使用express 和js开发一个博客系统

nodejs介绍和安装

nodejs介绍

node.js 是一个基于 Chrome v8 引擎的 javascript 运行环境。Node.js 使用了一个事件驱动、非阻塞式 I/O 的模型，使其轻量又高效。Node.js 提供的包管理器 npm，成为世界上最大的开放源代码的生态系统。

简单说：用于 js 开发服务端程序

特点： 单线程、异步、事件驱动。

学完 nodejs 我们就可以做后端！加上js本可以做前端，最后我们就可以做一个完整的网站。

nodejs 官网 https://nodejs.org/zh-cn/

nodejs 可以做什么

node.js 可以解析 js 代码（没有浏览器安全级别的限制），提供很多系统级别的 api，如：文件的读写、进程的管理、网络通信。。。。

简单来说是让js的执行脱离了浏览器。

下载和安装

下载地址 https://nodejs.org/zh-cn/download/ **LTS 为长期稳定版，下载这个即可。安装直接双击 nodejs.exe,疯狂下一步即可！**

安装完成 重启 vscode 或者系统，然后进入终端 执行 node -v 命令，查看 nodejs 安装版本号，如果输出版本号说明安装成功，反之安装失败。（如果安装失败，建议卸载，然后重启电脑-重新安装）

使用终端中验证

1. 打开终端，有两种方法，任选其一就可
    1. 在 vscode 选中任意一个文件，右键 在终端中打开。**注意：安装nodejs以后最好重启一下vscode！**
    2. 打开 任意一个文件所在的文件夹，在文件夹的地址栏中输入 cmd
2. 在终端中 node -v

node 环境下执行 js 文件

在你的项目中编写一个index.js文件，里面随便写一句js代码！

创建 index01.js 文件

```
console.log("hello nodejs");
```

终端中进入 index01.js 所在目录，在终端中执行

```
node index01.js
```

nodejs 模块化开发

所谓模块就是一个文件，文件 内部封装了某个功能的对象或者是函数

定义模块

ajax.js 文件

```
var obj={
  get:function(){
  },

  post:function(){
  }

}
module.exports=obj
```

加载模块

```
var  obj=require('./ajax')
obj.get()
obj.post()
```

nodejs 第三方模块

第三方模块，我们需要使用 node 自带的 npm 这个工具 下载

npm 类似于软件商店，任何程序猿使用 js 写的的功能都可以上传到 npm 这个网站上，www.npmjs.com

当然我们可以使用下载，下载的时候需要使用 npm 命令，接下来我们学习npm 的使用

npm 的使用

1-进入项目文件夹

vscode中直接选中文件夹右键，在终端中打开

2-初始化 npm init

终端 直接输入 npm init， 自动 package.json 文件， 这个文件是来记录我们的安装的第三方模块的。

3- 本地安装 第三方模块

我们项目中需要用到的第三库模块，也叫项目依赖，也可以说是项目中依赖的模块。

命令:

```
npm i 模块名
```

比如说的咱们的项目中会用到express模块

使用命令 npm i express，安装了 lodash模块，同时把安装信息写入到了 package.json 文件， 本地安装，安装 express被下载到了 node_modules 里面

```
npm install express
```

本地安装后 在 package.json 文件中会有 记录

```
"dependencies": {
    "express": "xxxx"
  }
```

4- 项目中安装的第三方模块

```
 let express = require('express')
```

5-本地安装 开发依赖

nodemon-------监听文件的变化 重新执行 node 命令（类似前端中使用的 liveserver）

想要使用 nodemon 可以配合 npm 脚本使用，也可以 直接全局安装 nodemon 使用

5-1 配合 本地安装 配合 npm 脚本使用

nodemon 是一个监听 js 文件的变化，如果 js 文件改变了，就会重新执行 js 文件

npm i nodemon -D 开发依赖 （工具） npm i nodemon --save-dev

```
 "devDependencies": {
    "nodemon": "^2.0.15"
  }
```

配合 npm 脚本

```
"scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "aaa": "nodemon index.js",
    "start": "nodemon index.js"
  }
```

终端中输入 npm run aaa 或者 npm start

5-2 直接全局安装使用

npm i nodemon -g

全局安装后直接 在终端中可以 输入 nodemon index.js

6-其他命令 npm i

自动检查 package.json 依赖 自动下载。

7-npm 下载源

直接使用 npm 下载速度可能比较慢

npmmirror.com是一个完整 `npmjs.org` 镜像，你可以用此代替官方版本(只读)，同步频率目前为 **10分钟** 一次以保证尽量与官方服务同步

更改 npm 的下载地址，下面两种任选一种就可以

1-修改npm 的下载地址

```
npm config set registry https://registry.npmmirror.com/
```

查看当前 npm 的下载源

```
npm config get registry
```

以后我们使用 npm 安装模块就直接去npmmirror下载了

2- cnpm

```
npm install -g cnpm --registry=https://registry.npmmirror.com
```

如果采用第二种，我们以后使用cnpm命令咱就可以了

cnpm i express

express框架的使用

express介绍

Express 介绍Express 是一个简洁而灵活的 node.js Web应用框架, 提供了一系列强大特性帮助你创建各种

Web 应用，和丰富的 HTTP 工具。使用 Express 可以快速地搭建一个完整功能的网站。

Express 框架核心特性：可以设置中间件来响应 HTTP 请求。

定义了路由表用于执行不同的 HTTP 请求动作。

可以通过向模板传递参数来动态渲染 HTML 页面。

express不对node.js本身的特性进行二次抽象 而是在基本功能上进行扩充

express完全是由路由和中间件构成的框架

从本质上来说一个express应用就是为了调用各种中间件

简单的说express可以很快速的让我们使用mvc的方式创建一个web应用（前后端可以分离，也可以不分离）

https://expressjs.com/ 英文官网

http://www.expressjs.com.cn/ 中文官网

express 创建服务器

新建项目文件夹 expressDemo1，

并且进入项目，使用npm init -y 初始化项目

本地安装 npm i express -D

```
const express = require('express');
const app = express();
//客户端发起get请求  如果请求的路径是/，返回hello world字符串
app.get('/', (req, res) =>
     res.send('Hello World!哈哈')
)
//启动服务器 然后我就可以通过IP地址和端口号访问这台服务器了   127.0.0.1:3000 或者
localhost:3000  或者自己电脑的ip:3000
app.listen(3000, () =>
     console.log('Example app listening on port 3000!')
)
```

终端中输入 node app.js 就可以启动项目

在浏览器中输入 localhost:3000 就发起了一个get请求，就能看到服务器返回的数据

服务器的主要作用就是来处理前端不同的请求，返回不同的数据，路由和中间件就是做这个事情的

路由和中间件

express完全是由路由和中间件构成的框架，从本质上来说一个express应用就是为了监听不同的路径调用各种中间件

中间件本质就是一个函数，也可以叫插件，只不过express里面我们更习惯叫中间件

路由

路由：前端访问不同的地址，后端要返回不同的数据，这就叫后端路由

如果/users/login 接口是登录用的，/users/reg 接口是注册用的。

监听不同的路径执行不同的函数，从而来处理不同的请求

这个路由一般是在app上使用的，一般我们也叫应用级别的路由！

通过路由挂载中间件

app.use(中间件监听的路径，中间件函数)

use里面面可以设置两个参数

第一个参数设置的路径，不传参数，默认是* 匹配任何路径

第二个参数是回调函数，如果匹配到路径，就会执行后的函数。很多时候后面都是第三方封装好的一个函数，这个函数我们称之为中间件。

请求的路径如果匹配到了路径，就会执行后面的函数

```
app.use （path，fn）

前端发起请求的路径匹配一个path的时候，就会执行后面的函数fn
-path称之为路径
-fn称之为中间件函数
-app.use我们可以让不同的路径（接口地址）执行不同的函数，这个叫做路由(注意use可以监听不同的请求方式，get-post等等)
app.use （'*'，fn1） // 和后面等效 app.use （fn1）
app.use （'/about-me'，fn1）
app.use （'/login'，fn2）
```

路由的匹配顺序时从上到下，如果我们一开始使用了`*`，那么匹配到的到了`*`下面的路径就不会执行了。

所以我们现在不用`*`,后面的我们会将到怎么避免这种情况。

http请求的方式很多，use可以监听到所有方式的请求。

除了use，也可以专门的去监听某种请求

app.get 可以监听get请求

app.post 可以监听post请求

app.put可以监听put请求

app.patch可以监听patch请求

app.delete可以监听delete请求

中间件

中间件本质就是一个函数

中间件函数中有三个形参

1-req就是请求，

2-res是服务器的相应，

3-next一旦调用，中间件会把请求交给下一个中间件去处理，如果不调用next请求到这里就结束了，不会把请求传递给其他的中间件了

test.js

```
// 中间件
function test(req,res,next){
    console.log('这是自定义中间件')
    console.log(req.url,'请求经过了test中间件');
    // 在中间件可以可以获取的前端请求的数据req
    // 如果不调用next() 那么后面的挂载的中间件就不会执行
    next()

}
module.exports = test;
```

app.js 中使用test中间件

```
// 加载模块
const test = require('./test')
// 挂载中间件
// 第一个参数不传，所有的路由，所有的请求都会经过这个中间件
app.use(test)
// 第一个参数为路径，只有包含这个路径的请求，才会经过这个中间件
app.use("/a",test)
```

启动项目，然后访问试试localhost:3000/a ，test中间件会调用两次，访问localhost:3000会调用test一次

路由中间件（专门处理路由的中间件）

路由中间件 是express内置的一个中间件。

前端调用了不同的接口，后端要做对应的处理。这里我们使用express自带的路由中间件帮我们处理不同的请求。

作用：

后端的接口提供的功能往往有很多，比如说 用户相关的操作（登录，注册），比如 文章相关的操作（发布文章，删除文章，修改文章，文章列表，文章详情等等）

这些如果都写在一个js文件中，代码量会比较大，所以我们可以借助路由中间件将 用户和文章相关的操作分散在不同的js文件中

这种路由主要对应用级别的路由，做了一个更加细分的处理，我们也叫路由的路由。或者叫路由级别的路由

获取路由中间件

新建routes/users.js

```
var router = express.Router()
```

处理get请求

router.get(监听请求的路径，处理回调函数)

处理post请求

router.post(监听请求的路径，处理回调函数)

导出router对象

```
module.exports=router
```

在app中使用

```
引入 router.js

app.use('/users',router)
```

完整如下

```
var router = express.Router();
// 监听get请求 /users/login
router.get('/login', function(req, res, next) {
    //   res.send('login');//返回普通文本
      res.json({code:1})//返回json数据
    });
    // 监听post请求 /users/register
router.post('/register', function(req, res, next) {
    //   res.send('reg');
      res.json({code:1})//返回json数据
```

```
});
// 监听get请求 /users/login
// 监听post请求 /users/register
app.use('/users',router)
```

测试 请求 /users/login 和 /users/register 看是否能返回正确的数据

利用中间件，我们可以把项目中每个功能的代码拆分到不同的文件中，不至于在app.js文件中写太多的代码。

新建routes/users.js 在处理 用户账号相关的请求

```
var express = require('express');
var router = express.Router();

/* 使用路由监听用户的请求 */
// 监听 /users
router.get('/', function(req, res, next) {
  res.send('respond with a resource');
});
// 监听get请求 /users/login
router.get('/login', function(req, res, next) {
//   res.send('login');//返回普通文本
  res.json({code:1})//返回json数据
});
// 监听post请求 /users/register
router.post('/register', function(req, res, next) {
//   res.send('reg');
  res.json({code:1})//返回json数据

});

module.exports = router;
```

路由写完，我们得让express服务器使用

app.js 添加代码

```
// 获取用户路由
let usersRouter = require('./routes/users')
// 监听 /users 路径，当用户访问/users路径的时候，使用usersRouter 处理
app.use('/users',usersRouter)
```

测试 发起get请求 /users/login 和 post请求 /users/register 看是否能返回正确的数据

get请求参数的获取

req.query

```
// 监听get请求 /users/login
router.get("/login", function (req, res, next) {

  console.log(req.query)//获取get请求的query参数 {username：aaa,password}

  res.json({ code: 1 }); //返回json数据
});
```

测试请求 localhost:3000/users/login?username=aaa&password=111

打印结果 { username: 'aaa', password: '111' }

get请求的另外一种传参

```
router.get("/test/:id", function (req, res, next) {
    //获取get请求的url中的参数
    // /users/test/12
    //{ id: '12' }
    console.log(req.params)

    res.json({ code: 1 }); //返回json数据
});
```

请求 /users/test/12，得到结果{ id: '12' }

post参数的获取

post请求参数的获取需要使用中间件express.urlencoded 解析req.body

app.js

```
app.use(express.urlencoded());//用来解析 x-www-form-urlencoded类型请求体
app.use(express.json());//用来解析 json类型请求体
```

users.js

```
// 监听post请求 /users/register
router.post("/register", function (req, res, next) {

  console.log(req.body)
  res.json({ code: 1 }); //返回json数据
})
```

测试post请求 /users/register

传入x-www-form-urlencoded类型的参数 username：111

打印的结果 {username:"1111"}

传入json类型的数据试试

静态资源托管中间件

```
打开一个网站，我们可能访问的 html页面，页面中会加载css，js文件，图片
```

html，css，js文件，图片等都叫静态资源。

我们网页中发起请求，请求接口返回的数据叫动态资源。
静态资源一般不需要做处理，直接就返回给浏览器
动态资源一般后端会处理，比如过解析前端的请求过来的url和参数，根据url和参数不同返回不同的json数据

使用express很容易就能创建一个静态资源服务器
express.static()

app.js

```
// 静态资源服务

// 将当前的文件夹下面的public文件夹设置为静态资源文件夹
// 一旦有请求过来，会先到请求资源文件夹里面查找看看有没有静态文件，有的话就直接返回静态文件
// 比如 访问 localhost:3000/1.js  就会去 public下面找1.js文件
app.use(express.static('public'));
```

在根目录下面创建public文件夹，里面可以存放静态资源 css，js，html，图片等资源

express脚手架创建项目

我们已经使用了好几个中间件了，express内置了很多中间件，我们一个一个配置比较麻烦，并且做一个后端项目，我们可以需要一些别的中间件，比如说日志插件，错误处理插件等等，需要考虑的比较多，这里我们借助于工具去创建一个通用的项目模板，就能省下这个配置过程

通过应用生成器工具 express-generator 可以快速创建一个应用的骨架。express-generator 包含了 express 命令行工具，也称脚手架。

通过如下命令即可安装

```
npm install express-generator -g
```

创建项目

```
express -e  express-demo3
```

-e是指定了项目中使用的模板引擎

提示如下内容，说明已经创建好一个项目骨架了

```
   create : express-demo3\
   create : express-demo3\public\
   create : express-demo3\public\javascripts\
   create : express-demo3\public\images\
   create : express-demo3\public\stylesheets\
   create : express-demo3\public\stylesheets\style.css
   create : express-demo3\routes\
   create : express-demo3\routes\index.js
   create : express-demo3\routes\users.js
   create : express-demo3\views\
   create : express-demo3\views\error.ejs
   create : express-demo3\views\index.ejs
   create : express-demo3\app.js
   create : express-demo3\package.json
   create : express-demo3\bin\
   create : express-demo3\bin\www


   change directory:
     $ cd express-demo3


   install dependencies:
     $ npm install


   run the app:
     $ DEBUG=express-demo3:* npm start
```

创建好以后，按照提示，需要进入项目，安装依赖，然后启动项目

创建好的骨架中包含的很多东西目录介绍

```
   create : project_name\public\   存放静态资源css，图片等
   create : project_name\routes\   存放路由模块
   create : project_name\views\     存放html模板
   create : project_name\app.js     express 配置
   create : project_name\package.json
   create : project_name\bin\www   服务器启动配置
```

app.js 里放express 的配置，是我们需要关注的，大部分都已经生成完毕。我们需要添加和改动的是路由配置

```
// http错误中间件
var createError = require('http-errors');
var express = require('express');
// 生成路径
var path = require('path');
// 解析cookie中间件
```

```
var cookieParser = require('cookie-parser');
// 日志
var logger = require('morgan');

// 路由配置
var indexRouter = require('./routes/index');
var usersRouter = require('./routes/users');

var app = express();

// view engine setup 视图（模板）引擎，在views文件夹中查询html模板
app.set('views', path.join(__dirname, 'views'));
// 模板类型 esj
app.set('view engine', 'ejs');

// 日志中间件
app.use(logger('dev'));
// 解析json 中间件
app.use(express.json());
// 解析 请求体 中间件
app.use(express.urlencoded({ extended: false }));
// 解析cookie 中间件
app.use(cookieParser());
// 静态资源中间件，设置静态资源根目录
app.use(express.static(path.join(__dirname, 'public')));

// 挂载 / 路由
app.use('/', indexRouter);
// 挂载/users路由
app.use('/users', usersRouter);

// 捕获错误
// catch 404 and forward to error handler
app.use(function(req, res, next) {
    next(createError(404));
});
// 错误处理
// error handler
app.use(function(err, req, res, next) {
    // set locals, only providing error in development
    res.locals.message = err.message;
    res.locals.error = req.app.get('env') === 'development' ? err : {};

    // render the error page
    res.status(err.status || 500);
        // 渲染error模板
    res.render('error');
});

module.exports = app;//导出模块
```

这里使用了一个模板引擎，一会儿来我们看看什么是模板引擎

前后端的爱恨情仇

未分离时代

MVC 是一种经典的设计模式，全名为 Model-View-Controller，即 模型-视图-控制器。　　大致就是：模型和视图需要通过 控制器 来进行粘合。例如，用户发送一个 HTTP 请求，此时该请求首先会进入控制器，然后控制器去获取数据并将其封装为模型，最后将模型传递到视图中进行展现。　　需要说明的是，这个View还可以采用 Velocity、Freemaker 等模板引擎。使用了这些模板引擎，可以使得开发过程中的人员分工更加明确，还能提高开发效率。

从MVC到前后端分离

　　前后端分离
前端负责开发页面，通过接口（Ajax）获取数据，采用Dom操作对页面进行数据绑定，最终是由前端把页面渲染出来。

前后端分离与耦合架构

　　前后端分离“已经成为互联网项目开发的业界标杆，通过Tomcat+Ngnix(也可以中间有个Node.js)，有效地进行解耦。并且前后端分离会为以后的大型分布式架构、弹性计算架构、微服务架构、多端化服务（多种客户端，例如：浏览器，车载终端，安卓，IOS等等）打下坚实的基础。

　　前后端分离(解耦)的核心思想是：前端Html页面通过Ajax调用后端的RestFul API并使用Json数据进行交互。

　　在互联网架构中,一般有两种服务器

　　web服务器：一般指像nginx，apache这类的服务器，他们一般只能解析静态资源。　　应用服务器：一般指像tomcat，jetty，resin这类的服务器可以解析动态资源也可以解析静态资源，但解析静态资源的能力没有web服务器好。(　一般只有Web服务器才能被外网访问，应用服务器只能内网访问。)

为什么前后端分离?

　　一般公司后端开发人员直接兼顾前端的工作，一边实现API接口，一边开发页面，两者互相切换着做，而且根据不同的url动态拼接页面，这也导致后台的开发压力大大增加。前后端工作分配不均。不仅仅开发效率慢，而且代码难以维护。而前后端分离的话，则可以很好的解决前后端分工不均的问题，将更多的交互逻辑分配给前端来处理，而后端则可以专注于其本职工作，比如提供API接口，进行权限控制以及进行运算工作。而前端开发人员则可以利用nodejs来搭建自己的本地服务器，直接在本地开发，然后通过一些插件来将api请求转发到后台，这样就可以完全模拟线上的场景，并且与后台解耦。前端可以独立完成与用户交互的整一个过程，两者都可以同时开工，不互相依赖，开发效率更快，而且分工比较均衡

从经典的JSP+Servlet+JavaBean的MVC时代，到SSM（Spring + SpringMVC + Mybatis）和SSH（Spring + Struts + Hibernate）的Java 框架时代，再到前端框架（KnockoutJS、AngularJS、vueJS、ReactJS）为主的MV*时代，然后是Nodejs引领的全栈时代，技术和架构一直都在进步。虽然“基于NodeJS的全栈式开发”模式很让人兴奋，但是把基于Node的全栈开发变成一个稳定，让大家都能接受的东西还有**很多路要走**。创新之路不会止步，无论是前后端分离模式还是其他模式，都是为了更方便得解决需求，但它们都只是一个“中转站”。前端项目与后端项目是两个项目，放在两个不同的服务器，需要独立部署，两个不同的工程，两个不同的代码库，不同的开发人员。前端只需要关注页面的样式与动态数据的解析及渲染，而后端专注于具体业务逻辑。

模板引擎-前后分离

EJS什么是模板引擎

模板引擎（Template Engine）是一个将页面模板和要显示的数据结合起来生成 HTML 页面的工具。 如果说上面讲到的 express 中的路由控制方法相当于 MVC 中的控制器的话，那模板引擎就相当于 MVC 中的视图。模板引擎的功能是将页面模板和要显示的数据结合起来生成 HTML 页面。它既可以运 行在服务器端又可以运行在客户端，大多数时候它都在服务器端直接被解析为 HTML，解析完成后再传输给客户端，因此客户端甚至无法判断页面是否是模板引擎生成的。有时候模板引擎也可以运行在客户端，即浏览器中。目前的主流还是由服务器运行模板引擎。 在 MVC 架构中，模板引擎包含在服务器端。控制器得到用户请求后，从模型获取数据，调用模板引擎。模板引擎以数据和页面模板为输入，生成 HTML 页面，然后返回给控制器，由控制器交回客户端。

简单来说，就是使用模板引擎可以产生html页面，后端直接产生页面，前端页面和后端程序混合在了一起（前后端不分离）

使用模板引擎

前面我们通过以下两行代码设置了模板文件的存储位置和使用的模板引擎

```
app.set('views', __dirname + '/views');
app.set('view engine', 'ejs');
```

通过调用 res.render() 渲染模版，并将其产生的页面直接返回给客户端。它接受两个参数，第一个是模板的名称，即 views 目录下的模板文件名，扩展名 .ejs 可选。第二个参数是传递给模板的数据对象，用于模板翻译。

routes/index.js

```
/* GET home page. */
router.get('/', function(req, res, next) {
  //get请求 / 路径时候 使用 { title: 'Express' } 渲染 views目录下的模板 index模板，把渲染后的数据 返回给客户端
  res.render('index', { title: 'Express' });
});
```

其实就是把一个对象传递给index.ejs文件，得到一个html字符串，把html字符串返回给了前端

index.ejs

```
<!DOCTYPE html>
<html>
  <head>
    <title><%= title %></title>
    <link rel='stylesheet' href='/stylesheets/style.css' />
  </head>
  <body>
    <h1><%= title %></h1>
    <p>Welcome to <%= title %></p>
  </body>
</html>
```

当我们 res.render('index', { title: 'Express' }); 时，模板引擎会把 <%= title %> 替换成 Express，然后把替换后的页面显示给用户。渲染后生成的页面代码为：

```
<!DOCTYPE html>
<html>
  <head>
    <title>Express</title>
    <link rel='stylesheet' href='/stylesheets/style.css' />
  </head>
  <body>
    <h1>Express</h1>
    <p>Welcome to Express</p>
  </body>
</html>
```

ejs 语法

ejs 的标签系统非常简单，它只有以下三种标签：

<% code %>：code是我们的JavaScript 代码。

<%= code %>：显示替换过 HTML 特殊字符的内容。（html不会被浏览器解析）

<%- code %>：显示原始 HTML 内容。（html会被浏览器）

注意： <%= code %> 和 <%- code %> 的区别，当变量 code 为普通字符串时，两者没有区别。

当 code 比如为

hello

这种字符串时，<%= code %> 会原样输出

hello

，

而 <%- code %> 则会显示 H1 大的 hello 字符串。hello

我们可以在 <% %> 内使用 JavaScript 代码。下面是 ejs 的官方示例：

```
arr: ['mop', 'broom', 'duster']
```

模板

```
<ul>
<% for(var i=0; i<arr.length; i++)
{%>
   <li><%= arr[i] %></li>
<% } %>
</ul>
```

渲染结果

```
<ul>
  <li>mop</li>
  <li>broom</li>
  <li>duster</li>
</ul>
```

接口开发restfulAPI-前后分离

简介

在主流公司的程序开发中，为了提高程序开发迭代的速度，基本都是前后端分离架构，而前端既包括网页、App、小程序等等，因此必须要有一个统一的规范用于约束前后端的通信，RESTful API则是目前比较成熟的API设计理论。

要想理解RESTful，就需要先明白REST。REST是 Roy T. Fielding 在其2000年的博士论文中提出的，是 `RE` presentational `S` tate `T` ransfer 词组的缩写，可以翻译为“表现层状态转移”，其实这个词组前面省略了个主语--“Resource”，加上主语后就是“资源表现层状态转移”。每个词都能看懂，连在一起不知道什么意思了有木有?

1. Resource（资源）

   所谓资源，就是互联网上的一个实体。URI（Uniform Resource Identifier）的全称就是统一资源标识符，和我们这里的资源一个意思。一个资源可以是一段文字、一张图片、一段音频、一个服务。

2. 表现层（Representation）

   "资源"是一种信息实体，它可以有多种外在表现形式。我们把"资源"具体呈现出来的形式，叫做它的"表现层"（Representation）。比如一篇文章，可以使用XML、JSON、HTML的形式呈现出来。

3. 状态转移（State Transfer）

   访问一个网站，就代表了客户端和服务器的一个互动过程。在这个过程中，势必涉及到数据和状态的变化。互联网通信协议HTTP协议，是一个无状态协议，这意味着所有的状态都保存在服务器端。因此，如果客户端想要操作服务器，必须通过某种手段，让服务器端发生"状态转化"（State Transfer）。而这种转化是建立在表现层之上的，所以就是"表现层状态转化"。

上面我们介绍了REST的基本概念，那么服务REST规范的设计，我们就可以称为RESTful。接下来我们就来看一下RESTful API的设计规范。

域名

`API` 的根入口点应尽可能保持足够简单，这里有两个常见的例子：

- api.example.com/* （子域名下）
- example.com/api/* （主域名下）

域名应该考虑拓展性，如果不确定API后续是否会拓展，应该将其放在子域名下，这样可以保持一定的灵活性。

路径

路径又称为端点，表示API的具体地址。在路径的设计中，需遵守下列约定：

- 命名必须全部`小写`
- 资源（resource）的命名必须是`名词`，并且必须是`复数形式`
- 如果要使用连字符，建议使用‘-’而不是‘’，“字符可能会在某些浏览器或屏幕中被部分遮挡或完全隐藏
- 易读

下面是一些反例：

- api.example.com/getUser
- api.example.com/addUser

下面是一些正例：

- api.example.com/zoos
- api.example.com/zoos/animal...

HTTP动词

对于如何操作资源，有相应的HTTP动词对应，常见的动词有如下五个（括号里表示SQL对应的命令）：

- GET（SELECT）：从服务器取出资源（一项或多项）
- POST（CREATE）：在服务器新建一个资源
- PUT（UPDATE）：在服务器更新资源（客户端提供改变后的完整资源）
- PATCH（UPDATE）：在服务器更新资源（客户端提供改变的属性）
- DELETE（DELETE）：从服务器删除资源

示例：

HTTP动词	路径	表述
GET	/zoos	获取所有动物园信息 /zoos?pagesize=10&pagenum=1
POST	/zoos	新建一个动物园-参数请求体中
GET	/zoos/:id	获取指定动物园的信息 接口文档一般这样写 /zoos/1
PUT	/zoos/:id	更新指定动物园的信息（前端提供该动物园的全部信息）/zoos/1,参数在请求体中
PATCH	/zoos/:id	更新某个指定动物园的信息（提供该动物园改动部分的信息）/zoos/1,参数在请求体中
DELETE	/zoos/:id	删除某个动物园 /zoos/1
GET	/zoos/:id/animals	获取某个动物园里面的所有动物信息

只要需要ID的，一般接口文档会这样写

get 请求 /zoos/:id ，使用的时候 /zoos/1 ，请求方式是使用get，意思是获取id为1的动物园的信息

不太标准的写法 /zoos?id=1

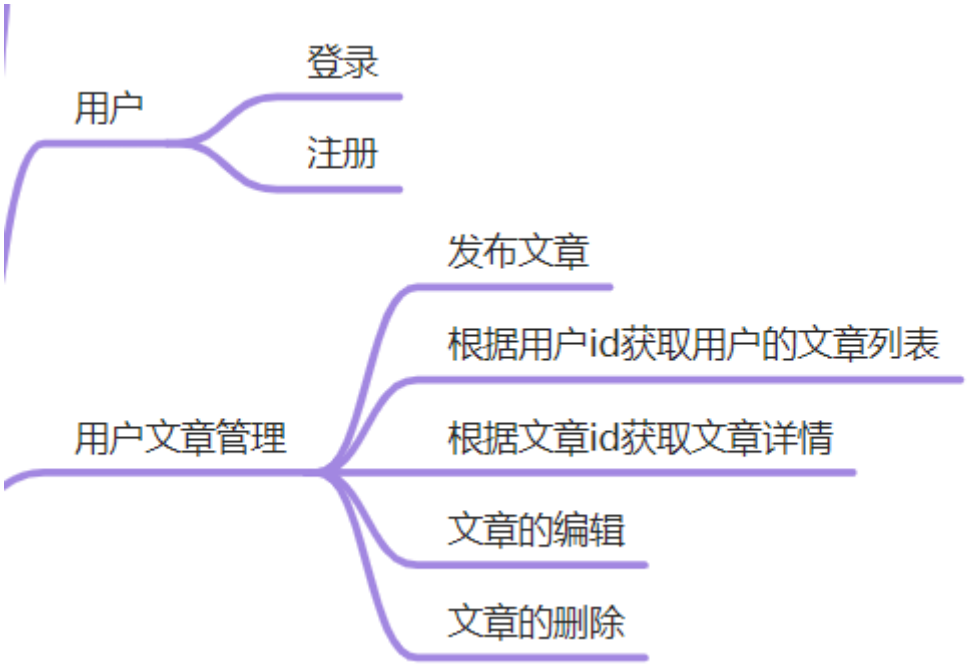
put请求 /zoos/:id ，使用的时候 /zoos/1 ，请求方式是使用put，参数放入请求体中，意思是更新d为1的动物园的信息

PATCH请求 /zoos/:id ，使用的时候 /zoos/1 ，请求方式是使用PATCH，参数放入请求体中，意思是更新id为1的动物园的信息

DELETE请求 /zoos/:id ，使用的时候 /zoos/1 ，请求方式是使用delete，，意思是删除id为1的动物园的信息

注册-登录-文章接口

注册-登录-文章相关



实现步骤

1-脚手架创建项目

2-安装依赖

3-配置路由

```
app.use('/api/users',...)
app.use('/api/aritcles',...)
```

创建routes/users.js

注册

```
请求方式 post
地址 /users/
参数 username  password  nickname
```

登录

```
请求方式get
地址 /api/users/
参数 username  password
```

创建routes/articles.js

用户发布文章

```
方式post
地址/api/articles
参数userid  title  content
```

查看某个用户的所有文章

```
方式get
地址/api/articles/users/:uid
参数uid
```

根据文章id查看对应的文章信息

```
方式get
地址/api/articles/:aid
参数aid
```

删除文章

```
方式delete
地址/api/articles/:aid
参数aid
```

编辑文章

```
方式patch
地址/api/articles/:aid
参数 aid  title content
```

上传文件中间件

有些时候，我们需要向服务器上传一些图片，比如发布博客的时候。

这个时候我们可以是使用multer中间件。

前端上传图片在post的请求体中使用的form data 数据上传的，

而multer可以解析 form data 数据

安装

安装npm i multer

创建upload路由

新建配置路由专门用来上传图片

app.js

```
var uploadRouter = require("./routes/upload.js");
```

route/upload.js

```
var express = require("express");
var router = express.Router();
router.post("/",function (req, res) {
  res.json({
    code: 1,
    msg: "上传成功",
    data: ‘test’,
  });
});

module.exports = router;
```

multer使用

multer的使用都是参考的 官方文档，我们不需要记忆，cv，理解就好

配置上传图片的地址配置

```
// 上传文件模块
var multer = require("multer");
//内置的path 模块 操作路径的模块
let path = require("path");
// 配置上传图片的路径
var storage = multer.diskStorage({
  //上传图片的路径
  destination: function (req, file, cb) {
```

```
    cb(null, "public/images");
  },
  filename: function (req, file, cb) {
    //path.extname(file.originalname) 获取前端上传图片的 后缀名
    //文件名字 以上传的时间戳为文件的名字
    cb(null, Date.now() + path.extname(file.originalname));
  },
});
```

创建upload中间件

根据地址配置 创建upload对象，同时配置请求头解析的参数名 img

```
//根据存储设置，创建upload
var upload = multer({ storage: storage }).single("img")
```

路由中挂载upload中间件，解析请求体

```
router.post("/", upload, function (req, res) {
  let file = req.file;
  console.log(file);
  let imgUrl = "/images/" + file.filename;
  res.json({
    code: 1,
    msg: "上传成功",
    data: imgUrl,
  });
}); //
```

当用户使用/api/upload 接口，post请求 上传图片的时候，会配置到上面的路由。

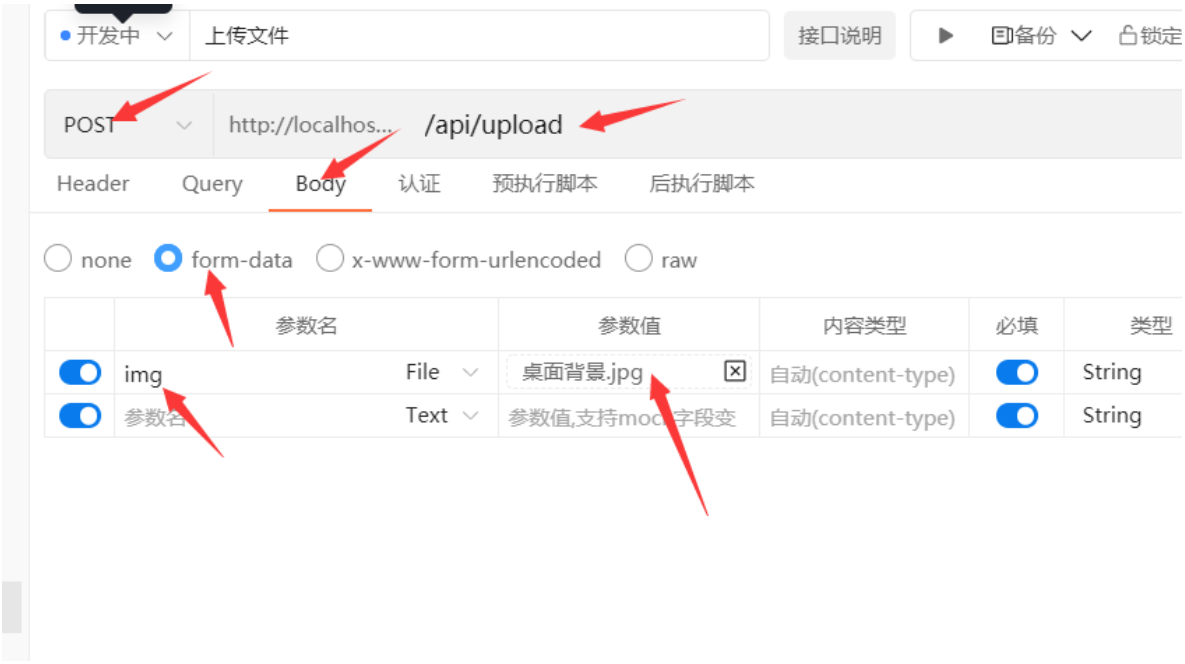
然后使用upload中间件来处理存储图片到 对应的文件夹下面，同时给req对象增加file属性，

我可以可以使用file对象获取图片的名字，并且返回图片的地址 /images/xxxxx.jpg !

注意：此时返回的路径不需要加public文件夹，因为用户 使用 http://localhost:3000/images/xxxxx.jpg 访问服务的图片的时候，自动会进入public下面去找静态资源文件。

测试一下

apipost测试上传 接口



服务器会返回图片的相对地址，我们加个服务器前缀就可以直接访问，并且此时，项目下面的public目录下面会多一张图片。

upload.js总体代码如下

```
var express = require("express");
var router = express.Router();

// 上传文件模块
//安装模块
var multer = require("multer");
//内置的path 模块 操作路径的模块
let path = require("path");
// 配置上传图片的路径
var storage = multer.diskStorage({
  //路径
  destination: function (req, file, cb) {
    //public/images/upload --- public路径下得存在 这个文件夹，否则会报错
    cb(null, "public/images");
  },
  filename: function (req, file, cb) {
    //path.extname(file.originalname) 获取前端上传图片的 后缀名
    //文件名字 以上传的时间戳为文件的名字
    cb(null, Date.now() + path.extname(file.originalname));
  },
});
var upload = multer({ storage: storage }).upload.single("img");
router.post("/", upload, function (req, res) {
  let file = req.file;
  console.log(file);
  let imgUrl = "/images/" + file.filename;
  res.json({
    code: 1,
    msg: "上传成功",
    data: imgUrl,
  });
}); //
```

```
module.exports = router;
```

身份验证jwt

有些接口的操作需要用登录以后才能使用，比如说，发布文章，删除文章等。

我们需要在用户发起请求的时候，知道用户已经登录了，并且还得知道登录的到底是谁。

HTTP 是一种没有状态的协议，也就是它并不知道是谁访问。客户端用户名密码通过了身份验证，不过下回这个客户端再发送请求时候，还得再验证，每次都输入用户名和密码体验肯定不好，所有就有了token验证。

token

什么是token

在计算机身份认证中是令牌（临时）的意思，在词法分析中是标记的意思。一般作为邀请、登录系统使用

1、Token的引入：Token是在客户端频繁向服务端请求数据，服务端频繁的去数据库查询用户名和密码并进行对比，判断用户名和密码正确与否，并作出相应提示，在这样的背景下，Token便应运而生。

2、Token的定义：Token是服务端生成的一串字符串，以作客户端进行请求的一个令牌，当第一次登录后，服务器生成一个Token便将此Token返回给客户端，以后客户端只需带上这个Token前来请求数据即可，无需再次带上用户名和密码。

3、使用Token的目的：Token的目的是为了减轻服务器的压力，减少频繁的查询数据库，使服务器更加健壮。

token机制，在服务端不需要存储用户的登录记录，全部发给客户端有客户端自己存(cookie,local)

1、客户端使用用户名跟密码请求登录
2、服务端收到请求，去验证用户名与密码
3、验证成功后，服务端会签发一个 Token（加了密的字符串），再把这个 Token 发送给客户端
4、客户端收到 Token 以后可以把它存储起来，比如放在 Cookie 里或者 Local Storage 里
5、客户端每次向服务端请求资源的时候需要带着服务端签发的 Token
6、服务端收到请求，然后去验证客户端请求里面带着的 Token，如果验证成功，就向客户端返回请求的数据

简单来说，用户在登录的时候，服务器返回一个加密字符串，给用户。用户下次请求带着这个加密串，去服务器访问就可以了。

有点像我们去看电影，想要看电影，得先用钱买个电影票，然后我们持着电影票就可以进入放映室。不需要再次用钱买了，已经买过了，电影票 就是类似token。

jwt

JWT 代表 JSON Web Token，它是一种用于认证头部的 token 格式。这个 token 帮你实现了在两个系统之间以一种安全的方式传递信息

jwt由三部分组成：

header(头部)、payload(载荷)、signature(签名) header.payload.signature

简单单说就是一个非常长的加密串

payload是允许我们存的数据，一般我的token中会存入用的id，或者username等信息。客户端下次请求的时候，我们只要获取了token就知道了用户是谁。

jwt的使用：

1-服务器生成jwt。客户端登录服务器，如果成功，服务器会返回token信息，

2-服务器验证客户端的jwt。客户端收到服务器返回的token之后，通常将其存储在localStorage或者sessionStorage中。

在后面的请求过程中，将token放在请求头的Authorization字段中 格式为：Bearer token，brearer意思是持票人，token就想是一张门票。

```
Authorization:Bearer xxxxxxxx
```

服务器生成jwt

安装jsonwebtoken

```
npm install  jsonwebtoken
```

生成jwt

通过jwt.sign

第一个参数 {username:'zhangsan'}是token存入的数据也叫payload

第二个参数 secretKey token加密的密码

第三个参数 是个对象

expiresIn token的有效时间,不带单位默认为秒 ,带单位: "1 days", "11h" “360s”

algorithm：“HS256” 表示加密算法

```
const jwt = require("jsonwebtoken")
let token = jwt.sign({username:'zhangsan'},'test123456',
{expiresIn:'30d',algorithm: "HS256"})
```

我们可以在登录成功以后返回jwt信息

routes/users.js

```
/* 登录请求 */
const jwt = require("jsonwebtoken");
router.get("/", function (req, res, next) {

  let token = jwt.sign({ username: "zhangsan" }, "test123456", {
    expiresIn: "360s",
    algorithm: "HS256",
  });

  console.log(req.query);
  res.json({
    code: 1,
    msg: "注册成功",
    token,
  });
});
```

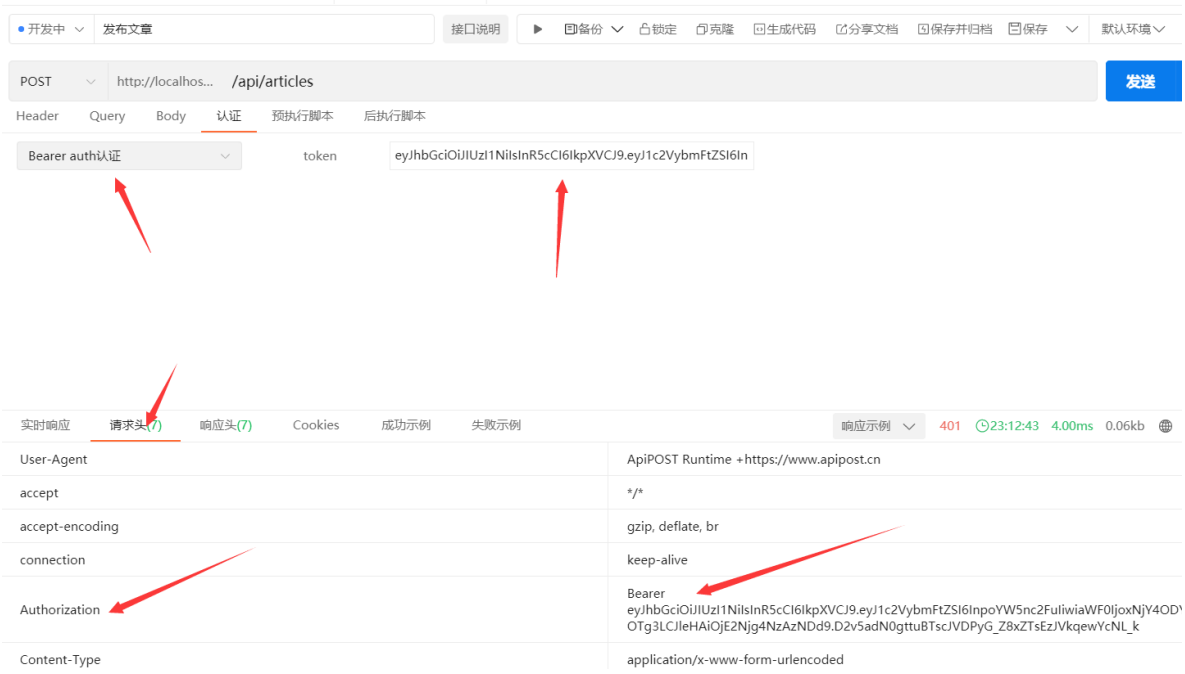
使用apipost测试登录

测试以后会返回token信息，正常前端应存入本地存储中，下次请求直接放入请求头中

```
Authorization:Bearer xxxxxxxx
```

我们可以登录返回的token，放入api post中的一个接口的请求头认证里面，下面请求测试的时候，可以解析到。

我发送请求的时候，可以再请求头中看到Authorization的信息



解析请求头中jwt

使用 这个插件express-jwt

安装

```
npm install express-jwt
```

使用express-jwt中间件解析jwt

我们在所有的路由之前挂载这个jwt处理中间件，所有的路由在匹配以前都会被检查是否请求头中包含token

secret 加密规则，字符串，一般不要告诉别人，只有开发者知道，否则，我们的token就可能被别人破解

algorithms 后面的是一个数组，数组放了一个字符串HS256，表示采用HS256加密算法

unless 参数里面有个path 属性，匹配不需要解析的请求，值是个数组,数组里元素匹配的路由都不会被检查是否有token，意思就是不用登录就可以使用。

```
var { expressjwt} = require("express-jwt");
app.use(
  expressjwt({
    secret: "test123456",
    algorithms: ["HS256"],
  }).unless({
    path: [
      "/api/users",
      // "/api/articles/users/:uid", 这样的路由，必须使用正则匹配
      /^\/api\/articles\/users\/\w+/,
      {
        url: /^\/api\/articles\/\w+/,
        methods: ["GET"],
      },
    ],
  })
);
```

"/api/users" 表示这个路由不会被检查是否头部有token值,相当于不需要token就可以使用。

对于/api/articles/users/:uid 这种路由，则需要使用正则表表达式匹配 /^\/api\/articles\/users\/\w+/

如果path里面有下面的对象，表示/api/articles/:aid 这样的路由不需要token 就是可以使用

```
{
        url: /^\/api\/articles\/\w+/,
        methods: ["GET"],
 }
```

解析的结果会被输出到req.auth

我们在需要登录的路由做测试打印一下，请求的时候需要在请求头中传入token才能成功被解析到

```
/*
 /api/articles
 发布文章
*/
router.post("/", function (req, res, next) {
// { username: 'zhangsan', iat: 1668870891, exp: 1668871251 }
  console.log(req.auth);
  console.log(req.body);
  res.json({
    code: 1,
    msg: "发布文章成功",
  });
});
/*
```

用户请求的时候携带的token，被解析成立一个对象，里面有用户名，这样我们就知道了这个请求是哪个用户发起了，并且知道他已经登陆了

检查失败的处理

有些接口需要传入token，用户调用的时候并没有传入，我们希望给用户一个提示。

只需要在app中404路由之前加个错误处理。

如果err的名字是未授权，意思是检查token失败或者没有token

```
app.use(function (err, req, res, next) {
  if (err.name === "UnauthorizedError") {
    res.status(401).json({ code: 0, msg: "无效的token或者没有没有传递token-请重新登
录" });
  } else {
    next(err);
  }
});
```

登录成功以后返回token

```

```

有客户端发起请求，请求到数据负责存贮

前端可以把token存贮在cookie里面或者localstorge里面,一次请求记得携带token，请求就不会报错了

测试一下

到此为止怎么写接口，上传图片，接口的身份验证我们都已经做了。

还差数据的存储，也就是数据库。注册时候的信息和发布的文章信息都存入数据库；登录的时候要从数据库中查找是否存在对应的用户名和密码，如果存在，我们才能给客户端返回token！

数据库MongoDB

+ 什么是数据库

- 就是存储数据的仓库

- 数据库是由后端访问操作的

- 后端获取到数据后，将数据返回给其前端

- 我们可以把数据库想象成一个文件夹

+ 如何存储数据的?

- 一个一个的 表格 进行存储

MongoDB介绍

MongoDB是一个基于分布式文件存储的数据库。由C++语言编写。旨在为WEB应用提供可扩展的高性能数据存储解决方案。它的特点:高性能、易部署、易使用，存储数据非常方便

MongoDB是一个介于关系数据库和非关系数据库之间的产品，是非关系数据库当中功能最丰富，最像关系数据库的。它支持的数据结构非常松散，是类似json的bson格式，因此可以存储比较复杂的数据类型。Mongo最大的特点是它支持的查询语言非常强大，其语法有点类似于面向对象的查询语言，几乎可以实现类似关系数据库单表查询的绝大部分功能

不管我们学习什么数据库都应该学习其中的基础概念，在mongodb中基本的概念是文档、集合、数据库

mysql术语/概念	MongoDB术语/概念	解释/说明
database	database	数据库
table	collection	数据库表/集合
row	document	数据记录行/文档
column	field	数据字段/域
index	index	索引
table joins		表连接,MongoDB不支持
primary key	primary key	主键,MongoDB自动将_id字段设置为主键

一个mongodb中可以建立多个数据库。

MongoDB的默认数据库为"db"，该数据库存储在data目录中。

MongoDB的单个实例可以容纳多个独立的数据库，每一个都有自己的集合和权限，不同的数据库也放置在不同的文件中。

数据库命名：通过标识符，一般是utf - 8字符串，不能为空，不能用local / admin / config这三个

文档

文档是一个键值(key-value)对MongoDB 的文档不需要设置相同的字段，并且相同的字段不需要相同的数据类型，这与关系型数据库有很大的区别，也是 MongoDB 非常突出的特点。

一个简单的文档例子如下：

{"title": "这个月还缺一个转介绍","content": "来千峰学编程，找翔哥"}

集合

集合就是 MongoDB 文档组，类似于 RDBMS （关系数据库管理系统：Relational Database Management System)中的表格。

集合存在于数据库中，集合没有固定的结构，这意味着你在对集合可以插入不同格式和类型的数据，但通常情况下我们插入集合的数据都会有一定的关联性。集合的命名不能是空字符串，也不能出现 - ，0等，不能以system，$开头

{"title": "文章标题1","content": "我是文章内容"}

{"title": "文章标题123","content": "我是文章内容2323"}

当第一个文档插入时，集合就会被创建。

MongoDB 数据类型

下表为MongoDB中常用的几种数据类型。

数据类型	描述
String	字符串。存储数据常用的数据类型。在 MongoDB 中，UTF-8 编码的字符串才是合法的。
Integer	整型数值。用于存储数值。根据你所采用的服务器，可分为 32 位或 64 位。
Boolean	布尔值。用于存储布尔值（真/假）。
Double	双精度浮点值。用于存储浮点值。
Min/Max keys	将一个值与 BSON（二进制的 JSON）元素的最低值和最高值相对比。
Array	用于将数组或列表或多个值存储为一个键。
Timestamp	时间戳。记录文档修改或添加的具体时间。
Object	用于内嵌文档。
Null	用于创建空值。
Symbol	符号。该数据类型基本上等同于字符串类型，但不同的是，它一般用于采用特殊符号类型的语言。
Date	日期时间。用 UNIX 时间格式来存储当前日期或时间。你可以指定自己的日期时间：创建 Date 对象，传入年月日信息。
Object ID	对象 ID。用于创建文档的 ID。
Binary Data	二进制数据。用于存储二进制数据。
Code	代码类型。用于在文档中存储 JavaScript 代码。
Regular expression	正则表达式类型。用于存储正则表达式。

ObjectId

ObjectId 类似唯一主键，可以很快的去生成和排序.

MongoDB 中存储的文档必须有一个 _id 键。这个键的值可以是任何类型的，默认是个 ObjectId 对象

字符串

BSON 字符串都是 UTF-8 编码

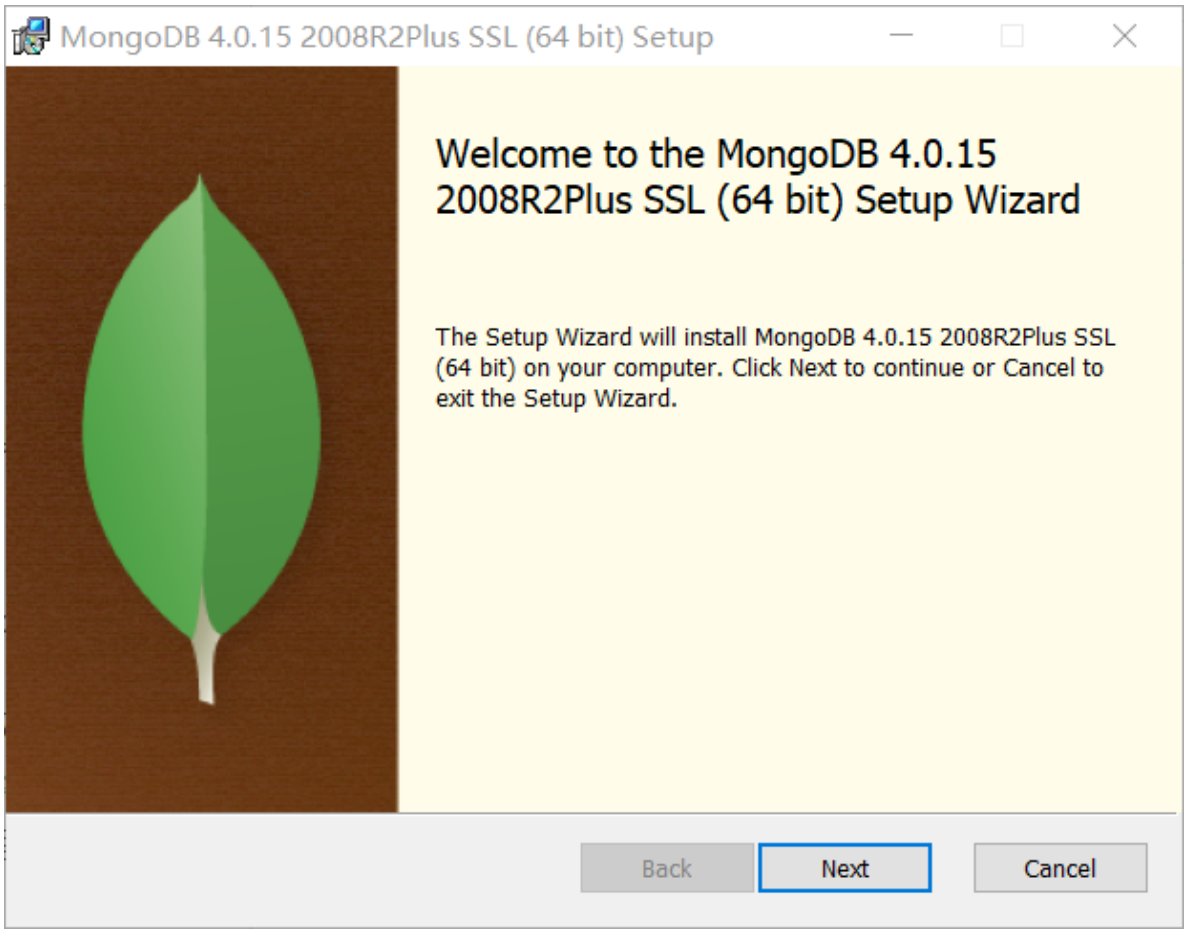
MongoDB 安装

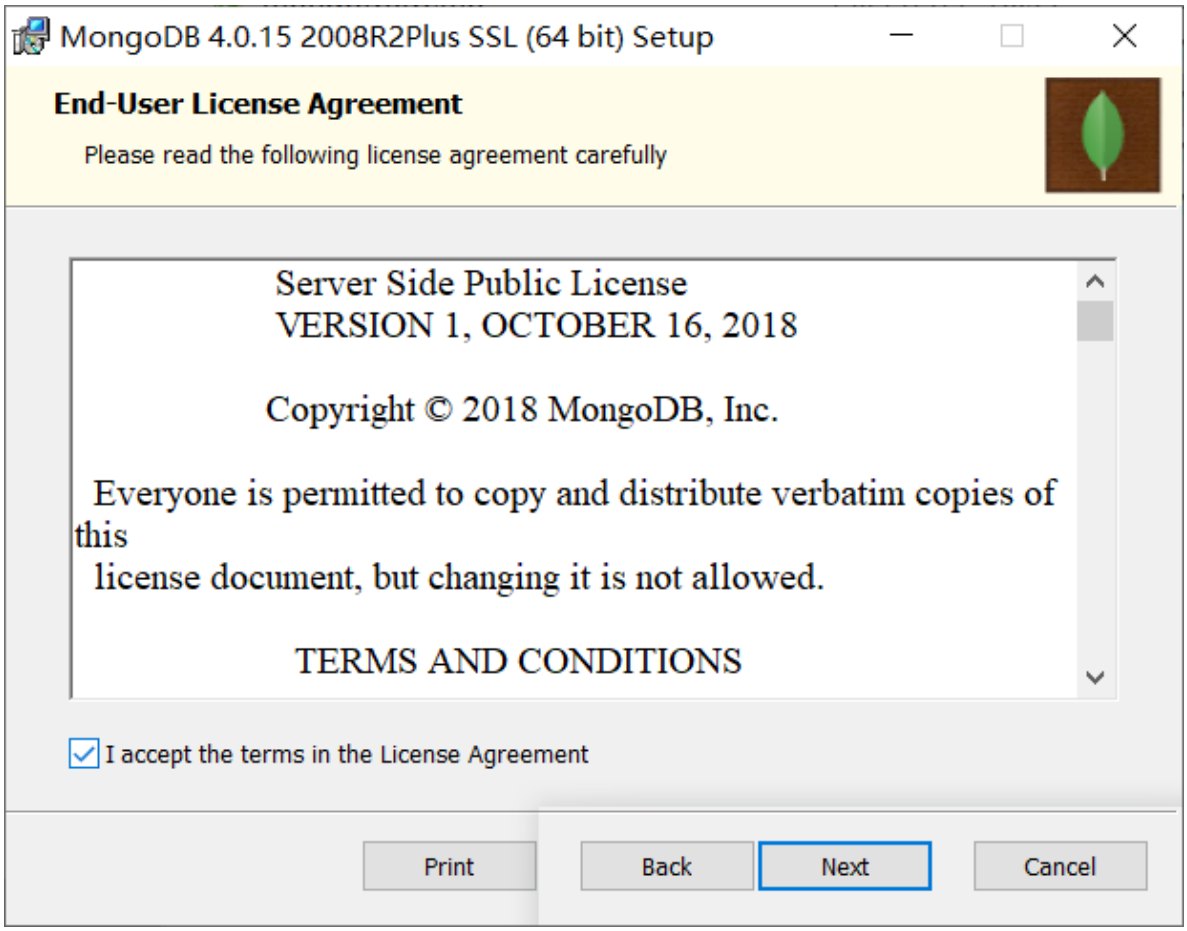
安装方式1

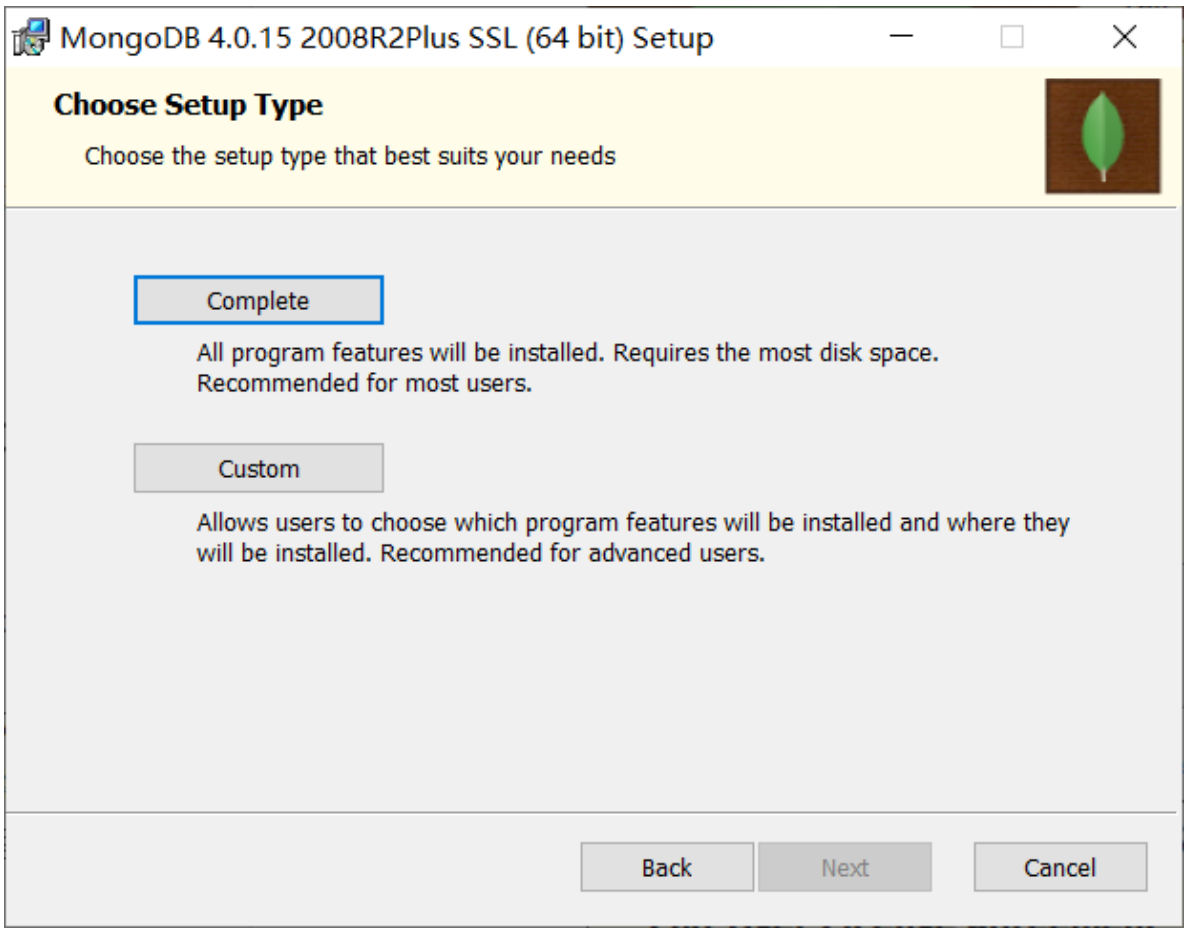
下载地址https://www.mongodb.com/download-center/community

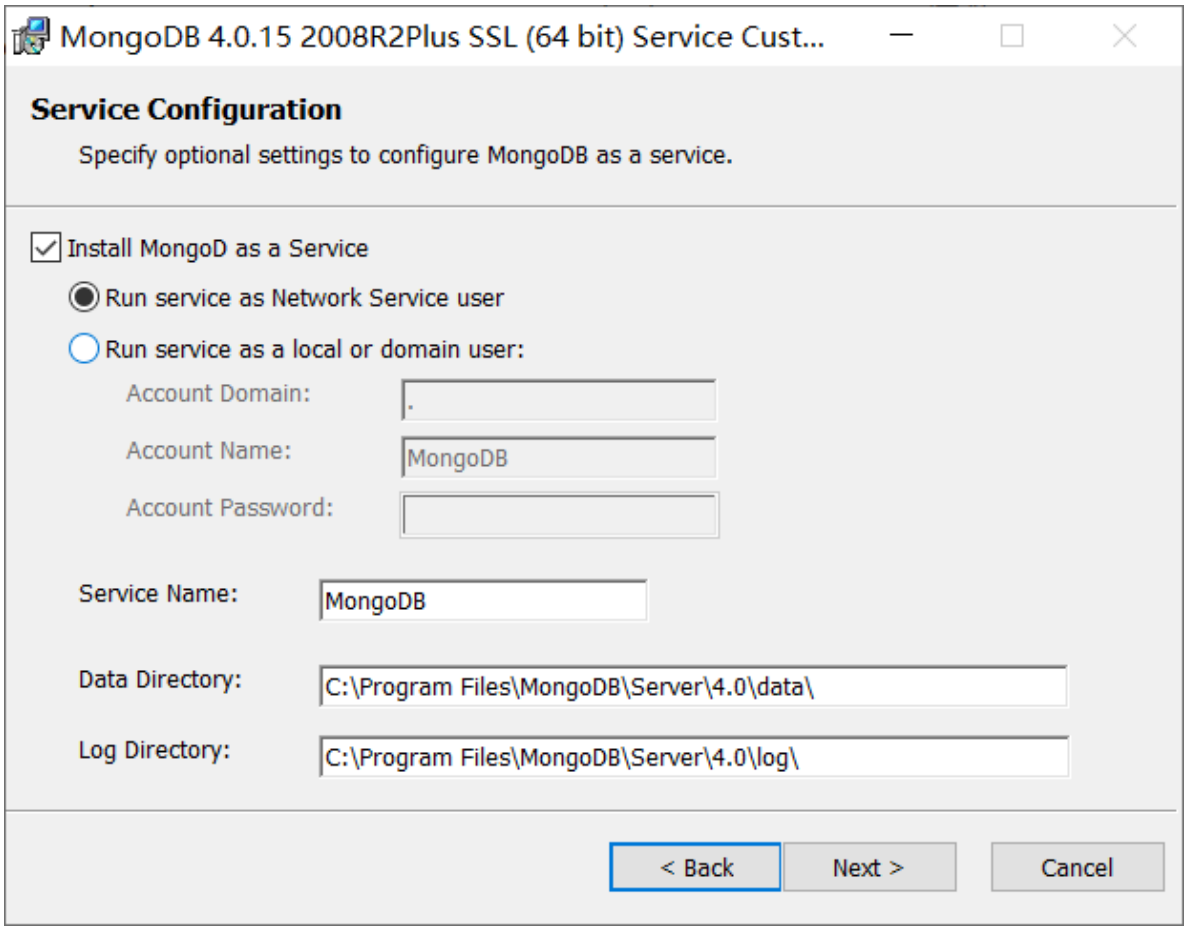
直接找到安装包中的mongodb-win32-x86_64-2008plus-ssl-4.0.15-signed.msi这个文件，双击安装

然后步骤如下图


MongoDB 4.0.15 2008R2Plus SSL (64 bit) Setup
Welcome to the MongoDB 4.0.15 2008R2Plus SSL (64 bit) Setup Wizard
The Setup Wizard will install MongoDB 4.0.15 2008R2Plus SSL (64 bit) on your computer. Click Next to continue or Cancel to exit the Setup Wizard.
Back
Next
Cancel

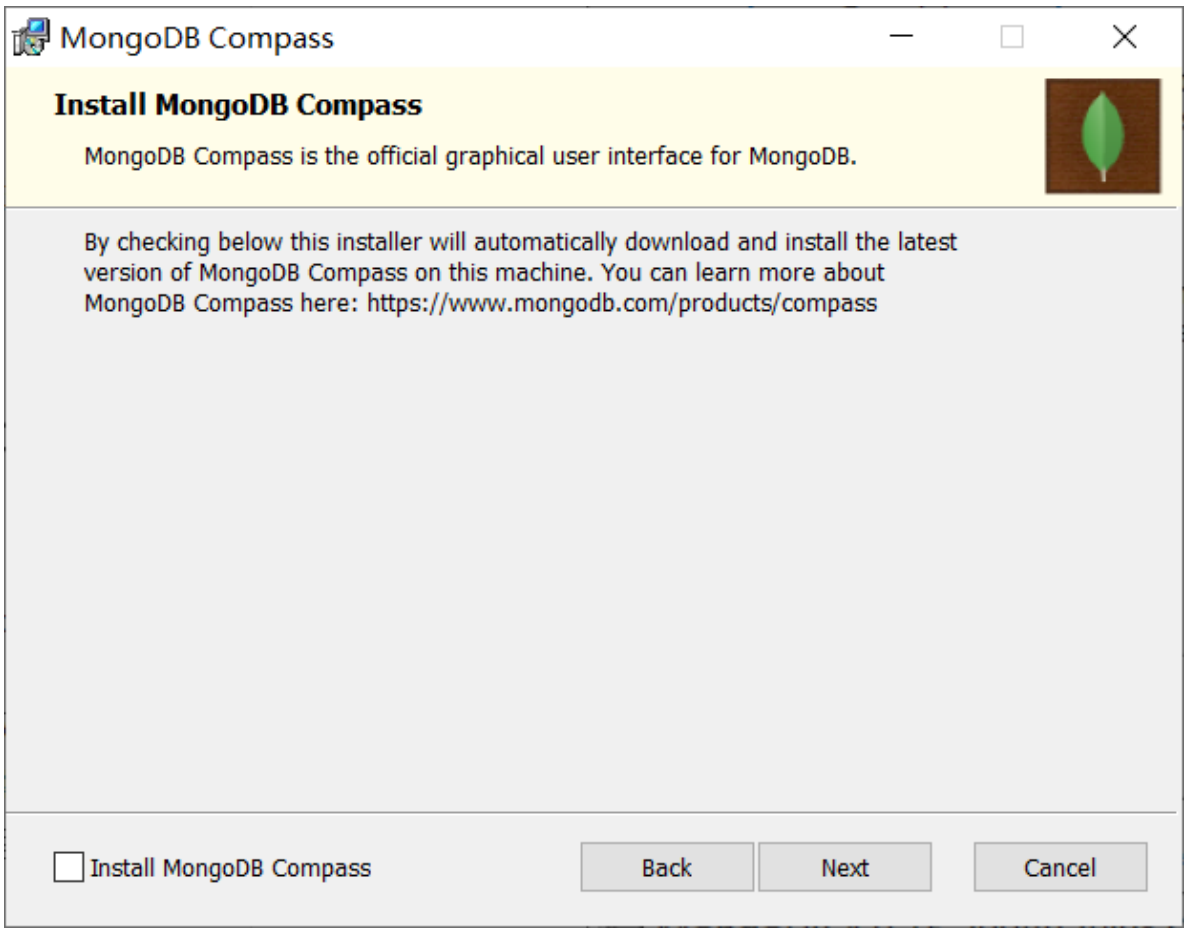

MongoDB 4.0.15 2008R2Plus SSL (64 bit) Setup
End-User License Agreement
Please read the following license agreement carefully
Server Side Public License
VERSION 1, OCTOBER 16, 2018
Copyright © 2018 MongoDB, Inc.
Everyone is permitted to copy and distribute verbatim copies of this
license document, but changing it is not allowed.
TERMS AND CONDITIONS
I accept the terms in the License Agreement
Print
Back
Next
Cancel



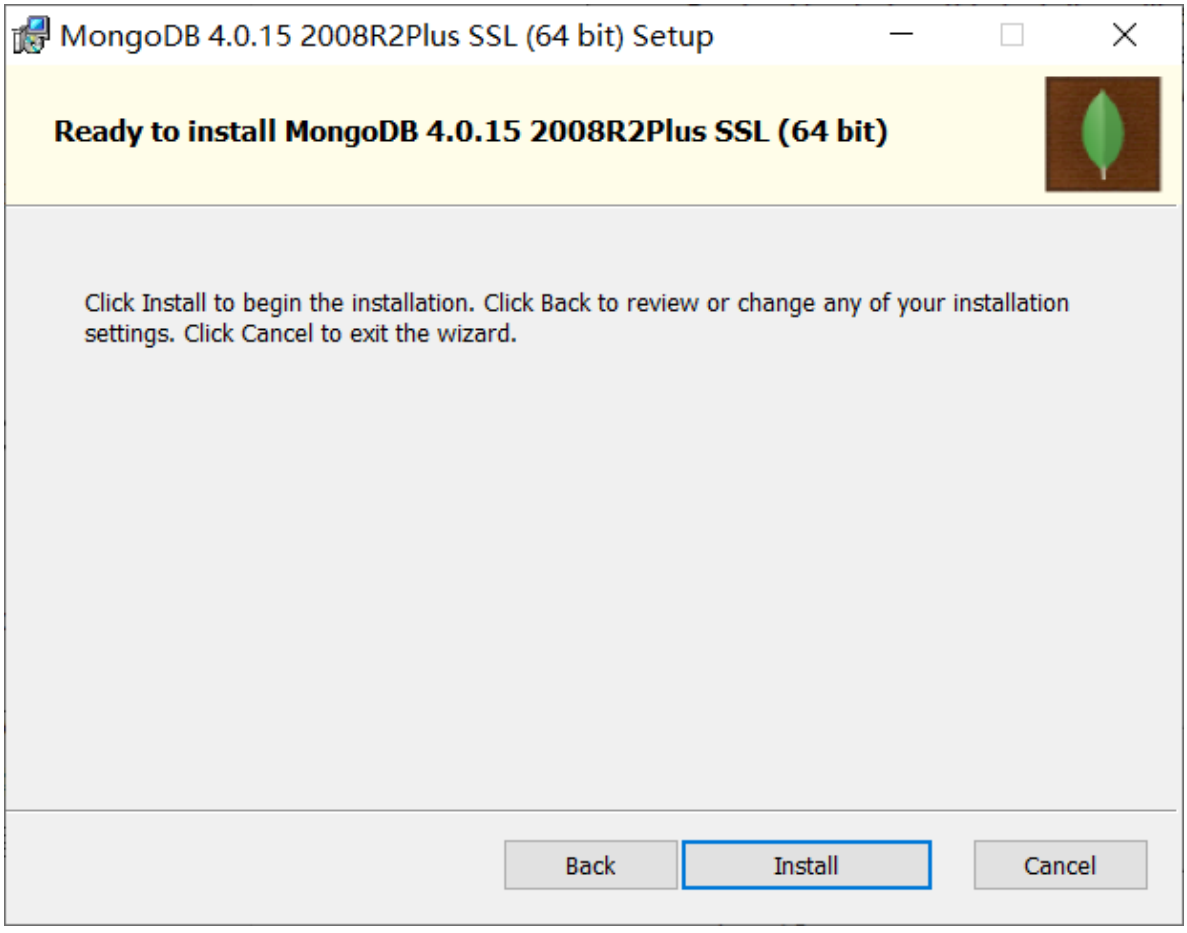


Data Directory 文件夹是数据数据的存储目录，可以自己修改路径

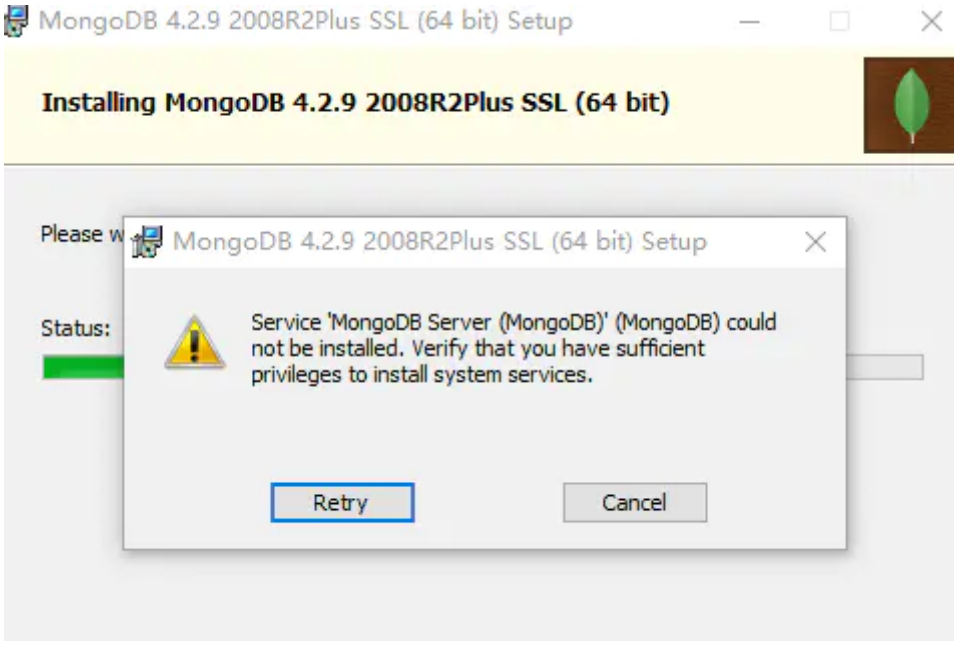
LogDirectory 文件夹是数据数据日志的存储目录，可以自己修改路径



注意把Compass 的勾选取消了



ps: 如果安装最后报错 Verify that you have sufficient privileges报错，关闭360或者电脑管家就行了，重新安装就行了。



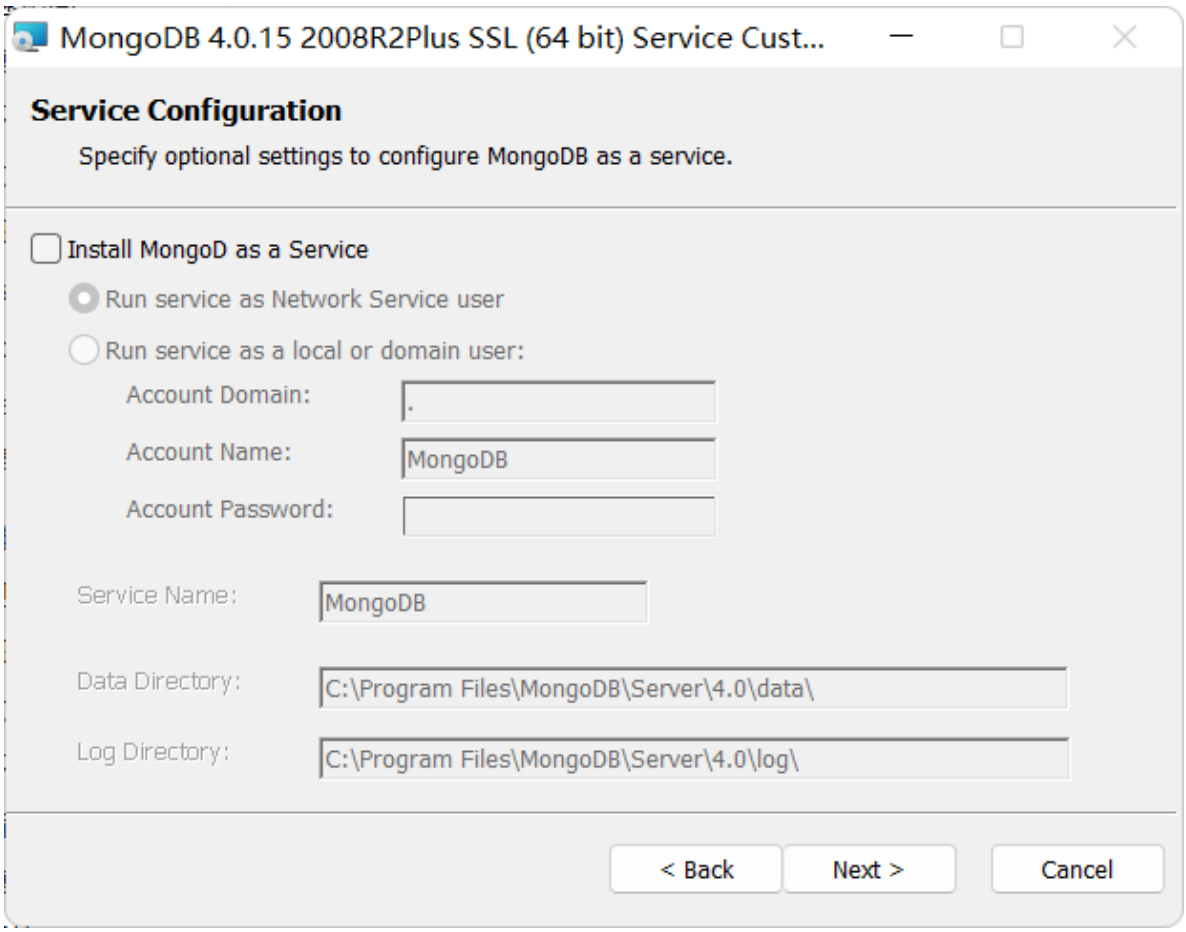
自动启动

我们这样安装，是把数据库安装成系统服务，这样不用每次启动，使用的时候会自动启动数据库。

安装方式2

如果上面安装不成功。也可以不安装成系统服务，使用的时候自己启动。

同样的安装方式，在第四个界面的时候，把 install MongoD as a Service 取消了



手动启动

1-创建数据储存目录

在c盘中创建文件夹,路径输如下

C:\MongoDB\

2-终端中启动数据库

终端中进入 C:\Program Files\MongoDB\Server\4.0\bin目录下

执行下面命令

```
mongod -dbpath C:\MongoDB
```

这条命令是开启服务，它会一直运行，只要你要使用Mongodb，这个窗口就不能关

当你看到 一段密集的文字提示 的时候就是数据库启动成功了

Mongodb 可视化工具

Robo3T 操作 MongoDB的数据操作

安装robo3t工具

+ 是一个数据库的可视化工具

+ 利用一个可视化界面来操作数据库的增删改查

+ 我们安装的目的，是为了我们使用代码操作的时候，更方便的查看操作的结果

\1. 官网下载

\2. 双击安装包安装

\3. 使用，双击图标开始使用

第一次启动的时候点击统一

- 你想要使用可视化工具，要保证数据库是开启状态

- 如果你的数据库是关闭状态，那么可视化工具就不能正常使用

Mongodb 命令了解

创建数据库，如果数据库不存在，则创建数据库，否则切换到指定数据库。

```
use DATABASE_NAME
```

如果你想查看所有数据库，可以使用 **show dbs** 命令

MongoDB 中使用 **createCollection()** 方法来创建集合

```
db.createCollection(name, options)
```

在 MongoDB 中，你不需要创建集合。当你插入一些文档时，MongoDB 会自动创建集合。

如果要查看已有集合，可以使用 **show collections** 或 **show tables** 命令：

document 文档操作

db.集合名.方法名

插入文档

```
db.posts.insertOne({title:"测试一下"})

db.posts.insertMany([{title:"test1"},{title:"test2"}])
```

查询文档

```
查找多条
db.posts.find({title:'aa'})
查找一条
db.posts.findOne({title:'aa'})
```

更新文档

```
 db.posts.update({title: "aaa"}, {$set: {age: 'changeName'}});

 db.users.update({name: 'Lisi'}, {$inc: {age: 50}});

相当于：update users set age = age + 50 where name = ‘Lisi';
```

删除文档

```
// 删除年龄是132

db.users.deleteOne({age: 132});

//删除users中所有数据

db.users.delete({}))
```

条件查询查询

范围查询

```
查询age = 22的记录

db.userInfo.find({"age": 22}); 相当于： select * from userInfo where age = 22;

查询age > 22的记录

db.userInfo.find({age: {$gt: 22}}); 相当于：select * from userInfo where age >22;

查询age < 22的记录

db.userInfo.find({age: {$lt: 22}}); 相当于：select * from userInfo where age <22;

查询age >= 25的记录

 db.userInfo.find({age: {$gte: 25}}); 相当于：select * from userInfo where age >=
25;

查询age <= 25的记录

db.userInfo.find({age: {$lte: 25}});

查询age >= 23 并且 age <= 26
```

```
db.userInfo.find({age: {$gte: 23, $lte: 26}});
```

筛选数据

```
查询指定列查询指定列name、age数据

db.userInfo.find({}, {name: 1, age: 1}); 相当于：select name, age from userInfo;

查询指定列name、age数据, age > 25
db.userInfo.find({age: {$gt: 25}}, {name: 1, age: 1}); 相当于：select name, age
from userInfo where age >25;
```

Limit 与 Skip 方法

```
查询前5条数据

db.userInfo.find().limit(5); 相当于：select top 5 * from userInfo;

查询10条以后的数据

db.userInfo.find().skip(10);相当于：select * from userInfo where id not in (
select top 10 * from userInfo );

限制数据量 / 几条数据

db.userInfo.find().limit(10).skip(5);
```

排序

```
按照年龄排序

升序：db.userInfo.find().sort({age: 1});

降序：db.userInfo.find().sort({age: -1});
```

Mongoose

介绍

http://www.mongoosejs.net/docs/guide.html

Mongoose 是一个第三方库，在 Express 中利用 Mongoose 库操作 Mongodb比较方便。

Mongoose为模型提供了一种直接的，基于scheme结构去定义你的数据模型。它内置数据验证，查询构建，

业务逻辑钩子等，开箱即用。

简单来说可以使用Mongoose定义数据的表结构，表结构在Mongoose 中叫scheme，然后直接使用表结构创建一个数据模型。

我们使用数据模型就可以操作数据库中的对应的表。

这里的表，我们不用自己创建，直接使用数据模块，他会自动创建表。

Mongoose 的使用方式

安装Mongoose

```
npm 初始化项目
安装mongoose
npm i mongoose
```

引入模块，连接数据库

创建models文件夹，新建文件index.js

```
// 引入模块
let mongoose = require('mongoose');
// 链接数据库
//mongodb://127.0.0.1数据库地址
// my-blog 是数据库名，没有就创建，有直接使用这个数据库-数据库我们只需要链接一次，时候的时候，他会自动连接和断开
mongoose.connect('mongodb://127.0.0.1/my-blog').then(res => {
    console.log('链接成功')
})
```

If the local connection fails then try using 127.0.0.1 instead of localhost. Sometimes issues may arise when the local hostname has been changed.

如果使用localhost失败，可以换成127.0.0.1

mongoose中的对象：

- `Schema` 模式对象（用于约束文档的结构-类似表头）
- `Model` 模型对象（即mongodb中的集合-表）
- `Document` 文档对象（即mongodb中的文档-表中一行数据）

Schema

Mongoose 的一切始于 Schema。每个 schema 都会映射到一个 MongoDB collection（表），并定义这个collection里的文档（一行数据）的构成。

```
// 获取创建的集合（表）的模块
```

```
let Schema = mongoose.Schema;
//新建文章表结构
let ArticleSchema = new Schema(
  {
    title: String, //文章标题
    content: String, //文章内容
    tag: String, //文章分类
    author: String, //文章的作者
    views: {
      type: Number,
      default: 0,
    }, //文章的浏览量
  },
  {
    timestamps: true, //启用时间戳时，Mongoose 会将 createdAt 和 updatedAt 属性添加到模型中。表中添加一行数据的时候会产生时间戳，记录，数据的创建时间和修改时间
  }
);
```

Model

Models 是从 `Schema` 编译来的构造函数。 它们的实例就代表着可以从数据库保存和读取的 documents。 从数据库创建和读取 document 的所有操作都是通过 model 进行的。

通过它可以对表的内容进行增删改查

```
// 将表绑定到当前数据库上，得到表构造函数，
// 用构造函数可以创建表中的数据，进行CRUD 增删改查
var Article = mongoose.model('Article', ArticleSchema);
```

第一个参数是跟 model 对应的集合（表）名字

将表结构AritcleSchema绑定到当前数据库上，并且得到操作表中内容的 构造函数，用这些构造函数可以创建表中的数据，进行CRUD 增删改查。

注意，当我们向表中插入数据的时候表才会生成，表的名字自动会使用复数的形式。比如刚刚的构造函数Aritcle，我们在使用的时候，会自动在数据库中创建一个Aritcles表。

创建文档数据，插入表中

```
// 创建数据，保存到数据库
Article.create({
  title: "春节杂谈1",
  content: "每逢佳节倍思亲",
  author: "zhangsan",
  tag: "社会",
}).then((res) => {
  console.log(res);
  console.log("保存成功");
});
//或者下面也可以
// 创建数据
let atr = new Article({
  title: "春节杂谈2",
  content: "每逢佳节倍思亲。。。",
```

```
  author: "zhangsan",
  tag: "社会",
});
// 保存到数据库
atr.save().then((res) => {
  console.log(res);
  console.log("保存成功");
});
```

终端进入models，然后执行 node index.js 看看能不能插入数据（记得启动MongoDB数据库）

执行成功，回返回类似如下的数据

```
{
  title: '春节杂谈',
  content: '每逢佳节倍思亲.。。',
  tag: '社会',
  author: 'zhangsan',
  views: 0,
  _id: new ObjectId("637b4f0045e1fcfb0e239d20"),
  createdAt: 2022-11-21T10:12:16.712Z,
  updatedAt: 2022-11-21T10:12:16.712Z,
  __v: 0
}
```

_id 就是这条文章的id，自动生成的，这个在数据库中也就主键-类似人的身份证，唯一的表示一个文章。createdAt是创建时间，updatedAt是更新时间，后面如果我们更新的文章，那么这个数据就会自动修改。

删除数据

```
// 删除数据，第一个参数是一个对象，对象内是条件
// 通过id删除
//删除 id 为 5f06ded47c5b015ecc379490的文章
Article.deleteOne({_id:'5f06ded47c5b015ecc379490'}).then(res=>{
  console.log('删除成功')
  console.log(res)
}).catch(err=>{
  console.log('删除失败')
  console.log(err)
})

// 删除多条数据
Article.deleteMany({title:"aaa"}).then(res=>{
  console.log('删除成功')
}).catch(err=>{
  console.log('删除失败')
})
```

修改数据

第一个参数 对象 是一个条件

第二个参数对象是一个要修改的内容

```
// 根据id为修改数据，
//把id 为5f06dd7010cb185ae005753e的文章的标题修改为aaa
Article.updateOne({_id:'5f06dd7010cb185ae005753e'},{title:'aaa'}).then(res=>{
  console.log('修改成功')
  console.log(res)
}).catch(err=>{
    console.log('修改失败')
})
```

根据id修改views字段，自加1,并且修改的时候不会让updateAt更新

```
Article.updateOne(
  { _id: "637b4f0045e1fcfb0e239d20" },
  { $inc: { views: 1 } },//views字段，自加1,
  { timestamps: false }//修改的时候不会让updateAt更新
)
  .then((res) => {
    console.log("修改成功");
    console.log(res);
  })
  .catch((err) => {
    console.log("修改失败");
  });
```

查询数据

查询一条

```
// 根据id查询一条数据
Article.findById('5f06dd7010cb185ae005753e').then(res => {
  console.log(res)
})
Article.findOne({ _id: '5f06dd7010cb185ae005753e' }).then(res => {
  console.log(res)
})
//根据id查询，并且更新数据
Article.findByIdAndUpdate(
  "637b4f0045e1fcfb0e239d20",
  { $inc: { views: 1 } },
  { timestamps: false }
).then((r) => {
  console.log(r);
});
```

查询多条

```
// 能查询多条，也可以传入条件
//第一个参数 是个对象{}不传参数就是查询所有，传入条件就是根据条件查询函
Article.find({
  views: { $gte: 0, $lt: 1000 }, //范围  views的值 大于等于0 小于 1000
  title:/哈哈/  //搜索 title包含 哈哈 ，搜索条件是正则
})
  .sort({ _id: -1 }) // 根据id 1 升序  -1 降序
  .skip(0) //跳过前0条
  .limit(10) //获取10条事件
  .select({ author: 0 }) //1 查询结果中出现 author 字段  0是不出现这个字段
  .exec() //执行查询
  .then((res) => {
    //查询成功
    console.log(res);
  });
```

##

到现在我们就可以对数据库的数据进行增删改查了，接下来我们设计博客网站需要的表设计一下。

表设计

来设计一下我们数据库中的表，用户表-文章表-评论表，每种数据尽量都用一个表去存储。

下面的表设计结构Schema 的时候都放到models/index.js文件中

用户表

设计用户表 username，password ，nickname,headImgUrl

```
let userSchema = new Schema({
  username: String,
  password: String,
  nickname: String,
  headImgUrl: String,
},
  {
    timestamps: true, //启用时间戳时
  });

let User = mongoose.model("User", userSchema);
```

文章表

标题，内容，文章标签，作业，浏览量，时间戳

```
// 获取创建的集合（表）的模块
let Schema = mongoose.Schema;
//新建文章表结构
let ArticleSchema = new Schema(
  {
    title: String, //文章标题
    content: String, //文章内容
```

```
    tag: String, //文章分类
    author: String, //文章的作者
    views: {
      type: Number,
      default: 0,
    }, //文章的浏览量
  },
  {
    timestamps: true, //启用时间戳时，Mongoose 会将 createdAt 和 updatedAt 属性添加到模型中。表中添加一行数据的时候会产生时间戳，记录，数据的创建时间和修改时间
  }
);
var Article = mongoose.model("Aritcle", ArticleSchema);
```

表关联

关联有什么用?

这样可以保证数据的唯一性，用户表中存用户的名字和用户id

文章表中想要表示一篇文章的作者是一个用户，文章表中不用再存用户名了。因为再存用户名，导致用户名存了两个地方，数据就不唯一了！假设用户修改的自己用户名，还得修改所有文章的表里面的用户名，就太麻烦了。

文章表中存用户id以后，想要知道文章是作者是谁，只需要根据作者的id去，用户名表查询用户信息即可。

于是就有了数据库关系查询，mongodb也是最像关系型数据库的非关系型数据库，就是让他们的相同类别数据存在同一个集合(表)中， 让行内的某个字段做集合与集合之间的对应关系，通常是用id作为集合之间的映射关系。

文章表-关联用户表-一对一关联

articles 存放文章信息，每篇文章都属于某个用户(用author字段和User表的_id字段做关联)

修改文章表 的作者字段 为 author: {type:Schema.Types.ObjectId, ref: 'User'}

这个叫表关联，可以 将用户表和文章表关联起来

文章表中的作者字段， 只存一个作者的id， 这个id的 对应的数据都在User表中.

```
//新建文章表结构
let ArticleSchema = new Schema(
  {
    title: String, //文章标题
    content: String, //文章内容
    tag: String, //文章分类
    author: {type:Schema.Types.ObjectId, ref: 'User'}, //文章的作者
    views: {
      type: Number,
      default: 0,
    }, //文章的浏览量
  },
```

```
  {
    timestamps: true, //启用时间戳时，Mongoose 会将 createdAt 和 updatedAt 属性添加到模型中。表中添加一行数据的时候会产生时间戳，记录，数据的创建时间和修改时间
  }
);
var Article = mongoose.model("Aritcle", ArticleSchema);
```

关联查询（一对一）

使用populate方法填充方法

参数1： 需要查询的关联字段

参数2： 关联查询出来的信息需要显示字段

```
Article.find().populate('author','username nickname')})
Article.find().populate('author',{username:1, nickname:1})})
```

这样查询出来的文章信息都带有用户信息，而不是只有一个用户id

测试一下

新建test.js在里测试一下

这样我们去新建一些文章数据和用户数据，创建文章的时候指定文章的作者id

，然后再去查询所有文章

新建用户

```
let { Article, User, Comment } = require("./index");


User.create({
  username: "lisi",
  password: "123456",
  headImgUrl: "http://aaa/aa.png",
  nickname: "我是mt",
}).then((r) => {
  console.log(r);
  //   username: 'lisi',
  //   password: '123456',
  //   nickname: '我是mt',
  //   headImgUrl: 'http://aaa/aa.png',
  //   _id: new ObjectId("637b65c445195465105f4580"),
  //   __v: 0
});
```

新建文章

记得指定作者id，多新建几篇文章，我们多查询一些文章

```
Article.create({
  title: "来的第一天",
  content: "xxxwewewe",
  tag: "前端",
  author: "637b65c445195465105f4580",
}).then((r) => {
  console.log(r);
  //   title: '来的第一天',
  //   content: 'xxxwewewe',
  //   tag: '前端',
  //   author: new ObjectId("637b65c445195465105f4580"),
  //   views: 0,
  //   _id: new ObjectId("637b66740887b523d45f54bd"),
  //   createdAt: 2022-11-21T11:52:20.466Z,
  //   updatedAt: 2022-11-21T11:52:20.466Z,
  //   __v: 0
});
```

查询文章

查询文章的时候使用 populate('author',{password:0}) 就可以把用户表的信息关联到author字段，并且不显示用户的password信息

```
Article.find({
  favs: { $gte: 0, $lt: 1000 }, //范围  favs的值 大于等于0 小于 1000
  title:/哈哈/ //搜索 title包含 哈哈 ，搜索条件是正则
})
  .sort({ _id: -1 }) // 根据id 1 升序  -1 降序
  .skip(0) //跳过前0条
  .limit(10) //获取10条事件
  .exec() //执行查询
  .then((res) => {
    //查询成功
    console.log(res);
  });
```

这样我们操作的文章信息，只有用户的id，没有用户的名字等信息

获取关联数据

使用 populate('author',{password:0})

可以获取author字段对应的关联表的信息，并且不显示用户的password信息

如果author的值用户id 1，使用populate('author')以后，就会去authors表中获取id为1的用户信息，填充到返回结果中

```
Article.find({
  favs: { $gte: 0, $lt: 1000 }, //范围  favs的值 大于等于0 小于 1000
  title:/哈哈/ //搜索 title包含 哈哈 ，搜索条件是正则
})
  .sort({ _id: -1 }) // 根据id 1 升序  -1 降序
  .skip(0) //跳过前0条
  .limit(10) //获取10条事件
  .populate('author',{password:0}) //关联author字段，并且不显示用户的password信息
```

```
  .exec() //执行查询
  .then((res) => {
    //查询成功
    console.log(res);
  });
```

查询的结果像是这样

```
[
  {
    _id: new ObjectId("637b66740887b523d45f54bd"),
    title: '来的第一天',
    content: 'xxxwewewe',
    tag: '前端',
    author: {
      _id: new ObjectId("637b65c445195465105f4580"),
      username: 'lisi',
      nickname: '我是mt',
      headImgUrl: 'http://aaa/aa.png',
      __v: 0
    },
    views: 0,
    createdAt: 2022-11-21T11:52:20.466Z,
    updatedAt: 2022-11-21T11:52:20.466Z,
    __v: 0
  },
  {
    _id: new ObjectId("637b664a9e6dbe811e23b926"),
    title: '来的第一天',
    content: 'xxxwewewe',
    tag: '前端',
    author: {
      _id: new ObjectId("637b65c445195465105f4580"),
      username: 'lisi',
      nickname: '我是mt',
      headImgUrl: 'http://aaa/aa.png',
      __v: 0
    },
    views: 0,
    createdAt: 2022-11-21T11:51:38.719Z,
    updatedAt: 2022-11-21T11:51:38.719Z,
    __v: 0
  }
]
```

新的评论表-关联了文章表和用户表

也应该单独放在一张表中

评论内容，评论的文章id，评论人的id

```
// 评论表
let CommentSchema = mongoose.Schema(
  {
    // 评论内容
    content: String,
    //评论所属文章的id
    article_id: { type: Schema.Types.ObjectId , ref: "Article"},
    //评论人id
    reply_user_id: { type: Schema.Types.ObjectId, ref: "User" },
  },
  {
    //记录创建时间和修改时间
    timestamps: true,
  }
);
var Comment = mongoose.model("Comment", CommentSchema);
```

新建评论

我们可以给一篇文章创建多个评论，然后添加到进评论表里面

```
let c1 = new Comment({
  content: "怎么了222",
  reply_user_id: "637b65c445195465105f4580",
  article_id: "637b66740887b523d45f54bd",
});
c1.save().then((r) => {
  console.log(r);
});
```

查询这篇文章的评论-关联用户-关联查询（一对一）

一个评论对应一个评论人

这样就会执行评论是哪个用户发布的了。

```
// 查一篇文章的所有评论
Comment.find({ article_id: "5f0731915f30dc5598bb39a2" })
  .populate("reply_user_id", { password: 0 })
  .then((res) => console.log(res));
```

关联查询（一对多）

如果想要查询文章的时候，然后查询出关于这篇的所有留言，这是一对多的关联。

在这里的话，需要用到虚拟字段

先给Article添加虚拟字段

```
// 这边使用virtual属性来完成对评论的列表提取，下面是对文章列表取数据的时带上评论数
ArticleSchema.virtual("coms", {
  ref: "Comment",
  localField: "_id",
  foreignField: "article_id",
  justOne: false, //取Array值- 会把文章对应的评论全部提取出来
  // count: true, //取总数  如果为true 只显示数组的长度，不显示数组的内容
});
// 下面这两句只有加上了， 虚拟字段才可以显性的看到，不然只能隐性使用
ArticleSchema.set("toObject", { virtuals: true });
ArticleSchema.set("toJSON", { virtuals: true });
```

然后查询文章 populate("coms")，就会所有的评论放入 coms 对应的数组中

```
Article.find({
  views: { $gte: 0, $lt: 1000 }, //范围  favs的值 大于等于0 小于 1000
  title: /第一天/, //搜索 title包含 哈哈 ，搜索条件是正则
})
  .sort({ _id: -1 }) // 根据id 1 升序  -1 降序
  .skip(0) //跳过前0条
  .limit(10) //获取10条事件
  .populate("author", { password: 0 }) //关联author的id对应的用户表中用户信息，并且不显示用户的password信息
  .populate("coms")
  .exec() //执行查询
  .then((res) => {
    //查询成功
    console.log(res);
    console.log(JSON.stringify(res));
  });
```

结合数据库实现接口

在接口文件我们要用到数据库操作，我们把数据库的相关内容导出

models/index.js

```
module.exports  = {
  Comment,Article,User
}
```

注册接口 routes/users.js

```
var express = require("express");
var router = express.Router();

let { User } = require("../models/index");

/* 注册请求 */
router.post("/", function (req, res, next) {
  let username = req.body.username;
  let password = req.body.password;
  let nickname = req.body.nickname;
  let headImgUrl = req.body.headImgUrl;
  console.log(req.body);
  if (username && password && nickname && headImgUrl) {
    // 创建用户 插入数据库
    User.create(req.body)
      .then((r) => {
        console.log(req.body);
        res.json({
          code: 1,
          msg: "注册成功",
        });
      })
      .catch((err) => {
        console.log(req.body);
        res.json({
          code: 0,
          msg: "注册失败",
          err: "用户名已经存在",
        });
      });
  } else {
    res.json({
      code: 0,
      msg: "注册失败-缺少参数",
    });
  }
});
```

登录接口 routes/users.js

登录接口稍加优化，我们昵称-用户id，用户头像，token等信息都返回

```
/* 登录请求 */
let jwt = require("jsonwebtoken");

router.get("/", function (req, res, next) {
  console.log(req.query);
  let { username, password } = req.query;
  User.findOne({ username, password }).then((r) => {
    console.log(r);
    if (r == null) {
      res.json({
        code: 0,
```

```
        msg: "登录失败",
      });
    } else {
      // 如果登录成功 返回jwt  ，并且在token 中存入用户名
      // 生成的token 的时候需要作者id 存入，下次请求的时候我们知道作者是谁
      let token = jwt.sign({ username: username, uid: r._id }, "test12345", {
        expiresIn: "365d",
        algorithm: "HS256",
      });

      res.json({
        code: 1,
        msg: "登录成功",
        token,
        uid: r._id,
        username,
        headImgUrl: r.headImgUrl,
        nickname: r.nickname,
      });
    }
  });
});

module.exports = router;
```

用户后台接口

发布文章 routes/articles.js

```
var express = require("express");
var router = express.Router();
let { Article, User } = require("../models/index");

/* GET home page. */
/*
 /api/articles
 发布文章
*/
router.post("/", function (req, res, next) {
  console.log(req.body);
  console.log(req.auth);
  // 作者的id 在
  // 发布文章的时候需要作者id
  // { username: 'admin', iat: 1668954424, exp: 1668954544 }
  Article.create({ ...req.body, author: req.auth.uid })
    .then((r) => {
      res.json({
```

```
        code: 1,
        msg: "发布文章成功",
        data: r,
      });
    })
    .catch((err) => {
      res.json({
        code: 1,
        msg: "发布文章成功",
      });
    });
});
```

获取文章列表

```
/*
 /api/articles/users/:uid
 根据用户id获取文章列表
*/
router.get("/users/:uid", function (req, res, next) {
  console.log(req.params); //{uid:11}
  Article.find({ author: req.params.uid })
    .populate("author", { password: 0 })
    .populate("coms")
    .then((r) => {
      res.json({
        code: 1,
        msg: "根据用户id获取文章列表成功",
        data: r,
      });
    })
    .catch((err) => {
      res.json({
        code: 1,
        msg: "根据用户id获取文章列表成功",
        data: err,
      });
    });
});
```

根据文章id获取文章详情

```
/*
 /api/articles/:aid
 根据文章id获取文章详情
*/
router.get("/:aid", function (req, res, next) {
  console.log(req.params); //{aid:11}
  //根据id 查询并且更新数据
  Article.findByIdAndUpdate(
    req.params.aid,
    { $inc: { views: 1 } },//views 增加1
```

```
    { new: true } //查询最新的结果
  )
    .populate("author", { password: 0 })
    .populate("coms")
    .then((r) => {
      res.json({
        code: 1,
        msg: "根据文章id获取文章详情",
        data: r,
      });
    });
});
```

删除文章

```
/*
 /api/articles/:aid
 根据文章id删除文章
*/
router.delete("/:aid", function (req, res, next) {
  console.log(req.params); //{uid:11}
  Article.findByIdAndDelete(req.params.aid).then((r) => {
    if (r) {
      res.json({
        code: 1,
        msg: "删除文章 成功",
      });
    } else {
      res.json({
        code: 0,
        msg: "删除文章-已经被删除",
      });
    }
    console.log(r);
  });
});
```

修改文章

```
/*
 /api/articles/:aid
 根据文章id编辑文章
*/
router.patch("/:aid", async function (req, res, next) {
  console.log(req.params); //{aid:11}
  console.log(req.body);
  let r = await Article.findByIdAndUpdate(
    req.params.aid,
    { ...req.body },
    { new: true }
  );
```

```
  res.json({
    code: 1,
    msg: "根据文章id修改文章",
    data: r,
  });
});

module.exports = router;
```

发布评论 routes/comments.js

```
/*
 /api/comments
 发布评论
*/
router.post("/", function (req, res, next) {
  console.log(req.body); //文章的id  评论内容
  console.log(req.auth);//uid
  // 作者的id 在
  // 发布文章的时候需要作者id
  // { username: 'admin', iat: 1668954424, exp: 1668954544 }
  Comment.create({ ...req.body, reply_user_id: req.auth.uid })
    .then((r) => {
      res.json({
        code: 1,
        msg: "发布评论成功",
        data: r,
      });
    })
    .catch((err) => {
      res.json({
        code: 1,
        msg: "发布评论失败",
      });
    });
});
```

评论列表展示

```
/*
 /api/articles/users/:uid
 根据文章id获取评论列表
*/
router.get("/articles/:aid", function (req, res, next) {
  console.log(req.params); //{uid:11}
  Comment.find({ article_id: req.params.aid })
    .populate("reply_user_id", { password: 0 })
    .then((r) => {
      res.json({
```

```
        code: 1,
        msg: "根据article id获取comment列表成功",
        data: r,
      });
    });
});
```

删除评论

```
/*
 /api/:cid
 根据评论id删除评论
*/
router.delete("/:cid", async function (req, res, next) {
  // req.auth.uid ===
  // 登录的uid  和 文章的作者的id一样--才能具有删除权限
  let { article_id } = await Comment.findById(req.params.cid);
  let { author } = await Article.findById(article_id);
  // req.auth.uid
  if (author == req.auth.uid) {
    console.log(req.params); //{uid:11}
    Article.findByIdAndDelete(req.params.aid).then((r) => {
      if (r) {
        res.json({
          code: 1,
          msg: "删除文章 成功",
        });
      } else {
        res.json({
          code: 0,
          msg: "删除文章-已经被删除",
        });
      }
      console.log(r);
    });
  } else {
    res.json({
      code: 0,
      msg: "删除文章-没有删除权限",
    });
  }
});
```

前端实现

前端页面中可以使用ajax技术发起请求，如果后端代码和前端代码不在同一个服务器上，就会出现跨域问题

跨域访问中间件

EE前端在开发中经常会被不能跨域访问所折磨，这次我们做后端，几句代码就可以支持跨域！（妈妈再也不用担心跨域报错了）

app.js中给所有的请求都设置响应头

```
//设置跨域访问
app.all("*", function (req, res, next) {
  //设置允许跨域的域名，*代表允许任意域名跨域
  res.header("Access-Control-Allow-Origin", req.headers.origin || '*');
  //允许的header类型
  res.header("Access-Control-Allow-Headers", "Content-Type, Authorization, X-
Requested-With");
  // //跨域允许的请求方式
  res.header("Access-Control-Allow-Methods", "PUT,POST,GET,DELETE,OPTIONS");
  // 可以带cookies
  res.header("Access-Control-Allow-Credentials", true);
  if (req.method == 'OPTIONS') {
    res.sendStatus(200).end();
  } else {
    next();
  }
})
```

跨域访问试试

皆苦支持跨域也可以安装cors插件 npm install cors

前端页面模板

我们提供了一套基本页面结构和内容已经写好的模块，我们只需要实现js部分，让页面动态变化即可

注册

登录

用户后台

博客列表-发布博客-删除-

编辑博客-评论发布-展示-删除

用户前台

获取文章列表-分页

根据文章id获取文章详情

评论列表展示

发布评论