安装成功

#### 使用

##### 服务端

###### windows

* 新建1个文件夹，用来存储数据；不能有中文、空格等字符
* 启动服务

cmd窗口中执行如下命令：

|  |
| --- |
| mongod --dbpath 目录 |

如：

|  |
| --- |
| mongod –dbpath E:\OneDrive\WebStorm\mongo\_data |

此时服务已经启动，窗口不关闭服务一直存在

###### MAC

brew services start mongodb

##### 客户端

打开新的cmd窗口

* 连接本地数据库

|  |
| --- |
| mongo |

* 连接远程数据库

|  |
| --- |
| mongo IP:Port |

如：

|  |
| --- |
| mongo 127.0.0.1:27017 |

## 操作

### 数据库操作

#### 查看所有数据库

|  |
| --- |
| show dbs |

如：



#### 使用数据库

|  |
| --- |
| use (数据库名称) |

如：



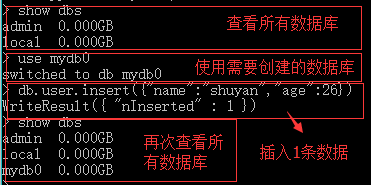
注意：

use命令可是用来使用不存在的数据库

#### 创建数据库

* mongodb没有创建数据库的命令，只能通过如下方法创建：
  1. 使用需要创建的数据库（此时数据库没有创建成功）
  2. 向该数据库的某个表中插入一条数据（此时数据库创建成功）
* 例

创建mydb0数据库



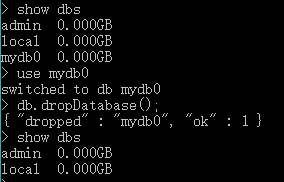
#### 删除数据库

* mongodb的删除数据库是删除所在的数据库，所以需要先使用数据库，在进行删除
* 命令

|  |
| --- |
| db.dropDatabase(); |

* 例

删除mydb0数据库



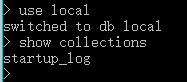
* 注意
  + 没有表的数据库会自动删除

### 集合（表）操作

#### 查看所有表

|  |
| --- |
| show collections |

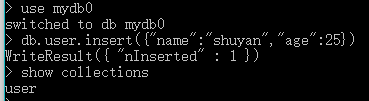
如：



#### 创建表

* mongodb没有创建表的命令，只能通过如下方法创建：
  1. 向数据库的需要创建的表中插入一条数据
* 例

创建user表

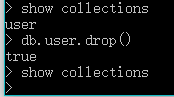


#### 删除表

|  |
| --- |
| db.表名.drop() |

如：

删除user表



### 数据操作

#### 插入数据

* 插入数据

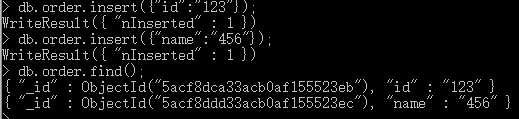
|  |
| --- |
| db.表名.insert(JSON格式数据) |

如：

向user表中插入1条数据：name=shuyan，age=25



* 注意
  + MongoDB同一张表中的字段可以不是同一的

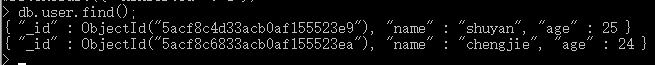


#### 查看数据

##### 查询所有数据

|  |
| --- |
| db.表名.find() |

如：



##### 查找key=value的数据

|  |
| --- |
| db.表名.find({key:value}) |

如：

查看age=24的数据



##### 查找key>value的数据

|  |
| --- |
| db.表名.find({key:{$gt:value}}) |

如：

查看age>24的数据

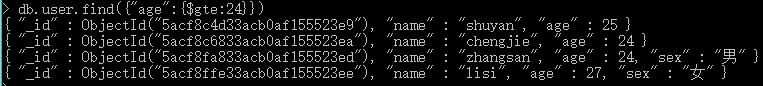


##### 查看key>=value的数据

|  |
| --- |
| db.表名.find(key:{$gte:value}) |

如：

查看age>=24的数据



##### 查看key<value的数据

|  |
| --- |
| db.表名.find(key:{$lt:value}) |

如：

查看age<24的数据

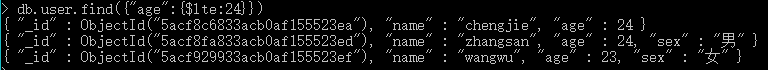


##### 查看key<=value的数据

|  |
| --- |
| db.表名.find(key:{$lte:value}) |

如：

查看age<=24的数据



##### 查看key在value0和value1内的数据

|  |
| --- |
| db.表名.find(key:{$lte:value0, $gte:value1 }) |

如：

查看age<27并且age>24的数据



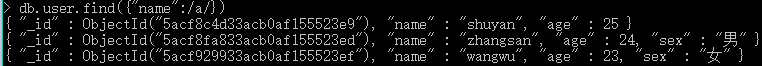
##### 模糊查询

###### 查找key字段中包含value的数据

|  |
| --- |
| db.表名.find({key:/value/}) |

如：

查找name字段中包含a的数据



###### 查找key字段以value开头的数据

|  |
| --- |
| db.表名.find({key:/^value/}) |

如：

查找name字段以s开头的数据

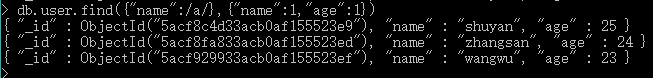


##### 将查询结果只显示指定列

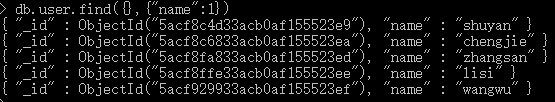
|  |
| --- |
| db.表名.find({条件},{key0:1,key1:1}) |

如：

查询name字段以包含a的数据，并将查询结果只显示name和age字段



查询所有数据，只显示name字段



##### 排序查询

升序：

按key字段升序排序

|  |
| --- |
| db.表名.find().sort({key:1}) |

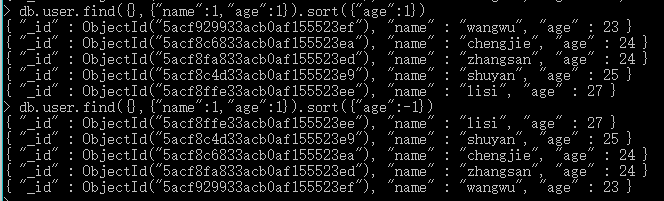
降序：

按key字段降序排序

|  |
| --- |
| db.表名.find().sort({key:-1}) |

如：

将所有数据只显示name和age字段，并按age升序和降序分别排序



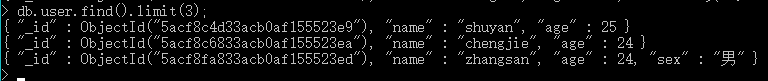
##### 分页查询

###### 查询前n条数据

|  |
| --- |
| db.表名.find().limit(n) |

如：

查询前3条数据



###### 查询第n条以后的数据

第n条：n从1开始

以后的数据：不包含第n条数据

|  |
| --- |
| db.表名.find().skip(n) |

如：



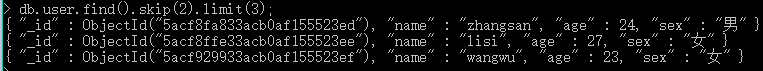
###### 查询第m-n条之间的数据

不包含第m条，包含第n条

|  |
| --- |
| db.表名.find().skip(m).limit(n-m) |

如：

显示第3条到第5条之间的数据，包含第3和5条



##### or或查询

查询满足条件1或条件2的数据

|  |
| --- |
| db.表名.find({$or:[条件1,条件2]}). |

如：

查询age=23或age=25的数据

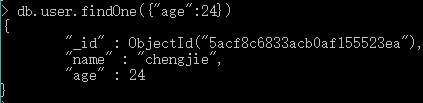


##### 查询第一条数据

|  |
| --- |
| db.表名.findOne() |

如：

查找age=24的第一条数据



##### 统计查询结果数量

|  |
| --- |
| db.表名.find(条件).count() |

如：

查找age=24的有几个人



#### 修改数据

修改数据使用如下语句修改

|  |
| --- |
| db.表名.update({条件},{新数据},upsert,multi) |

参数说明：

* + 条件：

修改数据只会修改符合条件的查询结果中的数据，在这里指定查询条件

* + 新数据

在这里指定日和修改数据

* + - 根据字段修改

|  |  |
| --- | --- |
| 名称 | 描述 |
| $inc | 将指定的值加上原字段值作为新值 |
| $mul | 将指定的值乘以原字段值作为新值 |
| $rename | 重命名字段 |
| $setOnInsert | 操作时,操作给相应的字段赋值 |
| $set | 用来指定一个键的值，如果不存在则创建它 |
| $unset | 用来指定一个键的值，如果不存在不创建创建它 |
| $min | 只有当现有字段值大于指定的值时才更新该字段。 |
| $max | 只有当现有字段值小于指定的值时才更新该字段。 |
| $currentDate | 设置当前日期字段的值，或者作为一个日期或时间戳。 |

* + - 整条替换

整条修改指的是不修改某个字段，而是用新数据完整替换现有数据

此时只需要将新数据作为参数即可，不需要使用上表中内容

* + upsert

boolean类型，默认false

用来指定如果没有匹配项是否插入新数据

* + multi

boolean类型，默认false

用来指定如果有多个匹配结果时，是否全部进行更新

false：只修改第一条，true：全部更新

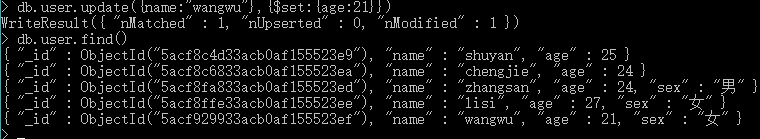
##### 修改字段的值

###### 直接指定新值

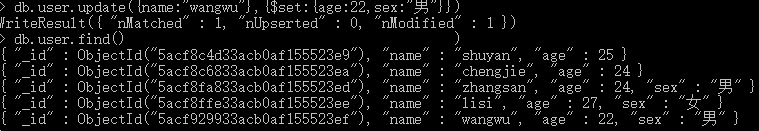
|  |
| --- |
| db.表名.update({条件},{$set:{新字段}}) |

如：

将wangwu的age改为21



将wangwu的age改为22，sex改为男



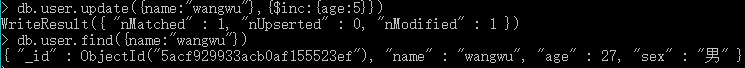
结论：使用同种方式修改多个字段时，可以将多个新字段放在同一个json中

###### 将字段加上指定值

|  |
| --- |
| db.表名.update({条件},{$set:{字段:指定值}}) |

如：

将wangwu的age加5

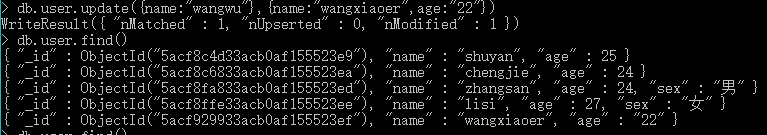


##### 整条替换

|  |
| --- |
| db.表名.update({条件},{新记录}) |

如：

将wangwu的所有数据替换为wangxiaoer的所有数据



##### 更新所有匹配项

update()默认只更新匹配的第一条记录，如果想更新所有匹配项，需要指定multi参数为true

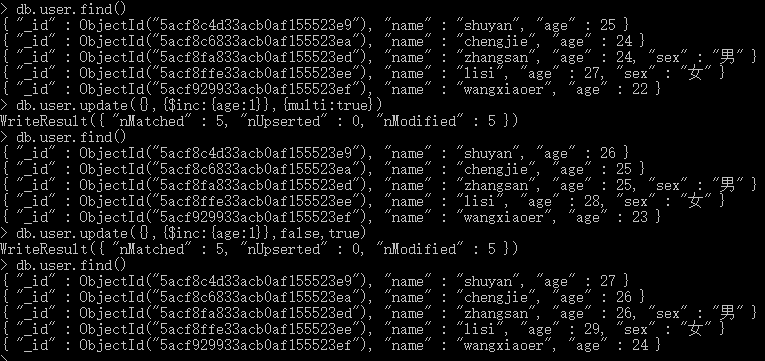
|  |
| --- |
| db.表名.update({条件},{新数据},{multi:true}) |

或

|  |
| --- |
| db.表名.update({条件},{新数据},false,true) |

如：

将所有人年龄+1



#### 删除数据

##### 删除所有匹配项

|  |
| --- |
| db.表名.remove({条件}) |

如：

删除所有name=shuyan的人



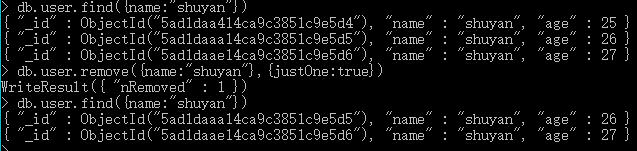
##### 删除第1个匹配项

remove默认删除所有匹配项，需要只删除第一个匹配项时，则将justOne参数指定为true

|  |
| --- |
| db.表名.remove({条件},{justOne:true}) |

如：

删除第一个name=shuyan的人

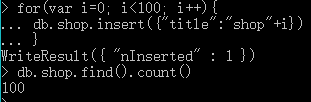


### 批量操作数据

当需要有规律的插入很多数据时，可以结合js语句和mongo的sql语句执行批量操作

如：

shop表中插入100条记录



### 查看语句执行时间

#### 查询方法

在执行语句后加上如下内容

|  |
| --- |
| .explain( "executionStats") |

查询结果中的executionTimeMillis的值就是执行时间，单位毫秒

#### 例

查寻name=shuyan的记录的执行时间



可见查询时间为0ms

### 索引

#### 介绍

索引是对数据库表中一列或多列的值进行排序的一种结构，可以让我们查询数据库变得更快索引基础

#### 索引基础

##### 创建索引

###### 创建索引

我们可以对表中的某列设置使用索引

|  |
| --- |
| db.表名.ensureIndex({字段名:索引方式}) |

索引方式：

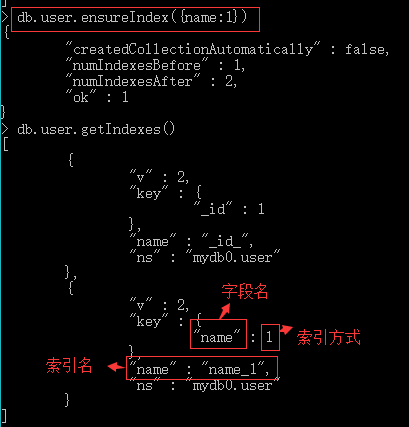
1：升序索引

-1:降序索引

表中的同一个字段可以同时存在升序索引和降序索引

MongoDB 会根据索引的 keyname 和索引方向为新创建的索引自动分配一个索引名

如：



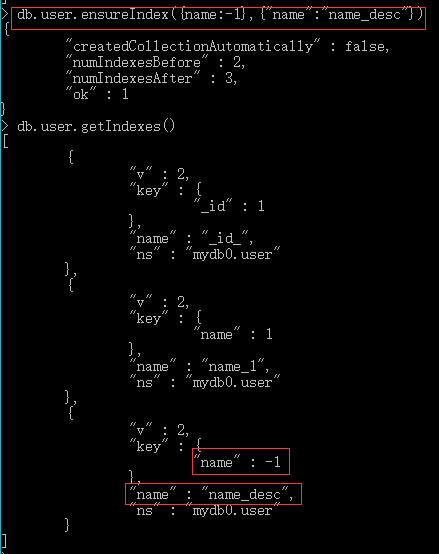
###### 指定索引名创建索引

下面的命令可以指定索引名创建索引

|  |
| --- |
| db.表名.ensureIndex({字段名:索引方式},{“name”:索引名}) |

如：

为user表的name字段创建降序索引，指定索引名为name\_desc

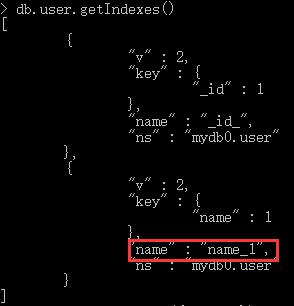


##### 查看索引

我们可以查看当前表下的所有设置的索引

|  |
| --- |
| db.表名.getIndexes() |

如：



##### 删除索引

|  |
| --- |
| db.表名.dropIndex({字段名:索引方式}) |

如：

删除user表中name字段的降序索引



##### 复合索引

###### 介绍

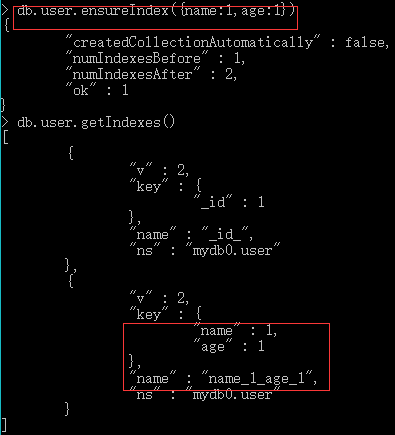
创建索引时，可以使用多个字段创建1个索引

|  |
| --- |
| db.表名.ensureIndex({"字段1":索引方式1, "字段2":索引方式2}) |

该索引被创建后，基于字段1和字段2的查询将会用到该索引，或者是基于字段1的查询也会用到该索引，但是只是基于字段2的查询将不会用到该复合索引。因此可以说，如果想用到复合索引，必须在查询条件中包含复合索引中的前N 个索引列。

然而如果查询条件中的键值顺序和复合索引中的创建顺序不一致的话，MongoDB 可以智能的帮助我们调整该顺序，以便使复合索引可以为查询所用。

###### 例



##### 唯一索引

###### 介绍

唯一索引指定的字段是不可重复的，插入的记录的唯一索引字段值如果与已经存在的记录重复，则会报错，插入失败

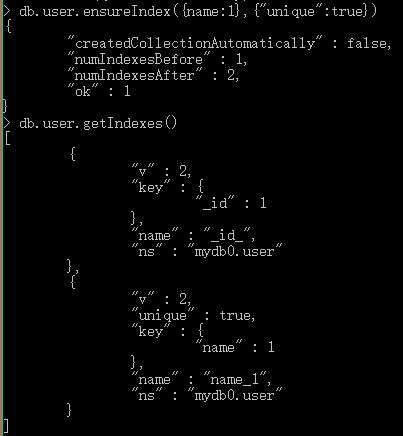
对已经存在数据的表设置唯一索引时，如果唯一索引字段已经存在重复时，则设置索引将失败

默认情况下创建的索引均不是唯一索引，创建唯一索引需要将unique属性指定为true

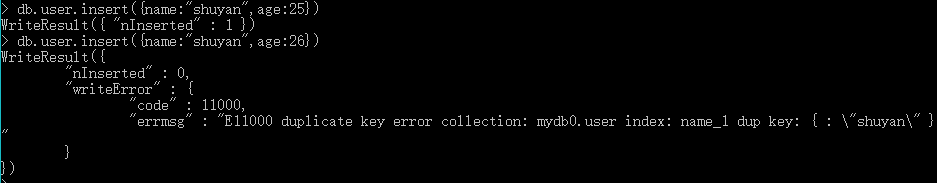
|  |
| --- |
| db.表名.ensureIndex({字段名:索引方式},{unique:true}) |

###### 例

设置name字段为唯一索引



插入两条name字段的值相同的数据



第一条数据插入成功，第二条数据插入失败

## Nodejs操作MongoDB

### 安装mongodb模块

* + 要在nodejs项目中使用mongodb，就需要在项目中安装mongodb模块

在项目根目录下执行如下命令：

|  |
| --- |
| npm install mongodb --save |

* + 注意

mongodb模块的2.x版本和3.x版本在连接数据库时存在差异，参见[连接数据库](#_连接数据库)

如果想使用指定版本的mongodb模块，使用如下方法进行切换：

* + - * mongodb模块安装之后默认3.x版本
      * 如果想切换版本，使用如下方法切换

将package.json中mongodb模块的版本修改为2.2.33或3.0.6，然后执行npm install

### 连接数据库

mongodb2.x与3.x之间只有连接数据库的方法存在差异

#### mongodb 2.x

|  |
| --- |
| **let httpModule** = *require*(**'http'**); **let** *app* = *require*(**'./modules/expressRouter'**); *//获取数据库的客户端引用* **let** MongoClient = *require*(**"mongodb"**).**MongoClient**; *//定义数据库的地址* **let** databaseUrl = **'mongodb://localhost:27017/mydb0'**;  **httpModule**.*createServer*(*app*).listen(80);  *app*.get(**'/'**,(req,res)=>{  *//连接数据库* MongoClient.connect(databaseUrl,(err,db)=>{  **if**(err)  **return**;  *//到这里，连接数据库成功* res. writeHead(200,{ **"Content-Type"**: **"text/html;charset=UTF8"**});  res.end(**'数据库连接成功'**);  *//接下来，执行数据库操作  //最后关闭数据库* db.close();  }) }); |

#### mongodb 3.x

|  |
| --- |
| **let httpModule** = *require*(**'http'**); **let** *app* = *require*(**'./modules/expressRouter'**); *//获取数据库的客户端引用* **let** MongoClient = *require*(**"mongodb"**).**MongoClient**; *//定义数据库的地址* **let** databaseUrl = **'mongodb://localhost:27017/'**; *//定义眼连接的数据库名称* **let** databaseName = **"mydb0"**;  **httpModule**.*createServer*(*app*).listen(80);  *app*.get(**'/'**,(req,res)=>{  *//连接指定url上的mongodb* MongoClient.connect(databaseUrl,(err,client)=>{  **if**(err)  **return**;  *//连接上面获得的mongodb上的databaseName数据库，并获取db对象* **let** db = client.db(databaseName);  *//执行数据库操作  //关闭数据库时，要关闭client对象* client.close();  }) }); |

### 增加数据

|  |
| --- |
| MongoClient.connect(databaseUrl,(err,db)=>{  **if**(err)  **return**;  *//插入数据* db.**collection**(**"user"**).insertOne({**'name'**:**'shuyan'**,**'age'**:25},(error,result)=>{  **if**(error)  {  db.close();  **return**;  } res. writeHead(200,{ **"Content-Type"**: **"text/html;charset=UTF8"**});  res.end(**'数据插入成功'**);  *//最后关闭数据库* db.close();  }) }) |

### 查询数据

|  |
| --- |
| MongoClient.connect(databaseUrl,(err,db)=>{  **if**(err)  **return**;  *//查询数据* **let** result = db.**collection**(**"user"**).find({**'name'**:**'shuyan'**});  *//定义数组用来存储查询结果* **let** list = [];  *//使用each方法逐条获取数据* result.each((error,doc)=>{  **if**(!error){  **if**(doc != **null**){ *//有数据时将数据放进数组* list.push(doc);  }**else**{ *//doc=null代表数据遍历结束* ***console***.log(list);  }  }  }) }) |

### 修改数据

|  |
| --- |
| MongoClient.connect(databaseUrl,(err,db)=>{  **if**(err)  **return**;  *//插入数据* db.**collection**(**"user"**).updateOne({**'name'**:**'shuyan'**},{**$set**:{**age**:30}},(error,result)=>{  **if**(error)  {  db.close();  **return**;  }  res. writeHead(200,{ **"Content-Type"**: **"text/html;charset=UTF8"**});  res.end(**'数据修改成功'**);  }) }) |

### 删除数据

|  |
| --- |
| MongoClient.connect(databaseUrl,(err,db)=>{  **if**(err)  **return**;  *//插入数据* db.**collection**(**"user"**).deleteOne({**'name'**:**'shuyan'**},(error,result)=>{  **if**(error)  {  db.close();  **return**;  }  res. writeHead(200,{ **"Content-Type"**: **"text/html;charset=UTF8"**});  res.end(**'数据删除成功'**);  }) }) |

# KOA

## 环境搭建

### 安装模块

在工程目录下执行：

|  |
| --- |
| npm install koa --save |

### 入门程序

|  |
| --- |
| **let** koaModule = *require*(**'koa'**);  **let** app = **new** koaModule();  *//向页面输出内容* app.use(**async** (ctx)=>{  ctx.**body**=**'你好 koa2'**; });  app.listen(80); |

## koa路由

### 安装路由模块

express中自带路由模块，但是KOA中需要额外安装路由模块

项目根路径下执行：

|  |
| --- |
| npm install koa-router --save |

### 入门程序

|  |
| --- |
| *//引入并实例化koa模块* **const Koa** = *require*(**'koa'**); **let** app = **new Koa**(); *//引入并实例化koa路由* **const** Koa\_router = *require*(**'koa-router'**); **let** router =**new** Koa\_router();  *//配置路由 /\* \* 1. 路径为‘/’的get请求的请求处理方法 \* 2. ctx:context对象，包含了request、response等信息 \* 3. ctx.body:向页面输出内容  \*/* router.get(**'/'**,**async** (ctx)=>{  ctx.**body**=**'首页'**; });  *//router.get与在上1个get()的后面接着get()是一样的* router.get(**'/news'**,**async** (ctx)=>{  ctx.**body**=**'新闻'**; }).get(**'/books'**,**async** (ctx)=>{  ctx.**body**=**'书城'**; });  *//启动路由* app.use(router.routes()); *//根据ctx.status设置response响应头* app.use(router.allowedMethods());  *//监听端口* app.listen(8001); |

### get、post等函数的参数

* 第1个参数

该参数是该路由匹配的URL

* 中间的参数

中间可以使用任意多个中间件

* 最后1个参数

最后1个参数本质上也是1个中间件，但是从概念上来说，他是1个controller

### 动态路由

#### 概念

将URL中的某/后面的内容作为参数在程序中获取，而不是将URL整体作为路由的匹配路径，这样的路由配置方式成为动态路由

#### 用法

|  |
| --- |
| router.get(**'/books/:aid'**,**async** (ctx)=>{}); |

* 该路由只能匹配“/books/xxxx”的URL，不能匹配“/books”或“/books/xxxx/xxxx”这样的URL
* 其中aid只是1个标识，该处可以使用其他名字
* 动态路由可以配置为多个参数的，如

|  |
| --- |
| router.get(**'/books/:aid/:cid'**,**async** (ctx)=>{}); |

但是格式必须完全一致才能匹配，多1个或少1个参数时无法匹配到。

* 例：

|  |
| --- |
| router.get(**'/books/:aid/:cid'**,**async** (ctx)=>{}); |

访问<http://localhost:8001/books/nodejs/1>时可以匹配到该路由

访问<http://localhost:8001/books/nodejs>时无法匹配到该路由

访问<http://localhost:8001/books/nodejs/1/2>时无法匹配到该路由

#### 获取动态路由参数

使用ctx.params就可以获取动态路由参数

|  |
| --- |
| router.get(**'/books/:aid/:cid'**,**async** (ctx)=>{  ***console***.log(ctx.**params**); }); |

访问：<http://localhost:8001/books/nodejs/1>

控制台打印：



## 请求方式

### 请求方式与传值方式

携带的信息可以放在HTTP协议的如下3个地方：

URL、header、body

理论上每种请求方式都可以获取这3种传值方式的传值，但是由于一些行业规范原因，每种方式可以获取传值的传值方式不同

不同的请求方式主要是根据功能划分的，与传值方式没有直接关系

### 请求方式分类

#### get

* 常用于从服务器获取资源
* 对应“查”操作
* 可以获取URL、header中的传值

#### post

* 常用于在服务器新建1个资源
* 对应“增”操作
* 可以获取URL、header、body中的传值

#### put

* 常用于在更新服务器的资源
* 对应“改”操作
* 可以获取URL、header、body中的传值

#### delete

* 常用于从服务器删除资源
* 对应“删”操作
* 可以获取URL、header中的传值

### 传值方式分类

#### 获取URL传值

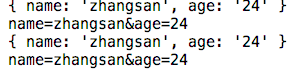
* koa中获取url传值通过如下两种方法获取：
* query属性

返回的是格式化好的参数对象

* querystring属性

返回的是请求字符串

* 上述两个属性在ctx对象和ctx.request对象中均有，用哪个对象中的属性都可以



#### 获取header传值

* 通过ctx.header获取header中的传值

#### 获取body传值

获取body传值需要使用koa-bodyparser模块

##### 安装模块

|  |
| --- |
| npm install koa-bodyparser --save |

##### 使用

* 引入模块

|  |
| --- |
| **const** Bodyparser = *require*(**'koa-bodyparser'**); |

* 加载bodyparser中间件
  + 使用默认参数加载

|  |
| --- |
| app.use(Bodyparser()); |

* 自定义参数加载

|  |
| --- |
| app.use(bodyparser({  **enableType**:[**'form'**,**'json'**,**'text'**],  **encode**:**'utf-8'**,  **formLimit**:**'56kb'**,  **jsonLimit**:**'1mb'**,  **textLimit**:**'1mb'**,  **strict**:**true**,  detectJSON:(ctx)=>{  **return** /\.json$/i.test(ctx.**path**);  },  **extendTypes**:{  **json**:[**'application/x-javascript'**]  },  onerror:(err,ctx)=>{  ctx.throw(**'body parser error'**,422);  },  **disableBodyParser**:**false** })); |

* 参数
  + enableType

只有请求类型是该类型中设置的类型时，bodyParser才能进行解析

可设值：“json”、“form”、“text”

默认值：[“json”，“form”]

* + encode

使用什么编码解析，默认utf-8，一般不需要修改此项

* + formLimit

可解析form表单的最大值，默认56kb，当form表单大小超过该值时，返回413错误

* + jsonLimit

可解析json数据的最大值，默认1mb，当json数据大小超过该值时，返回413错误

* + textLimit

可解析文本数据的最大值，默认56kb，当文本数据大小超过该值时，返回413错误

* + strict

如果设置为true，只能解析对象和数组，ctx.request.body也始终是1个对象或数组，这样能避免大量类型判断，如果body中是text，则始终返回字符串

默认为true

一般不需要修改此项

* + detectJson

自定义检查本次请求是否是json数据的函数

默认null

一般不需要设置此项

* + extendTypes

根据Content-Type请求头决定当做什么数据解析

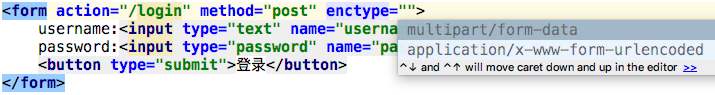
* + - * Content-Type请求头

使用body传输数据时，请求头中会有Content-Type字段



如：

form表单提交数据时，可以设置encType属性，该属性值就是请求头中Content-Type字段的值



* + - * 该属性作用

|  |
| --- |
| **extendTypes**:{  **json**:[**'****application/x-javascript'**],  **text**: [**'****application/xml'**, **'text/xml'**, **'text/plain'**] } |

上例中表示：

Content-Type请求头为“application/x-javascript”时，当做json数据解析

Content-Type请求头为“application/xml”、“ text/xml”或“text/plain”时，当做text数据解析

* + - * 默认值

各种类型数据的默认值如下：

|  |
| --- |
| *// default json types* **var** jsonTypes = [  **'application/json'**,  **'application/json-patch+json'**,  **'application/vnd.api+json'**,  **'application/csp-report'**, ];  *// default form types* **var** formTypes = [  **'application/x-www-form-urlencoded'**, ];  *// default text types* **var** textTypes = [  **'text/plain'**, ]; |

* + - * 注意

自己设置的该属性只是在默认值基础上追加，而不是替换

* + onerror

自定义解析出错时的回调函数

* + disableBodyParser

动态开启关闭解析功能的设置项

* 通过ctx.request.body获取body传值

ctx.request.body返回的是转换成对象的body传值

|  |
| --- |
| ***console***.log(ctx.**request**.**body**); |

* 通过ctx.request.rawBody获取原始body的内容

只有在使用bodyparser解析后才能通过该属性获得

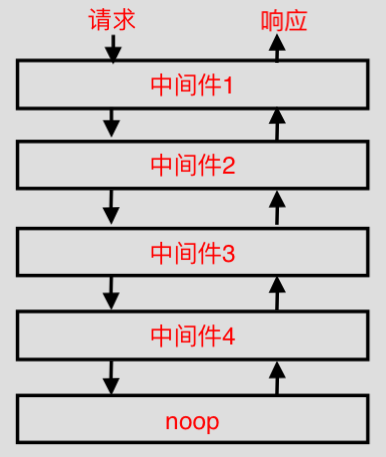
#### 例

|  |
| --- |
| **const Koa** = *require*(**'koa'**); **const** Koa\_router = *require*(**'koa-router'**); *//引入bodyparser* **const** Bodyparser = *require*(**'koa-bodyparser'**); **const** *Static* = *require*(**'koa-static'**);  **let** app = **new Koa**(); **let** router =**new** Koa\_router(); *//加载bodyparser中间件* app.use(Bodyparser()); app.use(*Static*(**'views'**));  router.post(**'/login'**,**async** (ctx,next)=>{  ***console***.log(ctx.**query**); *//url传值* ***console***.log(ctx.**header**); *//header传值* ***console***.log(ctx.**request**.**body**); *//body传值* **await** next(); });  app.use(router.routes()); app.use(router.allowedMethods()); app.listen(8001); |

## 中间件

### 什么叫中间件

* KOA2的整个框架就是由中间件串联组成



[入门程序](#入门程序)中的router.routes()和router.allowedMethods()也是中间件

### 功能

* 执行任意代码
* 修改请求和响应对象
* 终结请求-响应循环
* 调用堆栈中下一个中间件

### 使用

* 中间件统一使用app.use()函数加载
* 多个中间件会形成1个中间件栈，具备后进先出的特点
* 中间件函数默认接受2个参数：context对象、next函数

当在中间件中需要放行继续执行下面的中间件时，就调用next方法放行，

### 分类

#### 应用级中间件

##### 介绍

在匹配路由之前使用的中间件，叫做应用级中间件

##### 例

|  |
| --- |
| **const Koa** = *require*(**'koa'**); **const** Koa\_router = *require*(**'koa-router'**);  **let** app = **new Koa**(); **let** router =**new** Koa\_router();  *//定义1个中间件* **async function** *DateLog*(ctx,next){  ***console***.log(**new *Date***()); *//打印时间* **await** next(); *//继续执行下一个中间件* }  router.get(**'/'**,**async** (ctx)=>{  ctx.**body**=**'首页'**; });  router.get(**'/news'**,**async** (ctx)=>{  ctx.**body**=**'新闻'**; });  *//加载DateLog中间件* app.use(*DateLog*); app.use(router.routes()); app.use(router.allowedMethods()); app.listen(8001); |

上述代码中，DateLog函数就是1个中间件

在加载路由之前加载它，它就是1个应用级中间件

#### 路由级中间件

路由的配置按[路由](#路由)中的方法进行配置，其中配置路由时，get、post等方法的回调函数就是1个路由级中间件

#### 错误处理中间件

用于处理错误的中间件叫做错误处理中间件

#### 第三方中间件

koa-bodyparser、koa-static等都属于第三方中间件

## EJS模板引擎

### 安装模块

需要安装2个模块

|  |
| --- |
| npm install koa-views --save  npm install ejs --save |

### 使用

#### 步骤

* 引入koa-views模板

|  |
| --- |
| **const** *Views* = *require*(**'koa-views'**); |

* 应用ejs模板引擎

应用ejs模板引擎有2种方式

* + 按如下方式应用，模板必须为html文件

|  |
| --- |
| app.use(*Views*(**'views'**, { **map**: {**html**: **'ejs'** }})); |

* + 按如下方式应用，模板后缀名可以指定

|  |
| --- |
| app.use(*Views*(**'views'**, { **extension**: **'ejs'**})); |

上述两种方式中：

Views为引入的模板名称

views字符串为工程目录下存放模板的文件夹名称

我们使用第2种

* koa中使用模板

|  |
| --- |
| router.get(**'/'**,**async** (ctx)=>{  **await** ctx.render(**'index'**,{**msg**:**'世界'**}); }); |

* + 必须使用await，因为这是异步方法
  + 第1个参数：模板文件名
  + 第2个参数：传入模板的对象

注意：如果某些数据放在了ctx.state中，则不需要在这里传值，就可以在ejs模板中直接使用

#### 例

* index.ejs

|  |
| --- |
| <!DOCTYPE **HTML** PUBLIC **"-//W3C//DTD HTML 4.01 Transitional//EN"  "http://www.w3.org/TR/html4/loose.dtd"**> <**html**> <**head**>  <**title**>$Title$</**title**> </**head**> <**body**>  <**h1**>你好 <%=msg%></**h1**>  </**body**> </**html**> |

* app.js

|  |
| --- |
| **const Koa** = *require*(**'koa'**); **const** Koa\_router = *require*(**'koa-router'**); *//引入koa-views模板* **const** *Views* = *require*(**'koa-views'**);   **let** app = **new Koa**(); **let** router =**new** Koa\_router();  *//应用ejs模板引擎* app.use(*Views*(**'views'**, { **extension**: **'ejs'**}));  router.get(**'/'**,**async** (ctx)=>{  *//使用模板* **await** ctx.render(**'index'**,{**msg**:**'世界'**});  });  app.use(router.routes()); app.use(router.allowedMethods()); app.listen(8001); |

* 结果



## 静态资源

### 介绍

当浏览器需要访问服务器的静态资源时，需要使用到koa-static

koa-static用于浏览器访问服务器的静态资源时，可以将浏览器的静态资源返回

### koa-static

#### 安装

|  |
| --- |
| npm install koa-static --save |

#### 使用

* 引入模块

|  |
| --- |
| **const** *Static* = *require*(**'koa-static'**); |

* 加载koa-static中间件

|  |
| --- |
| app.use(*Static*(**'views'**)); |

其中：views为项目根路径下需要暴露出的静态资源所在的文件夹名称

#### 注意

* koa-static中间件应该放在加载路由中间件之前
* 如果在静态资源中间件中匹配到对应的资源，则将资源返回，不再执行下面的中间件；如果匹配不到，继续执行下面的中间件

## cookie

### 方法

#### 设置cookie

格式：ctx.cookies.set(key,value,{options})

参数： key：设置的cookie的key

value：设置的cookie的value

options：该参数是1个对象，代表这个cookie的设置项，该参数是缺省的

|  |  |
| --- | --- |
| option | 说明 |
| maxAge | 多少毫秒后销毁这个cookie |
| expires | 到什么时间销毁这个cookie |
| path | 访问哪个路径下的url时携带这个cookie，默认‘/’ |
| domain | 设置访问哪些域名时携带这个cookie  比如news.baidu.com设置的cookie想在sports.baidu.com的服务器使用，则需要设置该项 |
| secure | true：只有https协议可以使用该cookie  false：http协议和https协议均可使用该cookie（默认） |
| httponly | true：只有服务器端可以使用该cookie  false：服务器端可以使用该cookie，客户端页面上也可以使用js获取该cookie |

#### 获取cookie

格式：ctx.cookies.get(key)

参数： key：要获取的cookie的key

### 例

|  |
| --- |
| router.get(**'/'**,**async** (ctx,next)=>{  ctx.**body**=**'首页'**;  ctx.**cookies**.set(**'name'**,**'shuyan'**,{**maxAge**:1000\*3600});  **await** next(); });  router.get(**'/news'**,**async** (ctx,next)=>{  ctx.**body**=**'新闻'**;  ***console***.log(ctx.**cookies**.get(**'username'**));  **await** next(); }); |

### cookie中使用中文

#### 介绍

koa的cookie中不可以使用中文，否则运行报错；解决该问题采用如下方法：

将中文转换为base64字符传入cookie

获取cookie值得时候将base64字符再转换为中文

#### 中文与base64互转方法

* 中文转base64

|  |
| --- |
| **var** base64 = **new** Buffer(**'中文'**).toString(**'base64'**); |

* base64转中文

|  |
| --- |
| **var** Str = **new** Buffer(**'baseStr'**,**'base64'**).toString(); |

其中：

baseStr：中文转换得到的base64字符

base64：指定当前字符的字符格式

## session

### 安装koa-session模块

|  |
| --- |
| npm install koa-session --save |

### 使用

#### 步骤

* 引入模块

|  |
| --- |
| **const** Koa\_session = *require*(**'koa-session'**); |

* 加载koa-session中间件

|  |
| --- |
| app.**keys** = [**'some secret hurr'**]; **const** CONFIG = {  **key**: **'koa:sess'**,  **maxAge**: 86400000,  **overwrite**: **true**,  **httpOnly**: **true**,  **signed**: **true**,  **rolling**: **false**,  **renew**: **false**, }; app.use(Koa\_session(CONFIG, app)); |

以下内容是可以修改的字段，其他保持跟模板相同即可：

* + maxAge

设置过期时间，单位毫秒

* + httpOnly

true：只有服务器可以访问cookie（seesion的key）

false：浏览器的js代码中也可以访问cookie（session的key）

* + rolling

true：每次请求时，重置到期时间

false：不会重置到期时间

* + renew

true：session快过期时，产生1次请求则会重置过期时间

false：不会重置过期时间

* 使用

设置session：

ctx.session.key = value

获取session：

var value = ctx.session.key

|  |
| --- |
| router.get(**'/'**,**async** (ctx,next)=>{  ctx.**body**=**'首页'**;  ctx.**session**.**username** = **'shuyan'**;  **await** next(); });  router.get(**'/news'**,**async** (ctx,next)=>{  ctx.**body**=**'新闻'**;  ***console***.log(ctx.**session**.**username**);  **await** next(); }); |

## 文件上传下载

### 文件上传

文件上传需要使用koa-multer模块

#### 安装模块

|  |
| --- |
| npm install koa-multer --save |

#### 使用

##### 步骤

* 引入模块

|  |
| --- |
| **const** *koa\_multer* = *require*(**'koa-multer'**); |

* 指定上传目录获取中间件

|  |
| --- |
| **let** upload = *koa\_multer*({ **dest**: **'uploads'**}); |

其中：dest的属性值是上传文件在服务器的保存文件夹

* 指定上传文件的name属性加载中间件，实现文件上传

|  |
| --- |
| router.post(**'/upload'**,upload.single(**'file'**), **async** (ctx, next)=>{  **return** ctx.**body** = ctx.**req**.**file**; }); |

其中：single函数的参数就是需要保存的上传文件的name属性

* 获取上传文件信息

koa-multer模块将该模块获取到的上传文件的信息保存在了ctx.req对象上，其中ctx.req.file就是该上传文件的相关信息

##### 例

* index.html

|  |
| --- |
| <**form action="/upload" method="post" enctype="multipart/form-data"**>  <**input type="file" name="file1"**><**br**>  <**button type="submit"**>上传</**button**> </**form**> |

注意：使用了文件上传的form表单必须设置enctype属性

* app.js

|  |
| --- |
| **const Koa** = *require*(**'koa'**); **const** Koa\_router = *require*(**'koa-router'**); **const** bodyparser = *require*(**'koa-bodyparser'**); **const** *koa\_static* = *require*(**'koa-static'**); *//引入模块* **const** *koa\_multer* = *require*(**'koa-multer'**);  **let** app = **new Koa**(); **let** router =**new** Koa\_router(); *//指定上传目录获取中间件* **let** upload = *koa\_multer*({ **dest**: **'uploads'**});  app.use(bodyparser()); app.use(*koa\_static*(**'views'**));  *//路由中指定上传文件name属性实现文件上传* router.post(**'/upload'**,upload.single(**'file1'**), **async** (ctx, next)=>{  *//将上传文件信息打印在页面* **return** ctx.**body** = ctx.**req**.**file**; });  app.use(router.routes()); app.use(router.allowedMethods()); app.listen(8001); |

### 文件下载

文件下载分为2种：大文件下载，小文件下载

小文件下载时，只需要将文件使用静态文件托管即可

大文件上传时，需要使用koa-send模块

#### 模块安装

|  |
| --- |
| npm install koa-send --save |

#### 步骤

* 引入模块

|  |
| --- |
| **const** *koa\_send* = *require*(**'koa-send'**); |

* 设置响应头，标识文件下载

|  |
| --- |
| ctx.attachment(path); |

* + 作用

设置header中的Content-Disposition为attachment，来告诉客户端这是1个下载操作

* + path

path是文件在服务器的完整路径

* 发送文件

|  |
| --- |
| **await** *koa\_send*(ctx, path); |

* + ctx

koa中的context对象

* + path

path是文件在服务器的完整路径

#### 例

|  |
| --- |
| **const** Koa\_router = *require*(**'koa-router'**); *//引入模块* **const** *koa\_send* = *require*(**'koa-send'**);  **let** app = **new Koa**(); **let** router =**new** Koa\_router();  router.get(**'/download/:name'**,**async** (ctx,next)=>{  **let** path = **`uploads/**${ctx.**params**.**name**}**`**;  *//设置header中的Content-Disposition为attachment，来告诉客户端这是1个下载操作* ctx.attachment(path);  *//发送文件* **await** *koa\_send*(ctx, path);  **return**; });  app.use(router.routes()); app.use(router.allowedMethods()); app.listen(8001); |

## 跨域

### 基础知识

#### 什么是跨域

每个服务器都有1个唯一的域名，域名由以下几部分组成：

协议、域名、端口

如：http:www.baidu.com:80

当域名中存在任何一部分不同时，则认为是跨域

如：来自[www.baodu.com](http://www.baodu.com)的页面要请求[www.google.com](http://www.google.com)的资源，就是跨域

#### 浏览器如何进行跨域限制

跨域限制仅限于ajax请求

默认情况下，浏览器从互联网上请求下来的js脚本，都会记住他来自哪个域名，当某个js脚本想要发送与他的来源不同域的请求时，浏览器会先判断这是1个简单请求还是非简单请求，如果这是1个简单请求，则会自动添加1个origin请求头直接发送出去；如果是非简单请求，则会自动添加一些请求头先预检一下服务器是否允许当前域的请求，如果服务器允许，则真正的将请求发出去，否则就报错（参见[CORS请求](#CORS请求)）

举个🌰：

现在浏览器上运行的js脚本是来自A网站的，但是该页面向B站发送了1个ajax请求，浏览器一看，你这是跨域了，就会先判断该请求是否属于简单请求，如果是，则自动添加origin请求头直接将请求发出，如果不是，则先发一个“预检”请求，如果浏览器回复允许你访问，则真正的发出请求，否则就不让发了

#### 为什么要跨域

有的时候同一个公司可能有好几个子域名，他们之间需要跨域互相访问资源

当你的项目需要使用百度地图的api时，需要跨域让浏览器允许你的页面发送对百度地图所在的域名的请求

### 跨域方案

#### jsonp

现在没用到

#### 服务器代理

浏览器中来自A网站的资源想要请求B网站的资源时，浏览器中来自A网站的资源先请求A网站，告诉他我需要B网站的资源，A网站服务器接收到请求后由服务器去B网站拿资源，拿到后再进行响应

#### CORS请求

* 浏览器将CORS请求分成两类：简单请求（simple request）和非简单请求（not-so-simple request）。
* 只要同时满足以下两大条件，就属于简单请求。
  + 请求方法是以下三种方法之一：
* HEAD
* GET
* POST
  + HTTP的头信息不超出以下几种字段：
    - Accept
    - Accept-Language
    - Content-Language
    - Last-Event-ID
    - Content-Type：只限于三个值application/x-www-form-urlencoded、multipart/form-data、text/plain
* 凡是不同时满足上面两个条件，就属于非简单请求。
* 浏览器对这两种请求的处理，是不一样的。

##### 简单请求

* 对于简单请求，浏览器自动添加1个origin请求头，并将请求直接发出

例：

* localhost:8001中的123.html

|  |
| --- |
| <**body**>  <**div id="div1"**></**div**> </**body**> </**html**> <**script type="text/javascript"**>  **var *url*** = **'http://localhost:8002'**;  **var *xhr*** = **new** XMLHttpRequest();  ***xhr***.onreadystatechange = **function**(){  **if** ((***xhr***.**readyState** == 4) && (***xhr***.**status** == 200)){  document.getElementById(**'div1'**).**innerHTML** += ***xhr***.responseText;  }  };  ***xhr***.onerror = **function**(){  document.getElementById(**'div1'**).**innerHTML** += **'request error'**;  };  ***xhr***.open(**'GET'**, ***url***, **true**);  ***xhr***.**send**(**null**); </**script**> |

* cors简单请求的请求头

刷新123.html页面就会发出cors请求，在控制台看到请求头如下：



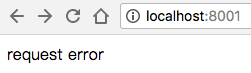
* 服务器接收到请求后，根据请求头中Origin字段值判断是否同意这次请求。
* 如果服务器没有实现CORS接口，或者判断Origin后，这个源不在允许范围内，则返回1个正常的响或将响应头中Access-Control-Allow-Origin字段设置为除请求头中Origin字段值以外的值。

浏览器接收到响应后发现响应头没有包含Access-Control-Allow-Origin字段，或包含了Access-Control-Allow-Origin字段，但是值不是当前域名，则认为出错，被XMLRequest的onerror回调函数捕获

所以上面的例程，如果返回1个普通的响应，响应头如下：



页面接收到响应后显示：



注意:这种错误无法通过状态码识别，因为HTTP回应的状态码有可能是200。此时onerror和onreadystatechange函数内“if ((xhr.readyState == 4) && (xhr.status == 200))”条件内的代码都会被得到执行

* 如果服务器实现了CORS接口，并且判断Origin指定的源在允许范围内，则需要在响应头中添加Access-Control-Allow-Origin和一些其他字段，其中Access-Control-Allow-Origin的值必须为请求头中Origin字段值或“\*”

浏览器接收到响应后发现响应头包含Access-Control-Allow-Origin字段，并且字段值就是当前域名，就认为服务器是允许被当前域名访问的，就不会调用onerror事件，但onreadystatechange函数内“if ((xhr.readyState == 4) && (xhr.status == 200))”条件内的代码可以正常被执行。

##### 非简单请求

非简单请求在正式请求之前，浏览器会自动增加1次预检请求，来询问服务器当前网页的域名是否在服务器的许可名单中，以及可以使用哪些请求方式和头信息字段，只有得到肯定答复，浏览器发正式发送XMLHttpRequest请求，否则报错

以如下代码为例：

|  |
| --- |
| <**body**>  <**div id="div1"**></**div**> </**body**> </**html**> <**script type="text/javascript"**>  **var *url*** = **'http://localhost:8002/test'**;  **var *xhr*** = **new** XMLHttpRequest();  ***xhr***.onreadystatechange = **function**(){  **if** ((***xhr***.**readyState** == 4) && (***xhr***.**status** == 200)){  document.getElementById(**'div1'**).**innerHTML** += ***xhr***.responseText;  }  };  ***xhr***.onerror = **function**(){  document.getElementById(**'div1'**).**innerHTML** += **'request error'**;  };  ***xhr***.open(**'PUT'**, ***url***, **true**);  ***xhr***.setRequestHeader(**'key1'**,**'value1'**);  ***xhr***.setRequestHeader(**'key2'**,**'value2'**);  ***xhr***.**send**(**null**); </**script**> |

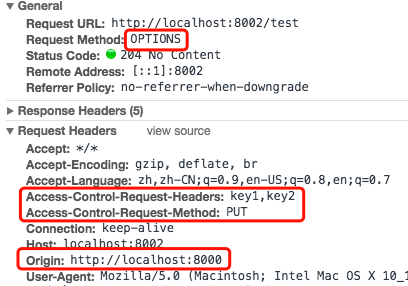
该CORS请求使用PUT请求，并且设置了2个自定义请求头：key1、key2

刷新该页面后，控制台可以看见3次请求：



###### 预检请求

* 点开上图中预检请求，可以看到如下请求头信息



* + 预检请求采用OPTION请求方式
  + 预检请求浏览器自动在请求头中加入如下字段：

[Origin](#Origin)

[Access-Control-Request-Method](#Access_Control_Request_Method)

[Access\_Control\_Request\_Headers](#Access_Control_Request_Headers)（请求前设置了自定义请求头字段时浏览器自动添加该字段，否则不使用该字段）

* 预检请求的响应头

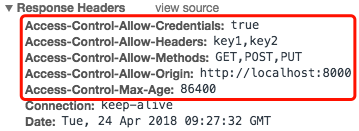
响应头中必须包含如下必选字段：

* + - 如果没有包含任何CORS请求相关的字段，则浏览器认为服务器拒绝
    - 如果没有[Access\_Control\_Allow\_Origin](#Access_Control_Allow_Origin)字段或者该字段值不是当前网页域名或不是“\*”，则表示服务器拒绝
    - 如果[Access\_Control\_Allow\_Methods](#Access_Control_Allow_Methods)中允许的请求方式不包括当前请求的请求方式，则浏览器认为服务器拒绝
    - 如果[Access\_Control\_Allow\_Headers](#Access_Control_Allow_Headers)没有全部包含当前页面的CORS请求前设置的自定义请求投字段，则浏览器认为服务器拒绝

响应头中也可能包含如下可选字段：

* + - [Access\_Control\_Allow\_Credentials](#Access_Control_Allow_Credentials)
    - [Access\_Control\_Max\_Age](#Access_Control_Max_Age)

如：



###### 正式请求

一旦服务器通过了预检请求，以后每次浏览器正常的CORS请求，就都跟简单请求一样了

##### cors头字段

###### Origin

* 用于CORS请求的请求头中
* 用以标识发送该请求的脚本来自哪个域名
* 该字段是浏览器自动添加上的

###### Access-Control-Allow-Origin

* 用于CORS请求的响应头中
* 如果允许该域名的请求，该字段是必须的
* 用于标识返回响应的服务器允许哪些域名的请求

\*：允许任意服务器访问

具体某个服务器的全域名：允许访问的域名

如果响应头中没有该字段，或该字段值不是当前域名，浏览器则报错

* 该字段需要服务器代码中进行添加

###### Access-Control-Allow-Credentials

* 用于简单请求的响应头中
* 该字段可选
* 布尔值，标识是否允许发送cookie，如果允许，设置该字段为true；如果不允许，不使用该字段即可

###### Access-Control-Expose-Headers

* 用于CORS请求的响应头中
* 该字段可选
* 作用

XMLHttpRequest对象的getResponseHeader()方法只能拿到6个基本字段：Cache-Control、Content-Language、Content-Type、Expires、Last-Modified、Pragma，其他的字段即使设置到响应头中，也无法通过该方法让你拿到这个字段的值

如：

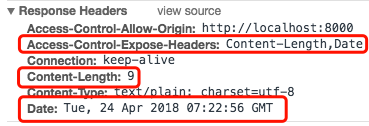
本次响应头如下：



其中Content-Length字段确确实实存在响应头中，但是通过getResponseHeader()方法取该字段值返回的是null

CORS请求时，如果服务器希望浏览器中可以通过getResponseHeader()方法拿到Content-Length字段的值，就将Content-Length等字段名以数组形式设置为Access-Control-Expose-Header字段的值，浏览器就可以通过getResponseHeader()方法获取到这些字段的值了

如：



Access-Control-Expose-Header字段中指定了Content-Length和Date这两个字段，则这两个字段的值就可以通过getResponseHeader()方法获取了

###### Access-Control-Request-Method

* 用于预检请求的请求头中
* 该字段浏览器自动添加，值就是非简单请求的请求方式

如：





###### Access-Control-Request-Headers

* 用于预检请求的请求头中
* 该字段值为逗号分隔的字符串，内容是浏览器发送CORS请求时额外发送的自定义请求头字段名

如：





* 如果浏览器没有设置自定义请求头字段，则预检请求头中没有该字段

###### Access-Control-Allow-Methods

* 用于预检请求的响应头
* 该字段值是逗号分隔的字符串，内容表示服务器允许的当前域名跨域请求的所有请求方式；如果当前页面的CORS请求的请求方式不在范围内，则浏览器抛出错误

###### Access-Control-Allow-Headers

* 用于预检请求的响应头
* 该字段值是逗号分隔的字符串，内容表示服务器允许的当前域名跨域请求的所有自定义请求头字段;如果当前页面的CORS请求在请求之前设置的自定义字段没有全部包含在内，则浏览器抛出错误

###### Access-Control-Max-Age

* 用于预检请求的响应头
* 该字段可选
* 用来指定本次预检请求有效期，单位秒，在此期间，不用发出另一条预检请求
* 如果不使用该参数，默认5s内不需要预检

##### withCredentials属性

* 该属性是XMLHttpRequest对象的属性
* 作用

CORS请求默认不发送cookie，[Access-Control-Allow-Credentials](#Access_Control_Allow_Credentials)可以设置服务器端同意使用cookie，但是浏览器端也需要同意才能真正的使用cookie

将该属性设置为true，表示浏览器端同意使用cookie，如果不设置该值，不同浏览器的处理方式不同，所以最好显示的设置这个属性的值

* 注意

如果要跨域使用cookie，必须注意以下几点：

* + Access-Control-Allow-Origin字段不能设为“\*”,必须设置为与请求网页一致的域名
  + 跨域使用cookie时，cookie依然遵循同源原则，必须合理设置cookie的domain属性，跨域使用cookie时cookie才会被上传
  + 原网页中document.cookie无论如何也无法读取到跨域服务器下的cookie

## ctx属性整理

ctx就是context对象，在koa中，将request对象和response对象的一些属性的引用放在了ctx中

### body

* 作用

用于设置响应正文内容

* 例

|  |
| --- |
| router.get(**'/'**,**async** (ctx)=>{  ctx.**body**=**'8002'**; }); |

显示效果：



### params

* 只有动态获取路由时使用
* 参见[获取动态路由参数](#获取动态路由参数)

### query

* 作用：获取url中拼接的参数
* 参见：[获取URL传值](#获取URL传值)

### querystring

* 作用：获取url中拼接的参数的原始字符串
* 参见：[获取URL传值](#获取URL传值)

### request

请求相关的属性全在这里

#### query

与[ctx.query](#ctx_query) 是同一个属性

#### querystring

与[ctx.querystring](#ctx_querystring)是同一个属性

#### body

* 只有使用了koa-bodyparser模块时，才有本属性，默认没有本属性
* 作用：koa-bodyparser将请求的body中的内容转换为对象保存在该属性中

#### rawBody

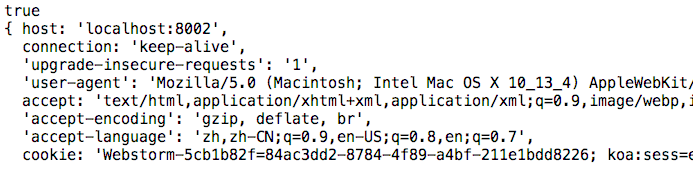
* 只有使用了koa-bodyparser模块时，才有本属性，默认没有本属性
* 作用：koa-bodyparser将请求的body中的原始内容保存在该属性中

### headers、header

* 这两个属性是同一个属性
* 作用：获取所有的请求头字段
* 例

|  |
| --- |
| router.get(**'/'**,**async** (ctx)=>{  ***console***.log(ctx.**header** === ctx.**headers**);  ***console***.log(ctx.**header**);  ctx.**body**=**'8002'**; }); |

结果：



### path

* 作用：获取相对于本服务器域名的相对请求路径
* 例

|  |
| --- |
| router.get(**'/test'**,**async** (ctx)=>{  ***console***.log(ctx.**path**);  ctx.**body**=**'8002'**; }); |

访问：<http://localhost:8002/test?name=111>

打印：/test

### throw(args. . .)

* 作用：抛出服务器错误，指定状态码和响应信息
* 例：

|  |
| --- |
| router.get(**'/test'**,**async** (ctx)=>{  ctx.throw(**"test error"**,400); }); |

页面显示内容：test error

响应状态码：400

### render()

* 只有使用了koa-views模块时，才有本属性方法，默认没有本属性方法
* 作用：渲染页面
* 参见[EJS模板引擎](#EJS模板引擎)

### state

* 作用：

如果一次请求响应过程中，1个地方设置了某个属性后，整个服务器希望可以直接获取到该属性，则可以将那个属性存放在state属性中

类似于javaEE中的域对象

* 例

比如在渲染ejs模板引擎时，将某个属性设置到state属性上，则不需要使用render函数传这个值，渲染时依然可以将该值渲染到页面上

|  |
| --- |
| app.use(**async** (ctx,next)=>{  ctx.**state**.**username**=**'zhangsan'**;  **await** next(); }); |

### cookies

里面是关于cookie的方法和属性

#### set

* 作用：用于设置cookie
* 参见[cookie](#cookie)

#### get

* 作用：用于获取cookie
* 参见[cookie](#cookie)

### session

* 只有使用koa-session模块时，才有该属性
* koa-session模块将关于session的属性和方法都放在了session属性中
* 向session属性中设置1对key和value，就能设置1个session，通过ctx.session.key即可获取session
* 参见[session](#session)

### req

与ctx.request不是同一个属性

#### file

* 只有使用koa-multer模块进行了文件上传操作是，才有该属性
* 作用：用来保存上传文件的信息
* 参见[文件上传](#文件上传)

#### attachment(filename)

* 根据下载文件的filename设置Content-Disposition响应头
* 参见：[文件下载](#文件下载)

### url

* 作用：获取相对于本服务器域名的相对访问路径和url中拼接的请求参数
* 例

|  |
| --- |
| router.get(**'/test'**,**async** (ctx)=>{  ***console***.log(ctx.**url**);  ctx.**body** = **'8002'**; }); |

访问：<http://localhost:8002/test?name=111>

打印：/test?name=111

# 模块

## 核心模块

### http

#### demo

|  |
| --- |
| *//引入http模块* **var http** = *require*(**'http'**); *//创建服务监听80端口，当有访问时自动调用回调函数* **http**.*createServer*(**function** (req,res) {  res.writeHead(200,{**"Content-Type"**:**"text/html;charset=utf-8"**});  res.write(**"你好 nodejs"**);  res.end();*//结束响应* }).listen(80); |

* 注意
  + utf-8不能再用单引号引起来，否则该设置项不生效
  + 有时因为异步编程可能会出现错误，提示“write after end”,更安全的做法是将数据使用end方法返回，而不是write方法

|  |
| --- |
| res.write(**"你好 nodejs"**); res.end();*//结束响应* |

改为

|  |
| --- |
| res.end(**"你好 nodejs"**);*//返回数据并结束响应* |

### url

#### 获取请求url

在http模块的基础上，通过request参数的url属性，即可拿到访问请求的url

如：

访问：<http://localhost/?a=1>

req.url：/?a=1

例：

|  |
| --- |
| **var http** = *require*(**'http'**);  **http**.*createServer*(**function** (req,res) {  ***console***.log(req.**url**);  res.end(); }).listen(80); |

访问：<http://localhost/?a=1>

打印：/?a=1

#### 请求过滤

1次访问，实际上是发送了2次请求

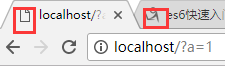
同样使用上面的例程，当访问<http://localhost/?a=1>时，打印结果如下：



其中：

第一行是真正的请求

第2行是请求图标的请求



我们只需要第一行的请求，所以需要进行过滤

|  |
| --- |
| **var url** = *require*(**'url'**); **var http** = *require*(**'http'**);  **http**.*createServer*(**function** (req,res) {  **if**(req.**url** !=**'/favicon.ico'**){  ***console***.log(req.**url**);  }  res.end(); }).listen(80); |

此时在访问<http://localhost/?a=1>时，打印结果如下：



#### 解析url

##### 方法

格式： var parse(url)

var parse(url, parseQueryString)

功能：解析url地址，将url中包含的信息解析为对象

参数： url：待解析的url

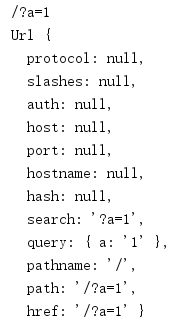
parseQueryString：是否将get’传值解析为对象，默认false

返回值：封装为对象后的解析结果

##### 例

|  |
| --- |
| **var url** = *require*(**'url'**); **var http** = *require*(**'http'**);  **http**.*createServer*(**function** (req,res) {  **if**(req.**url** !=**'/favicon.ico'**){  ***console***.log(req.**url**);  **var** result = **url**.*parse*(req.**url**,**true**);  ***console***.log(result);  }  res.end(); }).listen(80); |

结果：



### fs

#### 异步方法

* 异步方法不能直接获取结果，只能通过回调函数获取结果
* 异步方法执行过程中出错，会继续调用回调函数，并将错误传入回调函数

##### 判断是文件还是目录

|  |
| --- |
| **var** fs = *require*(**'fs'**);  *//stat函数用来判断是文件还是目录 //第1个参数为文件（夹）的路径，如果不是用全路径则是相对于*[*工程路径*](#_工程路径)*的相对路径 //第2个参数为回调函数，该函数有2个参数 // err：错误：如找不到该文件 // stats：状态，判断的结果保存在该参数中* fs.stat(**'demo1.js'**,(err,stats) =>{  **if**(err){  ***console***.log(err)  **return**;  }  *//stats的isFile()方法返回是否为文件* ***console***.log(**`是否为文件：**${stats.isFile()}**`**);  *//stats的isDirectory()方法返回是否为目录* ***console***.log(**`是否为目录：**${stats.isDirectory()}**`**); }) |

##### 创建目录

|  |
| --- |
| **var** fs = *require*(**'fs'**);  *//mkdir函数用来创建目录 //第1个参数为目录名，如果不是用全路径则是相对于*[*工程路径*](#_工程路径)*的相对路径下创建 //第2个参数为目录权限，默认0777 //第3个参数为回调函数，该函数有1个参数 // err：错误信息，如：目录已存在* fs.mkdir(**'css'**,(err)=>{  **if**(err){  ***console***.log(err);  **return false**;  }  ***console***.log(**'创建目录成功'**); }) |

##### 写入文件

|  |
| --- |
| **var** fs = *require*(**'fs'**);  *//writeFile函数用来写入文件，不存在该文件则创建，存在则覆盖 //参数： // filename：文件名 // data：文件内容，可以为字符串或buff // option：数组对象 // ·encoding：默认utf-8，data使用buff时，该处值应为ignored // ·mode：权限，默认438 // ·flag：默认'w' // callback：回调函数，传递1个异常参数err* fs.writeFile(**'t.txt'**,**'hello world'**,**'utf-8'**,(err)=>{  **if**(err){  ***console***.log(err);  **return false**;  }  ***console***.log(**'写入成功'**); }) |

##### 追加文件

|  |
| --- |
| **var** fs = *require*(**'fs'**);  *//appendFile函数用来追加文件，不存在该文件则创建，存在则追加 //参数： // filename：文件名 // data：文件内容，可以为字符串或buff // option：数组对象 // ·encoding：默认utf-8，data使用buff时，该处值应为ignored // ·mode：权限，默认438 // ·flag：默认'w' // callback：回调函数，传递1个异常参数err* fs.appendFile(**'t.txt'**,**'hello world'**,**'utf-8'**,(err)=>{  **if**(err){  ***console***.log(err);  **return false**;  }  ***console***.log(**'追加成功'**); }) |

##### 读取文件

|  |
| --- |
| **var** fs = *require*(**'fs'**);  *//readFile函数用来读取文件 //参数： // filename：文件名 // callback：回调函数，传递1个异常参数err，和保存读取结果的参数data // ·data：buff对象，存储的时ASCII码，需要使用toString方法转换为字符串* fs.readFile(**'t.txt'**,(err,data)=>{  **if**(err){  ***console***.log(err);  **return false**;  }  ***console***.log(data);  ***console***.log(data.toString()); }) |

##### 读取目录

|  |
| --- |
| **var** fs = *require*(**'fs'**);  *//readdir函数用来读取目录 //参数： // dirname：目录名 // callback：回调函数，传递1个异常参数err，和保存读取结果的参数data // ·data：数组，将该目录下的所有文件（夹）放在该数组中* fs.readdir(**'node\_modules'**,(err,data)=>{  **if**(err){  ***console***.log(err);  **return false**;  }  ***console***.log(data); }) |

##### 重命名、剪切

|  |
| --- |
| **var** fs = *require*(**'fs'**);  *//rename函数用来对文件、目录重命名，或剪切文件 //参数： // oldname：原名 // newname：新名 // callback：回调函数，传递1个异常参数err // 例：重命名* fs.rename(**'a.txt'**,**'t.txt'**,(err)=>{  **if**(err){  ***console***.log(err);  **return false**;  }  ***console***.log(**'重命名成功'**); }) *// 例：剪切* fs.rename(**'t.txt'**,**'node\_modules/t.txt'**,(err)=>{  **if**(err){  ***console***.log(err);  **return false**;  }  ***console***.log(**'剪切成功'**); }) |

##### 删除目录

|  |
| --- |
| **var** fs = *require*(**'fs'**);  *//rmdir函数用来删除目录 //参数： // dirname：目录名 // callback：回调函数，参数为异常信息err* fs.rmdir(**'css'**,(err)=>{  **if**(err){  ***console***.log(err);  **return false**;  }  ***console***.log(**'删除成功'**); }) |

##### 删除文件

|  |
| --- |
| **var** fs = *require*(**'fs'**);  *//unlink函数用来删除文件 //参数： // filename：文件名 // callback：回调函数，参数为异常信息err* fs.unlink(**'node\_modules/t.txt'**,(err)=>{  **if**(err){  ***console***.log(err);  **return false**;  }  ***console***.log(**'删除成功'**); }) |

##### 文件流读取

|  |
| --- |
| **var** fs = *require*(**'fs'**)  */\* \* 1.fs的文件流是分段读取的； \* 2.创建文件流时指定要读取的文件，并返回文件流对象 \* 3.注册接收数据事件'data'的回调函数，每读取到一段内容时就会调用该回调函数 \* 将读取到的内容传入该回调函数内 \* 4.注册读取结束事件'end'的回调函数，读取结束后调用该回调函数 \* 5.注册发生错误事件'error'的回调函数，读取过程中发生错误时调用该回调函数 \* 并将错误传入回调函数  \*/  //用于接收数据* **let** sdata = **''**;  */\*createReadStream函数用于创建读取文件流 \* 参数： \* path：要读取的文件 \* 返回值：返回文件流对象 \*/* **let** fsReadStream = fs.createReadStream(**'fsdemo.js'**);  */\* \* on函数用于注册事件的回调函数 \* 参数： \* eventType(String):事件类型，只能写指定的值 \* listenner(function):产生该事件时的回调函数  \*/  /\* \* data事件用于接收每次读取到的内容，每读取到一段数据，调用该事件的回调函数 \* 参数： \* data：本次读取到的内容  \*/* fsReadStream.**on**(**"data"**,(data)=>{  sdata+=data; })  */\* \* end事件在所有内容读取结束后产生，文件读取结束后，调用该事件的回调函数  \*/* fsReadStream.**on**(**'end'**,()=>{  ***console***.log(sdata); })  */\* \* error事件在读取过程中发生错误时产生，发生错误时，调用该事件的回调函数 \* 参数： \* error：错误信息  \*/* fsReadStream.**on**(**'error'**,(error)=>{  ***console***.log(error); }) |

##### 文件流写入

|  |
| --- |
| **var** fs = *require*(**'fs'**)  *//待写入数据* **var** sdata = **'hello world'**;  *//创建文件写入流，并指定要写入的文件（默认覆盖方式）* **var** fsWriteStream = fs.createWriteStream(**"fsWstream.txt"**);  *//写入数据* fsWriteStream.write(sdata);  *//写完后，执行该函数来关闭文件，必须执行该函数才会产生finish事件* fsWriteStream.end();  *//文件关闭后产生finish事件* fsWriteStream.**on**(**'finish'**,()=>{  ***console***.log(**'finished'**); })  *//写入过程中发生错误则产生该事件* fsWriteStream.**on**(**'error'**,(error)=>{  ***console***.log(error); }) |

##### 管道流

|  |
| --- |
| **var** fs = *require*(**'fs'**); */\* \* 管道流用于将1个文件中的内容复制到另一个文件  \*/* **var** fsReadStream = fs.createReadStream(**'fsdemo.js'**) **var** fsWriteStream = fs.createWriteStream(**'1.txt'**); fsReadStream.pipe(fsWriteStream); |

#### 同步方法

* 同步方法不需要使用回调函数获取结果，直接通过返回值获取结果
* 同步方法执行过程中如果产生错误程序将直接崩溃

##### 读取文件

|  |
| --- |
| **var** fs = *require*(**'fs'**); **var** data = fs.readFileSync(**"demo1.js"**); ***console***.log(data.toString()); |

使用同步方法

### path

* path模块提供了用于处理文件和目录路径的实用程序
* 该模块中的方法参见：

<https://nodejs.org/docs/latest/api/path.html>

#### 获取文件后缀名

|  |
| --- |
| **var** pathModule = *require*(**'path'**); ***console***.log(pathModule.extname(**"static/index.html"**));*// .html* |

### events

#### 事件驱动

|  |
| --- |
| **var** *eventModule* = *require*(**'events'**);  *//获得EventEmitter对象* **var** eventEmitter =**new** *eventModule*.EventEmitter();  */\* \* 1. on函数用来为指定的事件注册回调函数 \* 2. 参数： \* eventType：事件名称，可以随便定义， \* 只要注册与触发时事件名称对应即可 \* listener：回调函数，允许有多个参数用来传递数据  \*/* eventEmitter.**on**(**'testEvent'**,(data)=>{  ***console***.log(data); }); */\* \* 1. emit函数用来触发事件 \* 2. 参数： \* type：事件名称，必须与注册的事件名称对应 \* rest参数：后面是可变参数，这些参数都会被 \* 传入到回调函数中  \*/* eventEmitter.emit(**'testEvent'**,**'1223'**); |

## 第三方模块

### EJS

#### 介绍

EJS是后台模板引擎，用于将数据库和文件读取的数据加载到模板（ejs文件）中渲染之后显示到浏览器上

#### 安装

工程跟目录下执行

|  |
| --- |
| npm i ejs --save |

#### 后台渲染

##### renderFile

格式：renderFile(ejsfile, object, fun)

功能：

将object对象中的数据加载到ejsfile模板中进行渲染，渲染后的结果通过fun回调函数返回

参数：

ejsfile：模板文件

object：将模板中需要的数据封装成对象并传入该函数

fun：回调函数，回调函数有2个参数：err、data

##### 例

* index.ejs

|  |
| --- |
| <!DOCTYPE **HTML** PUBLIC **"-//W3C//DTD HTML 4.01 Transitional//EN"  "http://www.w3.org/TR/html4/loose.dtd"**> <**html**> <**head**>  <**title**>$Title$</**title**> </**head**> <**body**> <% **for**(let ***i***=0; ***i***<3; ***i***++){ %>  <**h3**><%= msg %></**h3**> <% } %> <%- node %> </**body**> </**html**> |

* staticWeb.js

|  |
| --- |
| **httpModule**.*createServer*((req,res)=>{res.writeHead(200,{**'Content-Type'**:**'text/html;charset=utf-8'**});  **var** msg = **'hello world'**;  **var** node = **'<h2>hello node</h2>'** ejsModule.renderFile(**'index.ejs'**,{**msg**:msg, **node**:node},(err,data)=>{  res.end(data);  }) }).listen(80); |

* 结果



### koa（koa2）

参见[环境搭建](#环境搭建)

### koa-router（koa路由）

参见：[koa路由](#koa路由)

### koa-bodyparser（获取http协议body中传值）

参见[获取body传值](#获取body传值)

### koa-views（页面渲染）

参见[EJS模板引擎](#EJS模板引擎)

### ejs（ejs）

参见[EJS模板引擎](#EJS模板引擎)

### koa-static（静态资源）

参见[静态资源](#静态资源)

### koa-session（koa中使用session）

参见[session](#session)

### koa-multer（文件上传）

参见[文件上传](#文件上传)

### koa-send（大文件下载）

参见[文件下载](#文件下载)

### archiver（打包）

#### 介绍

该模块用于将文件或文件夹打包成zip或tar格式压缩包

#### 使用

##### 模块安装

|  |
| --- |
| npm install archiver --save |

##### 代码

###### 多个单独文件打包

|  |
| --- |
| *// 将要打包的文件列表* **const** list = [{**name**: **'docs/1.docx'**},{**name**: **'docs/2.docx'**}]; *//创建zip文件的文件流，参数为打包后文件的保存路径* **const** zipStream = fs.createWriteStream(**'uploads/1.zip'**); *//获取zip格式的archiver对象* **const** zip = *archiver*(**'zip'**); *//就这么用* zip.pipe(zipStream); **for** (**let** i = 0; i < list.length; i++) {  */\*  \* 1. 添加单个文件到压缩包  \* 2. 参数：  \* 参数1：本地待压缩文件的读文件流  \* 参数2：{name:value}对象，value为该文件在zip文件中的文件名  \*/* zip.append(fs.createReadStream(**`./uploads/**${list[i].**name**}**`**), list[i]); } *//等待文件压缩完成* **await** zip.finalize(); |

###### 文件夹下文件打包

|  |
| --- |
| *//创建zip文件的文件流，参数为打包后文件的保存路径* **const** zipStream = fs.createWriteStream(**'uploads/1.zip'**); *//获取zip格式的archiver对象* **const** zip = *archiver*(**'zip'**); *//就这么用* zip.pipe(zipStream); *//将uploads/docs文件夹下内容打包，参数2指定zip文件中打包内容属于哪个目录* zip.directory(**'uploads/docs/'**,**'docs'**); *//等待文件压缩完成* **await** zip.finalize(); *//注意：生成的zip文件路径不可以在zip.directory参数1的目录下，否则一起把压缩包又打包了* |

### koa-onerror（异常处理）

#### 介绍

如果不用该模块，产生异常时，页面上显示的是“Internal Server Error”，用了该模块，页面上会显示一些更详细的信息。同样该模块也可以做产生异常时页面的跳转等…

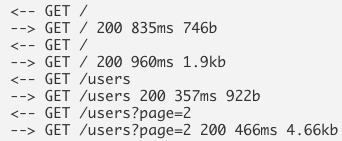
#### 例

|  |
| --- |
| **const Koa** = *require*(**'koa'**); **const** Koa\_router = *require*(**'koa-router'**); **const** onerror = *require*(**'koa-onerror'**);  **let** app = **new Koa**(); **let** router =**new** Koa\_router();  onerror(app);  router.get(**'/'**,**async** (ctx,next)=>{  **throw new *Error***(**'test onerror'**);  **return** ctx.**body** = **'test onerror'**; });  app.use(router.routes()); app.use(router.allowedMethods()); app.listen(8001); |

### koa-logger（日志：请求响应信息）

* 作用

将类似于如下请求响应信息log到指定的输出位置（控制台、文件等）



* 用法

安装：npm install koa-logger –save

加载中间件：

* + 使用默认输出（控制台）

|  |
| --- |
| app.use(*logger*()); |

* + 自定义输出

|  |
| --- |
| app.use(*logger*((str,args)=>{  ***console***.log(str); })); |

* + - 将str参数输出到指定的位置，来自定义输出位置，如：文件
    - args参数就是数组形式表示出str表示的内容，可以不用管这个参数
* 注意

这个中间件应该是最先加载的中间件

### winston（日志：自定义log）

#### 打印类型

winston在输出日志信息时，可以指定类型输出，打印类型分为以下几种：

* error
* error类log通过如下两种方法进行打印：

|  |
| --- |
| winston.error(**'\t error\_error'**);  *//等效于*  winston.log(**'error'**, **'\t error\_log\_error'**); |

* error类log是异步的，打印顺序不固定
* warn
* warn类log通过如下两种方法进行打印

|  |
| --- |
| winston.warn(**'\t warn\_warn'**); *//等效于* winston.log(**'warn'**,**'\t warn\_log\_warn'**); |

* warn类log是同步的，按顺序打印
* info
  + info类log通过如下两种方法进行打印：

|  |
| --- |
| winston.info(**'\t info\_info'**);  *//等效于*  winston.log(**info**, **'\t info\_log\_info'**); |

* info类log是同步的，按顺序打印
* verbose
* verbose类log通过如下两种方法进行打印：

|  |
| --- |
| winston.verbose(**'\t verbose'**); *//等效于* winston.log(**'verbose'**,**'\t log\_verbose'**); |

* verbose类log是同步的，按顺序打印
* debug
  + debug类log通过如下两种方法进行打印：

|  |
| --- |
| winston.debug(**'\t debug\_debug'**);  *//等效于*  winston.log(**'error'**, **'\t debug\_log\_debug'**); |

* debug类log是异步的，打印顺序不固定
* silly
* silly类log通过如下两种方法进行打印

|  |
| --- |
| winston.silly(**'\t silly'**); *//等效于* winston.log(**'silly'**,**'\t log\_silly'**); |

* silly类log是同步的，按顺序打印

#### 打印等级

* winston的每个实例都可以配置成不同级别的logger，打印等级分为以下几种（等级由低到高排序）：

error 🡪 warn 🡪 info 🡪 verbose 🡪 debug 🡪 silly

* 等级越低代表打印信息越重要
* 设置为某个等级后，这个等级对应的打印类型和比他低的等级对应的打印类型都会被打印出来

如：

error等级只打印error类型的log

warn等级打印error和warn类型的log

info等级打印error、warn和info类型的log

以此类推…

* 默认等级：info
* 配置等级的方法：

配置某个实例的打印等级有3中方法：

* 创建实例时直接指定

|  |
| --- |
| **let** logger = **new** winston.Logger({  **level**:**'info'** }); |

* 创建实例后用configure()配置

|  |
| --- |
| logger.configure({  **level**:**'debug'** }); |

* 创建实例后修改实例的level属性

|  |
| --- |
| logger.**level** = **'debug'**; |

#### transport

##### 介绍

* 每个transport就是1个日志输出的设备，比如：控制台、文件。
* winston可以为每个实例配置多个transport，每个transport可以在配置时单独指定打印等级、name属性等信息
* 默认logger的transports默认有1个Console；自定义的实例默认transports属性为{}
* winston模块中内置了3个核心的transport：

Console、File、Http

另外可以引入一些其他的模块，将其作为winston的transport

* 关于3个核心transport的参数说明和第三方transport的使用方法，详细内容参见参见：[github-winston](https://github.com/winstonjs/winston/blob/2.x/docs/transports.md#winston-core)

##### winston配置transports的方法

配置某个实例的transports有3种方法：

* 指定或增加transport
* 创建实例时直接指定transports属性

|  |
| --- |
| **let** logger = **new** winston.Logger({  **transports**:[  **new** winston.transports.Console({*/\* options \*/*}),  **new** winston.transports.File({*/\* options \*/*}),  ] }); |

* 创建实例后通过configure()配置

|  |
| --- |
| logger.configure({  **transports**:[  **new** winston.transports.Console({*/\* options \*/*}),  **new** winston.transports.File({*/\* options \*/*}),  ] }); |

* 创建实例后，通过实例的add()追加transport

|  |
| --- |
| logger.add(winston.transports.Console,*/\* options \*/*); |

* 配置多个同类型transport

配置transports时，可以在同一个实例上配置多个相同类型的transport，但是必须配置这几个transport的name属性不能相同。

|  |
| --- |
| **let** logger = **new** winston.Logger({  **transports**:[  **new** winston.transports.File({  **filename**:**'./test1.log'**,  **name**:**'transFile1'**}),  **new** winston.transports.File({  **filename**:**'./test2.log'**,  **name**:**'transFile2'**})  ] }); |

* 指定类型配置transport

使用前两种法师配置transport时，可以不指定transports属性，而是通过类型配置transport

|  |
| --- |
| */\*  \* 1. 用于在winston身上添加1个实例  \* 2. 参数  \* 参数1：实例的name属性  \* 参数2：实例的配置参数  \*/* winston.loggers.add(**'file1'**,{  */\*  \* 实例的配置参数  \*/   //配置1个console类型的transport  //使用此方法进行配置时，相同类型的transport只保留最后一个* **console**:{  *//console类型transport的参数配置* **level**:**'debug'** },   *//实例的level* **level**:**'info'** }); |

使用此方法配置时，相同类型的transport只保留最后1个，前面的会被覆盖掉

##### transport的配置参数

每个winston实例在配置transport时，可以为transport指定参数，下面几个参数是必须要说的

###### name

* 每个transport都有1个name属性，如果没有明确指定，则使用默认值创建
* 不同类型的transport的name属性默认值不同；相同类型的transport的name属性默认值相同
* 为同一个实例配置两个相同类型的transport时，如果没有明确指定，则会因为name属性相同而抛出错误

###### level

* 每个transport在配置时可以单独设置打印等级
* 如果没有为这个transport指定打印等级，则继承其所属的实例的打印等级；如果配置transport时指定了他的打印等级，则使用他自己的打印等级，而忽略其所属实例的打印等级

##### 移除transport

从实例身上移除某个transport时，需要使用remove()方法移除，其参数是该实例身上的某个transport的name属性值

|  |
| --- |
| logger.remove(**'transFile'**); |

注意：

以下方法是移除使用name属性默认值创建的Console类型的transport

|  |
| --- |
| logger.remove(winston.transports.Console); |

#### 默认logger与自己创建logger实例

##### 自己创建logger实例

创建logger实例有2种方法：

* 第一种

|  |
| --- |
| **let** logger = **new** winston.Logger({  **level**:**'error'**,  **transports**:[  **new** winston.transports.Console({**level**:**'debug'**})  ] }); |

* 第2种

|  |
| --- |
|  |

##### 默认logger

默认logger就是winston对象，默认logger其实也是1个winston实例，可以像winston实例一样正常使用

#### 使用元数据打印

* 就是直接打印对象

|  |
| --- |
| winston.log(**'info'**, **'metadata'**, { **key**: **'value'** }); |

打印结果： info: metadata key=value

* 注意

如果对象不是最后一个参数，则打印成{Key：value}形式，如果是最后1个，则打印成“key=value”形式

如果最后一个参数是1个空对象或函数，则不打印

|  |
| --- |
| logger.info(**'test '**,{**key0**:123}); logger.info(**'test '**,{**key0**:123},{**key1**:123}); logger.info(**'test '**,{**key0**:123},{**key1**:123},{**key2**:123}); logger.info(**'test '**,{**key0**:123},**'aaa'**); logger.info(**'test'** ,{}); logger.info(**'test'** ,{},123); logger.info(**'test'** ,**function** () {}); logger.info(**'test'** ,**function** () {},123);  *// info: test key0=123 // info: test { key0: 123 } key1=123 // info: test { key0: 123 } { key1: 123 } key2=123 // info: test { key0: 123 } aaa // info: test // info: test {} 123 // info: test // info: test function () {} 123* |

#### profile（计时功能）

* winston还提供了1个计时功能，计时结束后log出时间差，默认等级info
* 一共有2中方法：
  + 第一种

|  |
| --- |
| winston.profile(**'111'**);  *setTimeout*(**function** () {  winston.profile(**'111'**); }, 1000); |

1s后打印： info: 111 durationMs=1006

* 第2种

|  |
| --- |
| **var** timer = winston.startTimer() *setTimeout*(**function**(){  timer.done(**"222"**); }, 1000); |

1s后打印： info: 222 duration=1012ms

#### 字符串插值

* 字符串插值就是printf函数的用法，
* “% j”是对象的占位符
* 如果后面的参数个数超出了占位符的个数，则超出的参数按照[元数据](#元数据)的规则打印
* 如果前面的占位符个数超出了后面参数的个数，则超出的占位符原样打印出来

|  |
| --- |
| logger.info(**'my %s is %j'**,**'name'**,{}); logger.info(**'my %s is shuyan'**,**'name'**,{}); logger.info(**'my %s is shuyan'**,**'name'**,{},**'hi'**); logger.info(**'my %s is shuyan'**,**'name'**,{**age**:25});  *// info: my name is {} // info: my name is shuyan // info: my name is shuyan {} hi // info: my name is shuyan age=25* |

#### 查询log

|  |
| --- |
| **var** options = {  **from**: **new *Date*** - 24\* 600\*1000, *//起时间* **until**: **new *Date***, *//止时间* **start**: 20, *//跳过前20条记录* **order**: **'desc'**, *//将跳过前20条记录后的结果按时间戳倒叙排序* **limit**: 10, *//从排序后的结果中取出前10条记录* **fields**: [**'message'**,**'timestamp'**] *//显示message、timestamp字段* };  *//必须使用打印log的那个实例调用query方法* logger.query(options, **function** (err, results) {  **if** (err) {  **throw** err;  }  ***console***.log(results); }); |

#### 流日志

将日志从实例中回传

|  |
| --- |
| **const** winston = *require*(**'winston'**);  **let** logger = **new** winston.Logger({  **transports**:[  **new** winston.transports.File({  **filename**:**'./test.log'**,  **name**:**'transFile'** })  ] });  **var** index = 0; (**function** f(){  *setTimeout*(()=>{  logger.info(index ++);  *f*();  },1000); })();  logger.stream({ **start**: -1 }).**on**(**'log'**, **function**(log) {  ***console***.log(log);  ***console***.log(**'-----------------------------------------'**); }); |

打印结果：

先将原有日志读回，然后每次打印都会调用回调函数，将日志信息作为参数传入

#### 异常

winston可以用来捕获并打印uncaughtException

##### uncaughtException

就是未捕获的异常

##### winston记录uncaughtException的2种方式

* 实例化logger时指定

|  |
| --- |
| **let** logger = **new** winston.Logger({  **transports**: [  **new** winston.transports.File({  **filename**: **'./test.log'**,  **name**: **'transFile'** })  ],  **exceptionHandlers**:[  **new** winston.transports.Console()  ] }); **throw new *Error***(**'test'**); |

* 实例化后指定

|  |
| --- |
| logger.handleExceptions(  **new** winston.transports.File({**filename**: **'./exceptions.log'**})); **throw new *Error***(**'test'**); |

uncaughtException可以被多个transport捕获并打印

|  |
| --- |
| logger.handleExceptions([  **new** winston.transports.File({**filename**: **'./e1.log'**,**name**:**'e1'**}),  **new** winston.transports.File({**filename**: **'./e2.log'**,**name**:**'e2'**}) ]); **throw new *Error***(**'test'**); |

* 实例化后追加

|  |
| --- |
| logger.add(winston.transports.File, {  **filename**: **'./exception.log'**,  **handleExceptions**: **true**,  **humanReadableUnhandledException**: **true** }); **throw new *Error***(**'test'**); |

##### 捕捉异常后退出

* 默认情况下，winston捕捉并记录uncaughtException异常后就会退出，
* 如果捕获异常后不希望winston退出，则将实例的exitOnError属性设为fasle

|  |
| --- |
| **let** logger = **new** winston.Logger({  **exitOnError**:**false** }); *//或* logger.**exitOnError** = fasle; |

* exitOnError也可以接受1个回调函数，用来根据错误类型判断是否要退出

|  |
| --- |
| **let** logger = **new** winston.Logger({  exitOnError:(err)=>{  **return false**;  } }); *//或* logger.exitOnError = (err)=>{  **return false**; }; |

### mongoose（操作MongoDB）

#### 安装

|  |
| --- |
| npm install mongoose --save |

安装后引包：

|  |
| --- |
| **const** mongoose = *require*(**'mongoose'**); |

即可操作mongodb

#### 连接

##### connect

格式：mongoose.connect(url,option,(err)=>{})

功能：连接数据库

参数： url：要连接的数据库的url，

option：指定参数连接，参数后面补充

callback：连接成功或失败的回调函数

##### disconnect

格式：mongoose.disconnect(()=>{})

功能：断开连接

参数： callback：成功断开连接后的回调函数

#### schema

##### 作用

schema用于配置1张表中的所有字段信息，schema配置好之后，就可以用他创建1个model对象（对应某张表）

##### 用法

|  |
| --- |
| *//创建schema，对应1张表中的字段信息* **let** schema =**new** mongoose.**Schema**({**name**:***String***, **age**:***Number***}); |

* 从mongoose中获取Schema对象，使用该对象创建1个实例，字段信息配置在创建实例时的参数中
* 创建实例时还有1个参数option，后面补充

##### 字段类型

###### 包括

String 字符串

Number 数字

Date 日期

Buffer 二进制

Boolean 布尔值

Mixed 混合类型

ObjectId 对象ID

Array 数组

###### 配置字段类型2种方法

* 不用引号，首字母大写

|  |
| --- |
| {**name**:***String***, **age**:***Number***} |

* 用引号，首字母小写

|  |
| --- |
| {**name**:**'string'**, **age**:**'number'**} |

##### 追加字段信息

schema也可以创建之后再通过add方法追加字段信息

|  |
| --- |
| var MySchema = new Schema;  MySchema.add({ name: 'string', color: 'string', price: 'number' }); |

#### model

##### 作用

使用schema创建1个model，该model对应数据库中某张表

##### 用法

|  |
| --- |
| **let** Model = mongoose.model(**'model'**,schema,**'user'** ); |

* 使用前面创建的schema创建1个模型，该模型对应数据库中的1张表，对应的表名由第3个参数决定
* 第1个参数为模型名称。如果没有第3个参数，则使用模型名称生成1个名称作为对应到数据库中的表的名称；如果指定了第3个参数，该名称随意

生成规则为：先将模型名称全部转为小写，然后看最后1个字符是否为数字，如果不是数字符串再改为复数形式，是数字则不再改变

#### document

##### 作用

通过model对象创建1个实例，指定记录信息，来创建1个document对象，该对象对应数据库表中1条记录

##### 用法

|  |
| --- |
| **let** doc1 = **new** Model({**name**:**'shuyan'**,**age**:24}); *//将数据保存到数据库* doc1.save((err,doc)=>{  ***console***.log(doc); }); |

#### 自定义方法

##### 实例方法

自定义实例方法是通过Schema对象的methods是给document添加方法

可以通过Schema对象的methods属性给实例自定义扩展方法

|  |
| --- |
| var mongoose = require('mongoose');  mongoose.connect("mongodb://u1:123456@localhost/db1", function(err) {  if(!err){  var schema = new mongoose.Schema({ num:Number, name: String, size: String });  schema.methods.findSimilarSizes = function(cb){  return this.model('MyModel').find({size:this.size},cb);  }  var MyModel = mongoose.model('MyModel', schema);  var doc1 = new MyModel({ name:'doc1', size: 'small' });  var doc2 = new MyModel({ name:'doc2', size: 'small' });  var doc3 = new MyModel({ name:'doc3', size: 'big' });  doc1.save();  doc2.save();  doc3.save();  setTimeout(function(){  doc1.findSimilarSizes(function(err,docs){  docs.forEach(function(item,index,arr){  //doc1  //doc2  console.log(item.name)  })  })  },0)  }  }); |

##### 静态方法

自定义静态方法是通过Schema对象的statics属性给model添加方法

通过Schema对象的statics属性给 Model 添加静态方法

|  |
| --- |
| var mongoose = require('mongoose');  mongoose.connect("mongodb://u1:123456@localhost/db1", function(err) {  if(!err){  var schema = new mongoose.Schema({ num:Number, name: String, size: String });  schema.statics.findByName = function(name,cb){  return this.find({name: new RegExp(name,'i')},cb);  }  var MyModel = mongoose.model('MyModel', schema);  var doc1 = new MyModel({ name:'doc1', size: 'small' });  var doc2 = new MyModel({ name:'doc2', size: 'small' });  var doc3 = new MyModel({ name:'doc3', size: 'big' });  doc1.save();  doc2.save();  doc3.save();  setTimeout(function(){  MyModel.findByName('doc1',function(err,docs){  //[ { \_id: 5971e68f4f4216605880dca2,name: 'doc1',size: 'small',\_\_v: 0 } ]  console.log(docs);  })  },0)  }  }); |

##### 查询方法

自定义查询方法通过schema对象的query属性，给model添加查询方法

|  |
| --- |
| var mongoose = require('mongoose');  mongoose.connect("mongodb://u1:123456@localhost/db1", function(err) {  if(!err){  var schema = new mongoose.Schema({ age:Number, name: String});  schema.query.byName = function(name){  return this.find({name: new RegExp(name)});  }  var temp = mongoose.model('temp', schema);  temp.find().byName('huo').exec(function(err,docs){  //[ { \_id: 5971f93be6f98ec60e3dc86c, name: 'huochai', age: 27 },  // { \_id: 5971f93be6f98ec60e3dc86e, name: 'huo', age: 30 } ]  console.log(docs);  })  }  }); |

#### 增

增加记录有3种方法

##### save

参见[document](#document)

##### create

向model指定的表中增加2条数据

|  |
| --- |
| Model.create({**name**:**'shuyan'**,**age**:24}, {**name**:**'shuyan'**,**age**:24},(err, doc) => {  ***console***.log(doc); }); |

##### insertMany

向model指定的表中增加2条数据

|  |
| --- |
| Model.insertMany([{**name**:**'shuyan'**,**age**:24}, {**name**:**'shuyan'**,**age**:26}],(err, docs) => {  ***console***.log(docs); }); |

#### 查

##### find

根据查询条件查询所有记录

###### 参数

第一个参数表示查询条件

第二个参数用于控制返回的字段

第三个参数用于配置查询参数

第四个参数是回调函数，回调函数的形式为function(err,docs){}

###### 例

|  |
| --- |
| *//查询age>18的记录，只显示name字段，跳过第1条记录* Model.find({**age**:{**$gte**:18}},{**name**:1},{**skip**:1},(err,docs)=>{  ***console***.log(docs); }); *//查询所有记录，只显示name字段，跳过第1条记录* Model.find(**null**,{**name**:1,**\_id**:0},{**skip**:1},(err,docs)=>{  ***console***.log(docs); }); |

注意：

* 默认\_id字段会输出，如果不想输出，在第2个参数中，将这个字段设为0
* 需要使用后面的参数不需要使用前面参数时，前面参数设为null

##### findById

根据ID查询

###### 参数

第一个参数表示ID

第二个参数用于控制返回的字段

第三个参数用于配置查询参数

第四个参数是回调函数，回调函数的形式为function(err,doc){}

###### 例

|  |
| --- |
| Model.findById(**"5ad98bd5e8f8280de1431bae"**,{**name**:1},(err,doc)=>{  ***console***.log(doc); }); |

等效于

|  |
| --- |
| Model.findById(**"5ad98bd5e8f8280de1431bae"**,{**name**:1}).exec((err,doc)=>{  ***console***.log(doc); }); |

##### findOne

根据查询条件返回第一条记录

###### 参数

第一个参数表示查询条件

第二个参数用于控制返回的字段

第三个参数用于配置查询参数

第四个参数是回调函数，回调函数的形式为function(err,docs){}

###### 例

|  |
| --- |
| *//查询age>18的记录，只显示name字段，跳过第1条记录的第一条记录* Model.findOne({**age**:{**$gte**:18}},{**name**:1},{**skip**:1},(err,docs)=>{  ***console***.log(docs); }); |

##### 常用查询条件

$or　　　　或关系

$nor　　　 或关系取反

$gt　　　　大于

$gte　　　 大于等于

$lt　　　　小于

$lte　　　 小于等于

$ne　　　　不等于

$in　　　　在多个值范围内

$nin　　　 不在多个值范围内

$all　　　 匹配数组中多个值

$regex　　 正则，用于模糊查询

$size　　　匹配数组大小

$maxDistance　范围查询，距离（基于LBS）

$mod　　　　取模运算

$near　　　 邻域查询，查询附近的位置（基于LBS）

$exists　　 字段是否存在

$elemMatch　匹配内数组内的元素

$within　　　范围查询（基于LBS）

$box　　　　 范围查询，矩形范围（基于LBS）

$center　　　范围醒询，圆形范围（基于LBS）

$centerSphere　范围查询，球形范围（基于LBS）

$slice　　　　查询字段集合中的元素（比如从第几个之后，第N到第M个元素

##### $where

更复杂的查询可以使用$where，它可以使用js表达式字符串或函数作为查询一部分

###### 字符串

|  |
| --- |
| temp.find({$where:"obj.x == obj.y"},function(err,docs){  console.log(docs);  }) |

###### 函数

|  |
| --- |
| temp.find({$where:function(){  return obj.x !== obj.y;  }},function(err,docs){  console.log(docs);  }) |

##### 查询后处理

###### sort

* 作用：排序
* 例

按index升序，age降序排列

|  |
| --- |
| Model.find().sort(**'index -age'**).exec((err,docs)=>{  ***console***.log(docs); }); |

###### skip

* 作用：跳过几条记录

###### limit

* 作用：显示几条记录

###### select

* 作用：显示哪些字段，不显示哪些字段
* 例

显示name，不显示\_id

|  |
| --- |
| Model.find().select(**'name -\_id'**).exec((err,docs)=>{  ***console***.log(docs); }); |

###### count

* 作用：统计查询结果数量
* 例

|  |
| --- |
| Model.find().count((err,count)=>{  ***console***.log(count); }) |

###### distinct

* 作用：去重
* 例

将查询结果中age字段值去重后输出

|  |
| --- |
| Model.find().distinct(**'age'**,(err,distinct)=>{  ***console***.log(distinct); }) |

#### 改

##### update()

###### 参数

* 第一个参数conditions为查询条件
* 第二个参数doc为需要修改的数据
* 第三个参数options为控制选项

safe (boolean)： 默认为true。安全模式。

　　upsert (boolean)： 默认为false。如果不存在则创建新记录。

　　multi (boolean)： 默认为false。是否更新多个查询记录。

　　runValidators： 如果值为true，执行Validation验证。

　　setDefaultsOnInsert： 如果upsert为true，在新建时插入文档定义的默认值。

　　strict (boolean)： 以strict模式进行更新。

　　overwrite (boolean)： 默认为false。禁用update-only模式，允许覆盖记录。

* 第四个参数是回调函数，update的回调函数不能省略

###### 例

|  |
| --- |
| Model.update({**age**:{**$gt**:18}},{**age**:24},{**multi**:**true**},(err,raw)=>{  ***console***.log(raw); }); |

###### 注意

update()方法中的回调函数不能省略，否则数据不会被更新。如果回调函数里并没有什么有用的信息，则可以使用exec()简化代码

|  |
| --- |
| Model.update({**age**:{**$gt**:18}},{**age**:24},{**multi**:**true**}).exec(); |

##### updateMany()

updateMany与update唯一的区别在于，无论option中的multi设置为什么，都会更新多记录

##### find()+save()

如果需要更新的操作比较复杂，可以使用find()+save()方法来处理，比如找到年龄小于30岁的数据，名字后面添加'30'字符

|  |
| --- |
| temp.find({age:{$lt:20}},function(err,docs){  console.log(docs);  docs.forEach(function(item,index,arr){  item.name += '30';  item.save();  })  console.log(docs);  }); |

##### updateOne()

updateOne与update唯一的区别在于，无论option中的multi设置为什么，都会只更新一记录

##### findOne+save()

要实现updateOne的功能，但是操作很复杂时，使用该方法

#### 删

##### remove

删除所有匹配到的记录

###### 参数

第1个参数：条件

第2个参数：回调函数：(err,doc)=>{}，remove的回调函数不能省略

###### 例

|  |
| --- |
| Model.remove({**name**:**'shuyan'**},(err,raw)=>{  ***console***.log(raw); }) |

##### findOneAndRemove

回调函数不能省略

##### findByIdAndRemove

### koa-convert（generator代码转async）

#### 介绍

koa1中使用generator，但是在koa2中使用async、await，并计划在koa3中取消对generator的支持，但是有的人还是习惯使用generator，所有出现了这个模块

这个模块用于将generator语法的代码转换为async写法的代码

#### 例

|  |
| --- |
| const convert = require('koa-convert');  app.use(convert(bodyparser));  app.use(convert(json()));  app.use(convert(logger())); |

### koa-json（js对象转换为漂亮json字符串）

* 作用

需要将js对象转换为json字符串时，该模块的中间件会自动将js对象转换为漂亮格式的字符串

* 安装

npm install koa-json --save

* 用法
  + 以默认参数使用

|  |
| --- |
| app.use(json()); |

此时始终开启转换功能

* + 也可以指定参数使用

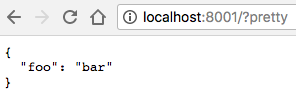
|  |
| --- |
| app.use(json({**pretty**:**false**,**param**:**'pretty'**})); |

此时：

如果http请求没有携带pretty参数，则本次响应不使用转换功能



如果http请求携带了pretty参数，则本次响应使用转换功能



* 参数
  + pretty

http请求不携带param参数指定的字符串时，是否启用转换功能

默认：true

* + param

当pretty参数为false时，http请求携带什么字符串时启动转换功能

默认：none

注意：http请求只携带字符串即可，无需指定该参数的值

* + spaces

漂亮格式中使用几个空格

### koa2-cors（跨域）

* 作用

用于处理[跨域](#跨域)问题

* 安装

npm install koa-cors --save

* 用法
  + 以默认参数使用

|  |
| --- |
| app.use(cors()); |

此时所有参数均使用默认值

* + 也可以指定参数使用

|  |
| --- |
| app.use(cors({  origin: **function**(ctx) {  *//跨多域名：判断本次请求的origin是否在允许的域名内  //跨本站点内多url：判断本次请求的url是否在允许的url内* **if**((origins.indexOf(ctx.**header**.**origin**) != -1) && (urls.indexOf(ctx.**url**) != -1)){  **return** ctx.**header**.**origin**;  }  },  **exposeHeaders**: [**'Content-Length'**,**'Date'**],  **maxAge**: 3600\*24,  **credentials**: **true**,  **allowMethods**: [**'GET'**, **'POST'**,**'PUT'**],  **allowHeaders**: [**'key1'**,**'key2'**] })); |

* 参数

所有参数都是用来设置[CORS头字段](#CORS头字段)中的响应头的，头字段含义参见链接

* + origin

用于配置Access-Control-Allow-Origin响应头

可以是字符串或“function(ctx)=>{}”

默认值为将过来的请求的请求头中的Origin字段值原封不动返回

* + exposeHeaders

用于配置Access-Control-Expose-Headers响应头

值为数组

默认不使用Access-Control-Expose-Headers响应头

* + maxAge

用于配置Access-Control-Max-Age响应头

值为数字

默认不使用Access-Control-Max-Age响应头

* + credentials

用于配置Access-Control-Allow-Credentials响应头

值为布尔值

默认不使用Access-Control-Allow-Credentials响应头

* + allowMethods

用于配置Access-Control-Allow-Methods响应头

值为数组

默认值：['GET', 'PUT', 'POST', 'PATCH', 'DELETE', 'HEAD', 'OPTIONS']

* + allowHeaders

用于配置Access-Control-Allow-Headers响应头

值为数组

默认将过来的请求的请求头中的Access-Control-Request-Headers字段值原封不动返回

### koa-range（范围请求）

* 范围请求概念

如果下载1个较大文件时，出现断网，就需要重新下载，此时就需要范围请求，每次只下载一段数据，即使断网也可以继续下载剩下的内容

范围请求需要在请求头中自定义Range字段，其值为“Byte：start-end”

如：



如果服务器支持范围请求，则返回206状态码，以及请求范围内的数据

如果服务器不支持范围请求，则返回200状态码，以及所有数据

* 作用

让服务器支持范围请求

* 安装

npm install koa-range –save

* 直接加载中间件即可，别的什么都不需要做

|  |
| --- |
| **const** range = *require*(**'koa-range'**);  app.use(range); |

### bluebird

#### 介绍

bluebird是1个promise库，相比于原生的promise功能更强

#### API

##### spread

作用与then方法相同，区别在于Promise.all().then方法指定的回调函数的参数是数组，spread方法指定的回调函数的参数是将数组展开后的多个元素

|  |
| --- |
| **const** Promise = *require*(**'bluebird'**); **const** fs = *require*(**'fs'**);  Promise.all([  fs.readFileSync(**'./package.json'**),  fs.readFileSync(**'./app.js'**) ]).spread((f1,f2)=>{  **if**(f1.toString() == f2.toString()){  ***console***.log(**'sample file'**);  }**else**{  ***console***.log(**'different file'**);  } }); *//等效于* Promise.all([  fs.readFileSync(**'./package.json'**),  fs.readFileSync(**'./app.js'**) ]).then((args)=>{  **if**(args[0].toString() == args[1].toString()){  ***console***.log(**'sample file'**);  }**else**{  ***console***.log(**'different file'**);  } }); |

##### Promise.map

* 函数

格式：Promise.map(iterator, function(item){}, {option})

作用：

遍历参数1的iterator，将每个元素都传入用参数2指定的回调函数执行1遍，iterator中所有元素遍历之后都执行成功，则执行map返回的promise执行成功，任意1个执行出错，则map返回的promise执行出错

参数：

iterator：可遍历的数据，如数组

function(item){}：遍历时需要执行的函数

{option}：设置参数

concurrency：

promise最大并发数量限制，

就是遍历对象时，如果有了该值为3，则最多只能有3个promise处于pending状态，必须有1个变为resolved状态了，下一个元素才能被遍历到。

* 例

|  |
| --- |
| **const** Promise = *require*(**'bluebird'**); **const** fs = *require*(**'fs'**);  **var** fileNames = [**'./spread.js'**,**'./package.jsaon'**,**'./package-lock.json'**];  Promise.map(fileNames, **function**(fileName) {  **return** fs.readFileSync(fileName).toString(); }).then(**function**(args) {  ***console***.log(args); }).catch((error)=>{  ***console***.log(error); });  *//等效于* **var** promises = []; **for** (**var** i = 0; i < fileNames.length; ++i) {  **let** p = **new** Promise((resolve,reject)=>{  **try** {  resolve(fs.readFileSync(fileNames[i]).toString());  }**catch**(err){  reject(err);  }  });  promises.push(p); } Promise.all(promises).then(**function**(args) {  ***console***.log(args); }).catch((error)=>{  ***console***.log(error); }); |

##### Primise.filter

* 函数

格式：Promise.filter(iterator, function(item){}, {option})

作用：

从参数1中传入的数组中筛选出一些需要保留的元素

遍历参数1的iterator，将每个元素都传入用参数2指定的回调函数执行1遍，回调函数中执行一些操作后判断当前数据是否需要保留，是返回true，不是返回false。如果所有元素遍历过程中没有出现未捕获异常，则所有元素都可以便利完成并执行filter返回的promise的resolve函数，将返回true的所有数据作为参数传入，如果便利过程中出现未捕获异常，则立即停止遍历，并调用filter返回的promise的reject函数

参数：

iterator：可遍历的数据，如数组

function(item){}：遍历时需要执行的函数

{option}：设置参数

concurrency：

promise最大并发数量限制，

就是遍历对象时，如果有了该值为3，则最多只能有3个promise处于pending状态，必须有1个变为resolved状态了，下一个元素才能被遍历到。

* 例

|  |
| --- |
| **const** Promise = *require*(**'bluebird'**); **const** fs = *require*(**'fs'**);  **var** fileNames = [**'./spread.js'**,**'./package.jsaon'**,**'./package-lock.json'**];  Promise.filter(fileNames, **function**(fileName) {  **try**{  fs.readFileSync(fileName).toString();  **return true**;  }**catch** (e) {  **return false**;  } }).then(**function**(args) {  ***console***.log(args); }).catch((error)=>{  ***console***.log(error); }); |