

目录

版权说明	1.1
第1章 项目介绍	1.2
1.1 项目描述	1.2.1
1.2 项目特点	1.2.2
1.3 数据交互	1.2.3
1.4 开发环境搭建	1.2.4
1.5 获取帮助	1.2.5
第2章 数据库支持	1.3
2.1 MySQL数据库支持	1.3.1
2.2 Oracle数据库支持	1.3.2
2.3 SQL Server数据库支持	1.3.3
2.4 PostgreSQL数据库支持	1.3.4
第3章 多数据源支持	1.4
3.1 多数据源配置	1.4.1
3.2 多数据源使用	1.4.2
3.3 源码讲解	1.4.3
第4章 基础知识讲解	1.5
4.1 Spring MVC使用	1.5.1
4.2 Swagger使用	1.5.2
4.3 Mybatis-plus使用	1.5.3
第5章 项目实战	1.6
5.1 需求说明	1.6.1
5.2 代码生成器	1.6.2
第6章 后端源码分析	1.7
6.1 前后端分离	1.7.1
6.2 权限设计思路	1.7.2
6.3 XSS脚本过滤	1.7.3
6.4 SQL注入	1.7.4
6.5 Redis缓存	1.7.5
6.6 异常处理机制	1.7.6
6.7 后端效验机制	1.7.7
6.8 系统日志	1.7.8
6.9 添加菜单	1.7.9
6.10 添加角色	1.7.10
6.11 添加管理员	1.7.11
6.12 定时任务模块	1.7.12

6.13 云存储模块	1.7.13
6.14 APP模块	1.7.14
第7章 生产环境部署	1.8
7.1 jar包部署	1.8.1
7.2 docker部署	1.8.2
7.3 集群部署	1.8.3

版权说明

本文档为付费文档，版权归人人开源（renren.io）所有，并保留一切权利，本文档及其描述的内容受有关法律的版权保护，对本文档以任何形式的非法复制、泄露或散布到网络提供下载，都将导致相应的法律责任。

免责声明

本文档仅提供阶段性信息，所含内容可根据项目的实际情况随时更新，以人人开源社区公告为准。如因文档使用不当造成的直接或间接损失，人人开源不承担任何责任。

文档更新

本文档由人人开源于2019年03月01日最后修订。

第1章 项目介绍

人人权限系统是一套轻量级的权限系统，主要包括用户管理、角色管理、部门管理、菜单管理、定时任务、参数管理、字典管理、文件上传、登录日志、操作日志、异常日志、文章管理、APP模块等功能。其中，还拥有多数据源、数据权限、国际化支持、Redis缓存动态开启与关闭、统一异常处理等技术特点。

1.1 项目描述

1.2 项目特点

1.3 数据交互

1.4 开发环境搭建

1.5 获取帮助

1.1 项目描述

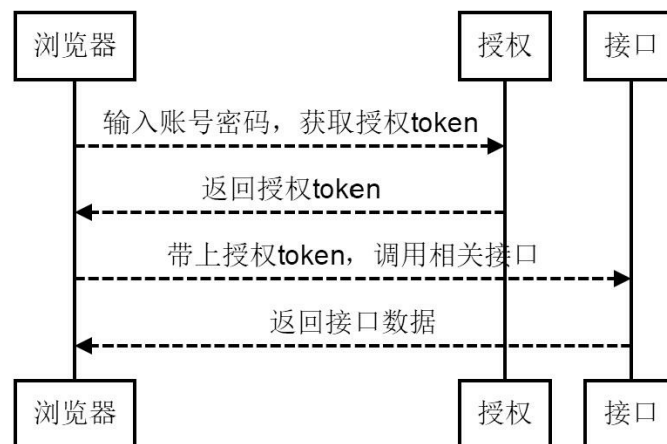
renren-fast是一套轻量级的权限系统，主要包括用户管理、角色管理、菜单管理、定时任务、文件上传、系统日志、APP模块等功能。其中，还拥有多数据源、Redis缓存动态开启与关闭、统一异常处理等技术特点。

1.2 项目特点

- [renren-fast](#)采用SpringBoot 2.1、MyBatis、Shiro框架，开发的一套权限系统，极低门槛，拿来即用。设计之初，就非常注重安全性，为企业系统保驾护航，让一切都变得如此简单。
- 灵活的权限控制，可控制到页面或按钮，满足绝大部分的权限需求
- 完善的 xss 防范及脚本过滤，彻底杜绝 xss 攻击
- 支持MySQL、Oracle、SQL Server、PostgreSQL等主流数据库
- 推荐使用阿里云服务器部署项目，免费领取阿里云优惠券，请点击【[免费领取](#)】

1.3 数据交互

- 一般情况下，web项目都是通过session进行认证，每次请求数据时，都会把jsessionId放在cookie中，以便与服务端保持会话
- 本项目是前后端分离的，通过token进行认证（登录时，生成唯一的token凭证），每次请求数据时，都会把token放在header中，服务端解析token，并确定用户身份及用户权限，数据通过json交互
- 数据交互流程，如下所示：



1.4 开发环境搭建

1.4.1 软件需求

- JDK 1.8+
- Maven 3.0+
- MySQL 5.5+
- Oracle 11g+
- SQL Server 2012+
- PostgreSQL 9.4+

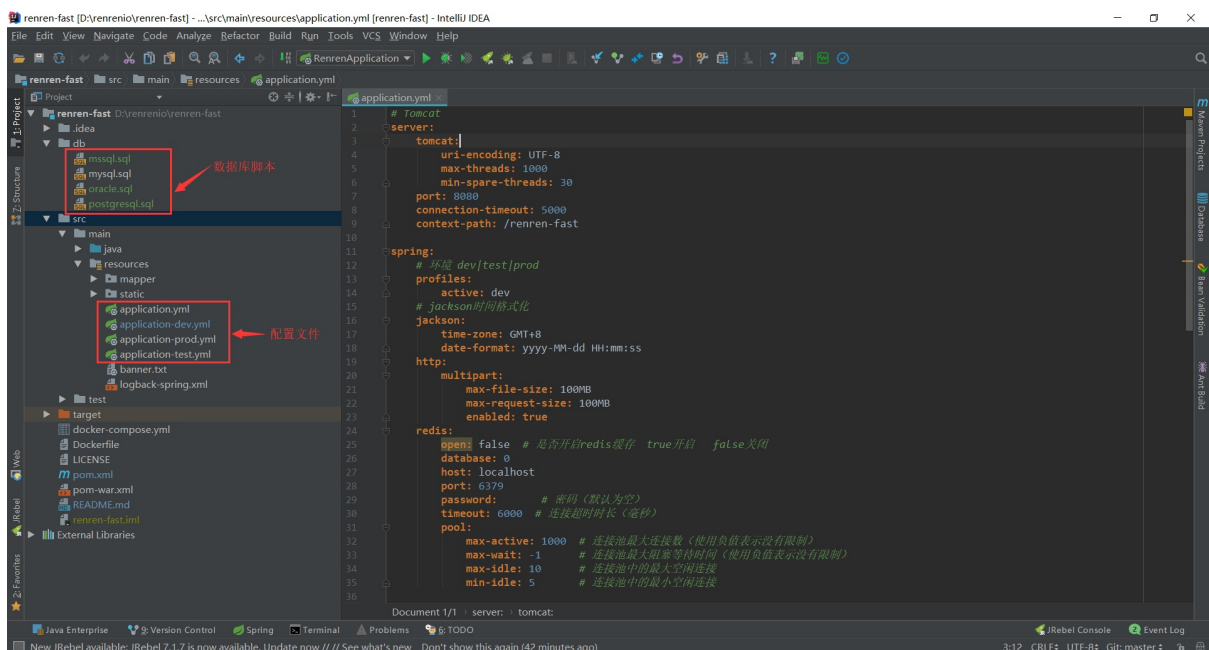
1.4.2 下载源码

- 通过 `git`，下载renren-fast源码，如下：

```
git clone https://gitee.com/renrenio/renren-fast.git
```

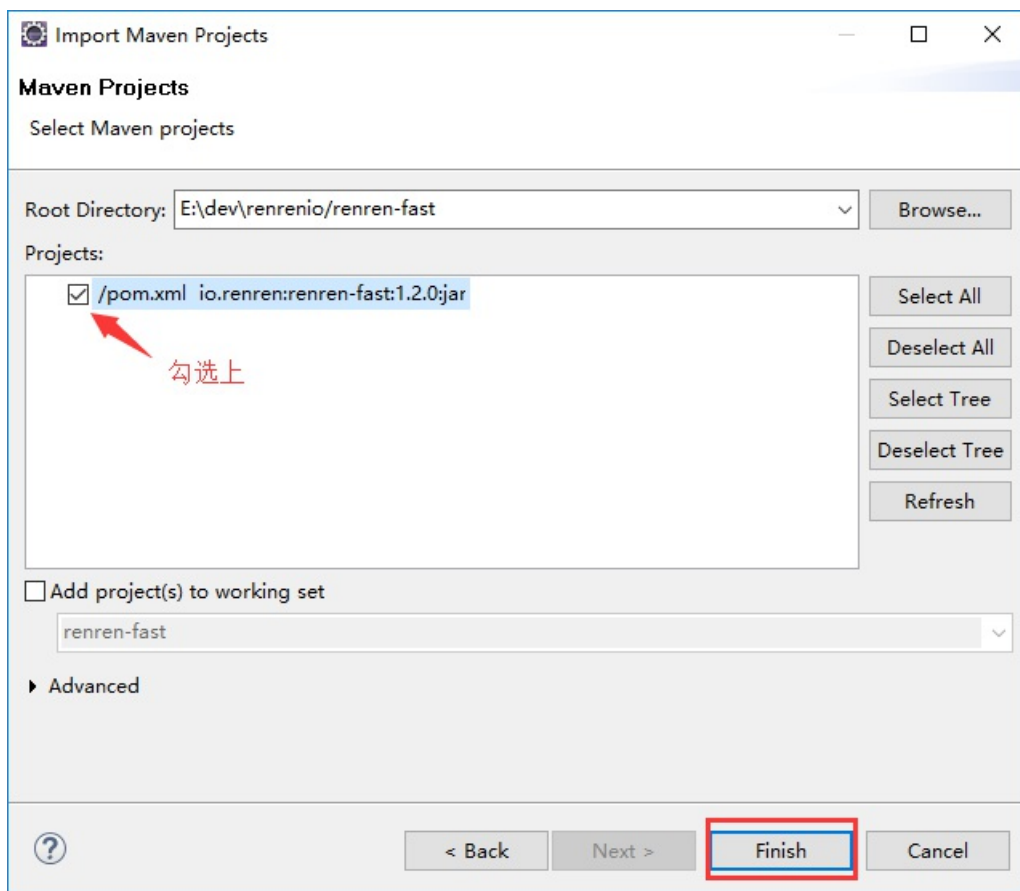
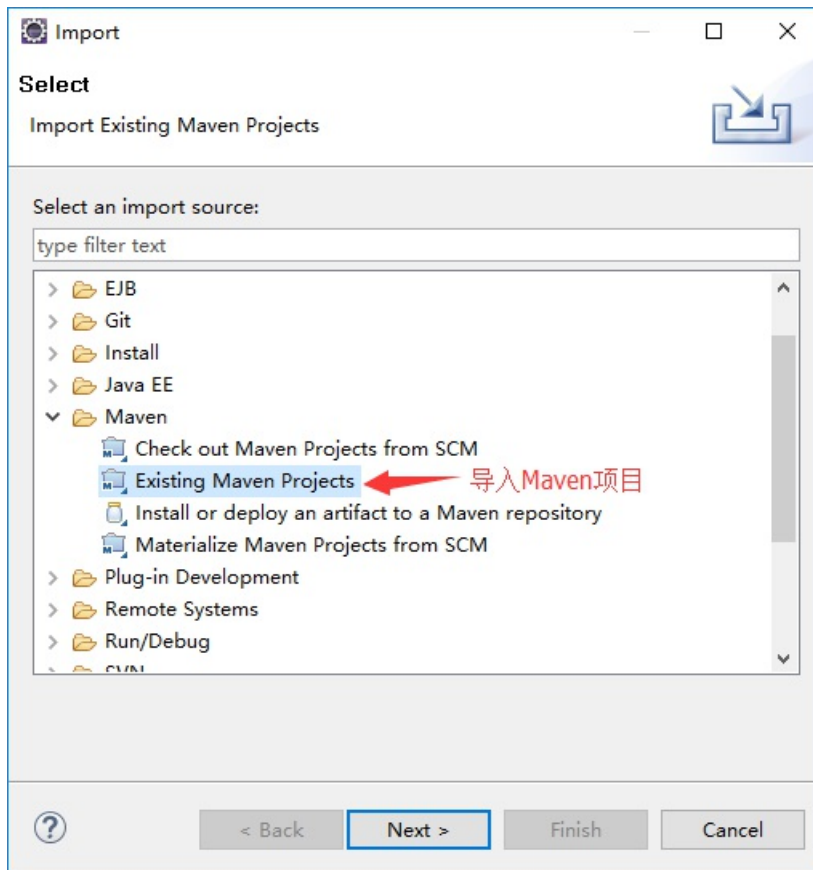
1.4.3 IDEA开发工具

- IDEA打开项目， `File -> Open` 如下图：



1.4.4 Eclipse开发工具

- Eclipse导入项目，如下图：



1.4.5 创建数据库

- 创建数据库 `renren_fast`，数据库编码为 `UTF-8`
- 执行 `db/mysql.sql` 文件，初始化数据（默认支持MySQL）

1.4.6 修改配置文件

- 修改 `application-dev.yml`，更新MySQL账号和密码
- 运行 `io.renren.RenrenApplication.java` 的 `main` 方法，则可启动项目
- Swagger路径: <http://localhost:8080/renren-fast/swagger/index.html>
- Swagger注解路径: <http://localhost:8080/renren-fast/swagger-ui.html>

1.4.7 前端部署

renren-fast-vue基于vue、element-ui构建开发，实现renren-fast后台管理前端功能，提供一套更优的前端解决方案。欢迎star或fork前端Git库，方便日后寻找，及二次开发

- 开发环境，需要安装node8.x最新版

```
# 克隆项目
git clone https://github.com/renrenio/renren-fast-vue.git

# 安装依赖
npm install --registry=https://registry.npm.taobao.org

# 启动服务
npm run dev
```

- 生产环境，打包并把dist目录文件，部署到Nginx里

```
# 构建生产环境(默认)
npm run build

# 构建测试环境
npm run build --qa

# 构建验收环境
npm run build --uat

# 构建生产环境
npm run build --prod

# 安装Nginx，并配置Nginx
server {
    listen      80;
    server_name localhost;

    location / {
        root    E:\\renren-fast-vue;
        index   index.html index.htm;
    }
}
```

```
}  
}
```

启动Nginx后，访问如下路径即可
<http://localhost>

- 登录的账号密码：admin/admin

1.5 获取帮助

- 后端地址: <https://gitee.com/renrenio/renren-fast>
- 前端地址: <https://github.com/renrenio/renren-fast-vue>
- 代码生成器: <https://gitee.com/renrenio/renren-generator>
- 官方社区: <https://www.renren.io/community>
- 如需寻求帮助、项目建议、技术讨论等, 请移步到官方社区, 我会在第一时间进行解答或回复
- 如需关注项目最新动态, 请Watch、Star项目, 同时也是对项目最好的支持

第2章 数据库支持

2.1 MySQL数据库支持

2.2 Oracle数据库支持

2.3 SQL Server数据库支持

2.4 PostgreSQL数据库支持

2.1 MySQL数据库支持

1) 修改数据库配置信息，开发环境的配置文件在application-dev.yml，如下所示：

```
spring:
  datasource:
    druid:
      driver-class-name: com.mysql.jdbc.Driver
      url: jdbc:mysql://localhost:3306/renren_fast?allowMultiQueries=true&useUnicode=true&characterEncoding=UTF-8&useSSL=false
      username: root
      password: 123456
```

2) 执行db/mysql.sql，创建表及初始化数据，再启动项目即可

2.2 Oracle数据库支持

1) 修改数据库配置信息，开发环境的配置文件在application-dev.yml，如下所示：

```
spring:
  datasource:
    druid:
      driver-class-name: oracle.jdbc.OracleDriver
      url: jdbc:oracle:thin:@192.168.10.10:1521:renren
      username: renren_fast
      password: 123456
```

2) 执行db/oracle.sql，创建表及初始化数据，再启动项目即可

2.3 SQL Server数据库支持

1) 修改数据库配置信息，开发环境的配置文件在application-dev.yml，如下所示：

```
spring:
  datasource:
    druid:
      driver-class-name: com.microsoft.sqlserver.jdbc.SQLServerDriver
      url: jdbc:sqlserver://192.168.10.10:1433;DatabaseName=renren_fast
      username: sa
      password: 123456
```

2) 执行db/sqlserver.sql，创建表及初始化数据，再启动项目即可

2.4 PostgreSQL数据库支持

1) 修改数据库配置信息，开发环境的配置文件在application-dev.yml，如下所示：

```
spring:
  datasource:
    druid:
      driver-class-name: org.postgresql.Driver
      url: jdbc:postgresql://192.168.10.10:5432/renren_fast
      username: renren
      password: 123456
```

2) 修改quartz配置信息，quartz配置文件 ScheduleConfig.java，打开注释，如下所示：

```
//PostgreSQL数据库，需要打开此注释
prop.put("org.quartz.jobStore.driverDelegateClass", "org.quartz.impl.jdbcjobstore.PostgreSQLDelegator");
```

3) 执行db/postgresql.sql，创建表及初始化数据，再启动项目即可

第3章 多数据源支持

3.1 多数据源配置

3.2 多数据源使用

3.3 源码讲解

3.1 多数据源配置

多数据源的应用场景，主要针对跨多个数据库实例的情况，如果是同实例中的多个数据库，则没必要使用多数据源。

#下面演示单实例，多数据库的使用情况

```
select * from db.table;
```

#其中，db为数据库名，table为数据库表名

- 配置多数据源，如果是开发环境，则修改 `application-dev.xml`，如下所示

#多数据源的配置

dynamic:

datasource:

slave1:

driver-class-name: com.microsoft.sqlserver.jdbc.SQLServerDriver

url: jdbc:sqlserver://192.168.10.10:1433;DatabaseName=renren_fast

username: sa

password: 123456

slave2:

driver-class-name: org.postgresql.Driver

url: jdbc:postgresql://192.168.10.10:5432/renren_fast

username: postgres

password: 123456

3.2 多数据源使用

多数据源的使用，只需在Service类、方法上添加`@DataSource("")`注解即可，比如在类上添加了`@DataSource("userDB")`注解，则表示该Service方法里的所有CURD，都会在 `userDB` 数据源里执行。

1) 多数据源注解使用规则

- 支持在Service类或方法上，添加多数据源的注解`@DataSource`
- 在Service类上添加了`@DataSource`注解，则该类下的所有方法，都会使用`@DataSource`标注的数据源
- 在Service类、方法上都添加了`@DataSource`注解，则方法上的注解会覆盖Service类上的注解

2) 编写DynamicDataSourceTestService.java，测试多数据源及事物

```
package io.renren.service;

import io.renren.commons.dynamic.datasource.annotation.DataSource;
import io.renren.dao.SysUserDao;
import io.renren.entity.SysUserEntity;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

/**
 * 测试多数据源
 *
 * @author Mark sunlightcs@gmail.com
 */
@Service
//@DataSource("slave1") 多数据源全局配置
public class DynamicDataSourceTestService {

    @Autowired
    private SysUserDao sysUserDao;

    @Transactional
    public void updateUser(Long id){
        SysUserEntity user = new SysUserEntity();
        user.setUserId(id);
        user.setMobile("13500000000");
        sysUserDao.updateById(user);
    }

    @Transactional
    @DataSource("slave1")
    public void updateUserBySlave1(Long id){
        SysUserEntity user = new SysUserEntity();
        user.setUserId(id);
        user.setMobile("13500000001");
        sysUserDao.updateById(user);
    }
}
```

```

    @DataSource("slave2")
    @Transactional
    public void updateUserBySlave2(Long id){
        SysUserEntity user = new SysUserEntity();
        user.setUserId(id);
        user.setMobile("13500000002");
        sysUserDao.updateById(user);

        //测试事物
        int i = 1/0;
    }
}

```

3) 运行测试类DynamicDataSourceTest.java，即可测试多数据源及事物是生效的

```

package io.renren.service;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;

/**
 * 多数据源测试
 *
 * @author Mark sunlightcs@gmail.com
 */
@RunWith(SpringRunner.class)
@SpringBootTest
public class DynamicDataSourceTest {

    @Autowired
    private DynamicDataSourceTestService dynamicDataSourceTestService;

    @Test
    public void test(){
        Long id = 1L;

        dynamicDataSourceTestService.updateUser(id);
        dynamicDataSourceTestService.updateUserBySlave1(id);
        dynamicDataSourceTestService.updateUserBySlave2(id);
    }

}

```

3) 其中，@DataSource("slave1")、@DataSource("slave2") 里的 slave1、slave2 值，是在application-dev.xml里配置的，如下所示：

```

dynamic:
  datasource:
    slave1:

```

```
driver-class-name: com.microsoft.sqlserver.jdbc.SQLServerDriver
url: jdbc:sqlserver://localhost:1433;DatabaseName=renren_security
username: sa
password: 123456
slave2:
driver-class-name: org.postgresql.Driver
url: jdbc:postgresql://localhost:5432/renren_security
username: renren
password: 123456
```

3.3 源码讲解

1) 定义多数据源注解类@DataSource，使用多数据源时，只需在Service方法上添加@DataSource注解即可

```
import java.lang.annotation.*;

/**
 * 多数据源注解
 *
 * @author Mark sunlightcs@gmail.com
 */
@Target({ElementType.METHOD, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
public @interface DataSource {
    String value() default "";
}
```

2) 定义读取多数据源配置文件的类，如下所示：

```
/**
 * 多数据源属性
 *
 * @author Mark sunlightcs@gmail.com
 */
public class DataSourceProperties {
    private String driverClassName;
    private String url;
    private String username;
    private String password;

    /**
     * Druid默认参数
     */
    private int initialSize = 2;
    private int maxActive = 10;
    private int minIdle = -1;
    private long maxWait = 60 * 1000L;
    private long timeBetweenEvictionRunsMillis = 60 * 1000L;
    private long minEvictableIdleTimeMillis = 1000L * 60L * 30L;
    private long maxEvictableIdleTimeMillis = 1000L * 60L * 60L * 7;
    private String validationQuery = "select 1";
    private int validationQueryTimeout = -1;
    private boolean testOnBorrow = false;
    private boolean testOnReturn = false;
    private boolean testWhileIdle = true;
    private boolean poolPreparedStatements = false;
}
```

```

private int maxOpenPreparedStatements = -1;
private boolean sharePreparedStatements = false;
private String filters = "stat,wall";

public String getDriverClassName() {
    return driverClassName;
}

public void setDriverClassName(String driverClassName) {
    this.driverClassName = driverClassName;
}

public String getUrl() {
    return url;
}

public void setUrl(String url) {
    this.url = url;
}

public String getUsername() {
    return username;
}

public void setUsername(String username) {
    this.username = username;
}

public String getPassword() {
    return password;
}

public void setPassword(String password) {
    this.password = password;
}

public int getInitialSize() {
    return initialSize;
}

public void setInitialSize(int initialSize) {
    this.initialSize = initialSize;
}

public int getMaxActive() {
    return maxActive;
}

public void setMaxActive(int maxActive) {
    this.maxActive = maxActive;
}

```



```

public int getMinIdle() {
    return minIdle;
}

public void setMinIdle(int minIdle) {
    this.minIdle = minIdle;
}

public long getMaxWait() {
    return maxWait;
}

public void setMaxWait(long maxWait) {
    this.maxWait = maxWait;
}

public long getTimeBetweenEvictionRunsMillis() {
    return timeBetweenEvictionRunsMillis;
}

public void setTimeBetweenEvictionRunsMillis(long timeBetweenEvictionRunsMillis) {
    this.timeBetweenEvictionRunsMillis = timeBetweenEvictionRunsMillis;
}

public long getMinEvictableIdleTimeMillis() {
    return minEvictableIdleTimeMillis;
}

public void setMinEvictableIdleTimeMillis(long minEvictableIdleTimeMillis) {
    this.minEvictableIdleTimeMillis = minEvictableIdleTimeMillis;
}

public long getMaxEvictableIdleTimeMillis() {
    return maxEvictableIdleTimeMillis;
}

public void setMaxEvictableIdleTimeMillis(long maxEvictableIdleTimeMillis) {
    this.maxEvictableIdleTimeMillis = maxEvictableIdleTimeMillis;
}

public String getValidationQuery() {
    return validationQuery;
}

public void setValidationQuery(String validationQuery) {
    this.validationQuery = validationQuery;
}

public int getValidationQueryTimeout() {
    return validationQueryTimeout;
}

```

```

public void setValidationQueryTimeout(int validationQueryTimeout) {
    this.validationQueryTimeout = validationQueryTimeout;
}

public boolean isTestOnBorrow() {
    return testOnBorrow;
}

public void setTestOnBorrow(boolean testOnBorrow) {
    this.testOnBorrow = testOnBorrow;
}

public boolean isTestOnReturn() {
    return testOnReturn;
}

public void setTestOnReturn(boolean testOnReturn) {
    this.testOnReturn = testOnReturn;
}

public boolean isTestWhileIdle() {
    return testWhileIdle;
}

public void setTestWhileIdle(boolean testWhileIdle) {
    this.testWhileIdle = testWhileIdle;
}

public boolean isPoolPreparedStatements() {
    return poolPreparedStatements;
}

public void setPoolPreparedStatements(boolean poolPreparedStatements) {
    this.poolPreparedStatements = poolPreparedStatements;
}

public int getMaxOpenPreparedStatements() {
    return maxOpenPreparedStatements;
}

public void setMaxOpenPreparedStatements(int maxOpenPreparedStatements) {
    this.maxOpenPreparedStatements = maxOpenPreparedStatements;
}

public boolean isSharePreparedStatements() {
    return sharePreparedStatements;
}

public void setSharePreparedStatements(boolean sharePreparedStatements) {
    this.sharePreparedStatements = sharePreparedStatements;
}

```

```

    public String getFilters() {
        return filters;
    }

    public void setFilters(String filters) {
        this.filters = filters;
    }
}

```

```

import org.springframework.boot.context.properties.ConfigurationProperties;

import java.util.LinkedHashMap;
import java.util.Map;

/**
 * 多数据源属性
 *
 * @author Mark sunlightcs@gmail.com
 */
@ConfigurationProperties(prefix = "dynamic")
public class DynamicDataSourceProperties {
    private Map<String, DataSourceProperties> datasource = new LinkedHashMap<>();

    public Map<String, DataSourceProperties> getDatasource() {
        return datasource;
    }

    public void setDatasource(Map<String, DataSourceProperties> datasource) {
        this.datasource = datasource;
    }
}

```

3) 扩展Spring的AbstractRoutingDataSource抽象类，AbstractRoutingDataSource中的抽象方法determineCurrentLookupKey是实现多数据源的核心，并对该方法进行Override，如下所示：

```

import org.springframework.jdbc.datasource.lookup.AbstractRoutingDataSource;

/**
 * 多数据源
 *
 * @author Mark sunlightcs@gmail.com
 */
public class DynamicDataSource extends AbstractRoutingDataSource {

    @Override
    protected Object determineCurrentLookupKey() {
        return DynamicContextHolder.peek();
    }
}

```

```
}
```

4) 多数据源上下文

```
/**
 * 多数据源上下文
 *
 * @author Mark sunlightcs@gmail.com
 */
public class DynamicContextHolder {
    @SuppressWarnings("unchecked")
    private static final ThreadLocal<Deque<String>> CONTEXT_HOLDER = new ThreadLocal() {
        @Override
        protected Object initialValue() {
            return new ArrayDeque();
        }
    };

    /**
     * 获得当前线程数据源
     *
     * @return 数据源名称
     */
    public static String peek() {
        return CONTEXT_HOLDER.get().peek();
    }

    /**
     * 设置当前线程数据源
     *
     * @param dataSource 数据源名称
     */
    public static void push(String dataSource) {
        CONTEXT_HOLDER.get().push(dataSource);
    }

    /**
     * 清空当前线程数据源
     */
    public static void poll() {
        Deque<String> deque = CONTEXT_HOLDER.get();
        deque.poll();
        if (deque.isEmpty()) {
            CONTEXT_HOLDER.remove();
        }
    }
}
```

5) 配置多数据源，如下所示：

```

import com.alibaba.druid.pool.DruidDataSource;
import io.renren.commons.dynamic.datasource.properties.DataSourceProperties;
import io.renren.commons.dynamic.datasource.properties.DynamicDataSourceProperties;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.boot.context.properties.EnableConfigurationProperties;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import java.util.HashMap;
import java.util.Map;

/**
 * 配置多数据源
 *
 * @author Mark sunlightcs@gmail.com
 */
@Configuration
@EnableConfigurationProperties(DynamicDataSourceProperties.class)
public class DynamicDataSourceConfig {
    @Autowired
    private DynamicDataSourceProperties properties;

    @Bean
    @ConfigurationProperties(prefix = "spring.datasource.druid")
    public DataSourceProperties dataSourceProperties() {
        return new DataSourceProperties();
    }

    //因为DynamicDataSource是继承与AbstractRoutingDataSource，而AbstractRoutingDataSource又是继承于AbstractDataSource，AbstractDataSource实现了统一的DataSource接口，所以DynamicDataSource也可以当做DataSource使用
    @Bean
    public DynamicDataSource dynamicDataSource(DataSourceProperties dataSourceProperties) {
        DynamicDataSource dynamicDataSource = new DynamicDataSource();
        dynamicDataSource.setTargetDataSources(getDynamicDataSource());

        //默认数据源
        DruidDataSource defaultDataSource = DynamicDataSourceFactory.buildDruidDataSource(dataSourceProperties);
        dynamicDataSource.setDefaultTargetDataSource(defaultDataSource);

        return dynamicDataSource;
    }

    private Map<Object, Object> getDynamicDataSource(){
        Map<Object, Object> targetDataSources = new HashMap<>();
        properties.getDatasource().forEach((k, v) -> {
            DruidDataSource druidDataSource = DynamicDataSourceFactory.buildDruidDataSource(v);
            targetDataSources.put(k, druidDataSource);
        });
    }
}

```

```

        return targetDataSources;
    }
}

```

```

import com.alibaba.druid.pool.DruidDataSource;
import io.renren.commons.dynamic.datasource.properties.DataSourceProperties;

import java.sql.SQLException;

/**
 * DruidDataSource
 *
 * @author Mark sunlightcs@gmail.com
 */
public class DynamicDataSourceFactory {

    public static DruidDataSource buildDruidDataSource(DataSourceProperties properties) {
        DruidDataSource druidDataSource = new DruidDataSource();
        druidDataSource.setDriverClassName(properties.getDriverClassName());
        druidDataSource.setUrl(properties.getUrl());
        druidDataSource.setUsername(properties.getUsername());
        druidDataSource.setPassword(properties.getPassword());

        druidDataSource.setInitialSize(properties.getInitialSize());
        druidDataSource.setMaxActive(properties.getMaxActive());
        druidDataSource.setMinIdle(properties.getMinIdle());
        druidDataSource.setMaxWait(properties.getMaxWait());
        druidDataSource.setTimeBetweenEvictionRunsMillis(properties.getTimeBetweenEvictionRun
sMillis());
        druidDataSource.setMinEvictableIdleTimeMillis(properties.getMinEvictableIdleTimeMilli
s());
        druidDataSource.setMaxEvictableIdleTimeMillis(properties.getMaxEvictableIdleTimeMilli
s());
        druidDataSource.setValidationQuery(properties.getValidationQuery());
        druidDataSource.setValidationQueryTimeout(properties.getValidationQueryTimeout());
        druidDataSource.setTestOnBorrow(properties.isTestOnBorrow());
        druidDataSource.setTestOnReturn(properties.isTestOnReturn());
        druidDataSource.setPoolPreparedStatements(properties.isPoolPreparedStatements());
        druidDataSource.setMaxOpenPreparedStatements(properties.getMaxOpenPreparedStatements(
));
        druidDataSource.setSharePreparedStatements(properties.isSharePreparedStatements());

        try {
            druidDataSource.setFilters(properties.getFilters());
            druidDataSource.init();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}

```

```

    }
    return druidDataSource;
}
}

```

5) @DataSource注解的切面处理类，动态切换的核心代码

```

import io.renren.commons.dynamic.datasource.annotation.DataSource;
import io.renren.commons.dynamic.datasource.config.DynamicContextHolder;
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Pointcut;
import org.aspectj.lang.reflect.MethodSignature;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.core.Ordered;
import org.springframework.core.annotation.Order;
import org.springframework.stereotype.Component;

import java.lang.reflect.Method;

/**
 * 多数据源，切面处理类
 *
 * @author Mark sunlightcs@gmail.com
 */
@Aspect
@Component
@Order(Ordered.HIGHEST_PRECEDENCE)
public class DataSourceAspect {
    protected Logger logger = LoggerFactory.getLogger(getClass());

    @Pointcut("@annotation(io.renren.commons.dynamic.datasource.annotation.DataSource) " +
        "|| @within(io.renren.commons.dynamic.datasource.annotation.DataSource)")
    public void dataSourcePointCut() {

    }

    @Around("dataSourcePointCut()")
    public Object around(ProceedingJoinPoint point) throws Throwable {
        MethodSignature signature = (MethodSignature) point.getSignature();
        Class targetClass = point.getTarget().getClass();
        Method method = signature.getMethod();

        DataSource targetDataSource = (DataSource)targetClass.getAnnotation(DataSource.class)
;
        DataSource methodDataSource = method.getAnnotation(DataSource.class);
        if(targetDataSource != null || methodDataSource != null){
            String value;
            if(methodDataSource != null){
                value = methodDataSource.value();

```

```
        }else {
            value = targetDataSource.value();
        }

        DynamicContextHolder.push(value);
        logger.debug("set datasource is {}", value);
    }

    try {
        return point.proceed();
    } finally {
        DynamicContextHolder.poll();
        logger.debug("clean datasource");
    }
}
}
```


第4章 基础知识讲解

4.1 Spring MVC使用

4.2 Swagger使用

4.3 Mybatis-plus使用

4.1 Spring MVC使用

对Spring MVC不太熟悉的，需要理解Spring MVC常用的注解，也方便日后排查问题，常用的注解如下所示：

4.1.1 @Controller注解

@Controller注解表明了一个类是作为控制器的角色而存在的。Spring不要求你去继承任何控制器基类，也不要求你去实现Servlet的那套API。当然，如果你需要的话也可以去使用任何与Servlet相关的特性。

```
@Controller
public class UserController {
    // ...
}
```

4.1.2 @RequestMapping注解

你可以使用@RequestMapping注解来将请求URL，如/user等，映射到整个类上或某个特定的处理器方法上。一般来说，类级别的注解负责将一个特定（或符合某种模式）的请求路径映射到一个控制器上，同时通过方法级别的注解来细化映射，即根据特定的HTTP请求方法（GET、POST方法等）、HTTP请求中是否携带特定参数等条件，将请求映射到匹配的方法上。

```
@Controller
public class UserController {

    @RequestMapping("/user")
    public String user() {
        return "user";
    }

}
```

以上代码没有指定请求必须是GET方法还是PUT/POST或其他方法，@RequestMapping注解默认会映射所有的HTTP请求方法。如果仅想接收某种请求方法，请在注解中指定之@RequestMapping(path = "/user", method = RequestMethod.GET)以缩小范围。

4.1.3 @PathVariable注解

在Spring MVC中你可以在方法参数上使用@PathVariable注解，将其与URI模板中的参数绑定起来，如下所示：

```
@RequestMapping(path="/user/{userId}", method=RequestMethod.GET)
public String userCenter(@PathVariable("userId") String userId, Model model) {

    UserDTO user = userService.get(userId);
    model.addAttribute("user", user);
}
```

```
        return "userCenter";  
    }  
}
```

URI模板"/user/{userId}"指定了一个变量名为userId。当控制器处理这个请求的时候，userId的值就会被URI模板中对应部分的值所填充。比如说，如果请求的URI是/user/1，此时变量userId的值就是1。

4.1.4 @GetMapping注解

@GetMapping是一个组合注解，是@RequestMapping(method = RequestMethod.GET)的缩写。该注解将HTTP GET映射到特定的处理方法上。可以使用@GetMapping("/user")来代替

@RequestMapping(path="/user",method= RequestMethod.GET)。还有@PostMapping、@PutMapping、@DeleteMapping等同理。

4.1.5 @RequestBody注解

该注解用于读取Request请求的body部分数据，使用系统默认配置的HttpMessageConverter进行解析，然后把相应的数据绑定到要返回的对象上，再把HttpMessageConverter返回的对象数据绑定到Controller中方法的参数上。

```
@Controller  
public class UserController {  
  
    @GetMapping("/user")  
    public String user(@RequestBody User user) {  
        //...  
        return "user";  
    }  
  
}
```

4.1.6 @ResponseBody注解

该注解用于将Controller的方法返回的对象，通过适当的HttpMessageConverter转换为指定格式后，写入到Response对象的body数据区。比如获取JSON数据，加上@ResponseBody后，会直接返回JSON数据，而不会被解析为视图。

```
@Controller  
public class UserController {  
  
    @ResponseBody  
    @GetMapping("/user/{userId}")  
    public User info(@PathVariable("userId") String userId) {  
        UserDTO user = userService.get(userId);  
        return user;  
    }  
  
}
```

4.1.7 @RestController注解

@RestController是一个组合注解，即@Controller + @ResponseBody的组合注解，请求完后，会返回JSON数据。

4.2 Swagger使用

Swagger是一个根据Swagger注解，即可生成接口文档的服务。

4.2.1 搭建Swagger环境

- 在pom.xml文件中添加swagger相关依赖，如下所示：

```
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger2</artifactId>
    <version>${springfox-version}</version>
</dependency>
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger-ui</artifactId>
    <version>${springfox-version}</version>
</dependency>
```

- 编写Swagger的Configuration配置文件，如下所示：

```
import io.swagger.annotations.ApiOperation;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.ResourceHandlerRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;
import springfox.documentation.builders.ApiInfoBuilder;
import springfox.documentation.builders.PathSelectors;
import springfox.documentation.builders.RequestHandlerSelectors;
import springfox.documentation.service.ApiInfo;
import springfox.documentation.service.ApiKey;
import springfox.documentation.spi.DocumentationType;
import springfox.documentation.spring.web.plugins.Docket;
import springfox.documentation.swagger2.annotations.EnableSwagger2;

import java.util.List;

import static com.google.common.collect.Lists.newArrayList;

@Configuration
@EnableSwagger2
public class SwaggerConfig implements WebMvcConfigurer {

    @Bean
    public Docket createRestApi() {
        return new Docket(DocumentationType.SWAGGER_2)
            .apiInfo(apiInfo())
            .select()
```

```

        //加了ApiOperation注解的类，才生成接口文档
        .apis(RequestHandlerSelectors.withMethodAnnotation(ApiOperation.class))
        //io.renren.controller包下的类，才生成接口文档
        //.apis(RequestHandlerSelectors.basePackage("io.renren.controller"))
        .paths(PathSelectors.any())
        .build()
        .directModelSubstitute(java.util.Date.class, String.class);
    }

    private ApiInfo apiInfo() {
        return new ApiInfoBuilder()
            .title("人人开源")
            .description("人人开源接口文档")
            .termsOfServiceUrl("https://www.renren.io/community")
            .version("1.0.0")
            .build();
    }
}

```

4.2.2 Swagger常用注解

- @Api注解用在类上，说明该类的作用。可以标记一个Controller类做为swagger文档资源，如下所示：

```

@Api(tags="用户管理")
@RestController
public class UserController {

}

```

- @ApiOperation注解用在方法上，说明该方法的作用，如下所示：

```

@Api(tags="用户管理")
@RestController
public class UserController {

    @GetMapping("/user/list")
    @ApiOperation("列表")
    public List<UserDTO> list(){
        List<UserDTO> list = userService.list();
        return list;
    }

}

```

- @ApiParam注解用在方法参数上，如下所示：

```

@Api(tags="用户管理")
@RestController
public class UserController {

```

```

@GetMapping("/user/list")
@ApiOperation("列表")
public List<UserDTO> list(@ApiParam(value= "用户名", required = true) String username){
    List<UserDTO> list = userService.list();
    return list;
}
}

```

- @ApiImplicitParams注解用在方法上，主要用于一组参数说明
- @ApiImplicitParam注解用在@ApiImplicitParams注解中，指定一个请求参数的信息，如下所示：

```

@GetMapping("page")
@ApiOperation("分页")
@ApiImplicitParams({
    @ApiImplicitParam(name = "page", value = "当前页码，从1开始", paramType = "query", required = true, dataType="int") ,
    @ApiImplicitParam(name = "limit", value = "每页显示记录数", paramType = "query", required = true, dataType="int") ,
    @ApiImplicitParam(name = "order_field", value = "排序字段", paramType = "query", dataType="String") ,
    @ApiImplicitParam(name = "order", value = "排序方式，可选值(asc、desc)", paramType = "query", dataType="String") ,
    @ApiImplicitParam(name = "username", value = "用户名", paramType = "query", dataType="String")
})
public Result<PageData<SysUserDTO>> page(@ApiIgnore @RequestParam Map<String, Object> params){
    PageData<SysUserDTO> page = sysUserService.page(params);

    return new Result<PageData<SysUserDTO>>>().ok(page);
}

```

- @ApiIgnore注解，可用于类、方法或参数上，表示生成Swagger接口文档时，忽略类、方法或参数。

4.3 Mybatis-plus使用

- 在项目的pom.xml里引入依赖，如下所示：

```
<dependency>
  <groupId>com.baomidou</groupId>
  <artifactId>mybatis-plus-boot-starter</artifactId>
  <version>${mybatisplus.version}</version>
</dependency>
```

- 在yml配置文件里，配置mybatis-plus，如下所示：

```
mybatis-plus:
  mapper-locations: classpath*:/mapper/**/*.xml
  #实体扫描，多个package用逗号或者分号分隔
  typeAliasesPackage: io.renren.modules.*.entity
  global-config:
    #数据库相关配置
    db-config:
      #主键类型 AUTO:"数据库ID自增", INPUT:"用户输入ID", ID_WORKER:"全局唯一ID (数字类型唯一ID)"
      , UUID:"全局唯一ID UUID";
      id-type: AUTO
      #字段策略 IGNORED:"忽略判断",NOT_NULL:"非 NULL 判断"),NOT_EMPTY:"非空判断"
      field-strategy: NOT_NULL
      #驼峰下划线转换
      column-underline: true
      logic-delete-value: -1
      logic-not-delete-value: 0
      banner: false
  #原生配置
  configuration:
    map-underscore-to-camel-case: true
    cache-enabled: false
    call-setters-on-nulls: true
    jdbc-type-for-null: 'null'
```


第5章 项目实战

5.1 需求说明

5.2 代码生成器

5.1 需求说明

我们来完成一个商品的列表、添加、修改、删除功能，熟悉如何快速开发自己的业务功能模块。

- 我们先建一个商品表tb_goods，表结构如下所示：

```
CREATE TABLE `tb_goods` (  
  `goods_id` bigint NOT NULL AUTO_INCREMENT COMMENT '商品ID',  
  `name` varchar(50) COMMENT '商品名',  
  `intro` varchar(500) COMMENT '介绍',  
  `price` decimal(10,2) COMMENT '价格',  
  `num` int COMMENT '数量',  
  PRIMARY KEY (`goods_id`)  
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COMMENT='商品管理';
```

5.1 代码生成器

- 使用代码生成器前，我们先来看下代码生成器的配置，看看那些是可配置的，打开renren-generator模块的配置文件generator.properties，如下所示：

```
#代码生成器，配置信息

mainPath=io.renren
#包名
package=io.renren.modules
moduleName=generator
#作者
author=Mark
#Email
email=sunlightcs@gmail.com
#表前缀(类名不会包含表前缀)
tablePrefix=tb_

#类型转换，配置信息
tinyint=Integer
smallint=Integer
mediumint=Integer
int=Integer
integer=Integer
bigint=Long
float=Float
double=Double
decimal=BigDecimal
bit=Boolean

char=String
varchar=String
tinytext=String
text=String
mediumtext=String
longtext=String

date=Date
datetime=Date
timestamp=Date

NUMBER=Integer
INT=Integer
INTEGER=Integer
BINARY_INTEGER=Integer
LONG=String
FLOAT=Float
BINARY_FLOAT=Float
```

```

DOUBLE=Double
BINARY_DOUBLE=Double
DECIMAL=BigDecimal
CHAR=String
VARCHAR=String
VARCHAR2=String
NVARCHAR=String
NVARCHAR2=String
CLOB=String
BLOB=String
DATE=Date
DATETIME=Date
TIMESTAMP=Date
TIMESTAMP(6)=Date

int8=Long
int4=Integer
int2=Integer
numeric=BigDecimal

```

上面的配置文件，可以配置包名、作者信息、表前缀、模块名称、类型转换等信息。其中，类型转换是指，MySQL中的类型与JavaBean中的类型，是怎么一个对应关系。如果有缺少的类型，可自行在generator.properties文件中补充。

- 再看看renren-generator模块的application.yml配置文件，我们只要修改数据库名、账号、密码，就可以了。其中，数据库名是指待生成的表，所在的数据库。

```

server:
  port: 80

# mysql
spring:
  datasource:
    type: com.alibaba.druid.pool.DruidDataSource
    #MySQL配置
    driverClassName: com.mysql.jdbc.Driver
    url: jdbc:mysql://localhost:3306/renren_fast?useUnicode=true&characterEncoding=UTF-8&useSSL=false
    username: renren
    password: 123456
    #oracle配置
    #   driverClassName: oracle.jdbc.OracleDriver
    #   url: jdbc:oracle:thin:@47.100.206.162:1521:xe
    #   username: renren
    #   password: 123456
    #SQLServer配置
    #   driverClassName: com.microsoft.sqlserver.jdbc.SQLServerDriver
    #   url: jdbc:sqlserver://192.168.10.10:1433;DatabaseName=renren_fast
    #   username: sa
    #   password: 123456
    #PostgreSQL配置
    #   driverClassName: org.postgresql.Driver

```

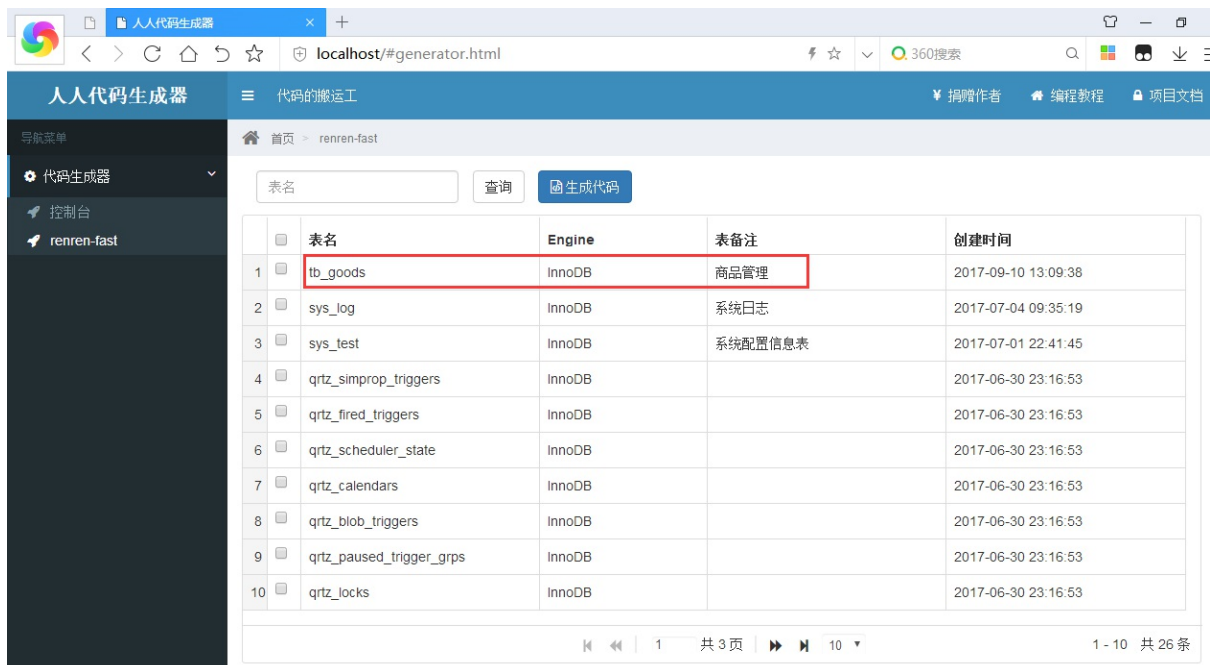
```
# url: jdbc:postgresql://192.168.10.10:5432/renren_fast
# username: postgres
# password: 123456
jackson:
  time-zone: GMT+8
  date-format: yyyy-MM-dd HH:mm:ss
resources:
  static-locations: classpath:/static/,classpath:/views/

mybatis:
  mapperLocations: classpath:mapper/**/*.xml

pagehelper:
  reasonable: true
  supportMethodsArguments: true
  params: count=countSql

#指定数据库，可选值有【mysql、oracle、sqlserver、postgresql】
renren:
  database: mysql
```

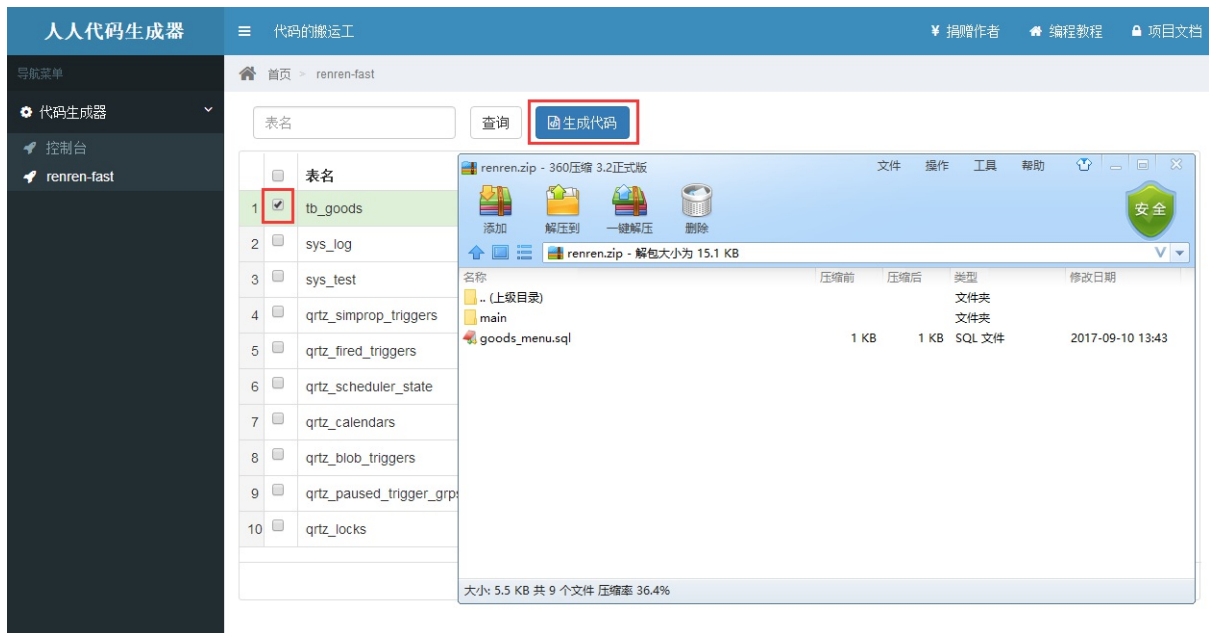
- 在数据库renren_fast中，执行建表语句，创建tb_goods表，再启动renren-generator项目(运行RenrenApplication.java的main方法即可)，如下所示：
- 在浏览器里输入项目地址(<http://localhost>)，如下所示：



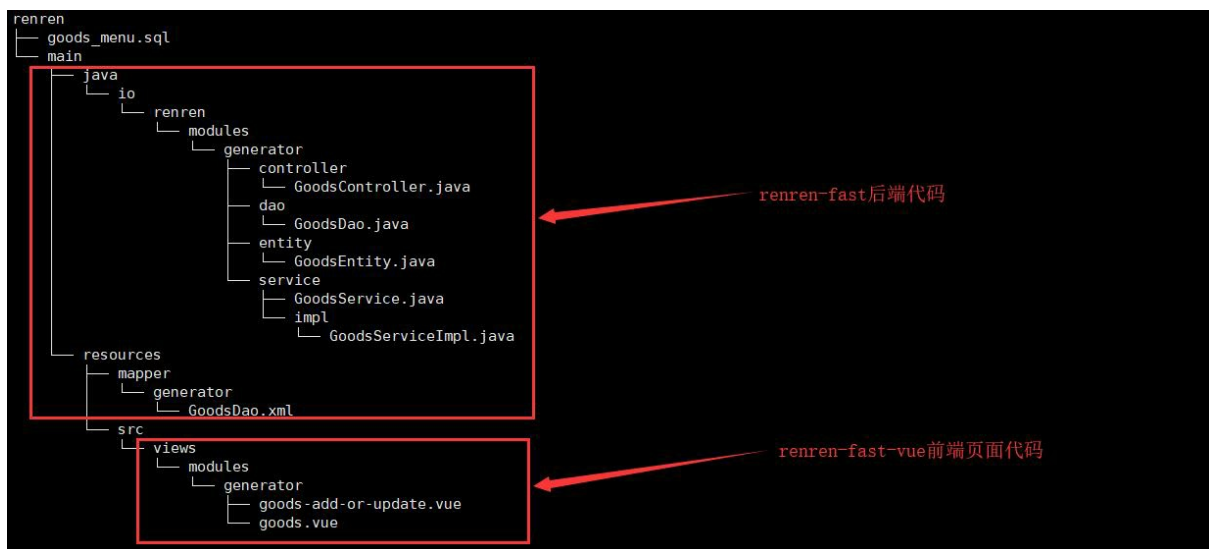
The screenshot shows the Renren-Generator web interface. The browser address bar shows `localhost/#generator.html`. The page title is "人人代码生成器". The left sidebar contains a navigation menu with "代码生成器" (Code Generator) selected. The main content area displays a table of database tables. The table has columns: "表名" (Table Name), "Engine", "表备注" (Table Remark), and "创建时间" (Creation Time). The table lists 10 tables, with "tb_goods" highlighted by a red box. The "tb_goods" table has a remark of "商品管理" (Goods Management). The "生成代码" (Generate Code) button is visible next to the "查询" (Query) button.

	表名	Engine	表备注	创建时间
1	tb_goods	InnoDB	商品管理	2017-09-10 13:09:38
2	sys_log	InnoDB	系统日志	2017-07-04 09:35:19
3	sys_test	InnoDB	系统配置信息表	2017-07-01 22:41:45
4	qrtz_simprop_triggers	InnoDB		2017-06-30 23:16:53
5	qrtz_fired_triggers	InnoDB		2017-06-30 23:16:53
6	qrtz_scheduler_state	InnoDB		2017-06-30 23:16:53
7	qrtz_calendars	InnoDB		2017-06-30 23:16:53
8	qrtz_blob_triggers	InnoDB		2017-06-30 23:16:53
9	qrtz_paused_trigger_grps	InnoDB		2017-06-30 23:16:53
10	qrtz_locks	InnoDB		2017-06-30 23:16:53

- 我们只需勾选tb_goods，点击【生成代码】按钮，则可生成相应代码，如下所示：



- 我们来看下生成的代码结构，如下所示：



- 生成好代码后，我们只需在数据库renren_fast中，执行goods_menu.sql语句，这个SQL是生成菜单的，SQL语句如下所示：

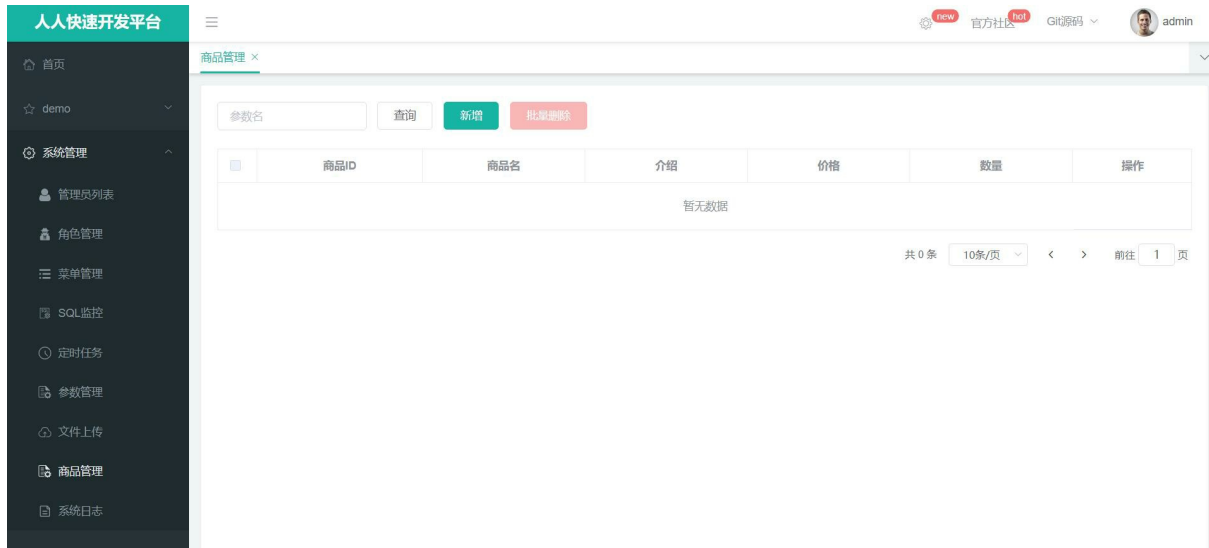
```
-- 菜单SQL
INSERT INTO `sys_menu` (`parent_id`, `name`, `url`, `perms`, `type`, `icon`, `order_num`)
VALUES ('1', '商品管理', 'generator/goods', NULL, '1', 'config', '6');

-- 按钮父菜单ID
set @parentId = @@identity;

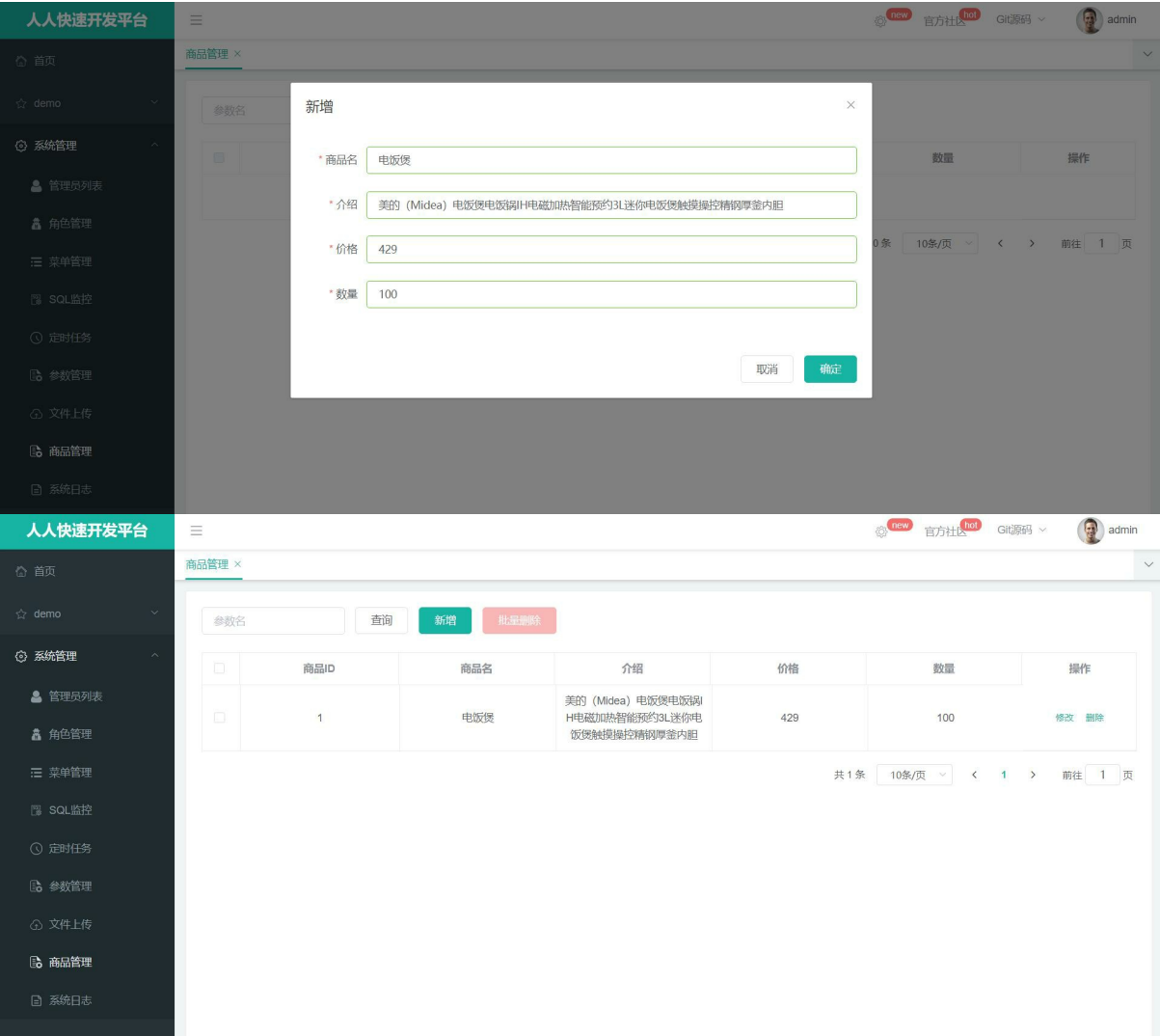
-- 菜单对应按钮SQL
INSERT INTO `sys_menu` (`parent_id`, `name`, `url`, `perms`, `type`, `icon`, `order_num`)
SELECT @parentId, '查看', null, 'generator:goods:list,generator:goods:info', '2', null, '6';
INSERT INTO `sys_menu` (`parent_id`, `name`, `url`, `perms`, `type`, `icon`, `order_num`)
SELECT @parentId, '新增', null, 'generator:goods:save', '2', null, '6';
```

```
INSERT INTO `sys_menu` (`parent_id`, `name`, `url`, `perms`, `type`, `icon`, `order_num`)
  SELECT @parentId, '修改', null, 'generator:goods:update', '2', null, '6';
INSERT INTO `sys_menu` (`parent_id`, `name`, `url`, `perms`, `type`, `icon`, `order_num`)
  SELECT @parentId, '删除', null, 'generator:goods:delete', '2', null, '6';
```

- 接下来，再把刚生成的后端代码，添加到项目renren-fast里，前端vue代码，添加到前端项目renren-fast-vue里，在启动renren-fast项目，如下所示：



- 现在，我们就可以新增、修改、删除等操作



第6章 后端源码分析

6.1 前后端分离

6.2 权限设计思路

6.3 XSS脚本过滤

6.4 SQL注入

6.5 Redis缓存

6.6 异常处理机制

6.7 后端效验机制

6.8 系统日志

6.9 添加菜单

6.10 添加角色

6.11 添加管理员

6.12 定时任务模块

6.13 云存储模块

6.14 APP模块

6.1 前后端分离

要实现前后端分离，需要考虑以下2个问题：

1. 项目不再基于session了，如何知道访问者是谁？
 2. 如何确认访问者的权限？
- 前后端分离，一般都是通过token实现，本项目也是一样；用户登录时，生成token及token过期时间，token与用户是一一对应关系，调用接口的时候，把token放到header或请求参数中，服务端就知道是谁在调用接口，登录如下所示：

```
/**
 * 验证码
 */
@GetMapping("captcha.jpg")
public void captcha(HttpServletResponse response, String uuid)throws ServletException, IOException {
    response.setHeader("Cache-Control", "no-store, no-cache");
    response.setContentType("image/jpeg");

    //获取图片验证码
    BufferedImage image = sysCaptchaService.getCaptcha(uuid);

    ServletOutputStream out = response.getOutputStream();
    ImageIO.write(image, "jpg", out);
    IOUtils.closeQuietly(out);
}

/**
 * 登录
 */
@PostMapping("/sys/login")
public Map<String, Object> login(@RequestBody SysLoginForm form)throws IOException {
    boolean captcha = sysCaptchaService.validate(form.getUuid(), form.getCaptcha());
    if(!captcha){
        return R.error("验证码不正确");
    }

    //用户信息
    SysUserEntity user = sysUserService.queryByUserName(form.getUsername());

    //账号不存在、密码错误
    if(user == null || !user.getPassword().equals(new Sha256Hash(form.getPassword(), user.getSalt()).toHex())) {
        return R.error("账号或密码不正确");
    }

    //账号锁定
    if(user.getStatus() == 0){
```

```

        return R.error("账号已被锁定,请联系管理员");
    }

    //生成token, 并保存到数据库
    R r = sysUserTokenService.createToken(user.getUserId());
    return r;
}

//生产token
public R createToken(long userId) {
    //生成一个token, 可以是uuid
    String token = TokenGenerator.generateValue();

    //当前时间
    Date now = new Date();
    //过期时间
    Date expireTime = new Date(now.getTime() + EXPIRE * 1000);

    //判断是否生成过token
    SysUserTokenEntity tokenEntity = queryByUserId(userId);
    if(tokenEntity == null){
        tokenEntity = new SysUserTokenEntity();
        tokenEntity.setUserId(userId);
        tokenEntity.setToken(token);
        tokenEntity.setUpdateTime(now);
        tokenEntity.setExpireTime(expireTime);

        //保存token
        save(tokenEntity);
    }else{
        tokenEntity.setToken(token);
        tokenEntity.setUpdateTime(now);
        tokenEntity.setExpireTime(expireTime);

        //更新token
        update(tokenEntity);
    }

    R r = R.ok().put("token", token).put("expire", EXPIRE);

    return r;
}

```

其中, 下面的这行代码, 是加盐操作; 可能有人不理解为何要加盐, 其目的是防止被拖库后, 黑客轻易的 (通过密码库对比), 就能拿到你的密码

```
new Sha256Hash(password, user.getSalt()).toHex())
```

- 调用接口时, 接受传过来的token后, 如何保证token有效及用户权限呢? 其实, shiro提供了 AuthenticatingFilter抽象类, 继承AuthenticatingFilter抽象类即可。

步骤1，所有请求全部拒绝访问

```
@Override
protected boolean isAccessAllowed(ServletRequest request, ServletResponse response, Object mappedValue) {
    return false;
}
```

步骤2，拒绝访问的请求，会调用onAccessDenied方法，onAccessDenied方法先获取token，再调用executeLogin方法

```
@Override
protected boolean onAccessDenied(ServletRequest request, ServletResponse response) throws Exception {
    //获取请求token，如果token不存在，直接返回401
    String token = getRequestToken((HttpServletRequest) request);
    if(StringUtils.isBlank(token)){
        HttpServletResponse httpResponse = (HttpServletResponse) response;
        String json = new Gson().toJson(R.error(HttpStatus.SC_UNAUTHORIZED, "invalid token"));

        httpResponse.getWriter().print(json);

        return false;
    }

    return executeLogin(request, response);
}

/**
 * 获取请求的token
 */
private String getRequestToken(HttpServletRequest httpRequest){
    //从header中获取token
    String token = httpRequest.getHeader("token");

    //如果header中不存在token，则从参数中获取token
    if(StringUtils.isBlank(token)){
        token = httpRequest.getParameter("token");
    }

    return token;
}
```

步骤3，阅读AuthenticatingFilter抽象类中executeLogin方法，我们发现调用了 `subject.login(token)`，这是shiro的登录方法，且需要token参数，我们自定义OAuth2Token类，只要实现AuthenticationToken接口，就可以了

```
//AuthenticatingFilter类中的方法
protected boolean executeLogin(ServletRequest request, ServletResponse response) throws Exception {
```

```

        AuthenticationToken token = createToken(request, response);
        if (token == null) {
            String msg = "createToken method implementation returned null. A valid non-null AuthenticationToken " +
                "must be created in order to execute a login attempt.";
            throw new IllegalStateException(msg);
        }
        try {
            Subject subject = getSubject(request, response);
            subject.login(token);
            return onLoginSuccess(token, subject, request, response);
        } catch (AuthenticationException e) {
            return onLoginFailure(token, e, request, response);
        }
    }
}

//OAuth2Filter类中的方法，继承了AuthenticatingFilter类
@Override
protected AuthenticationToken createToken(ServletRequest request, ServletResponse response) throws Exception {
    //获取请求token
    String token = getRequestToken((HttpServletRequest) request);

    if(StringUtils.isBlank(token)){
        return null;
    }

    return new OAuth2Token(token);
}

//subject.login(token)中的token对象，需要实现AuthenticationToken接口
public class OAuth2Token implements AuthenticationToken {
    private String token;

    public OAuth2Token(String token){
        this.token = token;
    }

    @Override
    public String getPrincipal() {
        return token;
    }

    @Override
    public Object getCredentials() {
        return token;
    }
}

```

步骤4，定义OAuth2Realm类，并继承AuthorizingRealm抽象类，调用 `subject.login(token)` 时，则会调用 `doGetAuthenticationInfo` 方法，进行登录

```

@Override
protected AuthenticationInfo doGetAuthenticationInfo(AuthenticationToken token) throws AuthenticationException {
    String accessToken = (String) token.getPrincipal();

    //根据accessToken, 查询用户信息
    SysUserTokenEntity tokenEntity = shiroService.queryByToken(accessToken);
    //token失效
    if(tokenEntity == null || tokenEntity.getExpireTime().getTime() < System.currentTimeMillis()){
        throw new IncorrectCredentialsException("token失效, 请重新登录");
    }

    //查询用户信息
    SysUserEntity user = shiroService.queryUser(tokenEntity.getUserId());
    //账号锁定
    if(user.getStatus() == 0){
        throw new LockedAccountException("账号已被锁定,请联系管理员");
    }

    SimpleAuthenticationInfo info = new SimpleAuthenticationInfo(user, accessToken, getName());
    return info;
}

```

步骤5, 登录失败后, 则调用onLoginFailure, 进行失败处理, 整个流程结束

```

@Override
protected boolean onLoginFailure(AuthenticationToken token, AuthenticationException e, ServletRequest request, ServletResponse response) {
    HttpServletResponse httpResponse = (HttpServletResponse) response;
    httpResponse.setContentType("application/json;charset=utf-8");
    try {
        //处理登录失败的异常
        Throwable throwable = e.getCause() == null ? e : e.getCause();
        R r = R.error(HttpStatus.SC_UNAUTHORIZED, throwable.getMessage());

        String json = new Gson().toJson(r);
        httpResponse.getWriter().print(json);
    } catch (IOException e1) {
    }

    return false;
}

```

步骤6, 登录成功后, 则调用doGetAuthorizationInfo方法, 查询用户的权限, 再调用具体的接口, 整个流程结束

```

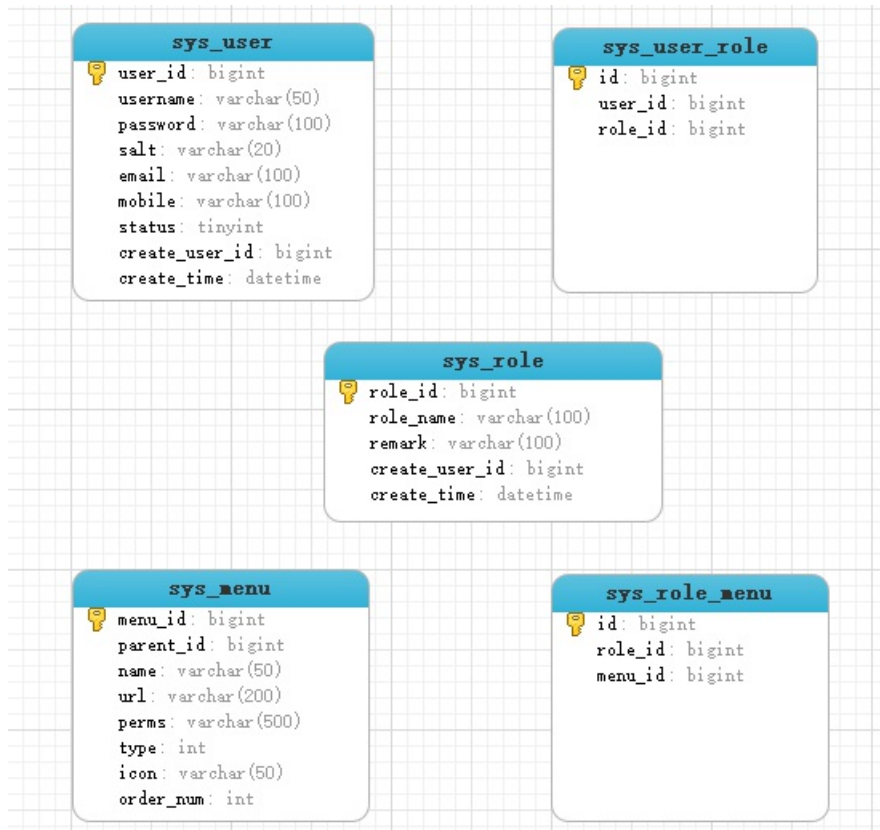
@Override

```

```
protected AuthorizationInfo doGetAuthorizationInfo(PrincipalCollection principals) {  
    SysUserEntity user = (SysUserEntity)principals.getPrimaryPrincipal();  
    Long userId = user.getUserId();  
  
    //用户权限列表  
    Set<String> permsSet = shiroService.getUserPermissions(userId);  
  
    SimpleAuthorizationInfo info = new SimpleAuthorizationInfo();  
    info.setStringPermissions(permsSet);  
    return info;  
}
```

6.2 权限设计思路

权限相关的表结构，如下图所示：



1. sys_user[用户]表，保存用户相关数据，通过sys_user_role[用户与角色关联]表，与sys_role[角色]表关联；sys_menu[菜单]表通过sys_role_menu[菜单与角色关联]表，与sys_role[角色]表关联
2. sys_menu表，保存菜单相关数据，并在perms字段里，保存了shiro的权限标识，也就是说，拥有此菜单，就拥有perms字段里的所有权限，比如，某用户拥有的菜单权限标识 `sys:user:info`，就可以访问下面的方法

```
@RequestMapping("/info/{userId}")
@RequiresPermissions("sys:user:info")
public R info(@PathVariable("userId") Long userId){

}
```

3. 在shiro配置代码里，配置为 `anon` 的，表示不经过shiro处理，配置为 `oauth2` 的，表示经过 `OAuth2Filter` 处理，前后端分离的接口，都会交给 `OAuth2Filter` 处理，这样就保证，没有权限的请求，拒绝访问

```
@Configuration
public class ShiroConfig {

    @Bean("sessionManager")
```



```

public SessionManager sessionManager(){
    DefaultWebSessionManager sessionManager = new DefaultWebSessionManager();
    sessionManager.setSessionValidationSchedulerEnabled(true);
    sessionManager.setSessionIdCookieEnabled(false);
    return sessionManager;
}

@Bean("securityManager")
public SecurityManager securityManager(OAuth2Realm oAuth2Realm, SessionManager sessionManager) {
    DefaultWebSecurityManager securityManager = new DefaultWebSecurityManager();
    securityManager.setRealm(oAuth2Realm);
    securityManager.setSessionManager(sessionManager);

    return securityManager;
}

@Bean("shiroFilter")
public ShiroFilterFactoryBean shiroFilter(SecurityManager securityManager) {
    ShiroFilterFactoryBean shiroFilter = new ShiroFilterFactoryBean();
    shiroFilter.setSecurityManager(securityManager);

    //oauth过滤
    Map<String, Filter> filters = new HashMap<>();
    filters.put("oauth2", new OAuth2Filter());
    shiroFilter.setFilters(filters);

    Map<String, String> filterMap = new LinkedHashMap<>();
    filterMap.put("/webjars/**", "anon");
    filterMap.put("/druid/**", "anon");
    filterMap.put("/app/**", "anon");
    filterMap.put("/sys/login", "anon");
    filterMap.put("/swagger/**", "anon");
    filterMap.put("/v2/api-docs", "anon");
    filterMap.put("/swagger-ui.html", "anon");
    filterMap.put("/swagger-resources/**", "anon");
    filterMap.put("/captcha.jpg", "anon");
    filterMap.put("/**", "oauth2");
    shiroFilter.setFilterChainDefinitionMap(filterMap);

    return shiroFilter;
}

@Bean("lifecycleBeanPostProcessor")
public LifecycleBeanPostProcessor lifecycleBeanPostProcessor() {
    return new LifecycleBeanPostProcessor();
}

@Bean
public AuthorizationAttributeSourceAdvisor authorizationAttributeSourceAdvisor(SecurityManager securityManager) {
    AuthorizationAttributeSourceAdvisor advisor = new AuthorizationAttributeSourceAdvisor

```

```
();  
    advisor.setSecurityManager(securityManager);  
    return advisor;  
}  
  
}
```

6.3 XSS脚本过滤

XSS跨站脚本攻击的基本原理和SQL注入攻击类似，都是利用系统执行了未经过滤的危险代码，不同点在于XSS是一种基于网页脚本的注入方式，也就是将脚本攻击载荷写入网页执行以达到对网页客户端访问用户攻击的目的，属于客户端攻击。程序员往往不太关心安全这块，这就给有心之人，提供了机会，本系统针对XSS攻击，提供了过滤功能，可以有效防止XSS攻击，代码如下：

```
public class XssFilter implements Filter {

    @Override
    public void init(FilterConfig config) throws ServletException {
    }

    public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
        throws IOException, ServletException {
        XssHttpServletRequestWrapper xssRequest = new XssHttpServletRequestWrapper(
            (HttpServletRequest) request);
        chain.doFilter(xssRequest, response);
    }

    @Override
    public void destroy() {
    }

}

@Configuration
public class FilterConfig {
    @Bean
    public FilterRegistrationBean xssFilterRegistration() {
        FilterRegistrationBean registration = new FilterRegistrationBean();
        registration.setDispatcherTypes(DispatcherType.REQUEST);
        registration.setFilter(new XssFilter());
        registration.addUrlPatterns("/");
        registration.setName("xssFilter");
        registration.setOrder(Integer.MAX_VALUE);
        return registration;
    }
}
```

- 自定义XssFilter过滤器，用来过滤所有请求，具体过滤还是在XssHttpServletRequestWrapper里实现的，如下所示：

```
public class XssHttpServletRequestWrapper extends HttpServletRequestWrapper {
    //没被包装过的HttpServletRequest（特殊场景，需要自己过滤）
    HttpServletRequest orgRequest;
    //html过滤
    private final static HTMLFilter htmlFilter = new HTMLFilter();
```

```

public XssHttpServletRequestWrapper(HttpServletRequest request) {
    super(request);
    orgRequest = request;
}

@Override
public ServletInputStream getInputStream() throws IOException {
    //非json类型, 直接返回
    if(!MediaType.APPLICATION_JSON_VALUE.equalsIgnoreCase(super.getHeader(HttpHeaders.CONTENT_TYPE))){
        return super.getInputStream();
    }

    //为空, 直接返回
    String json = IOUtils.toString(super.getInputStream(), "utf-8");
    if (StringUtils.isBlank(json)) {
        return super.getInputStream();
    }

    //xss过滤
    json = xssEncode(json);
    final ByteArrayInputStream bis = new ByteArrayInputStream(json.getBytes("utf-8"));
    return new ServletInputStream() {
        @Override
        public boolean isFinished() {
            return true;
        }

        @Override
        public boolean isReady() {
            return true;
        }

        @Override
        public void setReadListener(ReadListener readListener) {

        }

        @Override
        public int read() throws IOException {
            return bis.read();
        }
    };
}

@Override
public String getParameter(String name) {
    String value = super.getParameter(xssEncode(name));
    if (StringUtils.isNotBlank(value)) {
        value = xssEncode(value);
    }
}

```

```

        return value;
    }

    @Override
    public String[] getParameterValues(String name) {
        String[] parameters = super.getParameterValues(name);
        if (parameters == null || parameters.length == 0) {
            return null;
        }

        for (int i = 0; i < parameters.length; i++) {
            parameters[i] = xssEncode(parameters[i]);
        }
        return parameters;
    }

    @Override
    public Map<String,String[]> getParameterMap() {
        Map<String,String[]> map = new LinkedHashMap<>();
        Map<String,String[]> parameters = super.getParameterMap();
        for (String key : parameters.keySet()) {
            String[] values = parameters.get(key);
            for (int i = 0; i < values.length; i++) {
                values[i] = xssEncode(values[i]);
            }
            map.put(key, values);
        }
        return map;
    }

    @Override
    public String getHeader(String name) {
        String value = super.getHeader(xssEncode(name));
        if (StringUtils.isNotBlank(value)) {
            value = xssEncode(value);
        }
        return value;
    }

    private String xssEncode(String input) {
        return htmlFilter.filter(input);
    }

    /**
     * 获取最原始的request
     */
    public HttpServletRequest getOrgRequest() {
        return orgRequest;
    }

    /**
     * 获取最原始的request

```

```

    */
    public static HttpServletRequest getOrgRequest(HttpServletRequest request) {
        if (request instanceof XssHttpServletRequestWrapper) {
            return ((XssHttpServletRequestWrapper) request).getOrgRequest();
        }

        return request;
    }
}

```

如果需要处理富文本数据，可以通过 `XssHttpServletRequestWrapper.getOrgRequest(request)`，拿到原始的 `request` 对象后，再自行处理富文本数据，如：

```

public R data(HttpServletRequest request){
    HttpServletRequest orgRequest = XssHttpServletRequestWrapper.getOrgRequest(request);
    String content = orgRequest.getParameter("content");
    //富文本数据
    System.out.println(content);
    return R.ok();
}

```

6.4 SQL注入

本系统使用的是Mybatis，如果使用\${}拼接SQL，则存在SQL注入风险，可以对参数进行过滤，避免SQL注入，如下：

```
public class SQLFilter {

    /**
     * SQL注入过滤
     * @param str 待验证的字符串
     */
    public static String sqlInject(String str){
        if(StringUtils.isBlank(str)){
            return null;
        }
        //去掉'|' ';' '\字符
        str = StringUtils.replace(str, "'", "");
        str = StringUtils.replace(str, "\"", "");
        str = StringUtils.replace(str, ";", "");
        str = StringUtils.replace(str, "\\ ", "");

        //转换成小写
        str = str.toLowerCase();

        //非法字符
        String[] keywords = {"master", "truncate", "insert", "select", "delete", "update", "declare", "alter", "drop"};

        //判断是否包含非法字符
        for(String keyword : keywords){
            if(str.indexOf(keyword) != -1){
                throw new RuntimeException("包含非法字符");
            }
        }

        return str;
    }
}
```

像查询列表，涉及排序问题，排序字段是从前台传过来的，则存在SQL注入风险，需经如下处理：

```
public class Query<T> {

    public IPage<T> getPage(Map<String, Object> params) {
        return this.getPage(params, null, false);
    }

    public IPage<T> getPage(Map<String, Object> params, String defaultOrderField, boolean isA
```

```

sc) {
    //分页参数
    long curPage = 1;
    long limit = 10;

    if(params.get(Constant.PAGE) != null){
        curPage = Long.parseLong((String)params.get(Constant.PAGE));
    }
    if(params.get(Constant.LIMIT) != null){
        limit = Long.parseLong((String)params.get(Constant.LIMIT));
    }

    //分页对象
    Page<T> page = new Page<>(curPage, limit);

    //分页参数
    params.put(Constant.PAGE, page);

    //排序字段
    //防止SQL注入（因为sidx、order是通过拼接SQL实现排序的，会有SQL注入风险）
    String orderField = SQLFilter.sqlInject((String)params.get(Constant.ORDER_FIELD));
    String order = (String)params.get(Constant.ORDER);

    //前端字段排序
    if(StringUtils.isEmpty(orderField) && StringUtils.isEmpty(order)){
        if(Constant.ASC.equalsIgnoreCase(order)) {
            return page.setAsc(orderField);
        }else {
            return page.setDesc(orderField);
        }
    }

    //默认排序
    if(isAsc) {
        page.setAsc(defaultOrderField);
    }else {
        page.setDesc(defaultOrderField);
    }

    return page;
}
}

```


6.5 Redis缓存

缓存大家都很熟悉，但能否灵活运用，就不一定了。一般设计缓存架构时，我们需要考虑如下几个问题：

1. 查询数据的时候，尽量减少DB查询，DB主要负责写数据
2. 尽量不使用 `LEFT JOIN` 等关联查询，缓存命中率不高，还浪费内存
3. 多使用单表查询，缓存命中率最高
4. 数据库 `insert`、`update`、`delete` 时，同步更新缓存数据
5. 合理运用Redis数据结构，也许有质的飞跃
6. 对于访问量不大的项目，使用缓存只会增加项目的复杂度

本系统采用Redis作为缓存，并可配置是否开启redis缓存，主要还是通过Spring AOP实现的，配置如下所示：

```
redis:
  database: 0
  host: localhost
  port: 6379
  password:      # 密码（默认为空）
  timeout: 6000ms # 连接超时时长（毫秒）
  jedis:
    pool:
      max-active: 1000 # 连接池最大连接数（使用负值表示没有限制）
      max-wait: -1ms   # 连接池最大阻塞等待时间（使用负值表示没有限制）
      max-idle: 10     # 连接池中的最大空闲连接
      min-idle: 5      # 连接池中的最小空闲连接

renren:
  redis:
    open: false #是否开启redis缓存 true开启 false关闭
```

本项目中，使用Redis服务的代码，如下所示：

```
public class SysConfigServiceImpl implements SysConfigService {
    @Autowired
    private SysConfigDao sysConfigDao;
    @Autowired
    private SysConfigRedis sysConfigRedis;

    @Override
    @Transactional
    public void save(SysConfigEntity config) {
        sysConfigDao.save(config);
        sysConfigRedis.saveOrUpdate(config);
    }

    @Override
```

```

@Transactional
public void update(SysConfigEntity config) {
    sysConfigDao.update(config);
    sysConfigRedis.saveOrUpdate(config);
}

@Override
@Transactional
public void updateValueByKey(String key, String value) {
    sysConfigDao.updateValueByKey(key, value);
    sysConfigRedis.delete(key);
}

@Override
@Transactional
public void deleteBatch(Long[] ids) {
    sysConfigDao.deleteBatch(ids);

    for(Long id : ids){
        SysConfigEntity config = queryObject(id);
        sysConfigRedis.delete(config.getKey());
    }
}
}

-----

@Component
public class SysConfigRedis {
    @Autowired
    private RedisUtils redisUtils;

    public void saveOrUpdate(SysConfigEntity config) {
        if(config == null){
            return ;
        }
        String key = RedisKeys.getSysConfigKey(config.getKey());
        redisUtils.set(key, config);
    }

    public void delete(String configKey) {
        String key = RedisKeys.getSysConfigKey(configKey);
        redisUtils.delete(key);
    }

    public SysConfigEntity get(String configKey){
        String key = RedisKeys.getSysConfigKey(configKey);
        return redisUtils.get(key, SysConfigEntity.class);
    }
}

```

```

-----

package io.renren.common.aspect;

@Component
public class RedisUtils {
    @Autowired
    private RedisTemplate<String, Object> redisTemplate;
    @Autowired
    private ValueOperations<String, String> valueOperations;
    @Autowired
    private HashOperations<String, String, Object> hashOperations;
    @Autowired
    private ListOperations<String, Object> listOperations;
    @Autowired
    private SetOperations<String, Object> setOperations;
    @Autowired
    private ZSetOperations<String, Object> zSetOperations;
    /** 默认过期时长, 单位: 秒 */
    public final static long DEFAULT_EXPIRE = 60 * 60 * 24;
    /** 不设置过期时长 */
    public final static long NOT_EXPIRE = -1;
    private final static Gson gson = new Gson();

    public void set(String key, Object value, long expire){
        valueOperations.set(key, toJson(value));
        if(expire != NOT_EXPIRE){
            redisTemplate.expire(key, expire, TimeUnit.SECONDS);
        }
    }

    public void set(String key, Object value){
        set(key, value, DEFAULT_EXPIRE);
    }

    public <T> T get(String key, Class<T> clazz, long expire) {
        String value = valueOperations.get(key);
        if(expire != NOT_EXPIRE){
            redisTemplate.expire(key, expire, TimeUnit.SECONDS);
        }
        return value == null ? null : fromJson(value, clazz);
    }

    public <T> T get(String key, Class<T> clazz) {
        return get(key, clazz, NOT_EXPIRE);
    }

    public String get(String key, long expire) {
        String value = valueOperations.get(key);

```

```

        if(expire != NOT_EXPIRE){
            redisTemplate.expire(key, expire, TimeUnit.SECONDS);
        }
        return value;
    }

    public String get(String key) {
        return get(key, NOT_EXPIRE);
    }

    public void delete(String key) {
        redisTemplate.delete(key);
    }

    /**
     * Object转成JSON数据
     */
    private String toJson(Object object){
        if(object instanceof Integer || object instanceof Long || object instanceof Float ||
            object instanceof Double || object instanceof Boolean || object instanceof String){
            return String.valueOf(object);
        }
        return gson.toJson(object);
    }

    /**
     * JSON数据, 转成Object
     */
    private <T> T fromJson(String json, Class<T> clazz){
        return gson.fromJson(json, clazz);
    }
}

```

大家可能会有疑问, 认为这个项目必须要配置Redis缓存, 不然会报错, 因为有操作Redis的代码, 其实不然, 通过Spring AOP, 我们可以控制, 是否真的使用Redis, 代码如下:

```

@Aspect
@Component
public class RedisAspect {
    private Logger logger = LoggerFactory.getLogger(getClass());
    /**
     * 是否开启redis缓存 true开启 false关闭
     */
    @Value("${renren.redis.open: false}")
    private boolean open;

    @Around("execution(* io.renren.common.utils.RedisUtils.*(..))")
    public Object around(ProceedingJoinPoint point) throws Throwable {
        Object result = null;
        if(open){
            try{

```

```
        result = point.proceed();
    }catch (Exception e){
        logger.error("redis error", e);
        throw new RRException("Redis服务异常");
    }
}
return result;
}
}
```

6.6 异常处理机制

本项目通过RRException异常类，抛出自定义异常，RRException继承RuntimeException，不能继承Exception，如果继承Exception，则Spring事务不会回滚。

RRException代码如下所示：

```
public class RRException extends RuntimeException {
    private static final long serialVersionUID = 1L;

    private String msg;
    private int code = 500;

    public RRException(String msg) {
        super(msg);
        this.msg = msg;
    }

    public RRException(String msg, Throwable e) {
        super(msg, e);
        this.msg = msg;
    }

    public RRException(String msg, int code) {
        super(msg);
        this.msg = msg;
        this.code = code;
    }

    public RRException(String msg, int code, Throwable e) {
        super(msg, e);
        this.msg = msg;
        this.code = code;
    }

    public String getMsg() {
        return msg;
    }

    public void setMsg(String msg) {
        this.msg = msg;
    }

    public int getCode() {
        return code;
    }

    public void setCode(int code) {
        this.code = code;
    }
}
```

```

    }

}

```

如何处理抛出的异常呢，我们定义了RREExceptionHandler类，并加上注解@RestControllerAdvice，就可以处理所有抛出的异常，并返回JSON数据。@RestControllerAdvice是由@ControllerAdvice、@ResponseBody注解组合而来的，可以查找@ControllerAdvice相关的资料，理解@ControllerAdvice注解的使用。

RREExceptionHandler代码如下所示：

```

@RestControllerAdvice
public class RREExceptionHandler {
    private Logger logger = LoggerFactory.getLogger(getClass());

    /**
     * 处理自定义异常
     */
    @ExceptionHandler(RREException.class)
    public R handleRREException(RREException e){
        R r = new R();
        r.put("code", e.getCode());
        r.put("msg", e.getMessage());

        return r;
    }

    @ExceptionHandler(DuplicateKeyException.class)
    public R handleDuplicateKeyException(DuplicateKeyException e){
        logger.error(e.getMessage(), e);
        return R.error("数据库中已存在该记录");
    }

    @ExceptionHandler(AuthorizationException.class)
    public R handleAuthorizationException(AuthorizationException e){
        logger.error(e.getMessage(), e);
        return R.error("没有权限，请联系管理员授权");
    }

    @ExceptionHandler(Exception.class)
    public R handleException(Exception e){
        logger.error(e.getMessage(), e);
        return R.error();
    }
}

```


6.7 后端效验机制

本项目，后端效验使用的是Hibernate Validator校验框架，且自定义ValidatorUtils工具类，用来效验数据。

Hibernate Validator官方文档：http://docs.jboss.org/hibernate/validator/5.4/reference/en-US/html_single/

ValidatorUtils代码如下所示：

```
public class ValidatorUtils {
    private static Validator validator;

    static {
        validator = Validation.buildDefaultValidatorFactory().getValidator();
    }

    /**
     * 校验对象
     * @param object      待校验对象
     * @param groups      待校验的组
     * @throws RRException 校验不通过，则报RRException异常
     */
    public static void validateEntity(Object object, Class<?>... groups)
        throws RRException {
        Set<ConstraintViolation<Object>> constraintViolations = validator.validate(object, groups);
        if (!constraintViolations.isEmpty()) {
            ConstraintViolation<Object> constraint = (ConstraintViolation<Object>)constraintViolations.iterator().next();
            throw new RRException(constraint.getMessage());
        }
    }
}
```

使用案例：

```
@RestController
@RequestMapping("/sys/user")
public class SysUserController extends AbstractController {
    /**
     * 保存用户
     */
    @SysLog("保存用户")
    @RequestMapping("/save")
    @RequiresPermissions("sys:user:save")
    public R save(@RequestBody SysUserEntity user){
        //保存用户时，效验SysUserEntity里，带有AddGroup注解的属性
        ValidatorUtils.validateEntity(user, AddGroup.class);
    }
}
```

```

        user.setCreateUserId(getUserId());
        sysUserService.save(user);

        return R.ok();
    }

    /**
     * 修改用户
     */
    @SysLog("修改用户")
    @RequestMapping("/update")
    @RequiresPermissions("sys:user:update")
    public R update(@RequestBody SysUserEntity user){
        //修改用户时，效验SysUserEntity里，带有UpdateGroup注解的属性
        ValidatorUtils.validateEntity(user, UpdateGroup.class);

        user.setCreateUserId(getUserId());
        sysUserService.update(user);

        return R.ok();
    }
}

```

```

-----

public class SysUserEntity implements Serializable {
    /**
     * 用户ID
     */
    private Long userId;

    /**
     * 用户名
     */
    @NotBlank(message="用户名不能为空", groups = {AddGroup.class, UpdateGroup.class})
    private String username;

    /**
     * 密码
     */
    @NotBlank(message="密码不能为空", groups = AddGroup.class)
    private String password;

    /**
     * 盐
     */
    private String salt;

    /**
     * 邮箱

```

```

    */
    @NotBlank(message="邮箱不能为空", groups = {AddGroup.class, UpdateGroup.class})
    @Email(message="邮箱格式不正确", groups = {AddGroup.class, UpdateGroup.class})
    private String email;

    /**
     * 手机号
     */
    private String mobile;

    /**
     * 状态 0: 禁用 1: 正常
     */
    private Integer status;

    /**
     * 角色ID列表
     */
    private List<Long> roleIdList;

    /**
     * 创建者ID
     */
    private Long createUserId;

    /**
     * 创建时间
     */
    private Date createTime;
}

```

通过分析上面的代码，我们来理解Hibernate Validator校验框架的使用。其中，username属性，表示保存或修改用户时，都会校验username属性；而password属性，表示只有保存用户时，才会校验password属性，也就是说，修改用户时，password可以不填写，允许为空。如果不指定属性的groups，则默认属于javax.validation.groups.Default.class分组，可以通过ValidatorUtils.validateEntity(user)进行校验。

6.8 系统日志

系统日志是通过Spring AOP实现的，我们自定义了注解 `@SysLog`，且只能在方法上使用，如下所示：

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface SysLog {

    String value() default "";
}
```

下面是自定义注解 `@SysLog` 的使用方式，如下所示：

```
@RestController
@RequestMapping("/sys/user")
public class SysUserController extends AbstractController {

    @SysLog("保存用户")
    @RequestMapping("/save")
    @RequiresPermissions("sys:user:save")
    public R save(@RequestBody SysUserEntity user){
        ValidatorUtils.validateEntity(user, AddGroup.class);

        user.setCreateUserId(getUserId());
        sysUserService.save(user);

        return R.ok();
    }
}
```

我们可以发现，只需要在保存日志的请求方法上，加上 `@SysLog` 注解，就可以把日志保存到数据库里了。

具体是在哪里把数据保存到数据库里的呢，我们定义了 `SysLogAspect` 处理类，就是来干这事的，如下所示：

```
/**
 * 系统日志，切面处理类
 */
@Aspect
@Component
public class SysLogAspect {
    @Autowired
    private SysLogService sysLogService;

    @Pointcut("@annotation(io.renren.common.annotation.SysLog)")
    public void logPointCut() {
```

```

}

@Around("logPointCut()")
public Object around(ProceedingJoinPoint point) throws Throwable {
    long beginTime = System.currentTimeMillis();
    //执行方法
    Object result = point.proceed();
    //执行时长(毫秒)
    long time = System.currentTimeMillis() - beginTime;

    //保存日志
    saveSysLog(point, time);

    return result;
}

private void saveSysLog(ProceedingJoinPoint joinPoint, long time) {
    MethodSignature signature = (MethodSignature) joinPoint.getSignature();
    Method method = signature.getMethod();

    SysLogEntity sysLog = new SysLogEntity();
    SysLog syslog = method.getAnnotation(SysLog.class);
    if(syslog != null){
        //注解上的描述
        sysLog.setOperation(syslog.value());
    }

    //请求的方法名
    String className = joinPoint.getTarget().getClass().getName();
    String methodName = signature.getName();
    sysLog.setMethod(className + "." + methodName + "()");

    //请求的参数
    Object[] args = joinPoint.getArgs();
    try{
        String params = new Gson().toJson(args[0]);
        sysLog.setParams(params);
    }catch (Exception e){

    }

    //获取request
    HttpServletRequest request = HttpContextUtils.getHttpServletRequest();
    //设置IP地址
    sysLog.setIp(IPUtils.getIpAddr(request));

    //用户名
    String username = ((SysUserEntity) SecurityUtils.getSubject().getPrincipal()).getUsername();
    sysLog.setUsername(username);

    sysLog.setTime(time);
}

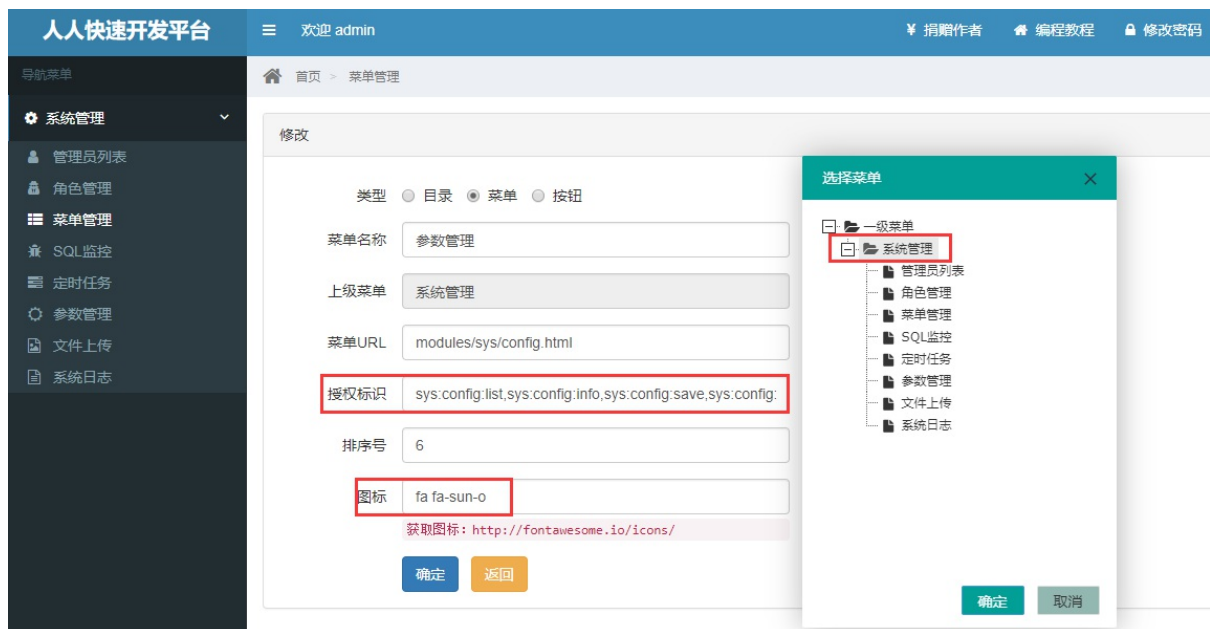
```

```
        sysLog.setCreateDate(new Date());  
        //保存系统日志  
        sysLogService.save(sysLog);  
    }  
}
```

`SysLogAspect` 类定义了一个切入点，请求 `@SysLog` 注解的方法时，会进入 `around` 方法，把系统日志保存到数据库中。

6.9 添加菜单

菜单管理，主要是对【目录、菜单、按钮】进行动态的新增、修改、删除等操作，方便开发者管理菜单。



上图是拿现有的菜单进行讲解。其中，授权标识与shiro中的注解@RequiresPermissions，定义的授权标识是一一对应的，如下所示：

```
@RestController
@RequestMapping("/sys/config")
public class SysConfigController extends AbstractController {

    @RequestMapping("/list")
    @RequiresPermissions("sys:config:list")
    public R list(@RequestParam Map<String, Object> params){

    }

    @RequestMapping("/info/{id}")
    @RequiresPermissions("sys:config:info")
    public R info(@PathVariable("id") Long id){

    }

    @RequestMapping("/save")
    @RequiresPermissions("sys:config:save")
    public R save(@RequestBody SysConfigEntity config){

    }

    @RequestMapping("/update")
```

```
@RequiresPermissions("sys:config:update")
public R update(@RequestBody SysConfigEntity config){

}

@RequestMapping("/delete")
@RequiresPermissions("sys:config:delete")
public R delete(@RequestBody Long[] ids){

}

}
```


6.10 添加角色

管理员权限是通过角色进行管理的，给管理员分配权限时，要先创建好角色。

下面创建了一个【开发角色】，如下图所示：

人人快速开发平台

欢迎 admin

捐赠作者 编程教程 修改密码

导航菜单

- 系统管理
- 管理员列表
- 角色管理
- 菜单管理
- SQL监控
- 定时任务
- 参数管理
- 文件上传
- 系统日志

新增

角色名称 开发角色

备注 开发人员使用

授权

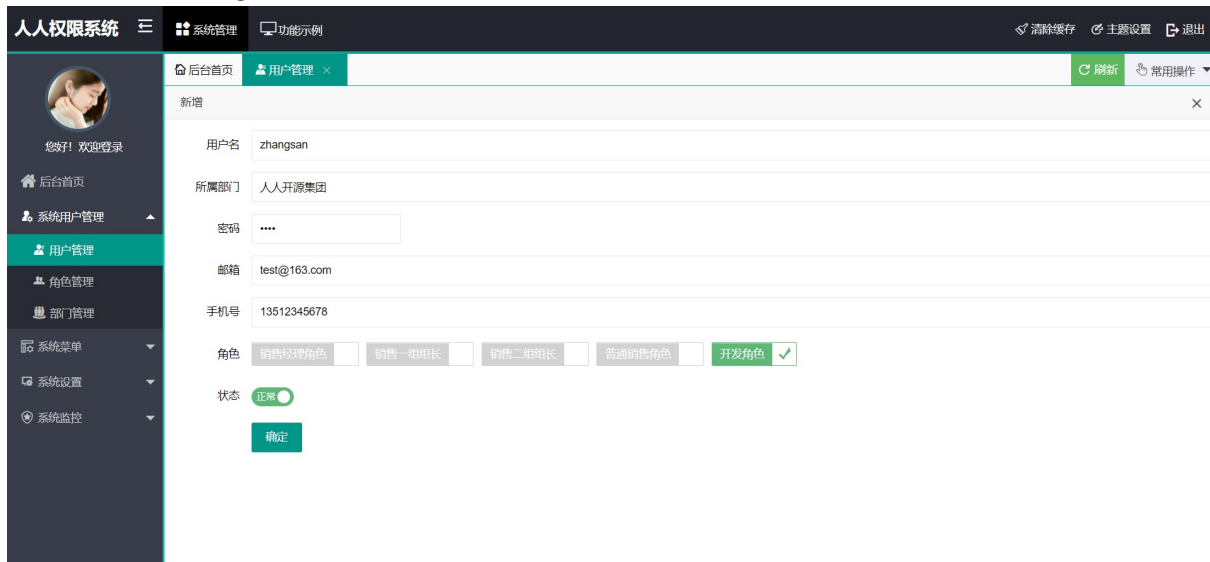
- ☒ 系统管理
 - ☒ 管理员列表
 - ☒ 角色管理
 - ☒ 菜单管理
 - ☒ SQL监控
 - ☒ 定时任务
 - ☒ 参数管理
 - ☒ 文件上传
 - ☒ 系统日志

确定 返回

6.11 添加管理员

本系统默认就创建了admin账号，无需分配任何角色，就拥有最高权限。一个管理员是可以拥有多个角色的。

下面创建一个【zhangsang】的管理员账号，并属于【开发角色】，如下所示：



The screenshot displays the 'User Management' section of the 'Human Rights System' (人人权限系统). The interface includes a sidebar with navigation options like 'System Management' (系统管理), 'User Management' (用户管理), 'Role Management' (角色管理), and 'Department Management' (部门管理). The main content area shows the 'Add User' (新增) form for creating a new user. The form fields are as follows:

- Username (用户名):** zhangsang
- Department (所属部门):** 人人开源集团
- Password (密码):** (masked with dots)
- Email (邮箱):** test@163.com
- Mobile Number (手机号):** 13512345678
- Role (角色):** A row of radio buttons for selecting a role. The 'Developer Role' (开发角色) is selected, indicated by a green checkmark.
- Status (状态):** A toggle switch set to 'Normal' (正常).
- Confirm (确定):** A green button to submit the form.

6.12 定时任务模块

本系统使用开源框架Quartz，实现的定时任务，已实现分布式定时任务，可部署多台服务器，不重复执行，以及动态增加、修改、删除、暂停、恢复、立即执行定时任务。Quartz自带了各数据库的SQL脚本，如果想更改成其他数据库，可参考Quartz相应的SQL脚本。


6.12.1 新增定时任务

新增一个定时任务，其实很简单，只要定义一个普通的Spring Bean即可，如下所示：

```
@Component("testTask")
public class TestTask implements ITask {
    private Logger logger = LoggerFactory.getLogger(getClass());

    @Override
    public void run(String params){
        logger.debug("TestTask定时任务正在执行，参数为: {}", params);
    }
}
```

如何让Quartz，定时执行testTask里的方法呢？只需要在管理后台，新增一个定时任务即可，如下图所示：

 [首页](#) > [定时任务](#)

修改

bean名称

testTask

参数

renren

cron表达式

0 0/30 * * * ?

备注

有参数测试

确定

返回

首页 > 定时任务

bean名称

查询

+ 新增

✎ 修改

🗑 删除

⏸ 暂停

▶ 恢复

⚡ 立即执行

日志列表

	任务ID	bean名称	参数	cron表达式	备注	状态
1	1	testTask	renren	0 0/30 * * * ?	有参数测试	正常

1

共 1 页

10

1 - 1 共 1 条

刚才配置的定时任务，每隔30分钟，就会调用TestTask的test方法了，是不是很简单啊。

6.12.2 源码分析

Quartz提供了相关的API，我们可以调用API，对Quartz进行增加、修改、删除、暂停、恢复、立即执行等。本系统中，ScheduleUtils 类就是对Quartz API进行的封装，代码如下所示：

```
public class ScheduleUtils {
    private final static String JOB_NAME = "TASK_";

    /**
     * 获取触发器key
     */
    private static TriggerKey getTriggerKey(Long jobId) {
        return TriggerKey.triggerKey(JOB_NAME + jobId);
    }

    /**
     * 获取jobKey
     */
    private static JobKey getJobKey(Long jobId) {
        return JobKey.jobKey(JOB_NAME + jobId);
    }

    /**
     * 获取表达式触发器
     */
    public static CronTrigger getCronTrigger(Scheduler scheduler, Long jobId) {
        try {
            return (CronTrigger) scheduler.getTrigger(getTriggerKey(jobId));
        } catch (SchedulerException e) {
            throw new RRException("getCronTrigger异常，请检查qrtz开头的表，是否有脏数据", e);
        }
    }
}
```

```

/**
 * 创建定时任务
 */
public static void createScheduleJob(Scheduler scheduler, ScheduleJobEntity scheduleJob)
{
    try {
        //构建job信息
        JobDetail jobDetail = JobBuilder.newJob(ScheduleJob.class).withIdentity(getJobKey(
            scheduleJob.getJobId())).build();

        //表达式调度构建器
        CronScheduleBuilder scheduleBuilder = CronScheduleBuilder.cronSchedule(scheduleJob
            .getCronExpression())
            .withMisfireHandlingInstructionDoNothing();

        //按新的cronExpression表达式构建一个新的trigger
        CronTrigger trigger = TriggerBuilder.newTrigger().withIdentity(getTriggerKey(scheduleJob
            .getJobId())).
            withSchedule(scheduleBuilder).build();

        //放入参数，运行时的方法可以获取
        jobDetail.getJobDataMap().put(ScheduleJobEntity.JOB_PARAM_KEY, new Gson().toJson(
            scheduleJob));

        scheduler.scheduleJob(jobDetail, trigger);

        //暂停任务
        if(scheduleJob.getStatus() == ScheduleStatus.PAUSE.getValue()){
            pauseJob(scheduler, scheduleJob.getJobId());
        }
    } catch (SchedulerException e) {
        throw new RRException("创建定时任务失败", e);
    }
}

/**
 * 更新定时任务
 */
public static void updateScheduleJob(Scheduler scheduler, ScheduleJobEntity scheduleJob)
{
    try {
        TriggerKey triggerKey = getTriggerKey(scheduleJob.getJobId());

        //表达式调度构建器
        CronScheduleBuilder scheduleBuilder = CronScheduleBuilder.cronSchedule(scheduleJob
            .getCronExpression())
            .withMisfireHandlingInstructionDoNothing();

        CronTrigger trigger = getCronTrigger(scheduler, scheduleJob.getJobId());

        //按新的cronExpression表达式重新构建trigger

```

```

        trigger = trigger.getTriggerBuilder().withIdentity(triggerKey).withSchedule(scheduler.getTriggerBuilder().build());

        //参数
        trigger.getJobDataMap().put(ScheduleJobEntity.JOB_PARAM_KEY, new Gson().toJson(scheduleJob));

        scheduler.rescheduleJob(triggerKey, trigger);

        //暂停任务
        if(scheduleJob.getStatus() == ScheduleStatus.PAUSE.getValue()){
            pauseJob(scheduler, scheduleJob.getJobId());
        }

    } catch (SchedulerException e) {
        throw new RRException("更新定时任务失败", e);
    }
}

/**
 * 立即执行任务
 */
public static void run(Scheduler scheduler, ScheduleJobEntity scheduleJob) {
    try {
        //参数
        JobDataMap dataMap = new JobDataMap();
        dataMap.put(ScheduleJobEntity.JOB_PARAM_KEY, new Gson().toJson(scheduleJob));

        scheduler.triggerJob(getJobKey(scheduleJob.getJobId()), dataMap);
    } catch (SchedulerException e) {
        throw new RRException("立即执行定时任务失败", e);
    }
}

/**
 * 暂停任务
 */
public static void pauseJob(Scheduler scheduler, Long jobId) {
    try {
        scheduler.pauseJob(getJobKey(jobId));
    } catch (SchedulerException e) {
        throw new RRException("暂停定时任务失败", e);
    }
}

/**
 * 恢复任务
 */
public static void resumeJob(Scheduler scheduler, Long jobId) {
    try {
        scheduler.resumeJob(getJobKey(jobId));
    } catch (SchedulerException e) {

```

```

        throw new RuntimeException("暂停定时任务失败", e);
    }
}

/**
 * 删除定时任务
 */
public static void deleteScheduleJob(Scheduler scheduler, Long jobId) {
    try {
        scheduler.deleteJob(getJobKey(jobId));
    } catch (SchedulerException e) {
        throw new RuntimeException("删除定时任务失败", e);
    }
}
}

```

以下是几个核心的方法：

- **createScheduleJob【创建定时任务】**：在管理后台新增任务时，会调用该方法，把任务添加到Quartz中，再根据cron表达式，定时执行任务。
- **updateScheduleJob【更新定时任务】**：修改任务时，调用该方法，修改Quartz中的任务信息。
- **run【立即执行定时任务】**：马上执行一次该任务，只执行一次。
- **pauseJob【暂停定时任务】**：这个不是暂停正在执行的任务，而是以后不再执行这个定时任务了。正在执行的任务，还是照常执行完。
- **resumeJob【恢复定时任务】**：这个是针对pauseJob来的，如果任务暂停了，以后都不会再执行，要想再执行，则需要调用resumeJob，使定时任务恢复执行。
- **deleteScheduleJob【删除定时任务】**：删除定时任务

其中，`createScheduleJob`、`updateScheduleJob` 在启动项目的时候，也会调用，把数据库里，新增或修改的任务，更新到Quartz中，如下所示：

```

@Service("scheduleJobService")
public class ScheduleJobServiceImpl implements ScheduleJobService {
    /**
     * 项目启动时，初始化定时器
     */
    @PostConstruct
    public void init(){
        List<ScheduleJobEntity> scheduleJobList = schedulerJobDao.queryList(new HashMap<>());
        for(ScheduleJobEntity scheduleJob : scheduleJobList){
            CronTrigger cronTrigger = ScheduleUtils.getCronTrigger(scheduler, scheduleJob.getJobId());

            //如果不存在，则创建
            if(cronTrigger == null) {
                ScheduleUtils.createScheduleJob(scheduler, scheduleJob);
            }else {
                ScheduleUtils.updateScheduleJob(scheduler, scheduleJob);
            }
        }
    }
}

```

```
}
```

大家是不是还有疑问呢，怎么就能定时执行，刚才在管理后台新增的任务testTask呢？下面我们再来分析下 `createScheduleJob` 方法，创建定时任务的时候，要调用该方法，代码如下所示：

```
//构建一个新的定时任务，JobBuilder.newJob()只能接受Job类型的参数
//把ScheduleJob.class作为参数传进去，ScheduleJob继承QuartzJobBean，而QuartzJobBean实现了Job接口
JobDetail jobDetail = JobBuilder.newJob(ScheduleJob.class).withIdentity(getJobKey(scheduleJob
.getJobId()))).build();

//构建cron，定时任务的周期
CronScheduleBuilder scheduleBuilder = CronScheduleBuilder.cronSchedule(scheduleJob.getCronExp
ression())
    .withMisfireHandlingInstructionDoNothing();

//根据cron，构建一个CronTrigger
CronTrigger trigger = TriggerBuilder.newTrigger().withIdentity(getTriggerKey(scheduleJob.getJ
obId())).
    withSchedule(scheduleBuilder).build();

//放入参数，运行时的方法可以获取
jobDetail.getJobDataMap().put(ScheduleJobEntity.JOB_PARAM_KEY, new Gson().toJson(scheduleJob)
);

//把任务添加到Quartz中
scheduler.scheduleJob(jobDetail, trigger);
```

把任务添加到 Quartz 后，等cron定义的时间周期到了，就会执行 ScheduleJob 类的 `executeInternal` 方法，ScheduleJob 代码如下所示：

```
public class ScheduleJob extends QuartzJobBean {
    private Logger logger = LoggerFactory.getLogger(getClass());

    @Override
    protected void executeInternal(JobExecutionContext context) throws JobExecutionException
    {
        ScheduleJobEntity scheduleJob = (ScheduleJobEntity) context.getMergedJobDataMap()
            .get(ScheduleJobEntity.JOB_PARAM_KEY);

        //获取spring bean
        ScheduleJobLogService scheduleJobLogService = (ScheduleJobLogService) SpringContextUt
            ils.getBean("scheduleJobLogService");

        //数据库保存执行记录
        ScheduleJobLogEntity log = new ScheduleJobLogEntity();
        log.setJobId(scheduleJob.getJobId());
        log.setBeanName(scheduleJob.getBeanName());
        log.setParams(scheduleJob.getParams());
        log.setCreateTime(new Date());

        //任务开始时间
```



```

        long startTime = System.currentTimeMillis();

        try {
            //执行任务
            logger.info("任务准备执行, 任务ID: " + scheduleJob.getJobId());

            Object target = SpringContextUtils.getBean(scheduleJob.getBeanName());
            Method method = target.getClass().getDeclaredMethod("run", String.class);
            method.invoke(target, scheduleJob.getParams());

            //任务执行总时长
            long times = System.currentTimeMillis() - startTime;
            log.setTimes((int)times);
            //任务状态    0: 成功    1: 失败
            log.setStatus(0);

            logger.info("任务执行完毕, 任务ID: " + scheduleJob.getJobId() + " 总共耗时: " + times + "毫秒");
        } catch (Exception e) {
            logger.error("任务执行失败, 任务ID: " + scheduleJob.getJobId(), e);

            //任务执行总时长
            long times = System.currentTimeMillis() - startTime;
            log.setTimes((int)times);

            //任务状态    0: 成功    1: 失败
            log.setStatus(1);
            log.setError(StringUtils.substring(e.toString(), 0, 2000));
        } finally {
            scheduleJobLogService.save(log);
        }
    }
}

```

6.13 云存储模块

图片、文件上传，使用的是七牛、阿里云、腾讯云的存储服务，不能上传到本地服务器。上传到本地服务器，不利于维护，访问速度慢等缺点，所以推荐使用云存储服务。

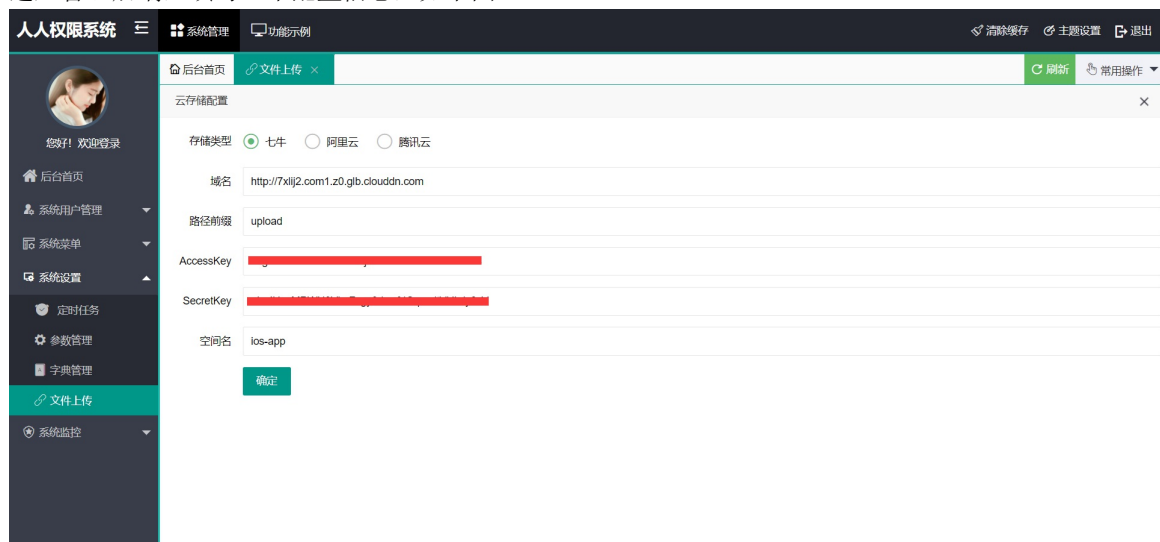
6.13.1 七牛的配置

如果没有七牛账号，则需要注册七牛账号，才能进行配置，下面演示注册七牛账号并配置，步骤如下：

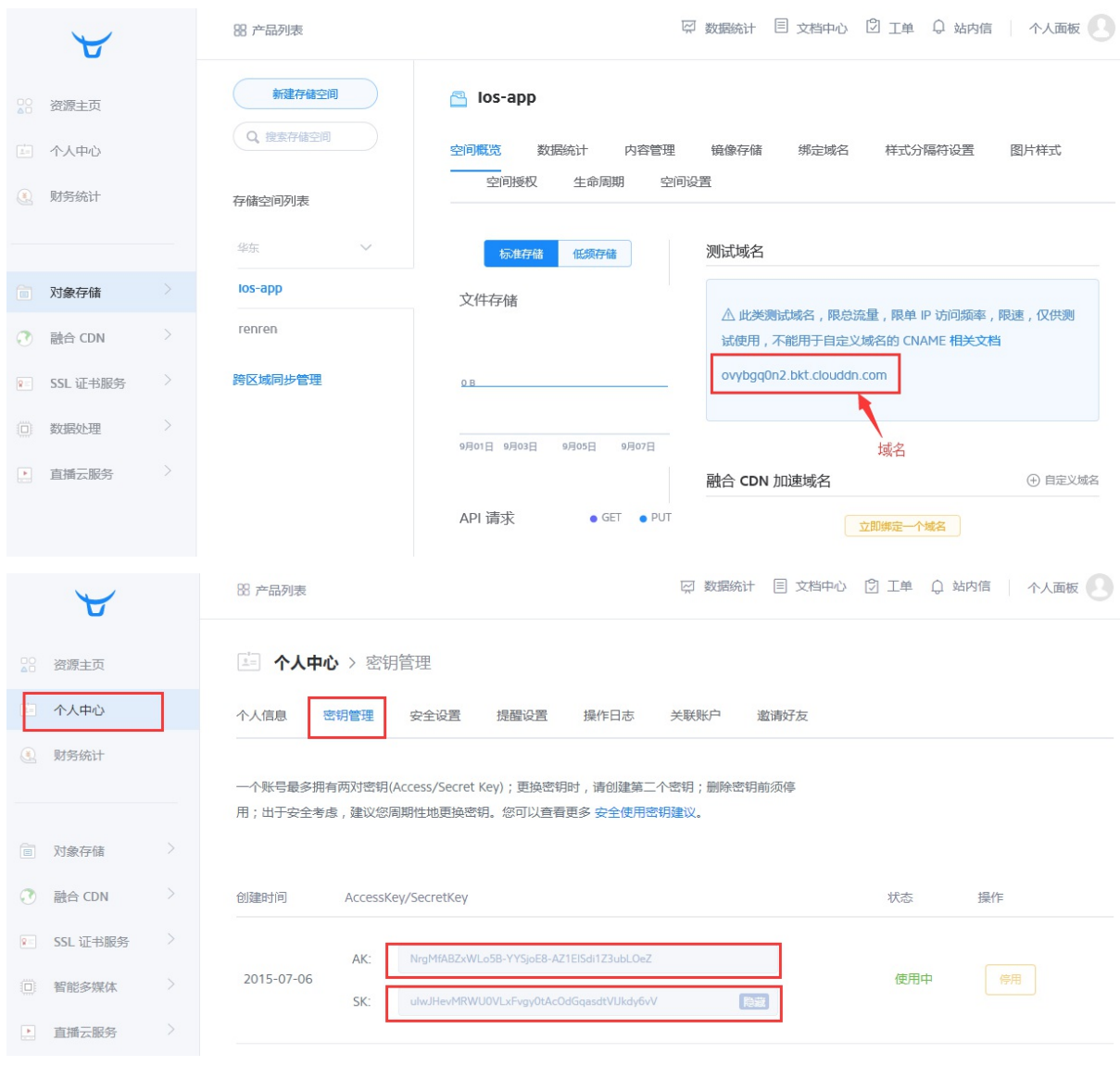
1. [注册七牛账号][34]，并登录后，再创建七牛空间，如下图：



2. 进入管理后端，填写七牛配置信息，如下图：



必填项有域名、AccessKey、SecretKey、空间名。其中，空间名就是才创建的空间名 `ios-app`，填进去就可以了。域名、AccessKey、SecretKey可以通过下图找到：



6.13.2 阿里云的配置

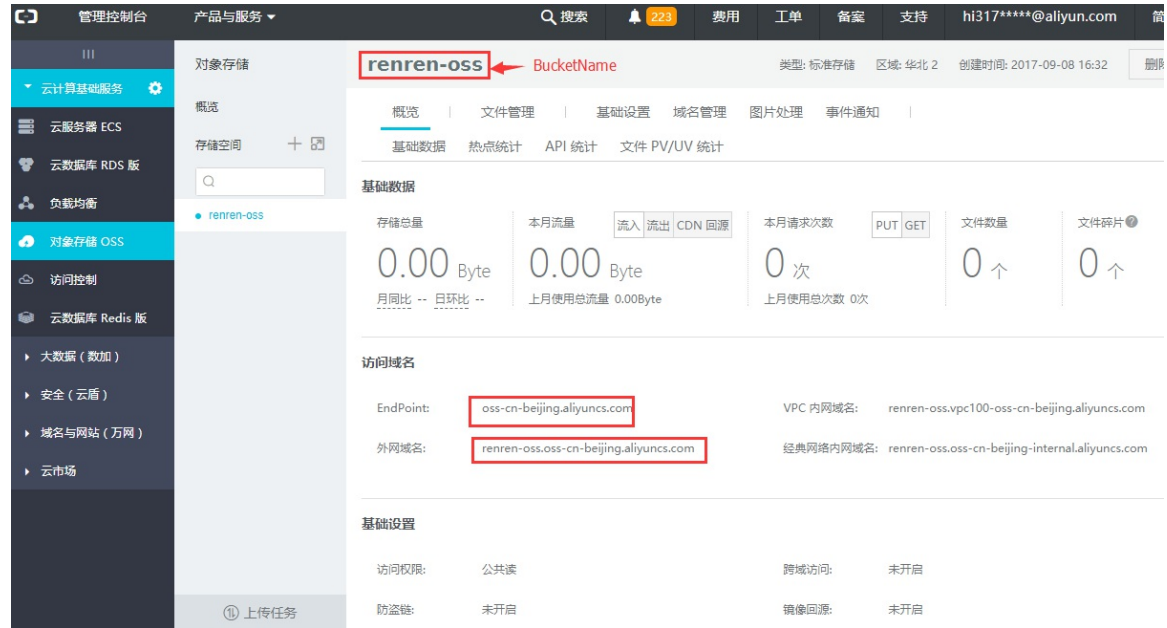
- 进入管理后端，填写阿里云配置信息，如下图：

The screenshot shows a web interface for system management. At the top, there are tabs for '系统管理' (System Management) and '功能示例' (Feature Examples). Below this is a navigation bar with '后台首页' (Backend Home) and '文件上传' (File Upload). The main section is titled '云存储配置' (Cloud Storage Configuration). It contains several input fields: '存储类型' (Storage Type) with radio buttons for '七牛' (Qiniu), '阿里云' (Alibaba Cloud), and '腾讯云' (Tencent Cloud); '域名' (Domain Name) with the value '阿里云绑定的域名'; '路径前缀' (Path Prefix) with the value '不设置默认为空'; 'EndPoint' with the value '阿里云EndPoint'; 'AccessKeyId' with the value '阿里云AccessKeyId'; 'AccessKeySecret' with the value '阿里云AccessKeySecret'; and 'BucketName' with the value '阿里云BucketName'. A green '确定' (Confirm) button is at the bottom.

- 进去阿里云管理后台，并创建Bucket，如下图：

The screenshot shows the Alibaba Cloud Management Console. On the left, the '对象存储 OSS' (Object Storage Service) menu item is highlighted. The main area shows the '新建 Bucket' (Create Bucket) dialog. The dialog has the following fields: '命名' (Name) with the value 'renren-oss'; 'Bucket 命名规范' (Bucket Naming Convention) with a list of rules; '所属地域' (Region) with the value '华北 2'; 'EndPoint' with the value 'oss-cn-beijing.aliyuncs.com'; '存储类型' (Storage Type) with the value '标准存储' (Standard Storage); and '读写权限' (Permissions) with the value '公共读' (Public Read). A green '确定' (Confirm) button is at the bottom right.

- 通过下面的界面，可以找到域名、BucketName、EndPoint



- 通过下面的界面，可以找到AccessKeyId、AccessKeySecret



6.13.3 腾讯云的配置

- 进入管理后端，填写腾讯云配置信息，如下图：

The screenshot shows a web interface for system management. At the top, there are tabs for '系统管理' (System Management) and '功能示例' (Feature Examples). Below this, there are tabs for '后台首页' (Backend Home) and '文件上传' (File Upload). The main section is titled '云存储配置' (Cloud Storage Configuration). It contains several input fields and a '确定' (Confirm) button at the bottom.

云存储配置

存储类型 ☐ 七牛 ☐ 阿里云 ☒ 腾讯云

域名

路径前缀

AppId

SecretId

SecretKey

BucketName

Bucket所属地区

- 进去腾讯云管理后台，并创建Bucket，如下图：

The screenshot shows the Tencent Cloud console. On the left, there is a sidebar with '云对象存储v4' (Cloud Object Storage v4) selected. Under it, 'Bucket列表' (Bucket List) is highlighted with a red box. The main area shows the '创建Bucket' (Create Bucket) dialog. The dialog has several fields and options for creating a new bucket.

腾讯云 总览 云产品 常用服务 English 备案 sunlightcs... 费用

云对象存储v4 < Bucket

Bucket

所属项目 默认项目

* 名称
仅支持小写字母、数字的组合，不能超过40字符。

地域
请根据您的业务就近存储，以提高访问速度。请注意，Bucket创建后不能修改所属地域，详见 [地域说明](#)

访问权限 ☐ 私有读写 ☒ 公有读私有写
公有读私有写: 可对object进行匿名读操作，写操作需要进行身份验证。

CDN加速 ☐ 开启 ☒ 关闭
开通腾讯云 CDN 来加速您访问。 [CDN 免费额度](#)

- 通过下面的界面，可以找到域名、BucketName、Bucket所属地区

The screenshot shows the Tencent Cloud COS console interface. On the left sidebar, the 'Bucket List' (Bucket列表) is highlighted. The main content area shows the 'Domain Management' (域名管理) tab, which lists default domains like 'renren-1251181044.cossh.myqcloud.com'. Below this, the 'Basic Information' (基本信息) tab is selected, showing details for the bucket 'renren', including its region 'Shanghai (华东) (sh)' and creation time '2017-09-08 23:18:31'.

- 通过下面的界面，可以找到AppId、SecretId、SecretKey

The screenshot shows the 'Personal API Key' (个人 API 密钥) page in the Tencent Cloud console. It includes a table with columns for AppID, Secret, Creation Time, Status, and Action. The first row shows AppID '1251181044' and Secret 'SecretId: AKIDMdebVAtjnnZ6MMpdwAQZ...'.

APPID	密钥	创建时间	状态	操作
1251181044	SecretId: AKIDMdebVAtjnnZ6MMpdwAQZ... SecretKey: ***** 显示	2017-03-29 13:57:20	已启用	禁用

6.13.4 源码分析

- 本项目的文件上传，使用的是七牛、阿里云、腾讯云，则需要引入他们的SDK，如下：

```
<dependency>
  <groupId>com.qiniu</groupId>
  <artifactId>qiniu-java-sdk</artifactId>
  <version>${qiniu.version}</version>
</dependency>
<dependency>
  <groupId>com.aliyun.oss</groupId>
  <artifactId>aliyun-sdk-oss</artifactId>
  <version>${aliyun.oss.version}</version>
</dependency>
<dependency>
  <groupId>com.qcloud</groupId>
  <artifactId>cos_api</artifactId>
  <version>${qcloud.cos.version}</version>
  <exclusions>
    <exclusion>
      <groupId>org.slf4j</groupId>
      <artifactId>slf4j-log4j12</artifactId>
    </exclusion>
  </exclusions>
</dependency>
```

- 定义抽象类 `CloudStorageService`，用来声明上传的公共接口，如下所示：

```
public abstract class CloudStorageService {
    /** 云存储配置信息 */
    CloudStorageConfig config;

    /**
     * 文件路径
     * @param prefix 前缀
     * @param suffix 后缀
     * @return 返回上传路径
     */
    public String getPath(String prefix, String suffix) {
        //生成uuid
        String uuid = UUID.randomUUID().toString().replaceAll("-", "");
        //文件路径
        String path = DateUtils.format(new Date(), "yyyyMMdd") + "/" + uuid;

        if(StringUtils.isNotBlank(prefix)){
            path = prefix + "/" + path;
        }

        return path + suffix;
    }

    /**
```



```

    * 文件上传
    * @param data      文件字节数组
    * @param path      文件路径, 包含文件名
    * @return          返回http地址
    */
    public abstract String upload(byte[] data, String path);

    /**
     * 文件上传
     * @param data      文件字节数组
     * @param suffix    后缀
     * @return          返回http地址
     */
    public abstract String uploadSuffix(byte[] data, String suffix);

    /**
     * 文件上传
     * @param inputStream 字节流
     * @param path        文件路径, 包含文件名
     * @return            返回http地址
     */
    public abstract String upload(InputStream inputStream, String path);

    /**
     * 文件上传
     * @param inputStream 字节流
     * @param suffix      后缀
     * @return            返回http地址
     */
    public abstract String uploadSuffix(InputStream inputStream, String suffix);
}

```

- 七牛上传的实现, 只需继承 `CloudStorageService` , 并实现相应的上传接口, 如下所示:

```

import com.qiniu.common.Zone;
import com.qiniu.http.Response;
import com.qiniu.storage.Configuration;
import com.qiniu.storage.UploadManager;
import com.qiniu.util.Auth;
import io.renren.common.exception.RRException;
import org.apache.commons.io.IOUtils;

import java.io.IOException;
import java.io.InputStream;

/**
 * 七牛云存储
 *
 * @author Mark sunlightcs@gmail.com
 */
public class QiniuCloudStorageService extends CloudStorageService {

```

```

private UploadManager uploadManager;
private String token;

public QiniuCloudStorageService(CloudStorageConfig config){
    this.config = config;

    //初始化
    init();
}

private void init(){
    uploadManager = new UploadManager(new Configuration(Zone.autoZone()));
    token = Auth.create(config.getQiniuAccessKey(), config.getQiniuSecretKey()).
        uploadToken(config.getQiniuBucketName());
}

@Override
public String upload(byte[] data, String path) {
    try {
        Response res = uploadManager.put(data, path, token);
        if (!res.isOK()) {
            throw new RuntimeException("上传七牛出错: " + res.toString());
        }
    } catch (Exception e) {
        throw new RuntimeException("上传文件失败, 请核对七牛配置信息", e);
    }

    return config.getQiniuDomain() + "/" + path;
}

@Override
public String upload(InputStream inputStream, String path) {
    try {
        byte[] data = IOUtils.toByteArray(inputStream);
        return this.upload(data, path);
    } catch (IOException e) {
        throw new RuntimeException("上传文件失败", e);
    }
}

@Override
public String uploadSuffix(byte[] data, String suffix) {
    return upload(data, getPath(config.getQiniuPrefix(), suffix));
}

@Override
public String uploadSuffix(InputStream inputStream, String suffix) {
    return upload(inputStream, getPath(config.getQiniuPrefix(), suffix));
}
}

```

- 阿里云上传的实现，只需继承 `CloudStorageService`，并实现相应的上传接口，如下所示：

```

import com.aliyun.oss.OSSClient;
import io.renren.common.exception.RRException;

import java.io.ByteArrayInputStream;
import java.io.InputStream;

/**
 * 阿里云存储
 *
 * @author Mark sunlightcs@gmail.com
 */
public class AliyunCloudStorageService extends CloudStorageService {
    private OSSClient client;

    public AliyunCloudStorageService(CloudStorageConfig config){
        this.config = config;

        //初始化
        init();
    }

    private void init(){
        client = new OSSClient(config.getAliyunEndPoint(), config.getAliyunAccessKeyId(),
            config.getAliyunAccessKeySecret());
    }

    @Override
    public String upload(byte[] data, String path) {
        return upload(new ByteArrayInputStream(data), path);
    }

    @Override
    public String upload(InputStream inputStream, String path) {
        try {
            client.putObject(config.getAliyunBucketName(), path, inputStream);
        } catch (Exception e){
            throw new RRException("上传文件失败, 请检查配置信息", e);
        }

        return config.getAliyunDomain() + "/" + path;
    }

    @Override
    public String uploadSuffix(byte[] data, String suffix) {
        return upload(data, getPath(config.getAliyunPrefix(), suffix));
    }

    @Override
    public String uploadSuffix(InputStream inputStream, String suffix) {
        return upload(inputStream, getPath(config.getAliyunPrefix(), suffix));
    }
}

```

- 腾讯云上传的实现，只需继承 `CloudStorageService`，并实现相应的上传接口，如下所示：

```
import com.qcloud.cos.COSClient;
import com.qcloud.cos.ClientConfig;
import com.qcloud.cos.request.UploadFileRequest;
import com.qcloud.cos.sign.Credentials;
import io.renren.common.exception.RRException;
import net.sf.json.JSONObject;
import org.apache.commons.io.IOUtils;

import java.io.IOException;
import java.io.InputStream;

/**
 * 腾讯云存储
 *
 * @author Mark sunlightcs@gmail.com
 */
public class QcloudCloudStorageService extends CloudStorageService {
    private COSClient client;

    public QcloudCloudStorageService(CloudStorageConfig config){
        this.config = config;

        //初始化
        init();
    }

    private void init(){
        Credentials credentials = new Credentials(config.getQcloudAppId(), config.getQcloudSecretId(),
            config.getQcloudSecretKey());

        //初始化客户端配置
        ClientConfig clientConfig = new ClientConfig();
        //设置bucket所在的区域，华南: gz 华北: tj 华东: sh
        clientConfig.setRegion(config.getQcloudRegion());

        client = new COSClient(clientConfig, credentials);
    }

    @Override
    public String upload(byte[] data, String path) {
        //腾讯云必需要以"/"开头
        if(!path.startsWith("/")) {
            path = "/" + path;
        }

        //上传到腾讯云
        UploadFileRequest request = new UploadFileRequest(config.getQcloudBucketName(), path,
            data);
```

```

        String response = client.uploadFile(request);

        JSONObject jsonObject = JSONObject.fromObject(response);
        if(jsonObject.getInt("code") != 0) {
            throw new RRException("文件上传失败, " + jsonObject.getString("message"));
        }

        return config.getQcloudDomain() + path;
    }

    @Override
    public String upload(InputStream inputStream, String path) {
        try {
            byte[] data = IOUtils.toByteArray(inputStream);
            return this.upload(data, path);
        } catch (IOException e) {
            throw new RRException("上传文件失败", e);
        }
    }

    @Override
    public String uploadSuffix(byte[] data, String suffix) {
        return upload(data, getPath(config.getQcloudPrefix(), suffix));
    }

    @Override
    public String uploadSuffix(InputStream inputStream, String suffix) {
        return upload(inputStream, getPath(config.getQcloudPrefix(), suffix));
    }
}

```

- 对外提供了OSSFactory工厂，可方便业务的调用，如下所示：

```

public final class OSSFactory {
    private static SysConfigService sysConfigService;

    static {
        OSSFactory.sysConfigService = (SysConfigService) SpringContextUtils.getBean("sysConfigService");
    }

    public static CloudStorageService build(){
        //获取云存储配置信息
        CloudStorageConfig config = sysConfigService.getConfigObject(ConfigConstant.CLOUD_STORAGE_CONFIG_KEY, CloudStorageConfig.class);

        if(config.getType() == Constant.CloudService.QINIU.getValue()){
            return new QiniuCloudStorageService(config);
        }else if(config.getType() == Constant.CloudService.ALIYUN.getValue()){
            return new AliyunCloudStorageService(config);
        }else if(config.getType() == Constant.CloudService.QCLOUD.getValue()){
            return new QcloudCloudStorageService(config);
        }
    }
}

```

```
    }  
  
    return null;  
}  
  
}
```

- 文件上传的例子，如下：

```
@RequestMapping("/upload")  
public R upload(@RequestParam("file") MultipartFile file) throws Exception {  
    if (file.isEmpty()) {  
        throw new RRException("上传文件不能为空");  
    }  
  
    //上传文件，并返回文件的http地址  
    String url = OSSFactory.build().upload(file.getBytes());  
}
```

6.14 APP模块

APP模块，是针对APP使用的，如IOS、Android等，主要是解决用户认证的问题。

6.14.1 APP的使用

APP的设计思路：用户通过APP，输入手机号、密码登录后，系统会生成与登录用户一一对应的token，用户调用需要登录的接口时，只需把token传过来，服务端就知道是谁在访问接口，token如果过期，则拒绝访问，从而保证系统的安全性。

使用很简单，看看下面的例子，就会使用了。仔细观察，我们会发现，有2个自定义的注解。其中，@LoginUser注解是获取当前登录用户的信息，有哪些信息，下面会分析的。@Login注解则是需要用户认证，没有登录的用户，不能访问该接口。

```
import io.renren.modules.app.annotation.Login;
import io.renren.modules.app.annotation.LoginUser;

@RestController
@RequestMapping("/app")
public class ApiTestController {

    /**
     * 获取用户信息
     */
    @Login
    @GetMapping("userInfo")
    public R userInfo(@LoginUser UserEntity user){
        return R.ok().put("user", user);
    }

    /**
     * 获取用户ID
     */
    @Login
    @GetMapping("userId")
    public R userInfo(@RequestAttribute("userId") Integer userId){
        return R.ok().put("userId", userId);
    }

    /**
     * 忽略Token验证测试
     */
    @GetMapping("notToken")
    public R notToken(){
        return R.ok().put("msg", "无需token也能访问。。。");
    }
}
```

6.14.2 源码分析

- 我们先来看看，APP用户登录的时候，都干了那些事情，如下所示：

```
@RestController
@RequestMapping("/app")
@Api("APP登录接口")
public class ApiLoginController {
    @Autowired
    private UserService userService;
    @Autowired
    private JwtUtils jwtUtils;

    /**
     * 登录
     */
    @PostMapping("login")
    @ApiOperation("登录")
    public R login(@RequestBody LoginForm form){
        //表单校验
        ValidatorUtils.validateEntity(form);

        //用户登录
        long userId = userService.login(form);

        //生成token
        String token = jwtUtils.generateToken(userId);

        Map<String, Object> map = new HashMap<>();
        map.put("token", token);
        map.put("expire", jwtUtils.getExpire());

        return R.ok(map);
    }
}

-----

/**
 * jwt工具类
 */
@ConfigurationProperties(prefix = "renren.jwt")
@Component
public class JwtUtils {
    private Logger logger = LoggerFactory.getLogger(getClass());

    private String secret;
    private long expire;
```



```

private String header;

/**
 * 生成jwt token
 */
public String generateToken(long userId) {
    Date nowDate = new Date();
    //过期时间
    Date expireDate = new Date(nowDate.getTime() + expire * 1000);

    return Jwts.builder()
        .setHeaderParam("typ", "JWT")
        .setSubject(userId+"")
        .setIssuedAt(nowDate)
        .setExpiration(expireDate)
        .signWith(SignatureAlgorithm.HS512, secret)
        .compact();
}

public Claims getClaimByToken(String token) {
    try {
        return Jwts.parser()
            .setSigningKey(secret)
            .parseClaimsJws(token)
            .getBody();
    }catch (Exception e){
        logger.debug("validate is token error ", e);
        return null;
    }
}

/**
 * token是否过期
 * @return true: 过期
 */
public boolean isTokenExpired(Date expiration) {
    return expiration.before(new Date());
}

public String getSecret() {
    return secret;
}

public void setSecret(String secret) {
    this.secret = secret;
}

public long getExpire() {
    return expire;
}

public void setExpire(long expire) {

```

```

        this.expire = expire;
    }

    public String getHeader() {
        return header;
    }

    public void setHeader(String header) {
        this.header = header;
    }
}

```

我们从上面的代码，可以看到，用户每次登录的时候，都会生成一个唯一的token，这个token是通过jwt生成的。

- APP模块的核心配置，如下所示：

```

import io.renren.modules.api.interceptor.AuthorizationInterceptor;
import io.renren.modules.api.resolver.LoginUserHandlerMethodArgumentResolver;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.method.support.HandlerMethodArgumentResolver;
import org.springframework.web.servlet.config.annotation.InterceptorRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurerAdapter;

@Configuration
public class WebMvcConfig extends WebMvcConfigurerAdapter {
    @Autowired
    private AuthorizationInterceptor authorizationInterceptor;
    @Autowired
    private LoginUserHandlerMethodArgumentResolver loginUserHandlerMethodArgumentResolver;

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(authorizationInterceptor).addPathPatterns("/app/**");
    }

    @Override
    public void addArgumentResolvers(List<HandlerMethodArgumentResolver> argumentResolvers) {
        argumentResolvers.add(loginUserHandlerMethodArgumentResolver);
    }
}

```

我们可以看到，配置了个Interceptor，用来拦截 /app 开头的请求，拦截后，会到 AuthorizationInterceptor类preHandle方法处理。只有以 /app 开头的请求，API模块认证才会起作用，如果要以 /api 开头，则需要修改此处。还配置了argumentResolver，别忽略了啊，下面会讲解。

温馨提示，别忘了配置shiro，不然会被shiro拦截掉的，如下所示：

```

@Configuration
public class ShiroConfig {

```

```

@Bean("shiroFilter")
public ShiroFilterFactoryBean shirFilter(SecurityManager securityManager) {

    //部分代码省略...

    Map<String, String> filterMap = new LinkedHashMap<>();
    //让shiro放过, 以/app开头的请求
    filterMap.put("/app/**", "anon");

    //部分代码省略...

    shiroFilter.setFilterChainDefinitionMap(filterMap);

    return shiroFilter;
}
}

```

- 分析AuthorizationInterceptor类, 我们可以发现, 拦截 /app 开头的请求后, 都干了些什么, 如下所示:

```

import io.jsonwebtoken.Claims;
import io.renren.common.exception.RRException;
import io.renren.modules.app.utils.JwtUtils;
import io.renren.modules.app.annotation.Login;
import org.apache.commons.lang.StringUtils;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.stereotype.Component;
import org.springframework.web.method.HandlerMethod;
import org.springframework.web.servlet.handler.HandlerInterceptorAdapter;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

/**
 * 权限(Token)验证
 */
@Component
public class AuthorizationInterceptor extends HandlerInterceptorAdapter {

    @Autowired
    private JwtUtils jwtUtils;

    public static final String USER_KEY = "userId";

    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {
        Login annotation;
        if(handler instanceof HandlerMethod) {
            annotation = ((HandlerMethod) handler).getMethodAnnotation(Login.class);
        }else{
            return true;
        }
    }
}

```

```

    }

    if(annotation == null){
        return true;
    }

    //获取用户凭证
    String token = request.getHeader(jwtUtils.getHeader());
    if(StringUtils.isBlank(token)){
        token = request.getParameter(jwtUtils.getHeader());
    }

    //凭证为空
    if(StringUtils.isBlank(token)){
        throw new RRException(jwtUtils.getHeader() + "不能为空", HttpStatus.UNAUTHORIZED.value());
    }

    Claims claims = jwtUtils.getClaimByToken(token);
    if(claims == null || jwtUtils.isTokenExpired(claims.getExpiration())){
        throw new RRException(jwtUtils.getHeader() + "失效, 请重新登录", HttpStatus.UNAUTHORIZED.value());
    }

    //设置userId到request里, 后续根据userId, 获取用户信息
    request.setAttribute(USER_KEY, Long.parseLong(claims.getSubject()));

    return true;
}
}

```

我们可以发现, 进入 `/app` 请求的接口之前, 会判断请求的接口, 是否加了`@Login`注解(需要token认证), 如果没有`@Login`注解, 则不验证token, 可以直接访问接口。如果有`@Login`注解, 则需要验证token的正确性, 并把userId放到request的USER_KEY里, 后续会用到。

- 此时, `@Login`注解的作用, 相信大家都明白了。再看看下面的代码, 加了`@LoginUser`注解后, `user`对象里, 就变成当前登录用户的信息, 这是什么时候设置进去的呢?

```

/**
 * 获取用户信息
 */
@GetMapping("userInfo")
public R userInfo(@LoginUser UserEntity user){
    return R.ok().put("user", user);
}

```

- 设置`user`对象进去, 其实是在`LoginUserHandlerMethodArgumentResolver`里干的, `LoginUserHandlerMethodArgumentResolver`是我们自定义的参数转换器, 只要实现`HandlerMethodArgumentResolver`接口即可, 代码如下所示:

```

import io.renren.modules.api.annotation.LoginUser;
import io.renren.modules.api.entity.UserEntity;
import io.renren.modules.api.interceptor.AuthorizationInterceptor;
import io.renren.modules.api.service.UserService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.core.MethodParameter;
import org.springframework.stereotype.Component;
import org.springframework.web.bind.support.WebDataBinderFactory;
import org.springframework.web.context.request.NativeWebRequest;
import org.springframework.web.context.request.RequestAttributes;
import org.springframework.web.method.support.HandlerMethodArgumentResolver;
import org.springframework.web.method.support.ModelAndViewContainer;

@Component
public class LoginUserHandlerMethodArgumentResolver implements HandlerMethodArgumentResolver
{
    @Autowired
    private UserService userService;

    @Override
    public boolean supportsParameter(MethodParameter parameter) {
        //如果方法的参数是UserEntity, 且参数前面有@LoginUser注解, 则进入resolveArgument方法, 进行处理
        return parameter.getParameterType().isAssignableFrom(UserEntity.class) && parameter.hasParameterAnnotation(LoginUser.class);
    }

    @Override
    public Object resolveArgument(MethodParameter parameter, ModelAndViewContainer container,
        NativeWebRequest request, WebDataBinderFactory factory) throws Exception {
        //获取用户ID, 之前设置进去的, 还有印象吧
        Object object = request.getAttribute(AuthorizationInterceptor.USER_KEY, RequestAttributes.SCOPE_REQUEST);
        if(object == null){
            return null;
        }

        //通过userId, 获取用户信息
        UserEntity user = userService.queryObject((Long)object);

        //把当前用户信息, 设置到UserEntity参数的user对象里
        return user;
    }
}

```

第7章 生产环境部署

部署项目前，需要准备JDK8、Maven、MySQL5.5+环境，参考开发环境搭建。

7.1 jar包部署

7.2 docker部署

7.3 集群部署

7.1 jar包部署

Spring Boot项目，推荐打成jar包的方式，部署到服务器上。

- Spring Boot内置了Tomcat，可配置Tomcat的端口号、初始化线程数、最大线程数、连接超时时长、https等等，如下所示：

```
server:
  tomcat:
    uri-encoding: UTF-8
    max-threads: 1000
    min-spare-threads: 30
  port: 8080
  connection-timeout: 5000ms
  servlet:
    context-path: /renren-fast
    session:
      cookie:
        http-only: true
  ssl:
    key-store: classpath:.keystore
    key-store-type: JKS
    key-password: 123456
    key-alias: tomcat
```

- 当然，还可以指定jvm的内存大小，如下所示：

```
java -Xms4g -Xmx4g -Xmn1g -server -jar renren-fast.jar
```

- 在windows下部署，只需打开cmd窗口，输入如下命令：

```
java -jar renren-fast.jar --spring.profiles.active=prod
```

- 在Linux下部署，只需输入如下命令，即可在Linux后台运行：

```
nohup java -jar renren-fast.jar --spring.profiles.active=prod > renren.log &
```

- 在Linux环境下，我们一般可以创建shell脚本，用于重启项目，如下所示：

```
#创建启动的shell脚本
[root@renren renren-fast]# vim start.sh
#!/bin/sh

process=`ps -fe|grep "renren-fast.jar" |grep -ivE "grep|cron" |awk '{print $2}'`
if [ ! $process ];
then
    echo "stop erp process $process ....."
```

```
        kill -9 $process
        sleep 1
    fi

    echo "start erp process....."
    nohup java -Dspring.profiles.active=prod -jar renren-fast.jar --server.port=8080 --server.servlet.context-path=/renren-fast 2>&1 | cronolog log.%Y-%m-%d.out >> /dev/null &

    echo "start erp success!"

#通过shell脚本启动项目
[root@renren renren-fast]# yum install -y cronolog
[root@renren renren-fast]# chmod +x start.sh
[root@renren renren-fast]# ./start.sh
```


7.2 docker部署

- 安装docker环境

```
#安装docker
[root@mark ~]# curl -sSL https://get.docker.com/ | sh

#启动docker
[root@mark ~]# service docker start

#查看docker版本信息
[root@mark ~]# docker version
Client:
 Version:      17.07.0-ce
 API version:  1.31
 Go version:   go1.8.3
 Git commit:   8784753
 Built:        Tue Aug 29 17:42:01 2017
 OS/Arch:      linux/amd64

Server:
 Version:      17.07.0-ce
 API version:  1.31 (minimum version 1.12)
 Go version:   go1.8.3
 Git commit:   8784753
 Built:        Tue Aug 29 17:43:23 2017
 OS/Arch:      linux/amd64
 Experimental: false
```

- 还需要准备java、maven环境，请自行安装
- 通过maven插件，构建docker镜像

```
#打包并构建项目镜像
[root@mark renren-fast]# mvn clean package docker:build
#省略打包log...
[INFO] Building image renren/fast
Step 1/6 : FROM java:8
---> d23bdf5b1b1b
Step 2/6 : EXPOSE 8080
---> Using cache
---> 8e33aadb2c18
Step 3/6 : VOLUME /tmp
---> Using cache
---> c5dc0c509062
Step 4/6 : ADD renren-fast-1.2.0.jar /app.jar
---> 831bc3ca84bc
Step 5/6 : RUN bash -c 'touch /app.jar'
---> Running in fe3ef9343e4c
```

```

---> b3d6dd6fc297
Removing intermediate container fe3ef9343e4c
Step 6/6 : ENTRYPOINT java -jar /app.jar
---> Running in 89adce4ae167
---> a4ae60970a77
Removing intermediate container 89adce4ae167
ProgressMessage{id=null, status=null, stream=null, error=null, progress=null, progressDetail=
null}
Successfully built a4ae60970a77
Successfully tagged renren/fast:latest

#查看镜像
[root@mark renren-fast]# docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
renren/fast          latest              a4ae60970a77       14 seconds ago     714MB
java                 8                  d23bdf5b1b1b       7 months ago       643MB

```

- 安装docker-compose，用来管理容器

```

#下载地址: https://github.com/docker/compose/releases

#下载docker-compose
[root@mark renren-fast]# curl -L https://github.com/docker/compose/releases/download/1.16.1/d
ocker-compose-`uname -s`-`uname -m` > /usr/local/bin/docker-compose

#增加可执行权限
[root@mark renren-fast]# chmod +x /usr/local/bin/docker-compose

#查看版本信息
[root@mark renren-fast]# docker-compose version
docker-compose version 1.16.1, build 6d1ac21
docker-py version: 2.5.1
CPython version: 2.7.13
OpenSSL version: OpenSSL 1.0.1t  3 May 2016

```

如果下载不了，可以用迅雷将https://github.com/docker/compose/releases/download/1.16.1/docker-compose-Linux-x86_64下载到本地，再上传到服务器

- 通过docker-compose，启动项目，如下所示：

```

#启动项目
[root@mark renren-fast]# docker-compose up -d
Creating network "renrenfast_default" with the default driver
Creating renrenfast_campus_1 ...
Creating renrenfast_campus_1 ... done

#查看启动的容器
[root@mark renren-fast]# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
PORTS              NAMES
f4e3fcdd8dd4       renren/fast        "java -jar /app.jar" 55 seconds ago     Up 3 secon

```

```
ds          0.0.0.0:8080->8080/tcp    renrenfast_renren-fast_1
```

#停掉并删除，docker-compose管理的容器

```
[root@mark renren-fast]# docker-compose down
```

```
Stopping renrenfast_renren-fast_1 ... done
```

```
Removing renrenfast_renren-fast_1 ... done
```

```
Removing network renrenfast_default
```

7.3 集群部署

本系统支持集群部署，集群部署，只需启动多个节点，并配置Nginx即可。

- 配置Nginx

```
http {
    upstream renren {
        server localhost:8080;
        server localhost:8081;
    }

    server {
        listen      80;
        server_name localhost;
        location /renren-fast {
            proxy_pass      http://renren;
            client_max_body_size 1024m;
            proxy_set_header X-Real-IP $remote_addr;
            proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
            proxy_set_header Host $host;
            proxy_redirect    off;
        }
    }
}
```

- 通过<http://localhost/renren-fast>，就可以访问了