LUO JIN 2020-05-17 (SUN)

MongDB

第一部分

1，数据库概述和环境搭建

1.1 为什么要使用数据库

动态网站的数据存储 http://www.czxy.com/artical?id=1 id不一样页面不一样

持久存储客户端通过表单手机的用户信息

数据库软件本身可以对数据进行搞笑的管理

1.2 什么数据库

数据库就是存储数据，可以将数据进行有序的分门别类的存储。他是独立于语言之外的软件，通过API操作 **Node.js** <--（**数据库提供的API**，**数据库反馈操作结果**）---> **数据库**

常见的数据库软件：mysql(php)，mongoDB，oracle

1.3 下载安装

1.4 MongoDB 可视化软件：compass

1.5 数据库

在一个数据库软件中，可以包含多个数据仓库，在每个数据仓库中可以包含多个数据ihe，每个数据集合可以包含多个文档（具体的数据）

术语	解释说明
database	数据库，mongoDB数据库软件中可以建立多个数据库
collection	集合，一组数据的集合，可以理解为JavaScript中的数组
document	文档，一条具体的数据，可以理解为JavaScript中的对象
field	字段，文档中的属性名称，可以理解为JavaScript中的对象属性

1.6 MongoDB第三方模块

使用node.js 操作MongoDB数据库需要依赖node,js的第三方包mongoose

Npm install mongoose

1.7 启动mongoDB

命令行工具：net start mongoDB / net stop mongodb

1.8 数据库连接

Connect

```
mongoose.connect('mongodb://localhost/playground')
    .then(() => console.log('数据库连接成功'))
    .catch(err => console.log('数据库连接失败', err));
```

1.9 创建数据库

不用显示的去创建数据库，如果正在使用的不存在，会自动帮你创建

3. MongoDB 增删查改操作

3.1 创建集合

1）创建集合规则

2）创建集合 mongoose.Schema构造函数的实例=创建集合

```
// 设定集合规则
const courseSchema = new mongoose.Schema({
    name: String,
    author: String,
    isPublished: Boolean
});
// 创建集合并应用规则
const Course = mongoose.model('Course', courseSchema); // courses
```

Model方法中的参数：

- 第一个参数：‘集合名称首字母大写‘，- 他自己在数据库创建的是小写而且是复数
- 第二个参数： 集合的规则

3.2 创建文档 - 向集合插入数据

3.2.1 方法一： 创建集合实例

1） 创建集合实例

2） 调用实例对象下的sava方法保存到数据库中

```
// 创建集合实例
const course = new Course({
    name: 'Node.js course',
    author: '黑马讲师',
    tags: ['node', 'backend'],
    isPublished: true
});
// 将数据保存到数据库中
course.save();
```

```
const mongoose = require('mongoose')

mongoose.connect('mongodb://localhost/playground', { useNewUrlParser: true,useUnifiedTopology: true  })
    .then(()=>{console.log("数据库连接成功！")})
    .catch(err => console.log(err, '数据库连接失败！'))


//1, 创建集合规则
const courseSchema = new mongoose.Schema ({
    name: String,  //课程名字
    author: String,  //课程作者
    isPublic: Boolean  //是否发布，状态
})
//2, 使用规则创建集合
const Course = mongoose.model('Course', courseSchema) //数据库中courses

//3. 插入文档数据
const course = new Course({
    name: 'nodejs basic',
    author: 'professor Li',
    isPublic: true
});
// 3.1 将文档插入到数据库中
course.save()
```

3.2.2 方法二：构造函数的create方法

```
Course.create({name: 'JavaScript基础', author: '黑马讲师', isPublish: true}, (err, doc) => {
    // 错误对象
    console.log(err)
    // 当前插入的文档
    console.log(doc)
});

Course.create({name: 'JavaScript基础', author: '黑马讲师', isPublish: true}
      .then(doc => console.log(doc))
      .catch(err => console.log(err))
```

Create也返回 Promise对象

参数一：对象

参数二：回调函数

在数据库中所有操作都是异

```
//3. 插入文档数据方法2
Course.create({
    name: 'JS',
    author: 'professor LUO',
    isPublic: false
},(err,result) => {
    console.log(err)
    console.log(result)
})
```

```
Course.create({name: 'JS2', author: 'professor LUO', isPublic: false})
.then(result => {
    console.log(result)
})
```

3.3 mongoDB 数据库导入数据

命令：mongoimport -d 数据库名称 -c 集合名称 -file 要导入的数据

但是我们不能使用现在，因为命令行回去计算机查找mongimport的可执行文件，**需要手动将命令添加到系统环境变量**中

mongoimport -d playground -c users --file ./user.json

3.4 查询文档

1） Find() **方法也返回promise对象**

```
// 根据条件查找文档（条件为空则查找所有文档）
Course.find().then(result => console.log(result))

//返回文档集合
[{
    _id: 5c0917ed37ec9b03c07cf95f,
    name: 'node.js基础',
    author: '黑马讲师'
},{
     _id: 5c09dea28acfb814980ff827,
     name: 'Javascript',
     author: '黑马讲师'
}]

// 根据条件查找文档
Course.findOne({name: 'node.js基础'}).then(result => console.log(result))

//返回文档
{
    _id: 5c0917ed37ec9b03c07cf95f,
    name: 'node.js基础',
    author: '黑马讲师'
}
```

//3，查询用户集合中的所有文档

//User.**find().**then(result => console.log(result));

//3.1 通过id字段查找

//User.find({_id: '5c09f267aeb04b22f8460968'}).then(result => console.log(result));

//4，返回一个，默认返回第一条

//User.**findOne().**then(result => console.log(result));

//4.1 可以加条件

User.**findOne({name: '李四'}).**then(result => console.log(result));

3） 匹配大于，小于

User.**find({age: {$gt: 20, $lt: 50}}).**then(result => console.log(result))

4） 匹配包含 **$in 网站搜索时，在后台查询的时候用**

User.find({hobbies:{$in:['敲代码']}}).then(result => console.log(result))

5） 选择要查询的字段

User.find().select('name email').then(result => console.log(result));

6） 根据年龄进行升序排列

User.find().sort('age').then(result => console.log(result));

7） 根据年龄进行降序排列

User.find().sort('-age').then(result => console.log(result));

8） Skip 跳过多少条数据，limit限制查询数量 = 翻页使用；跳过前两个文档，只显示3个数

User.find().skip(2).limit(3).then(result => console.log(result));

3.5 删除文档

1）删除单个，也返回promise对象，通过then；查找到一条文档并删除，返回删除的文档，

User.findOneAndDelete({_id: '5c09f2d9aeb04b22f846096b'}).then(result => console.log(result));

2）删除多个,不传全部删除，要小心，参数也是有条件的

User.deleteMany({}).then(result => console.log(result));

3.6 更新文档

1）更新文档

User.updateOne({查询条件}，{要修改的值}).then(result => console.log(result));

User.updateOne({name:'李四'},{name: '李狗蛋'}).then(result=>console.log(result));

2）更新多个

User.updateMany({查询条件}，{要修改的值}).then(result => console.log(result));

User.updateMany({},{age: 56}).then(result=>console.log(result));

3.6 mongoose验证

在创建集合规则时，可以设置当前字段的验证规则，验证失败就则插入失败

- Required: true 必传字段
- Min/max: 针对数值这样的字段类型

- required: true 必传字段
- minlength: 3 字符串最小长度
- maxlength: 20 字符串最大长度
- min: 2 数值最小为2
- max: 100 数值最大为100
- enum: ['html', 'css', 'javascript', 'node.js']
- trim: true 去除字符串两边的空格
- validate: 自定义验证器
- default: 默认值

```
const mongoose = require('mongoose')

mongoose.connect('mongodb://localhost/playground', {
    useNewUrlParser: true,useUnifiedTopology: true  })
    .then(()=>{console.log("数据库连接成功！")})
    .catch(err => console.log(err, '数据库连接失败！'))

const postSchema = new mongoose.Schema ({
    title: {
        type:String,
        required: [true, '请传入文章标题'], // 1,必选字段
        minlength: [2, '文章长度不能小于2'],
        maxlength: [5, '文章长度不能超过5'],
        trim: true  //2, 去除字符串两边的空格
    },
    age: {
        type:Number,
        min: [18, '年龄最小不能小于18岁'],
        max: [100, '年龄最大不能超过100岁']
    },
    publishDate: {
        type: Date,
        default: Date.now   // 3, 默认值
    },
    category: {
        type: String,
        enum: {   // 4, 枚举当前字段可以有的值
            values: ['html','css','js','node.js'],
            message: '分类名称要在一定的范围内才可以'
        }
    },
    author: {
        type: String,
        validate: {
            validator: (v) =>{
                // 5, 返回布尔值, true表示验证成功 v=要验证的值
                return v && v.length > 4
            },  // 6, 自定义错误信息
            message: '传入的值不符合验证规则'

        }
    }
})
const Post = mongoose.model('Post', postSchema) //数据库中courses
```

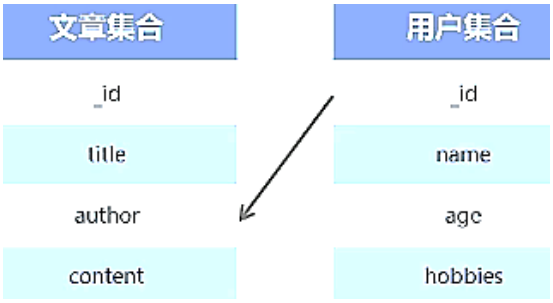
```
Post.create({
    title: 'hello',
    age: 35,
    category: 'java',
    author: 'bd'
}).then(result => console.log(result))
.catch(error => {   // 7, 获取错误的具体信息
    const err = error.errors;

    for(var attr in err) {
        console.log(err[attr]['message'])
    }
})
```

3.7 集合关联（实现）

通常**不同集合的数据之间是具有关系的**，例如文章信息和用户信息存在不同的集合中，但文章时某个用户发表的，要查询文章的所有信息包括发表用户，就需要用到集合关联

- 使用id对集合进行关联：查找当前发布文章的作者信息




```
// 用户集合
const User = mongoose.model('User', new mongoose.Schema({ name: { type:
String } }));
// 文章集合
const Post = mongoose.model('Post', new mongoose.Schema({
    title: { type: String },
    // 使用ID将文章集合和作者集合进行关联
    author: { type: mongoose.Schema.Types.ObjectId, ref: 'User' }
}));
//联合查询
Post.find()
      .populate('author')
      .then((err, result) => console.log(result));
```

- 1，创建用户集合规则 2，创建文章集合规则/数据库中创建首字母大写的用户集合和文章集合
- 在文章集合的作者中，关联用户集合 type: mongoose.Schema.Types.ObjectId，ref: 'User'
- 3，创建用户
- 4，创建一篇文章

- 5，查询作者信息 Post.find().populate('author').then(result=>console.log(result))

```
const mongoose = require('mongoose')

mongoose.connect('mongodb://localhost/playground', {
    useNewUrlParser: true,useUnifiedTopology: true  })
    .then(()=>{console.log("数据库连接成功！")})
    .catch(err => console.log(err, '数据库连接失败！'))

// 1，创建用户集合规则
const userSchema = new mongoose.Schema ({
    name: {
        type: String,
        required: true
    }
})

// 2，创建文章集合规则
const postSchema =  new mongoose.Schema ({
    title: {
        type: String
    },
    author: {
        type: mongoose.Schema.Types.ObjectId,
        ref: 'User'
    }
})

const User = mongoose.model('User',userSchema)
const Post = mongoose.model('Post', postSchema)

// 3，创建用户
//User.create({name: 'luojin'}).then(result=>console.log(result))

// 4，创建一篇文章
//Post.create({title: '你好', author: '5ec25ae06acb0d31482e2dc3'}).then(result=>console.log(result))

// 5，查询作者信息
Post.find().populate('author').then(result=>console.log(result))
```

3.8 案例： 用户的增删改查

1）搭建网站服务器，实现客户端与服务器端的通信

2）连接数据库，创建用户集合，向集合中插入文档

3）当用户访问/list时，将所有的用户信息查询出来

4）将用户访问/add时，呈现表单页面，并实现添加用户信息功能

6）当用户访问/modify时，呈现修改页面，并实现用户信息修改功能

7） 当用户访问/delete时，实现用户删除功能

第二部分

1，模板引擎得基础概念

1.1 模板引擎

第三方模块。

让开发者以更加友好得方式拼接字符串，是项目代码更加清晰，更加易于维护

```
// 未使用模板引擎的写法
var ary = [{ name: '张三', age: 20 }];
var str = '<ul>';
for (var i = 0; i < ary.length; i++) {
    str += '<li>\
        <span>'+ ary[i].name +'</span>\
        <span>'+ ary[i].age +'</span>\
    </li>';
}
str += '</ul>';
```

```
<!-- 使用模板引擎的写法 -->
<ul>
    {{each ary}}
        <li>{{$value.name}}</li>
        <li>{{$value.age}}</li>
    {{/each}}
</ul>
```

1.2 art-template模板引擎

1）种类很多，这个是由腾讯公司出品，目前运行最快的，公司使用最多的。

2）下载： npm install art-template

3) 使用const template = require('art-template') 引入

4）调用它，告诉模板引擎要拼接的数据和模板在哪 const html = template("模板路径"，数据);

1.3 案例

1） 后缀 .art

```
// 导入模板引擎模块
const template = require('art-template');
// 将特定模板与特定数据进行拼接
const html = template('./views/index.art',{
    data: {
        name: '张三',
        age: 20
    }
});
```

```
<div>
    <span>{{data.name}}</span>
    <span>{{data.age}}</span>
</div>
```

app.js — database\user ● | app.js — template × | index.

```
const template = require('art-template')
const path = require("path")

// template 方法 是用来拼接字符串的
// 1, 第一个字符串为 模板路径，绝对路径
// 2, 要在模板中显示的数据，对象类型

const views = path.join(__dirname, 'views', 'index.art')

const html = template(views, {
    name: 'haha',
    age: 20
})

console.log(html)
```

index.art

```
<!DOCTYPE html>
<html>
<head>
    <title></title>
</head>
<body>
    {{name}}
    {{age}}
</body>
</html>
```

2. 模板引擎语法

2.1 模板语法

1） art-template 同时支持两种模板语法：**标准语法，原始语法**

2）**标准语法**：容易就阅读 **{{数据}}**

原始语法：强大的裸机处理能力 **<%=数据 %>**

2.2 输出

```
<!-- 标准语法 -->
<h2>{{value}}</h2>
<h2>{{a ? b : c}}</h2>
<h2>{{a + b}}</h2>

<!-- 原始语法 -->
<h2><%= value %></h2>
<h2><%= a ? b : c %></h2>
<h2><%= a + b %></h2>
```

```
<!-- 标准语法 -->
<!-- {{name}}
{{age}} -->

<p>{{ name }}</p>
<p>{{ 1+1 }}</p>
<p>{{ 1 + 1 == 2 ? '=': '!='}}</p>

<p>{{ content }}</p>
<p>{{ @content }}</p>

<!-- 原始语法 -->
<p><%=name %></p>
<p><%= 1 + 2 %></p>
<p><%= 1 + 1 == 2 ? '=': '!=' %></p>

<p><%=content %></p>
<p><%-content %></p>
```

2.3 原文输出

1）如果数据中携带HTML标签，默认模板引擎不会解析标签，会将其转义后输出

2）标准语法：{{ @数据 }}

3）原始语法：<%-数据 %>

2.4 条件判断

1） 在模板中可以根据条件决定展是那块HTML代码

```
<!-- 标准语法 -->
{{if 条件}} ... {{/if}}
{{if v1}} ... {{else if v2}} ... {{/if}}

<!-- 原始语法 -->
<% if (value) { %> ... <% } %>
<% if (v1) { %> ... <% } else if (v2) { %> ... <% } %>
```

```
{{if age > 18}}
    年龄大于18
{{else if age < 15}}
    年龄小于15
{{else}}
    error
{{/if}}
```

```
<% if (age > 18) { %>
   年龄大于18了
<% } else if (age < 15) { %>
    年龄小于15
<% } else { %>
    error
<% } %>
```

2.5 循环

1）从数据库中查询数据，会返回一个数组，包含多个对象，如何展示在页面中？

2）标准语法：{{ each 数据}} {{/each}}

3) 原始语法： <% for() {%> <%}%>

```
<!-- 标准语法 -->
{{each target}}
    {{$index}} {{$value}}
{{/each}}
<!-- 原始语法 -->
<% for(var i = 0; i < target.length; i++){ %>
    <%= i %> <%= target[i] %>
<% } %>
```

03.art 文件清单.txt

```
<ul>
    {{each users}}
        <li>
            {{$value.name}}
            {{$value.age}}
            {{$value.sex}}
        </li>
    {{/each}}
</ul>



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

```
const template = require('art-template')
const path = require("path")

const views = path.join(__dirname, 'views', '03.art')

const html = template(views, {
    users: [{
        name: 'haha',
        age: 20,
        sex: 'nan'
    },{
        name: 'hehe',
        age: 80,
        sex: 'nv'
    },{
        name: 'hihi',
        age: 50,
        sex: 'nv'
    }]
})

console.log(html)

// 希望把这些数据展示在一个UL列表当中,
//有多少个对象就有多少个li
```

2.6 子模版

1）使用子模版可以净网站公共区块（头部，底部）抽离到单独的文件中去

2）标准语法：{{include ‘子模版的路径‘}}

3）原始语法： <%include(‘子模版的路径‘）%>

4）{{include ‘./header.art’}}

5）<%include(‘./header.art’）%>

04.art 文件清单.txt

```
{{include './common/header.art'}}

<div> {{ msg }} </div>
{{include './common/footer.art'}}

<% include('./common/footer.art') %>
```

04.js 04.art

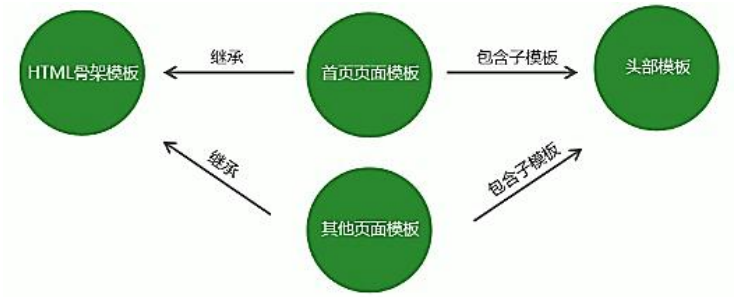
```
const template = require('art-template')
const path = require("path")

const views = path.join(__dirname, 'views', '04.art')
const html = template(views, {
    msg: '我是首页'
})
console.log(html)
```

2.7 模板继承

1）html骨架在每个页面也属于公共部分

2）模板继承可以将网站HTML股价抽离到单独的文件中，其他页面模板可以继承骨架文件



坑一：填充css内容

坑二：填充js内容

坑三：填充页面主体内容

填充一：main.css

填充二：index.js

填充三： Hello

2.8 模板继承示例

```
<!doctype html>
<html>
    <head>
        <meta charset="utf-8">
        <title>HTML骨架模板</title>
        {{block 'head'}}{{/block}}
    </head>
    <body>
        {{block 'content'}}{{/block}}
    </body>
</html>
```

```
<!--index.art 首页模板-->
{{extend './layout.art'}}
{{block 'head'}} <link rel="stylesheet" href="custom.css"> {{/block}}
{{block 'content'}} <p>This is just an awesome page.</p> {{/block}}
```

2.9 模板配置

1） 向模板中导入变量 template.defaults.imports.变量名 = 变量值

- 从数据库中查时间，是原始的，要处理
- 日期处理的第三方模块
- **{{dateFormat(time, 'yyyy-mm-dd')}}**

```
const dateFormat = require('dateformat');
const template = require('art-template')
const path = require("path")

const views = path.join(__dirname, 'views', '06.art')

// 导入模板变量
template.defaults.imports.dateFormat = dateFormat;

const html = template(views, {
    time: new Date()
})
console.log(html)
```

2）设置模板根目录 template.defaults.root = path.join(__dirname, 'views')

3）设置模板默认后缀

```
// 设置模板根目录
template.defaults.root = path.join(__dirname, 'views')

// 设置模板默认后缀
template.defaults.extnamem = '.art';

// 导入模板变量
template.defaults.imports.dateFormat = dateFormat;

const html = template('06', {
    time: new Date()
})
console.log(html)
```

```
// 设置模板根目录
template.defaults.root = path.join(__dirname, 'views')

// 设置模板默认后缀
template.defaults.extname = '.html';

// 导入模板变量
template.defaults.imports.dateFormat = dateFormat;

const html = template('06.art', {
    time: new Date()
});
console.log(template('07',{}));
console.log(html)
```

3， 案例

3.1 案例介绍 – 学生档案管理

- 目标：模板引擎应用，强化node.js项目制作流程
- 知识点：http请求响应，数据库，模板引擎，静态资源访问
- 后期优化代码

3.2 项目制作流程

1）建立项目文件夹并生成描述文件

2）创建数据库服务器实现客户端和服务器端通信

3）连接数据库并根据需求设计学院信息表

4）创建路由并实现页面模板呈递

- npm install router
- 下载模板引擎 npm i art-template

5）实现静态资源访问 – css, js, 图片文件等

6）实现学生信息添加功能

7）实现学生信息展示功能

3.3 第三方模块 router

- **功能**：实现路由
- **使用步骤：**1，获取路由对象

  2，调用路由对象提供的方法创建路由

3，启用路由，是路由生效

-代码： **npm install router**

```
const getRouter = require('router')
const router = getRouter();
router.get('/add', (req, res) => {
    res.end('Hello World!')
})
server.on('request', (req, res) => {
    router(req, res)
})
```

3.4 第三方模块 serve-static

- 功能：实现静态资源访问功能
- 使用步骤：

1，引入serve-static 模块获取创建静态资源服务功能的方法

2，调用方法创建静态资源服务并指定静态资源服务目录，当参数传递给他

3，启用静态资源服务功能：调用它

- 代码：npm install serve-static

```
const serveStatic = require('serve-static')
const serve = serveStatic('public')
server.on('request', () => {
    serve(req, res)
})
server.listen(3000)
```

3.5. 添加学生信息功能步骤分析

1）需要向服务器发请求的post，在模板的表单(action,method)中执行请求地址与请求方式

2）为每一个表单添加name属性

3）添加实现信息功能路由，处理请求

4）接收客户端请求来的学生信息

5）将学生信息太牛加到数据库

6）将页面重定向到学生信息页面

3.6 学生信息列表页面分析

1）将数据库中多有的学生信息查询出来

2）通过模板引擎将学生信息和HTML模板进行拼接

3）将拼接好的模板响应给客户端

- npm install dateformat 处理时间

model: 数据库相关代码

public: 静态资源

route: 路由

views: 模板

app.js: 入口文件，主文件

```
// 引入一些模块
// 配置模板引擎
// 连接数据库
// 创建网站服务器和客户端的请求和访问
// 监听端口

const http = require('http') // 引用http mongoose模块
const template = require('art-template') // 引入模板引擎
const path = require('path')
const serveStatic = require('serve-static') // 引入静态资源访问模块
const dateformat = require('dateformat') // 处理日期的第三方模块
const router = require('./route/index.js')
const serve = serveStatic(path.join(__dirname, 'pubilc')) // 使用静态资源访问功能

template.defaults.root = path.join(__dirname, 'views'); //配置模板根目录
template.defaults.imports.dateformat = dateformat //处理日期格式方法

require('./model/connect.js')


const app = http.createServer() //创建网站服务器
app.on('request', (req, res) =>{ //当客户端访问服务器端的时候
    router(req, res, () => {}) // 启用路由
    serve(req, res, () => {}) // 启用静态资源访问功能
});

app.listen(8080);
console.log('Server is booting');
```

```
const getRouter = require('router');// 引入路由模块
const router = getRouter();// 获取路由对象
const Student = require('../model/user.js');
const template = require('art-template');// 引入模板引擎
const querystring = require('querystring')// 处理字符串为对象格式

router.get('/add', (req, res) => {// 呈递学生档案信息页面,下载模板引擎 npm i art-template
    let html = template('index.art',{});
    res.end(html);
})

router.get('/list',async(req, res) => {// 呈递学生信息列表页面
    let students = await Student.find()// 查询all学生信息
    console.log(students)

    let html = template('list.art',{
        students: students
    });
    res.end(html);
})

router.post('/add',(req, res) => {// 实现学生信息添加功能
    let formData = '';// 接收post请求参数
    req.on('data', param => {
        formData += param;
    });
    req.on('end', async () => {// 将数据添加到数据库
        await Student.create(querystring.parse(formData))
        res.writeHead(301, {
            Location: '/list'
        });
        res.end()
    } )
})

module.exports = router

const mongoose = require('mongoose')// 数据库部分

mongoose.connect('mongodb://localhost/playground',{ useUnifiedTopology: true,useNewUrlParser: true })
.then(() => console.log('数据库连接成功'))
.catch(() => console.log('数据库连接失败'))
```

```
const mongoose = require('mongoose')
const studentSchema = new mongoose.Schema({// 设定学生集合规则
    name: {
        type: String,
        required: true,
        minlength: 2,
        maxlength: 10
    },
    age: {
        type: Number,
        min: 10,
        max: 40
    },
    sex: {
        type: String
    },
    email: {
        type: String
    },
    hobbies: [ String ],
    collage: String,
    enterDate: {
        type: Date,
        dafault: Date.now
    }
})
const Student = mongoose.model('Student', studentSchema);// 创建学生信息集合
module.exports = Student// 将学生信息集合导出
```