

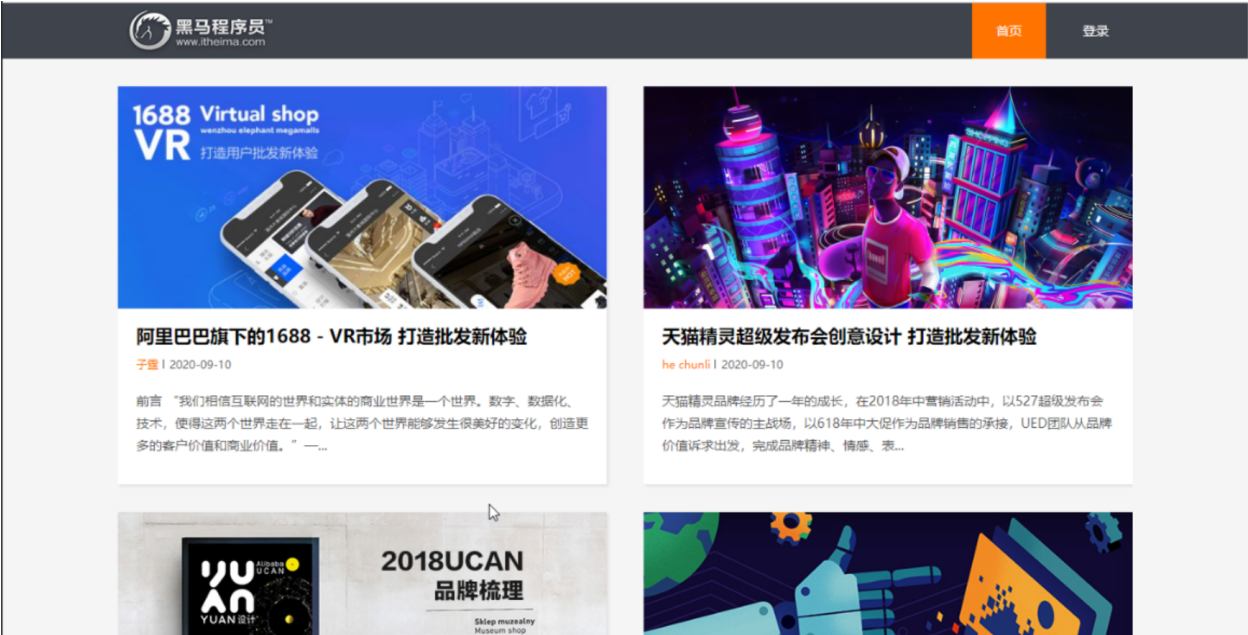
项目环境搭建

项目介绍

多人博客管理系统。

- 博客内容展示

1. 文章列表页面



2. 文章详情页面

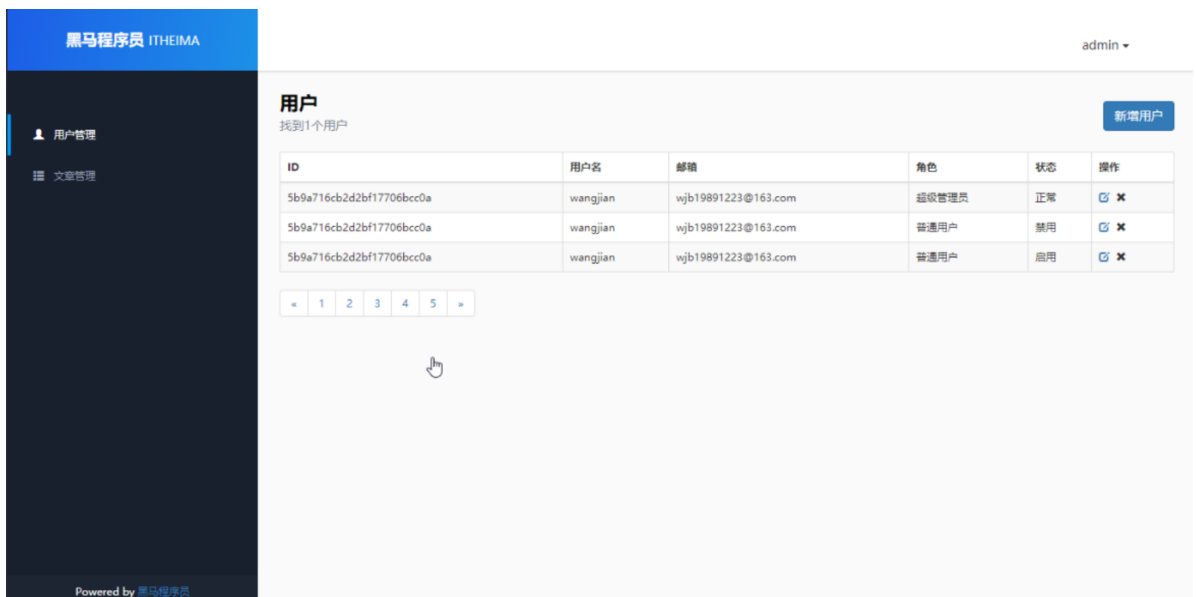


- 博客管理功能

1. 登录页面



2. 管理页面



案例初始化

1. 建立项目所需文件夹
 - public 静态资源
 - model 数据库操作
 - route 路由
 - views 模板
2. 初始化项目描述文件
 - npm init -y
3. 下载项目所需第三方模块
 - npm install express mongoose art-template express-art-template
4. 创建网站服务器

```
// 引用express框架
const express = require('express');
// 创建网站服务器
const app = express();
// 监听端口
app.listen(80);
console.log('网站服务器启动成功，请访问localhost')
```

5. 构建模块化路由

```
// 引入路由模块
const home = require('./route/home');
const admin = require('./route/admin');
// 为路由匹配请求路径
app.use('/home', home);
app.use('/admin', admin);
```

6. 构建博客管理页面模板

```
// 告诉express框架模板所在的位置
app.set('views', path.join(__dirname, 'views'));
// 告诉express框架模板的默认后缀是什么
app.set('view engine', 'art');
// 当渲染后缀为art的模板时 所使用的模板引擎是什么
app.engine('art', require('express-art-template'));

// 开放静态资源文件
app.use(express.static(path.join(__dirname, 'public')))
```

注意：模板中的相对路径是相对于地址栏中的请求路径的

所以在模板中引用其他文件中的资源的时候 需要使用服务器的绝对路径 也就是在src 或者 link的路径前加上“/”

```
<link rel="stylesheet" href="/admin/lib/bootstrap/css/bootstrap.min.css">
<link rel="stylesheet" href="/admin/css/base.css">
```

模板优化

抽离公共部分到单独模板中去

在common文件夹中创建头部公共模板文件

header.art

```

<!-- 头部 -->
<div class="header">
  <!-- 网站标志 -->
  <div class="logo fl">
    黑马程序员 <i>ITHEIMA</i>
  </div>
  <!-- /网站标志 -->
  <!-- 用户信息 -->
  <div class="info">
    <div class="profile dropdown fr">
      <span class="btn dropdown-toggle" data-toggle="dropdown">
        {{userInfo && userInfo.username}}
      <span class="caret"></span>
    </span>
    <ul class="dropdown-menu">
      <li><a href="user-edit.html">个人资料</a></li>
      <li><a href="/admin/logout">退出登录</a></li>
    </ul>
  </div>
</div>
<!-- /用户信息 -->
</div>
<!-- /头部 -->

```

创建aside.art

```

<!-- 侧边栏 -->
<div class="aside fl">
  <ul class="menu list-unstyled">
    <li>
      <a class="item active" href="user.html">
        <span class="glyphicon glyphicon-user"></span>
        用户管理
      </a>
    </li>
    <li>
      <a class="item" href="article.html">
        <span class="glyphicon glyphicon-th-list"></span>
        文章管理
      </a>
    </li>
  </ul>
  <div class="cppt">
    Powered by <a href="http://www.itheima.com/" target="_blank">黑马程序员</a>
  </div>
</div>
<!-- 侧边栏 -->

```

在use.art中引入

```

<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, user-scalable=no, initial-scale=1.0,
maximum-scale=1.0, minimum-scale=1.0">
  <title>Blog - Content Manager</title>
  <link rel="stylesheet" href="lib/bootstrap/css/bootstrap.min.css">
  <link rel="stylesheet" href="css/base.css">
</head>

<body>
  <!-- 头部 -->
  <!-- 子模板的相对路径相对的就是当前文件 因为它是由模板引擎解析的 而不是浏览器 -->
  {{include './common/header.art'}}
  <!-- /头部 -->
  <!-- 主体内容 -->
  <div class="content">
    <!-- 侧边栏 -->
    {{include './common/aside.art'}}
    <!-- 侧边栏 -->
    <div class="main">
      <!-- 分类标题 -->
      <div class="title">
        <h4>用户</h4>
        <span>找到1个用户</span>
        <a href="user-edit.html" class="btn btn-primary new">新增用户</a>
      </div>
      <!-- /分类标题 -->
      <!-- 内容列表 -->
      <table class="table table-striped table-bordered table-hover custom-table">
        <thead>
          <tr>
            <th>ID</th>
            <th>用户名</th>
            <th>邮箱</th>
            <th>角色</th>
            <th>状态</th>
            <th>操作</th>
          </tr>
        </thead>
        <tbody>
          <tr>
            <td>5b9a716cb2d2bf17706bcc0a</td>
            <td>wangjian</td>
            <td>wjb19891223@163.com</td>
            <td>超级管理员</td>
            <td>正常</td>
            <td>
              <a href="user-edit.html" class="glyphicon glyphicon-edit"></a>

              <i class="glyphicon glyphicon-remove" data-toggle="modal" data-

```

```

target=".confirm-modal"></i>
    </td>
</tr>
<tr>
    <td>5b9a716cb2d2bf17706bcc0a</td>
    <td>wangjian</td>
    <td>wjb19891223@163.com</td>
    <td>普通用户</td>
    <td>禁用</td>
    <td>
        <a href="user-edit.html" class="glyphicon glyphicon-edit"></a>
        <i class="glyphicon glyphicon-remove" data-toggle="modal" data-
target=".confirm-modal"></i>
    </td>
</tr>
<tr>
    <td>5b9a716cb2d2bf17706bcc0a</td>
    <td>wangjian</td>
    <td>wjb19891223@163.com</td>
    <td>普通用户</td>
    <td>启用</td>
    <td>
        <a href="user-edit.html" class="glyphicon glyphicon-edit"></a>
        <i class="glyphicon glyphicon-remove" data-toggle="modal" data-
target=".confirm-modal"></i>
    </td>
</tr>
</tbody>
</table>
<!-- /内容列表 -->
<!-- 分页 -->
<ul class="pagination">
    <li>
        <a href="#">
            <span>&laquo;</span>
        </a>
    </li>
    <li><a href="#">1</a></li>
    <li><a href="#">2</a></li>
    <li><a href="#">3</a></li>
    <li><a href="#">4</a></li>
    <li><a href="#">5</a></li>
    <li>
        <a href="#">
            <span>&raquo;</span>
        </a>
    </li>
</ul>
<!-- /分页 -->
</div>
</div>
<!-- /主体内容 -->

<!-- 删除确认弹出框 -->

```

```

<div class="modal fade confirm-modal">
  <div class="modal-dialog modal-lg">
    <form class="modal-content">
      <div class="modal-header">
        <button type="button" class="close" data-dismiss="modal"><span>&times;
</span></button>
        <h4 class="modal-title">请确认</h4>
      </div>
      <div class="modal-body">
        <p>您确定要删除这个用户吗?</p>
      </div>
      <div class="modal-footer">
        <button type="button" class="btn btn-default" data-dismiss="modal">取消
</button>
        <input type="submit" class="btn btn-primary">
      </div>
    </form>
  </div>
</div>
<!-- /删除确认弹出框 -->
<script src="lib/jquery/dist/jquery.min.js"></script>
<script src="lib/bootstrap/js/bootstrap.min.js"></script>
</body>

</html>

```

接下来对模板结构做进一步优化 提取页面结构

创建layout.art文件

```

<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, user-scalable=no, initial-scale=1.0,
maximum-scale=1.0, minimum-scale=1.0">
  <title>Blog - Content Manager</title>
  <link rel="stylesheet" href="/admin/lib/bootstrap/css/bootstrap.min.css">
  <link rel="stylesheet" href="/admin/css/base.css">
  {{block 'link'}}{{/block}}
</head>

<body>
  {{block 'main'}} {{/block}}
  <script src="/admin/lib/jquery/dist/jquery.min.js"></script>
  <script src="/admin/lib/bootstrap/js/bootstrap.min.js"></script>
  <script src="/admin/js/common.js"></script>
  {{block 'script'}} {{/block}}
</body>

</html>

```

修改user.art 模板 让其继承 layout.art的结构


```

{{extend './common/layout.art'}}

{{block 'main'}}
    <!-- 子模板的相对路径相对的就是当前文件 因为它是由模板引擎解析的 而不是浏览器 -->
    {{include './common/header.art'}}
    <!-- /头部 -->
    <!-- 主体内容 -->
    <div class="content">
        <!-- 侧边栏 -->
        {{include './common/aside.art'}}
        <!-- 侧边栏 -->
        <div class="main">
            <!-- 分类标题 -->
            <div class="title">
                <h4>用户</h4>
                <span>找到1个用户</span>
                <a href="user-edit.html" class="btn btn-primary new">新增用户</a>
            </div>
            <!-- /分类标题 -->
            <!-- 内容列表 -->
            <table class="table table-striped table-bordered table-hover custom-table">
                <thead>
                    <tr>
                        <th>ID</th>
                        <th>用户名</th>
                        <th>邮箱</th>
                        <th>角色</th>
                        <th>状态</th>
                        <th>操作</th>
                    </tr>
                </thead>
                <tbody>
                    <tr>
                        <td>5b9a716cb2d2bf17706bcc0a</td>
                        <td>wangjian</td>
                        <td>wjb19891223@163.com</td>
                        <td>超级管理员</td>
                        <td>正常</td>
                        <td>
                            <a href="user-edit.html" class="glyphicon glyphicon-edit"></a>
                            <i class="glyphicon glyphicon-remove" data-toggle="modal" data-
target=".confirm-modal"></i>
                        </td>
                    </tr>
                    <tr>
                        <td>5b9a716cb2d2bf17706bcc0a</td>
                        <td>wangjian</td>
                        <td>wjb19891223@163.com</td>
                        <td>普通用户</td>
                        <td>禁用</td>
                        <td>
                            <a href="user-edit.html" class="glyphicon glyphicon-edit"></a>
                            <i class="glyphicon glyphicon-remove" data-toggle="modal" data-

```



```
        <button type="button" class="btn btn-default" data-dismiss="modal">取消
    </button>
    <input type="submit" class="btn btn-primary">
  </div>
</form>
</div>
</div>
{{/block}}
```

项目功能实现

登录

1. 创建用户集合，初始化用户

- 连接数据库

```
// 引入mongoose第三方模块
const mongoose = require('mongoose');
// 连接数据库
mongoose.connect('mongodb://localhost/blog', {useNewUrlParser: true })
  .then(() => console.log('数据库连接成功'))
  .catch(() => console.log('数据库连接失败'))
```

- 创建用户集合

```

// 创建用户集合
// 引入mongoose第三方模块
const mongoose = require('mongoose');
// 创建用户集合规则
const userSchema = new mongoose.Schema({
  username: {
    type: String,
    required: true,
    minlength: 2,
    maxlength: 20
  },
  email: {
    type: String,
    // 保证邮箱地址在插入数据库时不重复
    unique: true,
    required: true
  },
  password: {
    type: String,
    required: true
  },
  // admin 超级管理员
  // normal 普通用户
  role: {
    type: String,
    required: true
  },
  // 0 启用状态
  // 1 禁用状态
  state: {
    type: Number,
    default: 0
  }
});

// 创建集合
const User = mongoose.model('User', userSchema);

// 将用户集合做为模块成员进行导出
module.exports = {
  User
}

```

- 初始化用户

```
// 为了方便我们实现登录功能我们先初始化一个用户
User.create({
  username: 'itheima',
  email: 'itheima@itcast.cn',
  password: '123456',
  role: 'admin',
  state: 0
}).then(() => {
  console.log('用户创建成功')
}).catch(() => {
  console.log('用户创建失败')
})
// 这段代码用来测试 执行过一次 创建过admin之后就应该删除或者注释掉
```

2. 为登录表单项设置请求地址、请求方式以及表单项name属性

```
<form action="/admin/login" method="post" id="loginForm">
  <div class="form-group">
    <label>邮件</label>
    <input name="email" type="email" class="form-control"
placeholder="请输入邮件地址">
  </div>
  <div class="form-group">
    <label>密码</label>
    <input name="password" type="password" class="form-control"
placeholder="请输入密码">
  </div>
  <button type="submit" class="btn btn-primary">登录</button>
</form>
```

3. 当用户点击登录按钮时，客户端验证用户是否填写了登录表单

4. 如果其中一项没有输入，阻止表单提交

```

<script type="text/javascript">
  // 为表单添加提交事件
  $('#loginForm').on('submit', function () {
    // 获取到表单中用户输入的内容
    var result = serializeToJson($(this))
    // 如果用户没有输入邮件地址的话
    if (result.email.trim().length == 0) {
      alert('请输入邮件地址');
      // 阻止程序向下执行
      return false;
    }
    // 如果用户没有输入密码
    if (result.password.trim().length == 0) {
      alert('请输入密码');
      // 阻止程序向下执行
      return false;
    }
  });
</script>

```

5. 服务器端接收请求参数，验证用户是否填写了登录表单

```

// 添加实现登录请求的路由
admin.post('/login', (req, res) => {
  // 接收请求参数
  const {email, password} = req.body;
  // 如果用户没有输入邮件地址
  // if (email.trim().length == 0 || password.trim().length == 0) return
  res.status(400).send('<h4>邮件地址或者密码错误</h4>');
  if (email.trim().length == 0 || password.trim().length == 0) return
  res.status(400).render('admin/error', {msg: '邮件地址或者密码错误'});
});

```

6. 如果其中一项没有输入，为客户端做出响应，阻止程序向下执行

7. 根据邮箱地址查询用户信息

```

const { User } = require('../model/user')

admin.post('/login', (req,res)=>{
  // 接收请求参数
  const {email, password} = req.body;
  // 如果用户没有输入邮件地址
  // if (email.trim().length == 0 || password.trim().length == 0) return
  res.status(400).send('<h4>邮件地址或者密码错误</h4>');
  if (email.trim().length == 0 || password.trim().length == 0) return
  res.status(400).render('admin/error', {msg: '邮件地址或者密码错误'});
  let user = await User.findOne({email});
  // 查询到了用户
  if (user) {
    // 将客户端传递过来的密码和用户信息中的密码进行比对
    // true 比对成功
    // false 对比失败

    // 如果密码比对成功
    if ( password = user.password ) {
      // 登录成功
      res.send('登录成功');
    } else {
      // 没有查询到用户
      res.status(400).render('admin/error', {msg: '邮箱地址或者密码错误'})
    }
  } else {
    // 没有查询到用户
    res.status(400).render('admin/error', {msg: '邮箱地址或者密码错误'})
  }
});

```

8. 如果用户不存在，为客户端做出响应，阻止程序向下执行
9. 如果用户存在，将用户名和密码进行比对
10. 比对成功，用户登录成功
11. 比对失败，用户登录失败

退出

删除session

删除cookie

重定向到用户登录页面

```

module.exports = (req, res) => {
  // 删除session
  req.session.destroy(function () {
    // 删除cookie
    res.clearCookie('connect.sid');
    // 重定向到用户登录页面
    res.redirect('/admin/login');
  });
}

```

登录拦截

添加一个中间件用于拦截请求

```

// 拦截请求 判断用户登录状态
app.use('/admin', (req, res, next) => {
  // 判断用户访问的是否是登录页面
  // 判断用户的登录状态
  // 如果用户是登录的 将请求放行
  // 如果用户不是登录的 将请求重定向到登录页面
  if (req.url !== '/login' && !req.session.username) {
    res.redirect('/admin/login');
  } else {
    // 用户是登录状态 将请求放行
    next();
  }
});

```

新增用户

1. 为用户列表页面的新增用户按钮添加链接
2. 添加一个连接对应的路由，在路由处理函数中渲染新增用户模板

```

// 创建用户编辑页面路由
admin.get('/user-edit', require('./admin/user-edit'));
// 创建实现用户添加功能路由
admin.post('/user-edit', require('./admin/user-edit-fn'));

```

3. 为新增用户表单指定请求地址、请求方式、为表单项添加name属性


```

<form class="form-container" action="{{link}}" method="post">
    <div class="form-group">
        <label>用户名</label>
        <input name="username" type="text" class="form-control" placeholder="请输入用户名" value="{{user && user.username}}">
    </div>
    <div class="form-group">
        <label>邮箱</label>
        <input type="email" class="form-control" placeholder="请输入邮箱地址"
name="email" value="{{user && user.email}}">
    </div>
    <div class="form-group">
        <label>密码</label>
        <input type="password" class="form-control" placeholder="请输入密码"
name="password">
    </div>
    <div class="form-group">
        <label>角色</label>
        <select class="form-control" name="role">
            <option value="normal" {{user && user.role == 'normal' ? 'selected'
: ''}}>普通用户</option>
            <option value="admin" {{user && user.role == 'admin' ? 'selected' :
''}}>超级管理员</option>
        </select>
    </div>
    <div class="form-group">
        <label>状态</label>
        <select class="form-control" name="state">
            <option value="0" {{user && user.state == '0' ? 'selected' : ''}}>
启用</option>
            <option value="1" {{user && user.state == '1' ? 'selected' : ''}}>
禁用</option>
        </select>
    </div>
    <div class="buttons">
        <input type="submit" class="btn btn-primary" value="{{button}}">
    </div>
</form>

```

4. 增加实现添加用户的功能路由
5. 接收到客户端传递过来的请求参数
6. 对请求参数的格式进行验证

```
// 定义对象的验证规则
const schema = {
  username: Joi.string().min(2).max(12).required().error(new Error('用户名不符合验证规则')),
  email: Joi.string().email().required().error(new Error('邮箱格式不符合要求')),
  password: Joi.string().regex(/^[a-zA-Z0-9]{3,30}$/).required().error(new Error('密码格式不符合要求')),
  role: Joi.string().valid('normal', 'admin').required().error(new Error('角色值非法')),
  state: Joi.number().valid(0, 1).required().error(new Error('状态值非法'))
};
// 实施验证
Joi.validate(user, schema);
```

7. 验证当前要注册的邮箱地址是否已经注册过

```
// 根据邮箱地址查询用户是否存在
let user = await User.findOne({email: req.body.email});
// 如果用户已经存在 邮箱地址已经被别人占用
if (user) {
  // 重定向回用户添加页面
  return res.redirect(`/admin/user-edit?message=邮箱地址已经被占用`);
}
```

8. 对密码进行加密处理

```
// 对密码进行加密处理
// 生成随机字符串
const salt = await bcrypt.genSalt(10);
// 加密
const password = await bcrypt.hash(req.body.password, salt);
// 替换密码
req.body.password = password;
```

9. 将用户信息添加到数据库中

10. 重定向页面到用户列表页面

```
// 将用户信息添加到数据库中
await User.create(req.body);
// 将页面重定向到用户列表页面
res.redirect('/admin/user');
```

用户列表展示-数据分页

展示查询到的数据

```

<tbody>
  {{each users}}
  <tr>
    <td>{{@$value._id}}</td>
    <td>{{@$value.username}}</td>
    <td>{{@$value.email}}</td>
    <td>{{@$value.role == 'admin' ? '超级管理员': '普通用户'}}</td>
    <td>{{@$value.state == 0 ? '启用': '禁用'}}</td>
    <td>
      <a href="/admin/user-edit?id={{@$value._id}}" class="glyphicon glyphicon-edit"></a>
      <i class="glyphicon glyphicon-remove" data-toggle="modal" data-target=".confirm-modal"></i>
    </td>
  </tr>
  {{/each}}
</tbody>

```

当数据库中的数据非常多时，数据需要分批次显示，这时就需要用到数据分页功能

分页功能的核心要素

1. 当前页，用户通过点击上一页或者下一页或者页码产生，客户端通过get参数方式传递到服务器端
2. 总页数，根据总页数判断当前页是否为最后一页，根据判断结果做响应操作

计算总页数

```

// 接收客户端传递过来的当前页参数
let page = req.query.page || 1;
// 每一页显示的数据条数
let pagesize = 10;
// 查询用户数据的总数
let count = await User.countDocuments({});
// 总页数
let total = Math.ceil(count / pagesize);

```

```

limit(2) // limit 限制查询数量 传入每页显示的数据数量
skip(1) // skip 跳过多少条数据 传入显示数据的开始位置

// 数据开始查询位置=（当前页-1）* 每页显示的数据条数

```

获取数据开始查询的位置

```

// 页码对应的数据查询开始位置
let start = (page - 1) * pagesize;

```

在列表展示页面生成分页器

```

<!-- 分页 -->
    <ul class="pagination">
      <li style="display: <%=page-1 < 1 ? 'none' : 'inline' %>">
        <a href="/admin/user?page=<%=page-1%>">
          <span>&laquo;</span>
        </a>
      </li>
      <% for (var i = 1; i <= total; i++) { %>
      <li><a href="/admin/user?page=<%=i %>">{{i}}</a></li>
      <% } %>
      <li style="display: <%= page-0+1 > total ? 'none' : 'inline' %>">
        <a href="/admin/user?page=<%=page-0+1%>">
          <span>&raquo;</span>
        </a>
      </li>
    </ul>
<!-- /分页 -->

```

完成分页查询

```

// 将用户信息从数据库中查询出来
let users = await User.find({}).limit(pagesize).skip(start)
// 渲染用户列表模块
res.render('admin/user', {
  users: users,
  page: page,
  total: total
});

```

用户信息修改

将要修改的用户ID传递到服务器端

```

// 在模板文件里的链接中添加参数
<a href="/admin/user-edit?id={{@$value._id}}" class="glyphicon glyphicon-edit"></a>

```

```

// 获取到地址栏中的id参数
const { message, id } = req.query;

// 如果当前传递了id参数
if (id) {
  // 修改操作
  let user = await User.findOne({_id: id});

  // 渲染用户编辑页面(修改)
  res.render('admin/user-edit', {
    message: message,
    user: user,
    link: '/admin/user-modify?id=' + id,
    button: '修改'
  });
} else {
  // 添加操作
  res.render('admin/user-edit', {
    message: message,
    link: '/admin/user-edit',
    button: '添加'
  });
}

```

建立用户信息修改功能对应的路由

```

// 用户修改功能路由
admin.post('/user-modify', require('./admin/user-modify'));

```

接收客户端表单传递过来的请求参数

```

// 接收客户端传递过来的请求参数
const { username, email, role, state, password } = req.body;

```

根据id查询用户信息，并将客户端传递过来的密码和数据库中的密码进行比对

```

// 即将要修改的用户id
const id = req.query.id;
// 根据id查询用户信息
let user = await User.findOne({_id: id});
// 密码比对
const isValid = await bcrypt.compare(password, user.password);

```

如果比对失败，对客户端做出响应

```

// 密码比对失败
let obj = {path: '/admin/user-edit', message: '密码比对失败,不能进行用户信息的修改', id:
id}
next(JSON.stringify(obj));

```

app.js中的错误处理中间件

```
app.use((err, req, res, next) => {
  // 将字符串对象转换为对象类型
  // JSON.parse()
  const result = JSON.parse(err);
  // {path: '/admin/user-edit', message: '密码比对失败,不能进行用户信息的修改', id: id}
  let params = [];
  for (let attr in result) {
    if (attr !== 'path') {
      params.push(attr + '=' + result[attr]);
    }
  }
  res.redirect(`${result.path}?${params.join('&')}`);
})
```

如果密码对比成功，将用户信息更新到数据库中

```
// 将用户信息更新到数据库中
await User.updateOne({_id: id}, {
  username: username,
  email: email,
  role: role,
  state: state
});

// 将页面重定向到用户列表页面
res.redirect('/admin/user');
```

用户删除功能

- 在确认删除框中添加隐藏域用以存储要删除用户的ID值
- 为删除按钮添自定义属性用以存储要删除用户的ID值
- 为删除按钮添加点击事件，在点击事件处理函数中获取自定义属性中存储的ID值并将ID值存储在表单的隐藏域中
- 为删除表单添加提交地址以及提交方式
- 在服务器端建立删除功能路由
- 接收客户端传递过来的id参数
- 根据id删除用户

```

<!-- 删除确认弹出框 -->
<div class="modal fade confirm-modal">
  <div class="modal-dialog modal-lg">
    <form class="modal-content" action="/admin/delete" method="get">
      <div class="modal-header">
        <button type="button" class="close" data-dismiss="modal"><span>&times;
</span></button>

        <h4 class="modal-title">请确认</h4>
      </div>
      <div class="modal-body">
        <p>您确定要删除这个用户吗?</p>
        <input type="hidden" name="id" id="deleteUserId">
      </div>
      <div class="modal-footer">
        <button type="button" class="btn btn-default" data-dismiss="modal">取消
</button>

        <input type="submit" class="btn btn-primary">
      </div>
    </form>
  </div>
</div>

```

```

module.exports = async (req, res) => {
  // 获取要删除的用户id
  // res.send(req.query.id)
  // 根据id删除用户
  await User.findOneAndDelete({_id: req.query.id});
  // 将页面重定向到用户列表页面
  res.redirect('/admin/user');
}

```

给侧边栏添加选中状态

```

<li>
  <a class="item {{currentLink == 'user' ? 'active' : ''}}" href="/admin/user">
    <span class="glyphicon glyphicon-user"></span>
    用户管理
  </a>
</li>
<li>
  <a class="item {{currentLink == 'article' ? 'active' : ''}}"
href="/admin/article">
    <span class="glyphicon glyphicon-th-list"></span>
    文章管理
  </a>
</li>

```

```

// 标识 标识当前访问的是用户管理页面
req.app.locals.currentLink = 'user';

```

```
// 标识 标识当前访问的是文章管理页面
req.app.locals.currentLink = 'article';
```

文章添加功能

创建文章集合

```
// 1.引入mongoose模块
const mongoose = require('mongoose');

// 2.创建文章集合规则
const articleSchema = new mongoose.Schema({
  title: {
    type: String,
    maxlength: 20,
    minlength: 4,
    required: [true, '请填写文章标题']
  },
  author: {
    type: mongoose.Schema.Types.ObjectId,
    ref: 'User',
    required: [true, '请传递作者']
  },
  publishDate: {
    type: Date,
    default: Date.now
  },
  cover: {
    type: String,
    default: null
  },
  content: {
    type: String
  }
});

// 3.根据规则创建集合
const Article = mongoose.model('Article', articleSchema);

// 4.将集合做为模块成员进行导出
module.exports = {
  Article
}
```

文件上传功能

使用formidable实现文件上传

form中的enctype的作用是指定表单数据的编码类型 默认值是 application/x-www-form-urlencoded

如果涉及达到了表单中文件上传 则需要设置为 multipart/form-data 将表单数据编码成二进制类型


```

<!--
    enctype 指定表单数据的编码类型
    application/x-www-form-urlencoded
    name=zhangsan&age=20
    multipart/form-data 将表单数据编码成二进制类型
-->
<!-- /分类标题 -->
<form class="form-container" action="/admin/article-add" method="post"
enctype="multipart/form-data">
    <div class="form-group">
        <label>标题</label>
        <input type="text" class="form-control" placeholder="请输入文章标题"
name="title">
    </div>
    <div class="form-group">
        <label>作者</label>
        <input name="author" type="text" class="form-control" readonly value="
{{@userInfo._id}}">
    </div>
    <div class="form-group">
        <label>发布时间</label>
        <input name="publishDate" type="date" class="form-control">
    </div>

    <div class="form-group">
        <label for="exampleInputFile">文章封面</label>
        <!--
            multiple 允许用户一次性选择多个文件
        -->
        <input type="file" name="cover" id="file" >
        <div class="thumbnail-waper">
            <img class="img-thumbnail" src="" id="preview">
        </div>
    </div>
    <div class="form-group">
        <label>内容</label>
        <textarea name="content" class="form-control" id="editor"></textarea>
    </div>
    <div class="buttons">
        <input type="submit" class="btn btn-primary">
    </div>
</form>

```

利用formidable实现文件上传

```

// 引入formidable第三方模块
const formidable = require('formidable');
const path = require('path');
const { Article } = require('../../model/article')

module.exports = (req, res) => {
  // 1.创建表单解析对象
  const form = new formidable.IncomingForm();
  // 2.配置上传文件的存放位置
  form.uploadDir = path.join(__dirname, '..', '..', 'public', 'uploads');
  // 3.保留上传文件的后缀
  form.keepExtensions = true;
  // 4.解析表单
  form.parse(req, async (err, fields, files) => {
    // 1.err错误对象 如果表单解析失败 err里面存储错误信息 如果表单解析成功 err将会是null
    // 2.fields 对象类型 保存普通表单数据
    // 3.files 对象类型 保存了和上传文件相关的数据
    res.send(files.cover.path.split('public')[1])
  })
  // res.send('ok');
}

```

实现图片的预览功能

使用FileReader 读取本地文件

```

// 选择文件上传控件
var file = document.querySelector('#file');
var preview = document.querySelector('#preview');
// 当用户选择完文件以后
file.onchange = function () {
  // 1 创建文件读取对象
  var reader = new FileReader();
  // 用户选择的文件列表
  // console.log(this.files[0])
  // 2 读取文件
  reader.readAsDataURL(this.files[0]);
  // 3 监听onload事件
  reader.onload = function () {
    console.log(reader.result)
    // 将文件读取的结果显示在页面中
    preview.src = reader.result;
  }
}

```

实现文章添加功能

```

form.parse(req, async (err, fields, files) => {
  // 1.err错误对象 如果表单解析失败 err里面存储错误信息 如果表单解析成功 err将会是null
  // 2.fields 对象类型 保存普通表单数据
  // 3.files 对象类型 保存了和上传文件相关的数据
  // res.send(files.cover.path.split('public')[1])
  await Article.create({
    title: fields.title,
    author: fields.author,
    publishDate: fields.publishDate,
    cover: files.cover.path.split('public')[1],
    content: fields.content,
  });
  // 将页面重定向到文章列表页面
  res.redirect('/admin/article');
})

```

文章列表页面展示

```

// 将文章集合的构造函数导入到当前文件中
const { Article } = require('../model/article');

module.exports = async (req, res) => {

  req.app.locals.currentLink = 'article'
  let articles = await Article.find().populate('author')
  res.send(articles)
}

```

实现文章列表展示中的分页

```

<tbody>
  {{each articles.records}}
    <tr>
      <td>{{@$value._id}}</td>
      <td>{{ $value.title}}</td>
      <td>{{dateFormat($value.publishDate, 'yyyy-mm-dd')}}</td>
      <td>{{ $value.author.username}}</td>
      <td>
        <a href="article-edit.html" class="glyphicon glyphicon-edit"></a>
        <i class="glyphicon glyphicon-remove" data-toggle="modal" data-
target=".confirm-modal"></i>
      </td>
    </tr>
  {{/each}}
</tbody>

```

<!-- 分页 -->

```

<ul class="pagination">
  {{if articles.page > 1}}
    <li>
      <a href="/admin/article?page={{articles.page - 1}}">
        <span>&laquo;</span>
      </a>
    </li>
  {{/if}}

  {{each articles.display}}
    <li><a href="/admin/article?page={{ $value }}">{{ $value }}</a></li>
  {{/each}}

  {{if articles.page < articles.pages}}
    <li>
      <a href="/admin/article?page={{articles.page - 0 + 1}}">
        <span>&raquo;</span>
      </a>
    </li>
  {{/if}}
</ul>
<!-- /分页 -->

```

```
// page 指定当前页
// suze 指定每页显示的数据条数
// display 指定客户端要显示的页码数量
// exec 向数据库中发送查询请求
// 查询所有文章数据
let articles = await
pagination(Article).find().page(page).size(2).display(3).populate('author').exec();

// res.send(articles);

// 渲染文章列表页面模板
res.render('admin/article.art', {
  articles: articles
});
```

博客前台功能

添加前台路由

```
// 为路由匹配请求路径
app.use('/home', home);
```

```
// 引用express框架
const express = require('express');
// 创建博客展示页面路由
const home = express.Router();

// 博客前台首页的展示页面
home.get('/', require('./home/index'));

// 博客前台文章详情展示页面
home.get('/article', require('./home/article'));

// 将路由对象做为模块成员进行导出
module.exports = home;
```

博客首页

```

const { Article } = require('../..../model/article');
// 导入分页模块
const pagination = require('mongoose-sex-page');

module.exports = async (req, res) => {
  // 获取页码值
  const page = req.query.page;

  // 从数据库中查询数据
  let result = await
  pagination(Article).page(page).size(4).display(5).find().populate('author').exec();

  // res.send('欢迎来到博客首页')
  // 渲染模板并传递数据
  res.render('home/default.art', {
    result: result
  });
}

```

数据处理

```

<div class="content">
  <a class="article-title" href="#">{{value.title}}</a>
  <div class="article-info">
    <span class="author">{{value.author.username}}</span>
    <span>{{dateFormat(value.publishDate, 'yyyy-mm-dd')}}</span>
  </div>
  <div class="brief">
    {{@value.content.replace(/<[>]+>/g, '').substr(0, 90) + '...'}}
  </div>
</div>

```

分页页码

```

<!-- 分页开始 -->
<div class="page w1100">
  {{if result.page > 1}}
  <a href="/home/?page={{result.page-1}}">上一页</a>
  {{/if}}
  {{each result.display}}
  <a href="/home/?page={{value}}" class="{{value == result.page ? 'active' : ''}}">
  {{value}}</a>
  {{/each}}
  {{if result.page < result.pages}}
  <a href="/home/?page={{result.page - 0 + 1}}">下一页</a>
  {{/if}}
</div>
<!-- 分页结束 -->

```

文章详情

添加跳转链接

```
<a href="/home/article?id={{@$value._id}}" class="thumbnail">
  
</a>
```

获取详情页数据

```
// 接收客户端传递过来的文章id值
const id = req.query.id;
// 根据id查询文章详细信息
let article = await Article.findOne({_id: id}).populate('author');
```

文章评论

1. 创建评论集合

```
// 引入mongoose模块
const mongoose = require('mongoose');

// 创建评论集合规则
const commentSchema = new mongoose.Schema({
  // 文章id
  aid: {
    type: mongoose.Schema.Types.ObjectId,
    ref: 'Article'
  },
  // 评论人用户id
  uid: {
    type: mongoose.Schema.Types.ObjectId,
    ref: 'User'
  },
  // 评论时间
  time: {
    type: Date
  },
  // 评论内容
  content: {
    type: String
  }
});

// 创建评论集合
const Comment = mongoose.model('Comment', commentSchema);

// 将评论集合构造函数作为模块成员进行导出
module.exports = {
  Comment
}
```

2. 判断用户是否登录，如果登录，再允许用户提交评论表单

```

    {{if userInfo}}
    <h4>评论</h4>
    <form class="comment-form" action="/home/comment" method="post">
      <textarea class="comment" name="content"></textarea>
      <input type="hidden" name="uid" value="{{@userInfo._id}}">
      <input type="hidden" name="aid" value="{{@article._id}}">
      <div class="items">
        <input type="submit" value="提交">
      </div>
    </form>
    {{else}}
    <div><h2>先进行登录，再对文章进行评论</h2></div>

```

3. 在服务器端创建文章评论功能对应的路由
4. 在路由请求处理函数中接收客户端传递过来的评论信息
5. 将评论信息存储在评论集合中
6. 将页面重定向回文章详情页

```

// 将评论集合构造函数进行导入
const { Comment } = require('../../model/comment');

module.exports = async (req, res) => {
  // 接收客户端传递过来的请求参数
  const { content, uid, aid } = req.body;

  // 将评论信息存储到评论集合中
  await Comment.create({
    content: content,
    uid: uid,
    aid: aid,
    time: new Date()
  });

  // 将页面重定向回文章详情页面
  res.redirect('/home/article?id=' + aid);
}

```

7. 在文章详情页面路由中获取文章评论信息并展示在页面中


```
<div class="comment-list">
  {{each comments}}
  <div class="mb10">
    <div class="article-info">
      <span class="author">{{value.uid.username}}</span>
      <span>{{dateFormat(value.time, 'yyyy-mm-dd')}}</span>
      <span>{{value.uid.email}}</span>
    </div>
    <div class="comment-content">
      {{value.content}}
    </div>
  </div>
  {{/each}}
</div>
```

项目包含的知识点

密码加密**bcrypt**

哈希加密是单程加密方式: 1234=>abcd

在加密的密码中加入随机字符串可以增加密码被破解的难度

```
// 导入bcrypt模块
const bcrypt = require('bcrypt')
// 生成随机字符串 gen => generate 生成 salt 盐
let salt = await bcrypt.genSalt(10)
// 使用随机字符串对密码进行加密
let pass = await bcrypt.hash('明文密码', salt)
```

bcrypt 依赖的其他环境

1. python 2.x
2. node-gyp npm install -g node-gyp
3. windows-build-tools
4. npm install --global --production windows-build-tools

bcrypt的基本使用

```
// 导入bcrypt
const bcrypt = require('bcrypt');

async function run () {
  // 生成随机字符串
  // genSalt方法接收一个数值作为参数
  // 数值越大 生成的随机字符串复杂度越高
  // 数值越小 生成的随机字符串复杂度越低
  // 默认值是 10
  // 返回生成的随机字符串
  const salt = await bcrypt.genSalt(10);
  // 对密码进行加密
  // 1. 要进行加密的明文
  // 2. 随机字符串
  // 返回值是加密后的密码
  const result = await bcrypt.hash('123456', salt);
  console.log(salt);
  console.log(result);
}

run();
```

密码对比

```
// 密码比对
let isEqual = await bcrypt.compare('明文密码', '加密密码');
```

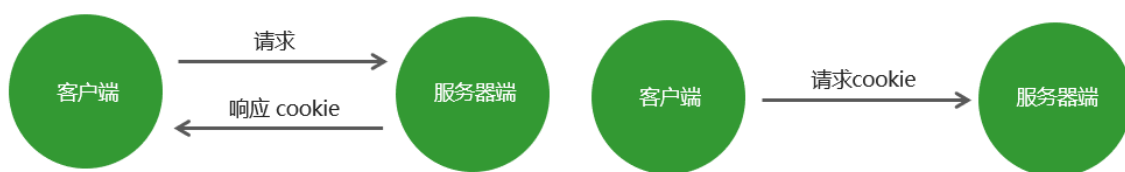
在admin.js中引入功能

```
// 查询到了用户
if (user) {
  // 将客户端传递过来的密码和用户信息中的密码进行比对
  // true 比对成功
  // false 对比失败
  let isValid = await bcrypt.compare(password, user.password);
  // 如果密码比对成功
  if ( isValid ) {
    // 登录成功
    // 将用户名存储在请求对象中
    req.username = user.username;
    // res.send('登录成功');
    req.app.locals.userInfo = user;
    // 重定向到用户列表页面
    res.redirect('/admin/user');
  } else {
    // 没有查询到用户
    res.status(400).render('admin/error', {msg: '邮箱地址或者密码错误'})
  }
} else {
  // 没有查询到用户
  res.status(400).render('admin/error', {msg: '邮箱地址或者密码错误'})
}
```

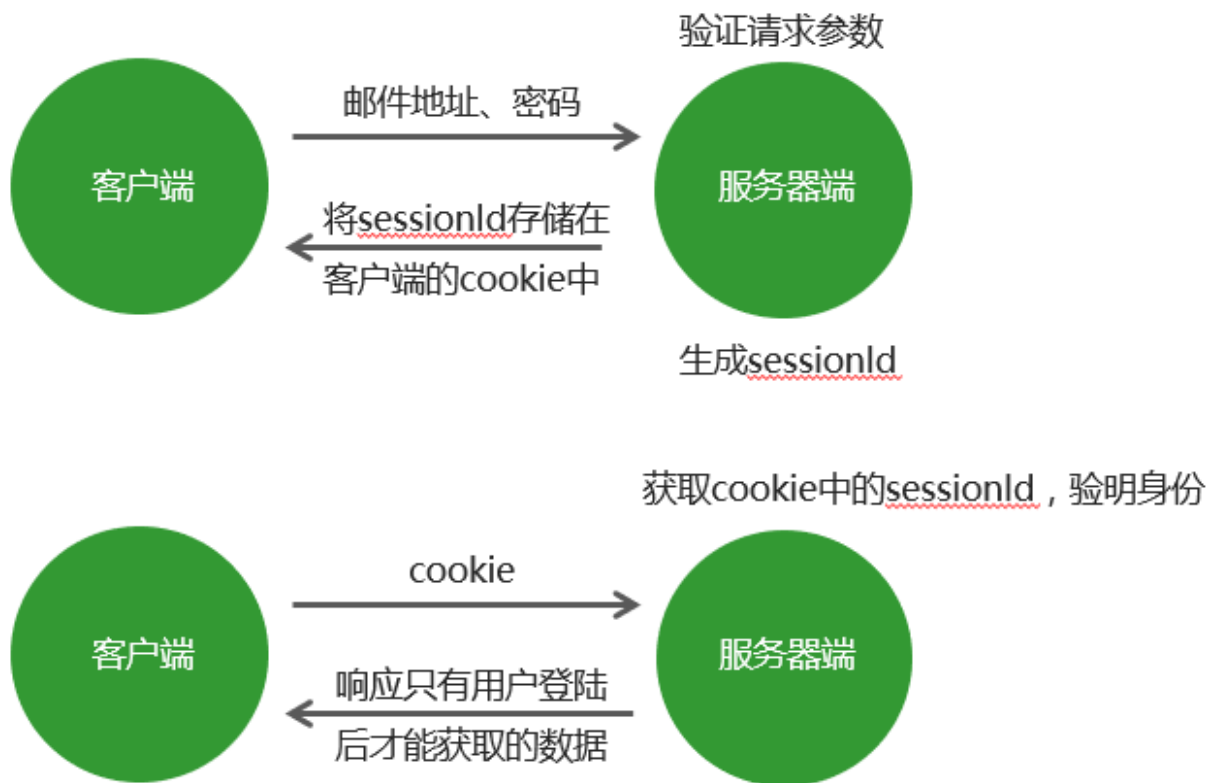
cookie与session

cookie: 浏览器在电脑硬盘中开辟的一块空间，主要供服务器端存储数据。

- lcookie中的数据是以域名的形式进行区分的。
- lcookie中的数据是有过期时间的，超过时间数据会被浏览器自动删除。
- lcookie中的数据会随着请求被自动发送到服务器端。



session: 实际上就是一个对象，存储在服务器端的内存中，在session对象中也可以存储多条数据，每一条数据都有一个sessionid做为唯一标识。



在node中使用express-session模块实现session功能

```
const session = require('express-session');  
app.use(session({ secret: 'secret key' }));
```

在app.js中引入模块

```
// 导入express-session模块  
const session = require('express-session');  
// 配置session  
app.use(session({  
  secret: 'secret key',  
  saveUninitialized: false,  
  cookie: {  
    maxAge: 24 * 60 * 60 * 1000  
  }  
}));
```

在用户页面从session中获取数据

```
// 创建用户列表路由
admin.get('/user', (req, res) => {
  res.render('admin/user', {
    msg: req.session.username
  })
})
```

Joi

JavaScript对象的规则描述语言和验证器

安装 npm install joi

```
const Joi = require('joi');
const schema = {
  username: Joi.string().alphanum().min(3).max(30).required().error(new Error('错误信息')),
  password: Joi.string().regex(/^[a-zA-Z0-9]{3,30}$/),
  access_token: [Joi.string(), Joi.number()],
  birthyear: Joi.number().integer().min(1900).max(2013),
  email: Joi.string().email()
};
Joi.validate({ username: 'abc', birthyear: 1994 }, schema);
```

定义项目中的表单验证

```
// 引入joi模块
const Joi = require('joi');

// 定义对象的验证规则
const schema = {
  username: Joi.string().min(2).max(5).required().error(new Error('username属性没有通过验证')),
  birth: Joi.number().min(1900).max(2020).error(new Error('birth没有通过验证'))
};

async function run () {
  try {
    // 实施验证
    await Joi.validate({username: 'ab', birth: 1800}, schema);
  } catch (ex) {
    console.log(ex.message);
    return;
  }
  console.log('验证通过')
}

run();
```

formidable

作用：解析表单，支持get请求参数，post请求参数、文件上传

安装：npm install formidable

```
// 引入formidable模块
const formidable = require('formidable');
// 创建表单解析对象
const form = new formidable.IncomingForm();
// 设置文件上传路径
form.uploadDir = "/my/dir";
// 是否保留表单上传文件的扩展名
form.keepExtensions = false;
// 对表单进行解析
form.parse(req, (err, fields, files) => {
  // fields 存储普通请求参数
  // files 存储上传的文件信息
});
```

FileReader

```
var reader = new FileReader();
reader.readAsDataURL('文件');
reader.onload = function () {
  console.log(reader.result);
}
```

数据分页 mongoose-sex-page

```
const pagination = require('mongoose-sex-page');
pagination(集合构造函数).page(1) .size(20) .display(8) .exec();
```

mongoDB数据库添加账号

1. 以系统管理员的方式运行powershell
2. 连接数据库 mongo
3. 查看数据库 show dbs
4. 切换到admin数据库 use admin
5. 创建超级管理员账户 db.createUser()
6. 切换到blog数据 use blog
7. 创建普通账号 db.createUser()
8. 卸载mongodb服务

停止服务 net stop mongod

mongod --remove

9. 创建mongodb服务

```
mongod --logpath="C:\Program Files\MongoDB\Server\4.1\log\mongod.log" --dbpath="C:\Program Files\MongoDB\Server\4.1\data" --install --auth
```

10. 启动mongodb服务 net start mongod

11. 在项目中使用账号连接数据库

```
mongoose.connect('mongodb://user:pass@localhost:port/database')
```

开发环境与生产环境

什么是开发环境与生产环境？

环境，就是指项目运行的地方，当项目处于开发阶段，项目运行在开发人员的电脑上，项目所处的环境就是开发环境。当项目开发完成以后，要将项目放到真实的网站服务器电脑中运行，项目所处的环境就是生产环境。

为什么要区分开发环境与生产环境

因为在不同的环境中，项目的配置是不一样的，需要在项目代码中判断当前项目运行的环境，根据不同的环境应用不同的项目配置

```
if (process.env.NODE_ENV == 'development') {  
    // 开发环境  
} else {  
    // 生产环境  
}
```

安装morgan

```
npm install morgan
```

```
// 获取系统环境变量 返回值是对象  
if (process.env.NODE_ENV == 'development') {  
    // 当前是开发环境  
    console.log('当前是开发环境')  
    // 在开发环境中 将客户端发送到服务器端的请求信息打印到控制台中  
    app.use(morgan('dev'))  
} else {  
    // 当前是生产环境  
    console.log('当前是生产环境')  
}
```

第三方模块config

作用：允许开发人员将不同运行环境下的应用配置信息抽离到单独的文件中，模块内部自动判断当前应用的运行环境，

并读取对应的配置信息，极大提供应用配置信息的维护成本，避免了当运行环境重复的多次切换时，手动到项目代码

中修改配置信息

使用步骤:

1. 使用npm install config命令下载模块
2. 在项目的根目录下新建config文件夹
3. 在config文件夹下面新建default.json、development.json、production.json文件
4. 在项目中通过require方法，将模块进行导入
5. 使用模块内部提供的get方法获取配置信息

```
{
  "db": {
    "user": "itcast",
    "host": "localhost",
    "port": "27017",
    "name": "blog"
  }
}
```

```
mongoose.connect(`mongodb://${config.get('db.user')}:${config.get('db.pwd')}@${config.get('db.host')}:${config.get('db.port')}/${config.get('db.name')}`, {useNewUrlParser: true })
  .then(() => console.log('数据库连接成功'))
  .catch(() => console.log('数据库连接失败'))
```

将敏感配置信息存储在环境变量中

1. 在config文件夹中建立custom-environment-variables.json文件
2. 配置项属性的值填写系统环境变量的名字
3. 项目运行时config模块查找系统环境变量，并读取其值作为当前配置项属于的值

```
{
  "db": {
    "pwd": "APP_PWD"
  }
}
```

