

renren-security开发文档3.2_完整版

0. 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2018年4月最后修订。

1. 介绍

1.1. 项目描述

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、系统日志、APP模块等功能。其中，还拥有多数据源、数据权限、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-security](#)采用SpringBoot、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
- 完善的部门管理及数据权限，通过注解实现数据权限的控制
- 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. 项目介绍

项目一共分为四个模块

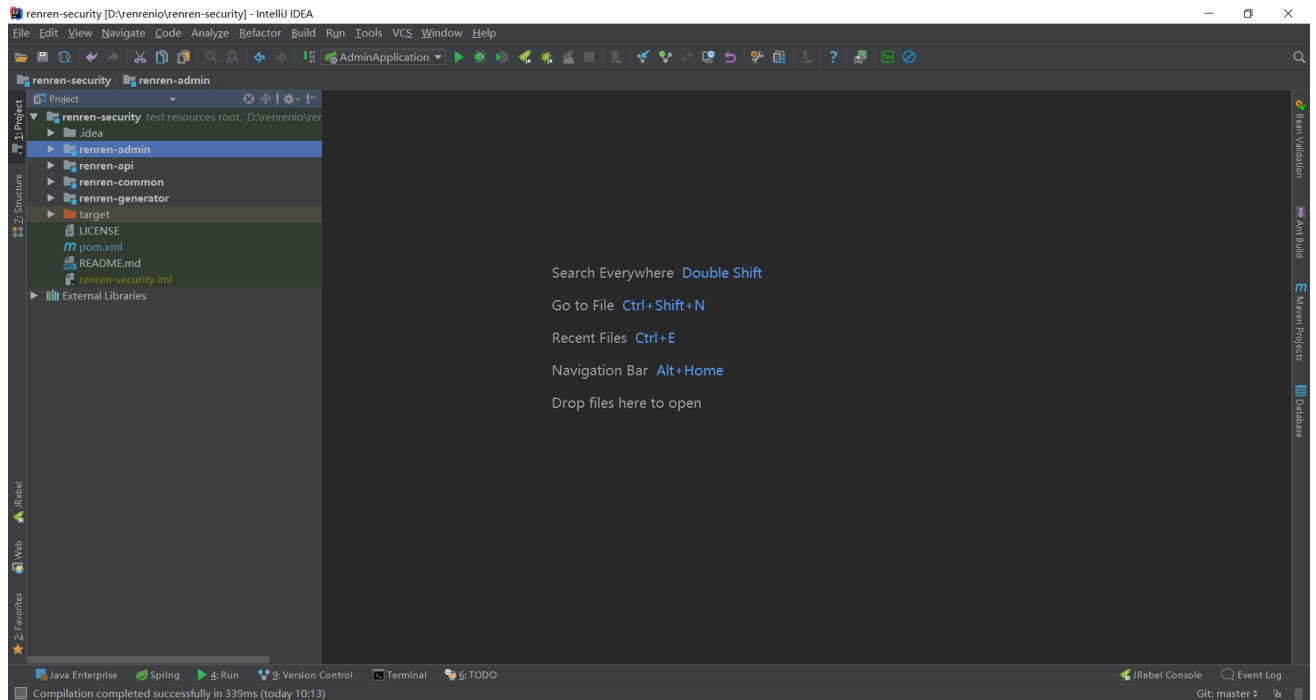
- renren-common为公共模块，其他模块以jar包的形式引入进去，主要提供些工具类，以及renren-admin、renren-api模块公共的entity、mapper、dao、service服务，防止一个功能重复多次编写代码。
- renren-admin为后台模块，也是系统的核心，用来开发后台管理系统，可以打包成jar，部署到服务器上运行，或者打包成war，放到Tomcat8.5+容器里运行。
- renren-api为接口模块，主要是简化APP开发，如：为微信小程序、IOS、Android提供接口，拥有一套单独的用户体系，没有与renren-admin用户表共用，因为renren-admin用户表里存放的是企业内部人员账号，具有后台管理员权限，可以登录后台管理系统，而renren-api用户表里存放的是我们的真实用户，不具备登录后台管理系统的权限。renren-api主要是实现了用户注册、登录、接口权限认证、获取登录用户等功能，为APP接口的安全调用，提供一套优雅的解决方案，从而简化APP接口开发。
- renren-generator为代码生成器模块，只需在MySQL数据库里，创建好表结构，就可以生成新增、修改、删除、查询、导出等操作的代码，包括entity、mapper、dao、service、controller、页面等所有代码，项目开发神器。

1.4. 本地部署

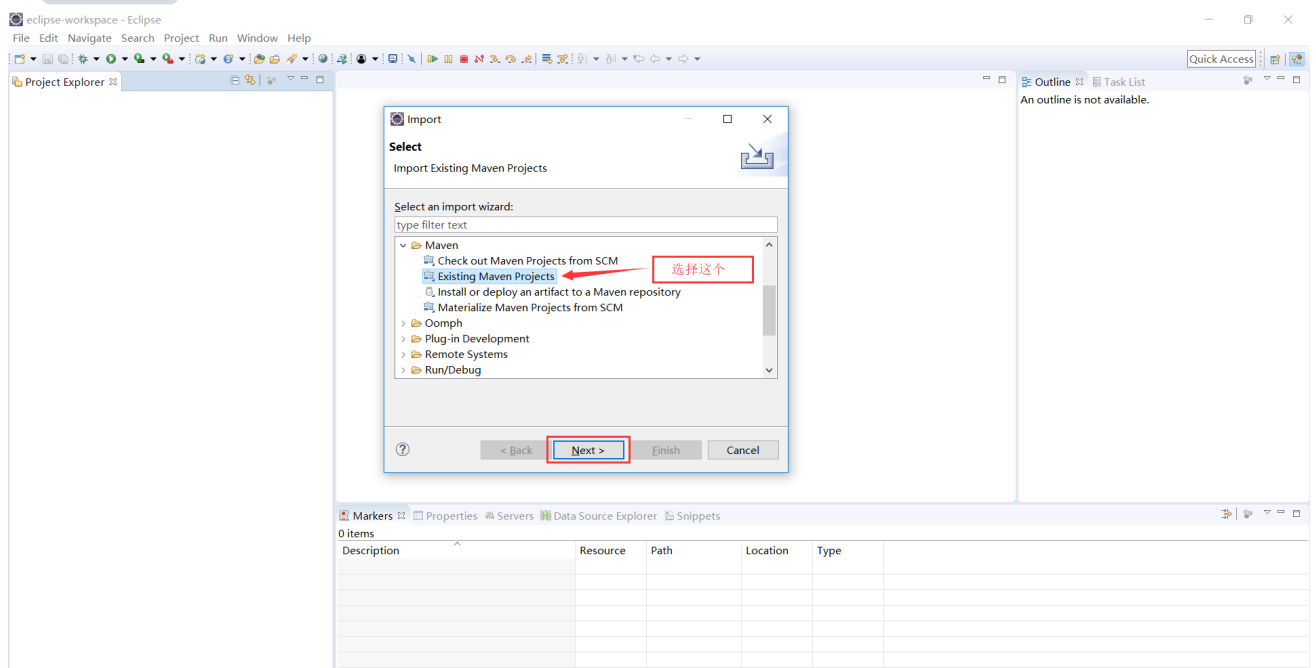
- 环境要求 JDK1.8、Tomcat8.5+、MySQL5.5+
- 通过 git ，下载renren-security源码，如下：

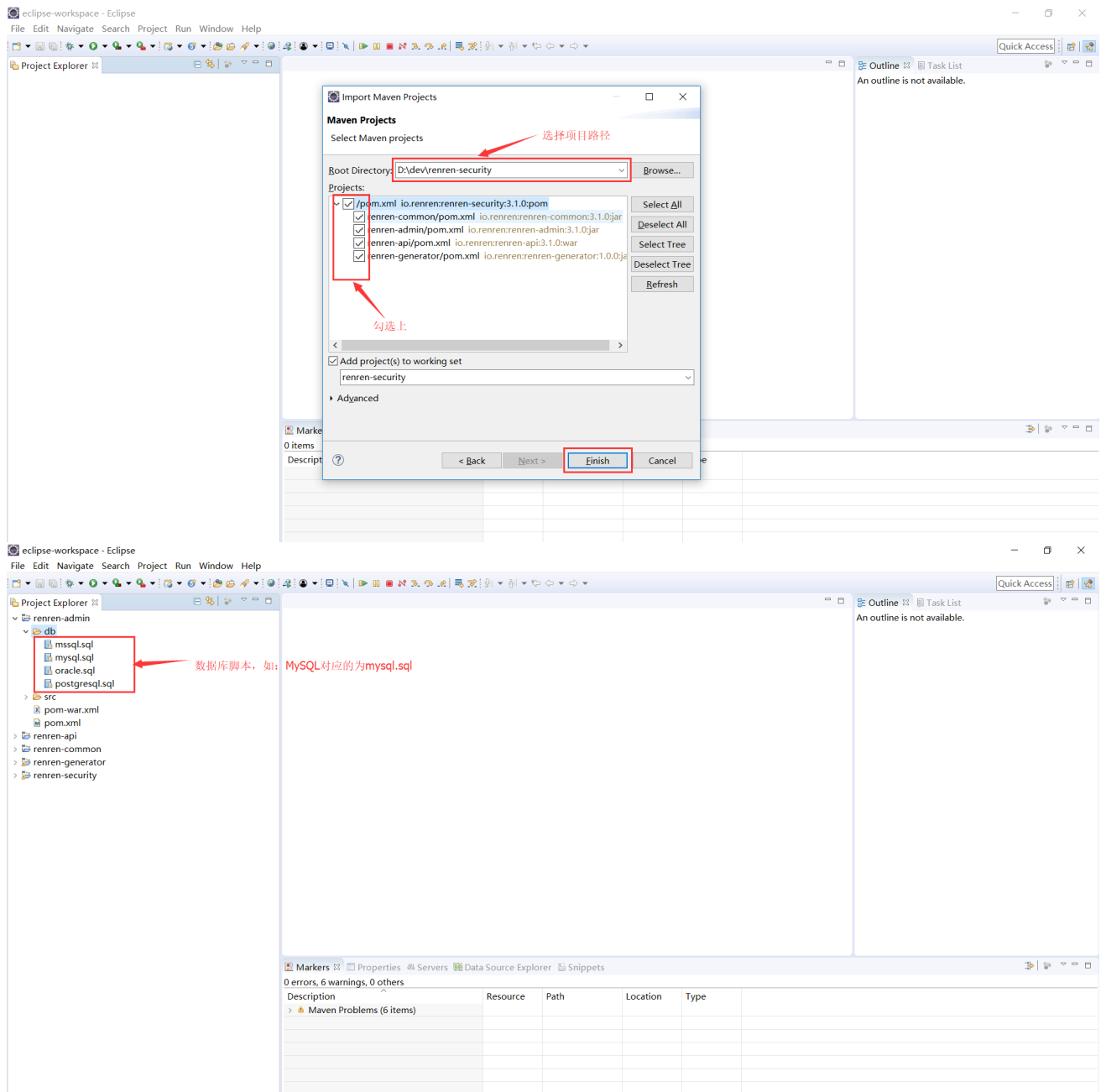
```
1. git clone https://gitee.com/renrenio/renren-security.git
```

- 用 idea 打开项目，File -> Open 如下图：



- 用 eclipse 打开项目，如下图：





- 创建数据库 `renren_security`，数据库编码为 `UTF-8`
- 执行数据库脚本，如MySQL数据库，则执行 `db/mysql.sql` 文件，初始化数据
- 修改 `application-dev.yml`，更改数据库账号和密码

【启动renren-admin项目】

- 运行 `io.renren.AdminApplication.java` 的 `main` 方法，则可启动renren-admin项目

- 项目访问路径：<http://localhost:8080/renren-admin>
 - 账号密码：admin/admin
 - Swagger路径：<http://localhost:8080/renren-admin/swagger/index.html>
 - Swagger注解路径：<http://localhost:8080/renren-admin/swagger-ui.html>
-

【启动renren-api项目】

- Eclipse、IDEA运行ApiApplication.java，则可启动项目【renren-api】
 - Swagger路径：<http://localhost:8081/renren-api/swagger-ui.html>
-

【启动renren-generator项目】

- Eclipse、IDEA运行GeneratorApplication.java，则可启动项目【renren-generator】
 - 项目访问路径：<http://localhost:8082/renren-generator>
-

1.5. 获取帮助

- Git地址：<https://gitee.com/renrenio/renren-security>
 - 官方社区：<http://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
-

2. 项目实战

2.1. 功能描述

我们来完成一个商品的列表、添加、修改、删除、导出功能，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：

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

2.2. 使用代码生成器

- 使用代码生成器前，我们先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：

```
1. #代码生成器，配置信息  
2.  
3. mainPath=io.renren  
4. #包名  
5. package=io.renren.modules  
6. moduleName=demo  
7. #作者  
8. author=chenshun  
9. #Email  
10. email=sunlightcs@gmail.com  
11. #表前缀 (类名不会包含表前缀)  
12. tablePrefix=tb_  
13.  
14. #类型转换，配置信息  
15. tinyint=Integer  
16. smallint=Integer  
17. mediumint=Integer  
18. int=Integer  
19. integer=Integer  
20. bigint=Long  
21. float=Float  
22. double=Double
```

```
23.    decimal=BigDecimal
24.    bit=Boolean
25.
26.    char=String
27.    varchar=String
28.    tinytext=String
29.    text=String
30.    mediumtext=String
31.    longtext=String
32.
33.    date=Date
34.    datetime=Date
35.    timestamp=Date
```

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。

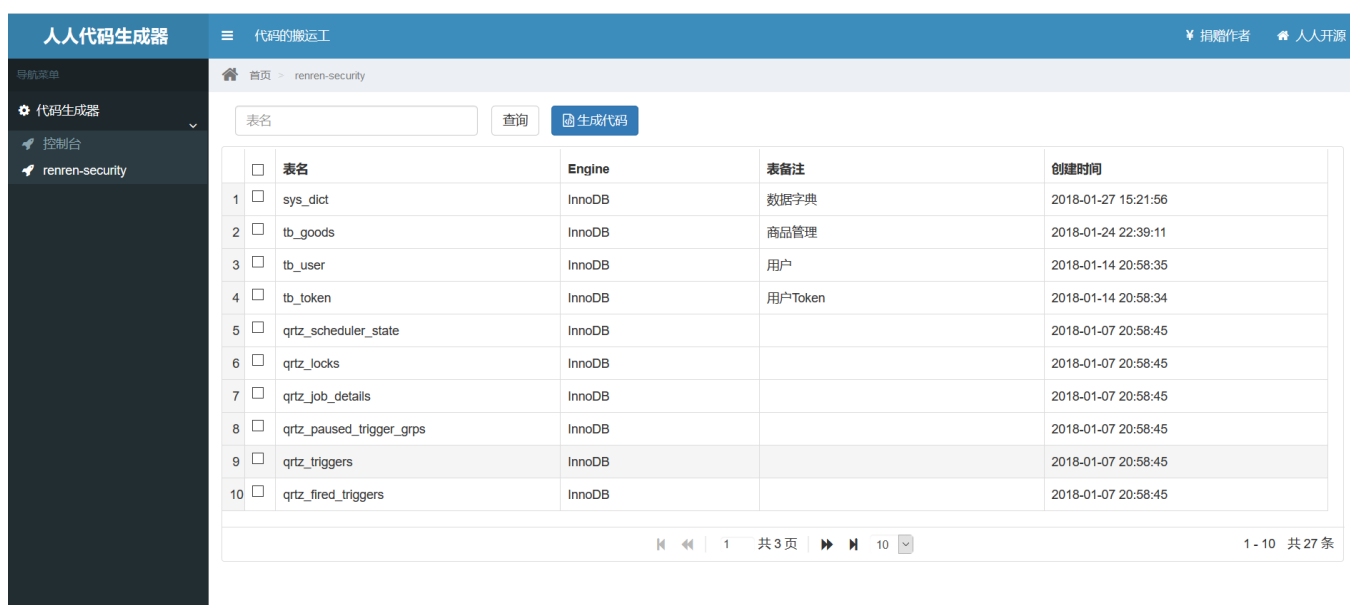
```
1.    # Tomcat
2.    server:
3.        tomcat:
4.            uri-encoding: UTF-8
5.            max-threads: 1000
6.            min-spare-threads: 30
7.        port: 8082
8.        context-path: /renren-generator
9.
10.   # mysql
11.   spring:
12.       datasource:
13.           type: com.alibaba.druid.pool.DruidDataSource
14.           driverClassName: com.mysql.jdbc.Driver
15.           url: jdbc:mysql://localhost:3306/renren_security?useUnicode=true&characterEncoding=UTF-8
16.           username: renren
17.           password: 123456
18.       jackson:
19.           time-zone: GMT+8
20.           date-format: yyyy-MM-dd HH:mm:ss
21.       resources:
```

```

22.         static-locations: classpath:/static/,classpath:/views/
23.
24.     # Mybatis配置
25.     mybatis:
26.         mapperLocations: classpath:mapper/**/*.xml

```

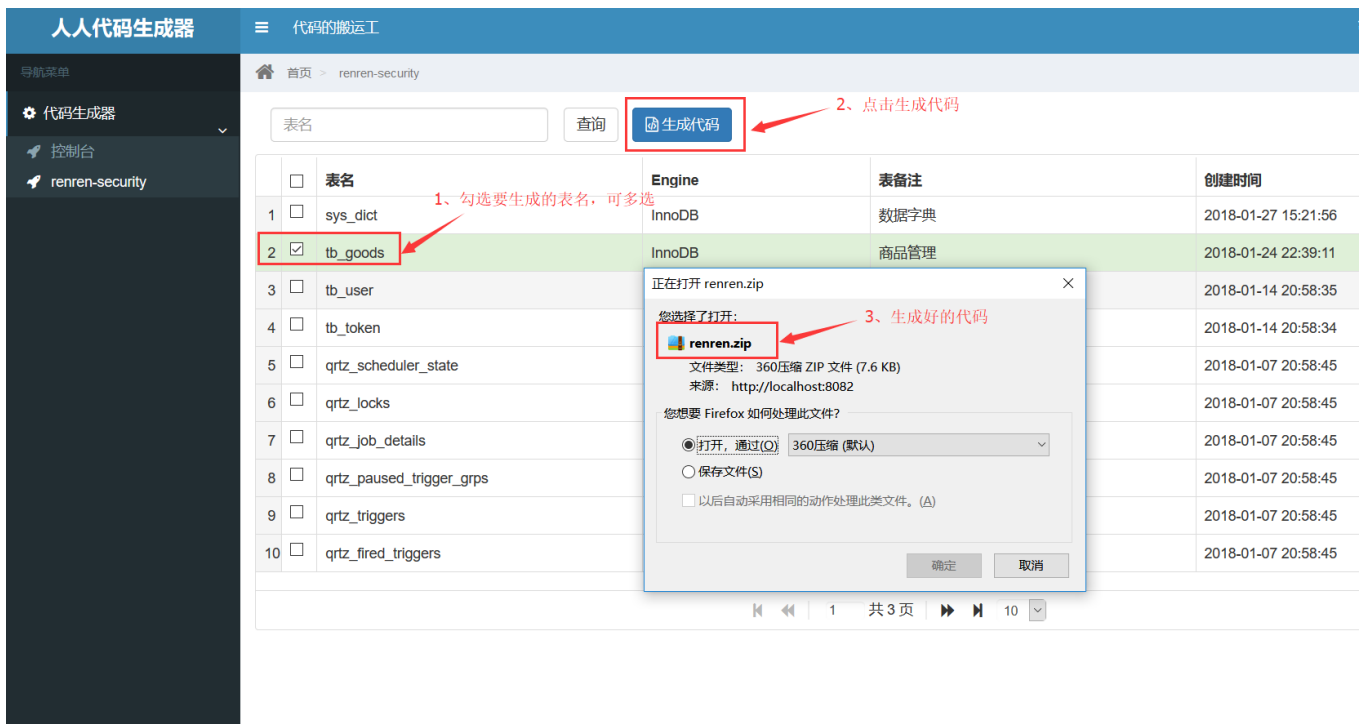
- 在数据库renren_security中，执行建表语句，创建tb_goods表，再启动renren-generator项目，运行GeneratorApplication.java的main方法即可
- 项目访问路径：<http://localhost:8082/renren-generator>
- 在浏览器里输入项目地址，如下所示：



The screenshot shows the '人人代码生成器' (Renren-Generator) web application. The interface includes a sidebar with navigation options like '代码生成器' and '控制台'. The main area displays a table of database tables with columns for '表名' (Table Name), 'Engine', '表备注' (Table Remark), and '创建时间' (Creation Time). The table lists 10 tables, including 'sys_dict', 'tb_goods', 'tb_user', 'tb_token', and several 'qrtz' tables. The 'tb_goods' table is highlighted. At the bottom, there is a pagination bar showing '1 - 10 共 27 条'.

	<input type="checkbox"/>	表名	Engine	表备注	创建时间
1	<input type="checkbox"/>	sys_dict	InnoDB	数据字典	2018-01-27 15:21:56
2	<input type="checkbox"/>	tb_goods	InnoDB	商品管理	2018-01-24 22:39:11
3	<input type="checkbox"/>	tb_user	InnoDB	用户	2018-01-14 20:58:35
4	<input type="checkbox"/>	tb_token	InnoDB	用户Token	2018-01-14 20:58:34
5	<input type="checkbox"/>	qrtz_scheduler_state	InnoDB		2018-01-07 20:58:45
6	<input type="checkbox"/>	qrtz_locks	InnoDB		2018-01-07 20:58:45
7	<input type="checkbox"/>	qrtz_job_details	InnoDB		2018-01-07 20:58:45
8	<input type="checkbox"/>	qrtz_paused_trigger_grps	InnoDB		2018-01-07 20:58:45
9	<input type="checkbox"/>	qrtz_triggers	InnoDB		2018-01-07 20:58:45
10	<input type="checkbox"/>	qrtz_fired_triggers	InnoDB		2018-01-07 20:58:45

- 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：



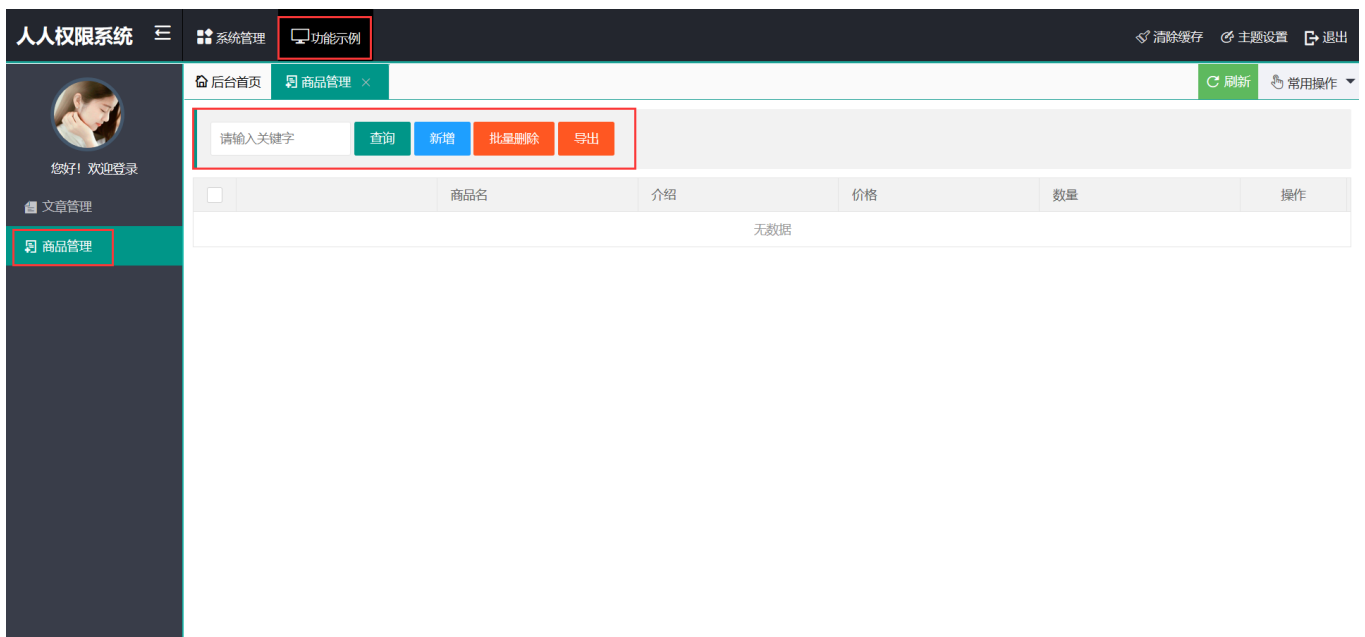
- 生成好代码后，我们只需在数据库renren_security中，执行goods_menu.sql语句，这个SQL是生成菜单的，SQL语句如下所示：

```

1.  -- 菜单SQL
2.  INSERT INTO `sys_menu` (`parent_id`, `name`, `url`, `perms`, `type`, `icon`, `order_num`)
3.      VALUES ('48', '商品管理', 'modules/demo/goods.html', NULL, '1', 'larry-10109', '6');
4.
5.  -- 按钮父菜单ID
6.  set @parentId = @@identity;
7.
8.  -- 菜单对应按钮SQL
9.  INSERT INTO `sys_menu` (`parent_id`, `name`, `url`, `perms`, `type`, `icon`, `order_num`)
10.     SELECT @parentId, '查看', null, 'demo:goods:list,demo:goods:info', '2', null, '6';
11.  INSERT INTO `sys_menu` (`parent_id`, `name`, `url`, `perms`, `type`, `icon`, `order_num`)
12.     SELECT @parentId, '新增', null, 'demo:goods:save', '2', null, '6';
13.  INSERT INTO `sys_menu` (`parent_id`, `name`, `url`, `perms`, `type`, `icon`, `order_num`)
14.     SELECT @parentId, '修改', null, 'demo:goods:update', '2', null, '6';
15.  INSERT INTO `sys_menu` (`parent_id`, `name`, `url`, `perms`, `type`, `icon`, `order_num`)
16.     SELECT @parentId, '删除', null, 'demo:goods:delete', '2', null, '6';

```

- 接下来，再把main目录覆盖renren-admin里的main目录即可，启动renren-admin项目，如下所示：



- 现在，我们就可以新增、修改、删除等操作

后台首页商品管理 ×

刷新常用操作

修改

商品名

电饭煲

介绍

高压电饭煲

价格

299

数量

10

确定

后台首页商品管理 ×

刷新常用操作

请输入关键字

查询

新增

批量删除

导出

<1>

到第

1

页

确定

共 1 条

20 条/页

3. 数据库支持

项目已支持MySQL、Oracle、SQL Server、PostgreSQL数据库，后续会支持更多常用数据库

3.1. MySQL数据库支持

系统默认支持MySQL数据库，执行db/mysql.sql，创建表及初始化数据，再启动项目即可

3.2. Oracle数据库支持

- 步骤1，引入oracle驱动，只需在公共的pom.xml里，去掉相应的jar依赖注释，如下所示：

```
1. <!-- oracle驱动 -->
2. <dependency>
3.     <groupId>com.oracle</groupId>
4.     <artifactId>ojdbc6</artifactId>
5.     <version>${oracle.version}</version>
6. </dependency>
```

- 步骤2，修改主键生成策略，配置文件在application.yml，把id-type修改成2，如下所示：

```
#mybatis
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:
    #主键类型 0:"数据库ID自增", 1:"用户输入ID", 2:"全局唯一ID (数字类型唯一ID)", 3:"全局唯一ID UUID";
    id-type: 2
    #字段策略 0:"忽略判断", 1:"非 NULL 判断"), 2:"非空判断"
    field-strategy: 2
    #驼峰下划线转换
    db-column-underline: true
    #刷新mapper 调试神器
    refresh-mapper: true
    #数据库大写下划线转换
    #capital-mode: true
    #序列接口实现类配置
    #key-generator: com.baomidou.springboot.xxx
    #逻辑删除配置
    logic-delete-value: -1
    logic-not-delete-value: 0
    #自定义填充策略接口实现
    #meta-object-handler: com.baomidou.springboot.xxx
    #自定义SQL注入器
    sql-injector: com.baomidou.mybatisplus.mapper.LogicSqlInjector
  configuration:
    map-underscore-to-camel-case: true
    cache-enabled: false
    call-setters-on-nulls: true
```

- 步骤3，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：

```
1.  spring:
2.      datasource:
3.          type: com.alibaba.druid.pool.DruidDataSource
4.          driverClassName: oracle.jdbc.OracleDriver
5.          druid:
6.              first: #数据源1
7.                  url: jdbc:oracle:thin:@192.168.10.10:1521:renren
8.                  username: renren
9.                  password: 123456
```

- 步骤4，执行db/oracle.sql，创建表及初始化数据，再启动项目即可
-

3.3. SQL Server数据库支持

- 步骤1，引入SQL Server驱动，只需在公共的pom.xml里，去掉相应的jar依赖注释，如下所示：

```
1.  <!-- mssql驱动 -->
2.  <dependency>
3.      <groupId>com.microsoft.sqlserver</groupId>
4.      <artifactId>sqljdbc4</artifactId>
5.      <version>${mssql.version}</version>
6.  </dependency>
```

- 步骤2，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：

```
1.  spring:
2.      datasource:
3.          type: com.alibaba.druid.pool.DruidDataSource
4.          driverClassName: com.microsoft.sqlserver.jdbc.SQLServerDriver
5.          druid:
6.              first: #数据源1
7.                  url:
8.                  jdbc:sqlserver://192.168.10.10:1433;DatabaseName=renren_security
9.                  username: sa
10.                 password: 123456
```

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

3.4. PostgreSQL数据库支持

- 步骤1，引入PostgreSQL驱动，只需在公共的pom.xml里，去掉相应的jar依赖注释，如下所示：

```
1.  <!-- postgresql驱动 -->
2.  <dependency>
3.      <groupId>org.postgresql</groupId>
4.      <artifactId>postgresql</artifactId>
5.  </dependency>
```

- 步骤2，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：

```
1.  spring:
2.      datasource:
3.          type: com.alibaba.druid.pool.DruidDataSource
4.          driverClassName: org.postgresql.Driver
5.          druid:
6.              first: #数据源1
7.              url:
8.                  jdbc:postgresql://192.168.10.10:5432/renren_security
9.                  username: renren
10.                 password: 123456
```

- 步骤3，修改quartz配置信息，quartz配置文件ScheduleConfig.java，打开注释，如下所示：

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

- 步骤4，执行db/postgresql.sql，创建表及初始化数据，再启动项目即可
-

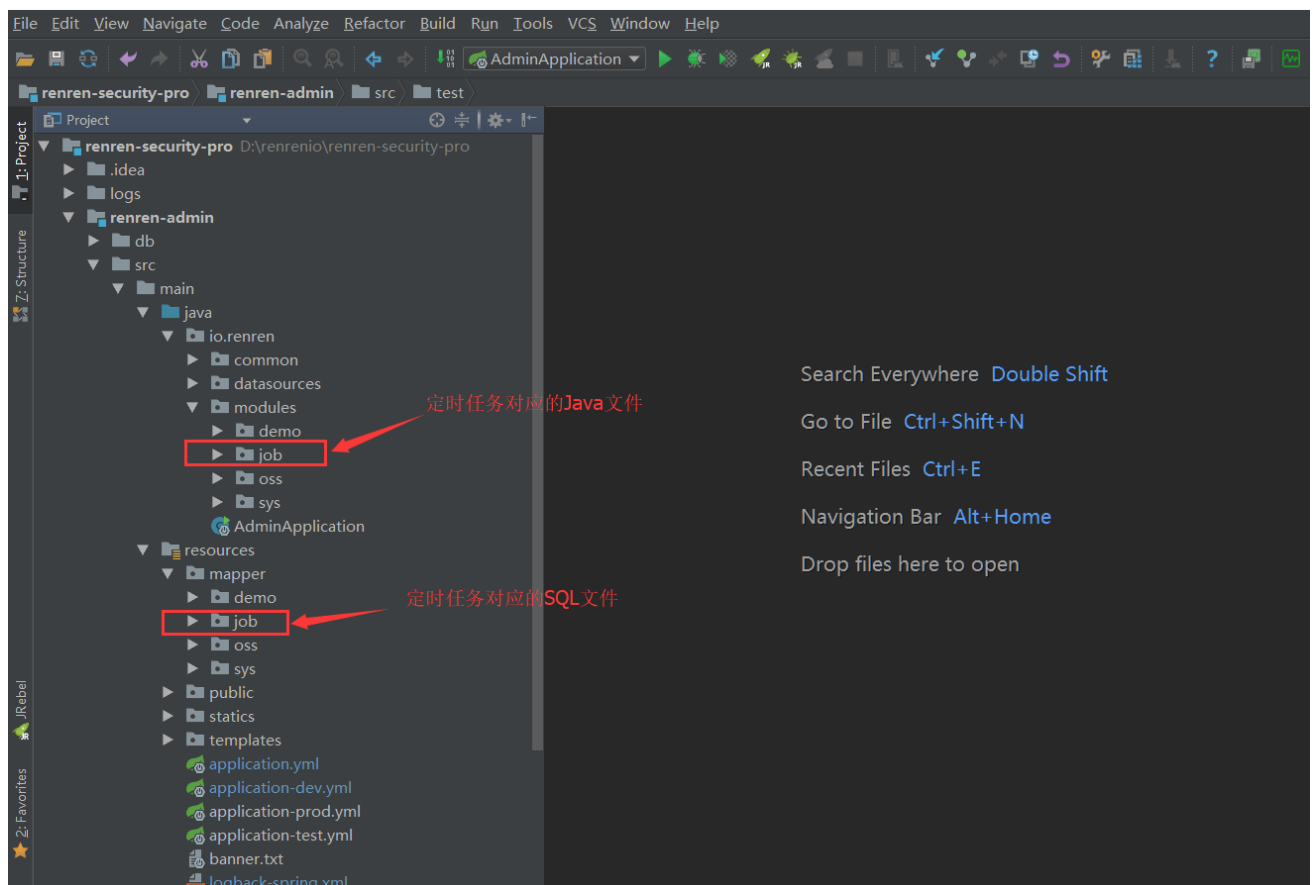
4. 后端源码分析

4.1. 功能模块移除

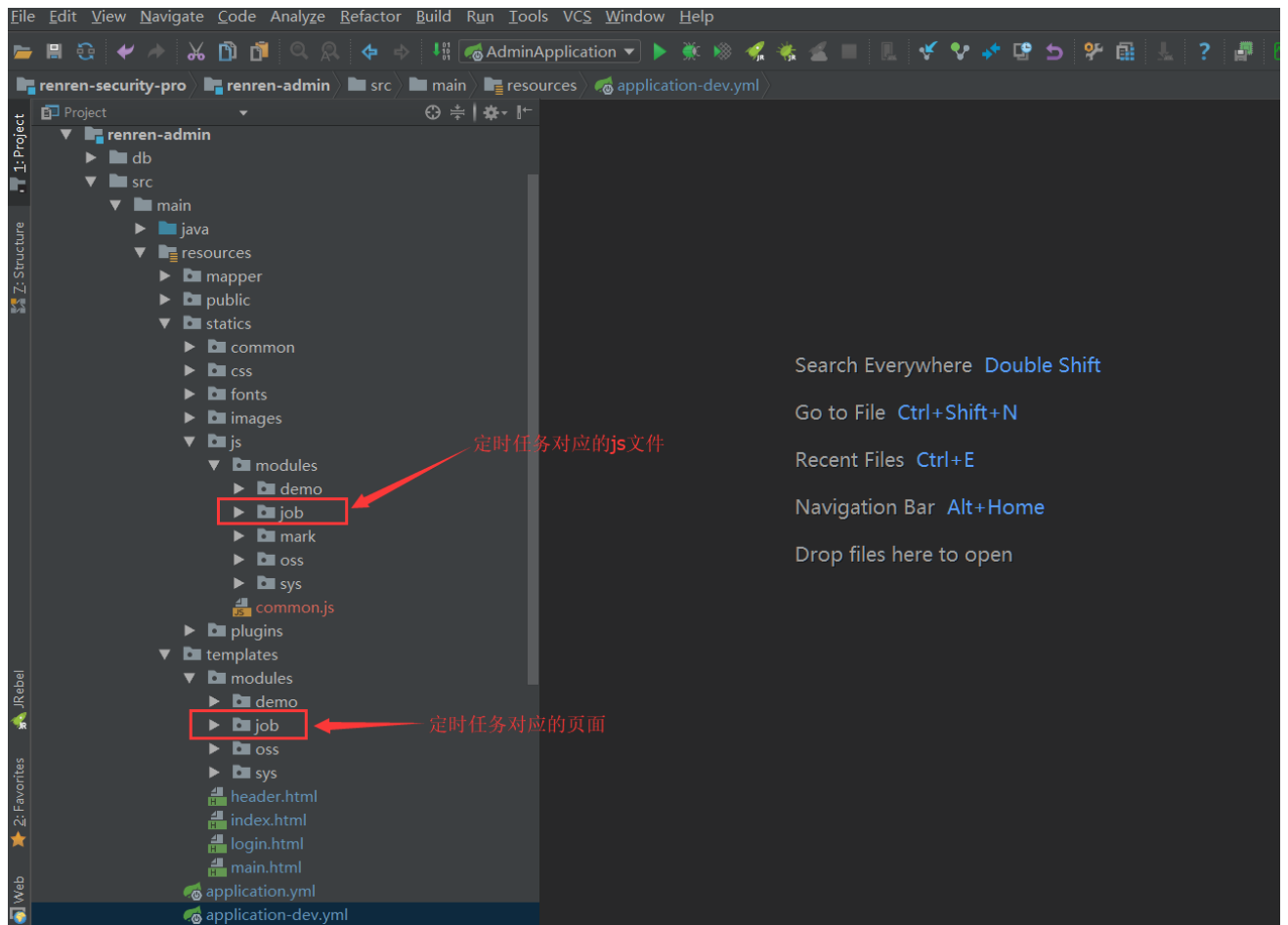
项目已包含用户管理、角色管理、部门管理、菜单管理、定时任务、参数管理、字典管理、文件上传、系统日志、文章管理、APP模块等功能，有些功能点，可能不需要，怎么才能移除掉不要的功能点呢？

项目结构是按模块划分的，功能点代码都在modules目录下面，所以，想要移除某个功能，只需删除modules目录对应的代码即可；比如，定时任务在我们的系统里，是不需要的，我们想移除掉，只需进行如下操作

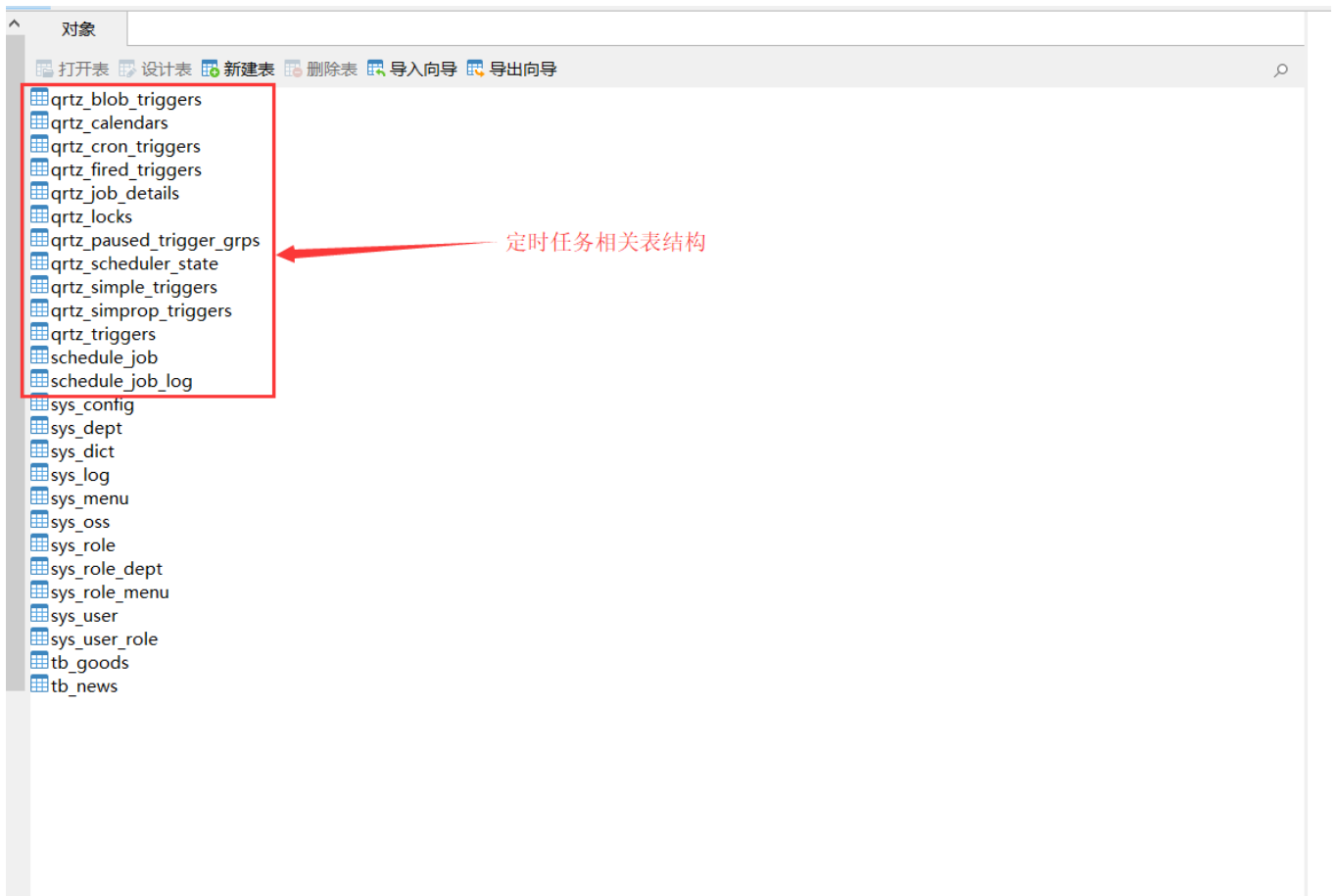
- 步骤1，删除定时任务的Java文件及SQL文件，如下图



- 步骤2，删除定时任务的js文件及页面，如下图



- 步骤3，删除对应的表结构，如下图



- 步骤4，删除对应的菜单，如下图

人人权限系统

系统管理 功能示例

后台首页 菜单管理

新增 修改 删除

步骤3

菜单ID	菜单名称	上级菜单	图标	类型	排序号	菜单URL	授权标识
1	系统管理			目录	0		
42	系统设置	系统管理		目录	2		
6	定时任务	系统设置		菜单	1	modules/job/schedul...	
7	查看	定时任务		按钮	0	sys:schedule:list,sys:schedule:info	
8	新增	定时任务		按钮	0	sys:schedule:save	
9	修改	定时任务		按钮	0	sys:schedule:update	
10	删除	定时任务		按钮	0	sys:schedule:delete	
11	暂停	定时任务		按钮	0	sys:schedule:pause	
12	恢复	定时任务		按钮	0	sys:schedule:resume	
13	立即执行	定时任务		按钮	0	sys:schedule:run	
14	日志列表	定时任务		按钮	0	sys:schedule:log	
27	参数管理	系统设置		菜单	2	modules/sys/config...	sys:config:list sys:config:info sys:config:save

步骤2，需要先删除按钮，再删除定时任务菜单

步骤1

4.2. 多数据源

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。

```
1.  #下面演示单实例，多数据库的使用情况
2.
3.  select * from db.table;
4.
5.  #其中，db为数据库名，table为数据库表名
```

4.2.1 实现多数据源

- 步骤1，在spring boot中，增加多数据源的配置，如下所示：

```
1.  spring:
2.      datasource:
3.          type: com.alibaba.druid.pool.DruidDataSource
4.          driverClassName: com.mysql.jdbc.Driver
5.          druid:
6.              first: #数据源1
7.                  url: jdbc:mysql://192.168.1.10:3306/renren_security?allowMultiQueries=true&useUnicode=true&characterEncoding=UTF-8
8.                  username: renren
9.                  password: 123456
10.             second: #数据源2
11.                 url: jdbc:mysql://192.168.1.11:3306/order?allowMultiQueries=true&useUnicode=true&characterEncoding=UTF-8
12.                 username: root
13.                 password: root
```

- 步骤2，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：

```
1.  public class DynamicDataSource extends AbstractRoutingDataSource {
2.      private static final ThreadLocal<String> contextHolder = new Thread
3.      Local<>();
```

```

4.     public DynamicDataSource (DataSource defaultTargetDataSource, Map<Ob
ject, Object> targetDataSources) {
5.         //设置默认数据源
6.         super.setDefaultTargetDataSource (defaultTargetDataSource);
7.         super.setTargetDataSources (targetDataSources);
8.         super.afterPropertiesSet ();
9.     }
10.
11.     @Override
12.     protected Object determineCurrentLookupKey () {
13.         //获取数据源, 没有指定, 则为默认数据源
14.         return getDataSource ();
15.     }
16.
17.     public static void setDataSource (String dataSource) {
18.         contextHolder.set (dataSource);
19.     }
20.
21.     public static String getDataSource () {
22.         return contextHolder.get ();
23.     }
24.
25.     public static void clearDataSource () {
26.         contextHolder.remove ();
27.     }
28.
29. }
30.
31.
32. public interface DataSourceNames {
33.     String FIRST = "first";
34.     String SECOND = "second";
35.
36. }

```

- 步骤3, 配置DataSource, 指定数据源的信息, 如下所示:

```

1.     @Configuration
2.     public class DynamicDataSourceConfig {
3.
4.         //数据源1, 读取spring.datasource.druid.first下的配置信息
5.         @Bean
6.         @ConfigurationProperties ("spring.datasource.druid.first")
7.         public DataSource firstDataSource () {

```

```

8.         return DruidDataSourceBuilder.create().build();
9.     }
10.
11.     //数据源2, 读取spring.datasource.druid.second下的配置信息
12.     @Bean
13.     @ConfigurationProperties("spring.datasource.druid.second")
14.     public DataSource secondDataSource() {
15.         return DruidDataSourceBuilder.create().build();
16.     }
17.
18.     //加了@Primary注解, 表示指定DynamicDataSource为Spring的数据源
19.     //因为DynamicDataSource是继承与AbstractRoutingDataSource, 而AbstractR
    outingDataSource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
20.     @Bean
21.     @Primary
22.     public DynamicDataSource dataSource(DataSource firstDataSource, Dat
    aSource secondDataSource) {
23.         Map<Object, Object> targetDataSources = new HashMap<>();
24.         targetDataSources.put(DataSourceNames.FIRST, firstDataSource);
25.         targetDataSources.put(DataSourceNames.SECOND, secondDataSource)
    ;
26.         return new DynamicDataSource(firstDataSource, targetDataSources
    );
27.     }

```

- 步骤4, 通过注解, 实现多数据源, 如下所示:

```

1.     /**
2.      * 多数据源注解类
3.      */
4.     @Target(ElementType.METHOD)
5.     @Retention(RetentionPolicy.RUNTIME)
6.     @Documented
7.     public @interface DataSource {
8.         String name() default "";
9.     }
10.
11.
12.     /**
13.      * 多数据源, 切面处理类
14.      */
15.     @Aspect
16.     @Component

```

```

17. public class DataSourceAspect implements Ordered {
18.     protected Logger logger = LoggerFactory.getLogger(getClass());
19.
20.     @Pointcut("@annotation(io.renren.datasources.annotation.DataSource)")
21.     public void dataSourcePointCut() {
22.
23.     }
24.
25.     @Around("dataSourcePointCut()")
26.     public Object around(ProceedingJoinPoint point) throws Throwable {
27.         MethodSignature signature = (MethodSignature) point.getSignature();
28.         Method method = signature.getMethod();
29.
30.         DataSource ds = method.getAnnotation(DataSource.class);
31.         if(ds == null){
32.             DynamicDataSource.setDataSource(DataSourceNames.FIRST);
33.             logger.debug("set datasource is " + DataSourceNames.FIRST);
34.         }else {
35.             DynamicDataSource.setDataSource(ds.name());
36.             logger.debug("set datasource is " + ds.name());
37.         }
38.
39.         try {
40.             return point.proceed();
41.         } finally {
42.             DynamicDataSource.clearDataSource();
43.             logger.debug("clean datasource");
44.         }
45.     }
46.
47.     @Override
48.     public int getOrder() {
49.         return 1;
50.     }
51. }

```

4.2.2. 测试多数据源

```

1. @RunWith(SpringRunner.class)
2. @SpringBootTest
3. public class DynamicDataSourceTest {

```

```

4.     @Autowired
5.     private DataSourceTestService dataSourceTestService;
6.
7.     @Test
8.     public void test() {
9.         //数据源1
10.        SysUserEntity user1 = dataSourceTestService.queryUser(1L);
11.        System.out.println(ToStringBuilder.reflectionToString(user1));
12.
13.        //数据源2
14.        SysUserEntity user2 = dataSourceTestService.queryUser2(1L);
15.        System.out.println(ToStringBuilder.reflectionToString(user2));
16.
17.        //数据源1
18.        SysUserEntity user3 = dataSourceTestService.queryUser(1L);
19.        System.out.println(ToStringBuilder.reflectionToString(user3));
20.    }
21.
22. }
23.
24.
25.
26. @Service
27. public class DataSourceTestService {
28.     @Autowired
29.     private SysUserService sysUserService;
30.
31.     public SysUserEntity queryUser(Long userId) {
32.         return sysUserService.selectById(userId);
33.     }
34.
35.     @DataSource(name = DataSourceNames.SECOND)
36.     public SysUserEntity queryUser2(Long userId) {
37.         return sysUserService.selectById(userId);
38.     }
39. }

```

4.2.3. 增加多数据源

```

1.     spring:
2.         datasource:
3.             type: com.alibaba.druid.pool.DruidDataSource
4.             driverClassName: com.mysql.jdbc.Driver

```

```

5.         druid:
6.             first:  #数据源1
7.                 url: jdbc:mysql://192.168.1.10:3306/renren_security?allowMultiQueries=true&useUnicode=true&characterEncoding=UTF-8
8.                 username: renren
9.                 password: 123456
10.            second:  #数据源2
11.                 url: jdbc:mysql://192.168.1.11:3306/order?allowMultiQueries=true&useUnicode=true&characterEncoding=UTF-8
12.                 username: root
13.                 password: root
14.            third:  #数据源3
15.                 url: jdbc:mysql://192.168.1.12:3306/user?allowMultiQueries=true&useUnicode=true&characterEncoding=UTF-8
16.                 username: root
17.                 password: root
18.
19.            #数据源4、5、6.....

```

```

1.  public interface DataSourceNames {
2.      String FIRST = "first";
3.      String SECOND = "second";
4.      String THIRD = "third";
5.  }
6.
7.
8.  @Configuration
9.  public class DynamicDataSourceConfig {
10.
11.      @Bean
12.      @ConfigurationProperties("spring.datasource.druid.first")
13.      public DataSource firstDataSource() {
14.          return DruidDataSourceBuilder.create().build();
15.      }
16.
17.      @Bean
18.      @ConfigurationProperties("spring.datasource.druid.second")
19.      public DataSource secondDataSource() {
20.          return DruidDataSourceBuilder.create().build();
21.      }
22.
23.      @Bean
24.      @ConfigurationProperties("spring.datasource.druid.third")
25.      public DataSource thirdDataSource() {

```

```

26.         return DruidDataSourceBuilder.create().build();
27.     }
28.
29.     @Bean
30.     @Primary
31.     public DynamicDataSource dataSource(DataSource firstDataSource, DataSource secondDataSource, DataSource thirdDataSource) {
32.         Map<Object, Object> targetDataSources = new HashMap<>();
33.         targetDataSources.put(DataSourceNames.FIRST, firstDataSource);
34.         targetDataSources.put(DataSourceNames.SECOND, secondDataSource);
35.         ;
36.         targetDataSources.put(DataSourceNames.THIRD, thirdDataSource);
37.         return new DynamicDataSource(firstDataSource, targetDataSources);
38.     }

```

4.2.4. 移除多数据源

本项目，默认是实现了多数据源的，如果自己项目不需要多数据源，也可以移除多数据源，具体操作步骤，如下所示。

步骤1，修改数据源的配置，如下所示：

```

1.  spring:
2.     datasource:
3.         type: com.alibaba.druid.pool.DruidDataSource
4.         driverClassName: com.mysql.jdbc.Driver
5.         druid:
6.             url: jdbc:mysql://localhost:3306/renren_security?allowMultiQueries=true&useUnicode=true&characterEncoding=UTF-8
7.             username: renren
8.             password: 123456

```

步骤2，删除 `io.renren.datasources` 包下的所有类，则完成了，多数据源的移除

4.3. 核心模块

4.3.1. 功能权限设计

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

```
1. @RequestMapping("/info/{userId}")
2. @RequiresPermissions("sys:user:info")
3. public R info(@PathVariable("userId") Long userId){
4.
5. }
```

3) sys_dept表，保存部门相关数据，数据权限也是根据部门进行过滤的，下一小节具体讲解

4) 在shiro配置代码里，配置为anon的，表示不经过shiro处理，配置为authc的，表示要经过shiro处理，这样就保证了，没有权限的请求，拒绝访问

```
1.  @Configuration
2.  public class ShiroConfig {
3.
4.      @Bean("sessionManager")
5.      public SessionManager sessionManager(RedisShiroSessionDAO redisShir
6.      oSessionDAO,
7.
8.          @Value("${renren.redis.open}")
9.          boolean redisOpen,
10.
11.          @Value("${renren.shiro.redis}")
12.          boolean shiroRedis){
13.
14.          DefaultWebSessionManager sessionManager = new
15.          DefaultWebSessionManager();
16.          //设置session过期时间为1小时(单位：毫秒)，默认为30分钟
17.          sessionManager.setGlobalSessionTimeout(60 * 60 * 1000);
18.          sessionManager.setSessionValidationSchedulerEnabled(true);
19.          sessionManager.setSessionIdUrlRewritingEnabled(false);
20.
21.          //如果开启redis缓存且renren.shiro.redis=true, 则shiro session存到r
22.          edis里
23.          if(redisOpen && shiroRedis){
24.              sessionManager.setSessionDAO(redisShiroSessionDAO);
25.          }
26.          return sessionManager;
27.      }
28.
29.      @Bean("securityManager")
30.      public SecurityManager securityManager(UserRealm userRealm, Session
31.      Manager sessionManager) {
32.          DefaultWebSecurityManager securityManager = new
33.          DefaultWebSecurityManager();
34.          securityManager.setRealm(userRealm);
35.          securityManager.setSessionManager(sessionManager);
36.
37.          return securityManager;
38.      }
39.
40.      @Bean("shiroFilter")
41.      public ShiroFilterFactoryBean shiroFilter(SecurityManager
```

```

securityManager) {
33.     ShiroFilterFactoryBean shiroFilter = new ShiroFilterFactoryBean
        ();
34.     shiroFilter.setSecurityManager(securityManager);
35.     shiroFilter.setLoginUrl("/login.html");
36.     shiroFilter.setUnauthorizedUrl("/");
37.
38.     Map<String, String> filterMap = new LinkedHashMap<>();
39.     filterMap.put("/swagger/**", "anon");
40.     filterMap.put("/v2/api-docs", "anon");
41.     filterMap.put("/swagger-ui.html", "anon");
42.     filterMap.put("/webjars/**", "anon");
43.     filterMap.put("/swagger-resources/**", "anon");
44.     filterMap.put("/statics/**", "anon");
45.     filterMap.put("/login.html", "anon");
46.     filterMap.put("/sys/login", "anon");
47.     filterMap.put("/favicon.ico", "anon");
48.     filterMap.put("/captcha.jpg", "anon");
49.     filterMap.put("/**", "authc");
50.     shiroFilter.setFilterChainDefinitionMap(filterMap);
51.
52.     return shiroFilter;
53. }
54.
55. @Bean("lifecycleBeanPostProcessor")
56. public LifecycleBeanPostProcessor lifecycleBeanPostProcessor() {
57.     return new LifecycleBeanPostProcessor();
58. }
59.
60. @Bean
61. public DefaultAdvisorAutoProxyCreator
defaultAdvisorAutoProxyCreator() {
62.     DefaultAdvisorAutoProxyCreator proxyCreator = new DefaultAdviso
rAutoProxyCreator();
63.     proxyCreator.setProxyTargetClass(true);
64.     return proxyCreator;
65. }
66.
67. @Bean
68. public AuthorizationAttributeSourceAdvisor
authorizationAttributeSourceAdvisor(SecurityManager securityManager) {
69.     AuthorizationAttributeSourceAdvisor advisor = new Authorization
AttributeSourceAdvisor();
70.     advisor.setSecurityManager(securityManager);
71.     return advisor;

```

```
72.     }
73. }
```

5) 上面的配置，我们可以看出来，只有访问下面这些路径，就不用登录，也就是不会经过shiro处理，其他访问路径，都会经过shiro处理，没有对应的权限，都会拒绝访问。

```
1.  Map<String, String> filterMap = new LinkedHashMap<>();
2.  filterMap.put("/swagger/**", "anon");
3.  filterMap.put("/v2/api-docs", "anon");
4.  filterMap.put("/swagger-ui.html", "anon");
5.  filterMap.put("/webjars/**", "anon");
6.  filterMap.put("/swagger-resources/**", "anon");
7.  filterMap.put("/statics/**", "anon");
8.  filterMap.put("/login.html", "anon");
9.  filterMap.put("/sys/login", "anon");
10. filterMap.put("/favicon.ico", "anon");
11. filterMap.put("/captcha.jpg", "anon");
```

6) 接下来，我们看看shiro框架，具体是怎么效验功能权限的，系统登录代码如下：

```
1.  @Controller
2.  public class SysLoginController {
3.      @Autowired
4.      private Producer producer;
5.
6.      @RequestMapping("captcha.jpg")
7.      public void captcha(HttpServletRequest response) throws IOException
8.      {
9.          response.setHeader("Cache-Control", "no-store, no-cache");
10.         response.setContentType("image/jpeg");
11.
12.         //生成文字验证码
13.         String text = producer.createText();
14.         //生成图片验证码
15.         BufferedImage image = producer.createImage(text);
16.         //保存到shiro session
17.         ShiroUtils.setSessionAttribute(Constants.KAPTCHA_SESSION_KEY, text);
18.
19.         ServletOutputStream out = response.getOutputStream();
20.         ImageIO.write(image, "jpg", out);
21.     }
```

```

21.
22.     /**
23.      * 登录
24.      */
25.     @ResponseBody
26.     @RequestMapping(value = "/sys/login", method = RequestMethod.POST)
27.     public R login(String username, String password, String captcha) {
28.         String kaptcha =
ShiroUtils.getKaptcha(Constants.KAPTCHA_SESSION_KEY);
29.         if(!captcha.equalsIgnoreCase(kaptcha)){
30.             return R.error("验证码不正确");
31.         }
32.
33.         try{
34.             Subject subject = ShiroUtils.getSubject();
35.             UsernamePasswordToken token = new UsernamePasswordToken(use
rname, password);
36.             subject.login(token);
37.         }catch (UnknownAccountException e) {
38.             return R.error(e.getMessage());
39.         }catch (IncorrectCredentialsException e) {
40.             return R.error("账号或密码不正确");
41.         }catch (LockedAccountException e) {
42.             return R.error("账号已被锁定,请联系管理员");
43.         }catch (AuthenticationException e) {
44.             return R.error("账户验证失败");
45.         }
46.
47.         return R.ok();
48.     }
49.
50.     /**
51.      * 退出
52.      */
53.     @RequestMapping(value = "logout", method = RequestMethod.GET)
54.     public String logout() {
55.         ShiroUtils.logout();
56.         return "redirect:login.html";
57.     }
58.
59. }

```

登录的时候，调用了 `subject.login(token)` 方法，分析源码，我们可以知道，会调用自定义 Realm 中的 `doGetAuthenticationInfo` 方法，如下所示：

```
1.  @Component
2.  public class UserRealm extends AuthorizingRealm {
3.      @Autowired
4.      private SysUserDao sysUserDao;
5.      @Autowired
6.      private SysMenuDao sysMenuDao;
7.
8.      /**
9.       * 认证(登录时调用)
10.     */
11.     @Override
12.     protected AuthenticationInfo doGetAuthenticationInfo(
13.         AuthenticationToken authcToken) throws
14.     AuthenticationException {
15.         UsernamePasswordToken token = (UsernamePasswordToken) authcToken
16.         ;
17.
18.         //查询用户信息
19.         SysUserEntity user = new SysUserEntity();
20.         user.setUsername(token.getUsername());
21.         user = sysUserDao.selectOne(user);
22.
23.         //账号不存在
24.         if(user == null) {
25.             throw new UnknownAccountException("账号或密码不正确");
26.         }
27.
28.         //账号锁定
29.         if(user.getStatus() == 0){
30.             throw new LockedAccountException("账号已被锁定,请联系管理员");
31.         }
32.
33.         SimpleAuthenticationInfo info = new SimpleAuthenticationInfo(user,
34.             user.getPassword(), ByteSource.Util.bytes(user.getSalt()), getName
35.             ());
36.         return info;
37.     }
38.
39.     @Override
40.     public void setCredentialsMatcher(CredentialsMatcher
41.     credentialsMatcher) {
42.         HashedCredentialsMatcher shaCredentialsMatcher = new HashedCred
43.         entialsMatcher();
44.         shaCredentialsMatcher.setHashAlgorithmName(ShiroUtils.hashAlgor
45.         ithmName);
```

```

39.         shaCredentialsMatcher.setHashIterations(ShiroUtils.hashIterations);
40.         super.setCredentialsMatcher(shaCredentialsMatcher);
41.     }
42. }

```

这个 `doGetAuthenticationInfo` 方法里，会通过用户名，查询用户信息，如果用户不存在，则直接抛 `UnknownAccountException` 异常；如果用户被禁用，则抛出 `LockedAccountException` 异常；如果用户存在，则封装成 `SimpleAuthenticationInfo` 对象，并返回，Shiro就会帮我们验证密码是否正确，如果密码错误，则会抛出 `IncorrectCredentialsException` 异常；登录的时候，我们会对这些异常进行捕获，并以相应的错误消息，提示前端，如下所示：

```

1.     try{
2.         Subject subject = ShiroUtils.getSubject();
3.         UsernamePasswordToken token = new UsernamePasswordToken(username, password);
4.         subject.login(token);
5.     }catch (UnknownAccountException e) {
6.         return R.error(e.getMessage());
7.     }catch (IncorrectCredentialsException e) {
8.         return R.error("账号或密码不正确");
9.     }catch (LockedAccountException e) {
10.        return R.error("账号已被锁定,请联系管理员");
11.    }catch (AuthenticationException e) {
12.        return R.error("账户验证失败");
13.    }

```

其中，下面的这段代码，可能不明白是什么意思，这个就告诉shiro，验证登录密码的时候，登录密码需要进

行 `new SimpleHash(hashAlgorithmName, password, salt, hashIterations).toString()` 处理后，再与数据库密码进行比较

```

1.     SimpleAuthenticationInfo info = new SimpleAuthenticationInfo(user, user.getPassword(), ByteSource.Util.bytes(user.getSalt()), getName());
2.
3.     @Override
4.     public void setCredentialsMatcher(CredentialsMatcher credentialsMatcher) {

```

```

5.     HashedCredentialsMatcher shaCredentialsMatcher = new HashedCredenti
alsMatcher();
6.     //指定sha算法为sha-256, 需要和生成密码时的一样
7.
shaCredentialsMatcher.setHashAlgorithmName(ShiroUtils.hashAlgorithmName
);
8.     //hash迭代次数, 需要和生成密码时的一样
9.
shaCredentialsMatcher.setHashIterations(ShiroUtils.hashIterations);
10.    super.setCredentialsMatcher(shaCredentialsMatcher);
11.    }

```

其中, salt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。

我们在注册账号的时候, 数据库不是存储的明文密码, 而

是 `ShiroUtils.sha256(user.getPassword(), user.getSalt())` 处理过后的密码, 如下所示:

```

1.    public void save(SysUserEntity user) {
2.        //sha256加密
3.        String salt = RandomStringUtils.randomAlphanumeric(20);
4.        user.setSalt(salt);
5.        user.setPassword(ShiroUtils.sha256(user.getPassword(), user.getSalt
()));
6.        this.insert(user);
7.    }
8.
9.    public class ShiroUtils {
10.        /** 加密算法 */
11.        public final static String hashAlgorithmName = "SHA-256";
12.        /** 循环次数 */
13.        public final static int hashIterations = 16;
14.
15.        public static String sha256(String password, String salt) {
16.            return new SimpleHash(hashAlgorithmName, password, salt, hashIt
erations).toString();
17.        }
18.    }

```

7) 登录验证通过后, 是不是就可以调用所有接口了呢, 如下所示, `/sys/user/list` 接口


```

1.  @RestController
2.  @RequestMapping("/sys/user")
3.  public class SysUserController extends AbstractController {
4.      @Autowired
5.      private SysUserService sysUserService;
6.      @Autowired
7.      private SysUserRoleService sysUserRoleService;
8.
9.      /**
10.       * 所有用户列表
11.       */
12.      @RequestMapping("/list")
13.      @RequiresPermissions("sys:user:list")
14.      public R list(@RequestParam Map<String, Object> params) {
15.          PageUtils page = sysUserService.queryPage(params);
16.
17.          return R.ok().put("page", page);
18.      }
19.  }

```

当然不能，调用有 `@RequiresPermissions("sys:user:list")` 注解的接口时，Shiro会验证当前登录用户，是否有 `sys:user:list` 权限；Shiro验证是否有权限时，会调用自定义Realm的 `doGetAuthorizationInfo` 方法，如下所示：

```

1.  @Component
2.  public class UserRealm extends AuthorizingRealm {
3.      @Autowired
4.      private SysUserDao sysUserDao;
5.      @Autowired
6.      private SysMenuDao sysMenuDao;
7.
8.      /**
9.       * 授权 (验证权限时调用)
10.       */
11.      @Override
12.      protected AuthorizationInfo
13.      doGetAuthorizationInfo(PrincipalCollection principals) {
14.          SysUserEntity user =
15.          (SysUserEntity) principals.getPrimaryPrincipal();
16.          Long userId = user.getUserId();
17.
18.          List<String> permsList;

```

```

18.         //系统管理员, 拥有最高权限
19.         if(userId == Constant.SUPER_ADMIN) {
20.             List<SysMenuEntity> menuList = sysMenuDao.selectList(null);
21.             permsList = new ArrayList<>(menuList.size());
22.             for(SysMenuEntity menu : menuList){
23.                 permsList.add(menu.getPerms());
24.             }
25.         }else{
26.             permsList = sysUserDao.queryAllPerms(userId);
27.         }
28.
29.         //用户权限列表
30.         Set<String> permsSet = new HashSet<>();
31.         for(String perms : permsList){
32.             if(StringUtils.isBlank(perms)){
33.                 continue;
34.             }
35.             permsSet.addAll(Arrays.asList(perms.trim().split(",")));
36.         }
37.
38.         SimpleAuthorizationInfo info = new SimpleAuthorizationInfo();
39.         info.setStringPermissions(permsSet);
40.         return info;
41.     }
42. }

```

上面的代码，会判断用户是否有超级管理员，如果是超级管理员，则拥有所有菜单的权限；如果不是超级管理员，则查询用户所拥有的菜单权限。

上面的 `doGetAuthorizationInfo` 方法，我们可以发现，如果调用下面的接口，是需要 `mall.order:update` 权限的，菜单里如果没有配置这个权限，拥有该菜单，也是没权限访问这个接口的。

```

1.     @RequestMapping("/update")
2.     @RequiresPermissions("mall.order:update")
3.     public R update(@RequestParam Map<String, Object> params){
4.         //省略若干代码....
5.         return R.ok();
6.     }

```

4.3.2. 数据权限设计

本系统采用注解的方式，实现了数据权限的功能，代码更加灵活简洁。在需要数据权限的service方法上，添加 `@DataFilter` 注解，就可以达到数据过滤的功能，也就是我们常说的数据权限。该实现方式，适应绝大多数企业后台管理系统，对数据权限的要求。

- 数据权限是通过 `@DataFilter` 注解实现的，代码如下所示

```
1.  @Target(ElementType.METHOD)
2.  @Retention(RetentionPolicy.RUNTIME)
3.  @Documented
4.  public @interface DataFilter {
5.      /** 表的别名 */
6.      String tableAlias() default "";
7.
8.      /** true：没有本部门数据权限，也能查询本人数据 */
9.      boolean user() default true;
10.
11.     /** true：拥有子部门数据权限 */
12.     boolean subDept() default false;
13.
14.     /** 部门ID */
15.     String deptId() default "dept_id";
16.
17.     /** 用户ID */
18.     String userId() default "user_id";
19. }
```

上面的注解类，定义了tableAlias、user、subDept、deptId、userId及每个方法的作用，这个仅仅是定义，如需赋予功能，还需要编写具体的实现代码。

- 上面定义好了注解，我们再来看下具体的实现，也就是 `@DataFilter` 注解的切面实现类 `DataFilterAspect`，如下所示：

```
1.  @Aspect
2.  @Component
3.  public class DataFilterAspect {
4.      @Autowired
5.      private SysDeptService sysDeptService;
6.      @Autowired
7.      private SysUserRoleService sysUserRoleService;
```

```

8.      @Autowired
9.      private SysRoleDeptService sysRoleDeptService;
10.
11.      @Pointcut("@annotation(io.renren.common.annotation.DataFilter)")
12.      public void dataFilterCut() {
13.
14.      }
15.
16.      @Before("dataFilterCut()")
17.      public void dataFilter(JoinPoint point) throws Throwable {
18.          Object params = point.getArgs()[0];
19.          if(params != null && params instanceof Map){
20.              SysUserEntity user = ShiroUtils.getUserEntity();
21.
22.              //如果不是超级管理员，则进行数据过滤
23.              if(user.getUserId() != Constant.SUPER_ADMIN){
24.                  Map map = (Map)params;
25.                  map.put(Constant.SQL_FILTER, getSQLFilter(user, point))
;
26.              }
27.
28.              return ;
29.          }
30.
31.          throw new RRException("数据权限接口，只能是Map类型参数，且不能为NULL"
);
32.      }
33.
34.      /**
35.       * 获取数据过滤的SQL
36.       */
37.      private String getSQLFilter(SysUserEntity user, JoinPoint point){
38.          MethodSignature signature = (MethodSignature) point.getSignatur
e();
39.          DataFilter dataFilter = signature.getMethod().getAnnotation(Data
aFilter.class);
40.          //获取表的别名
41.          String tableAlias = dataFilter.tableAlias();
42.          if(StringUtils.isNotBlank(tableAlias)){
43.              tableAlias += ".";
44.          }
45.
46.          //部门ID列表
47.          Set<Long> deptIdList = new HashSet<>();
48.

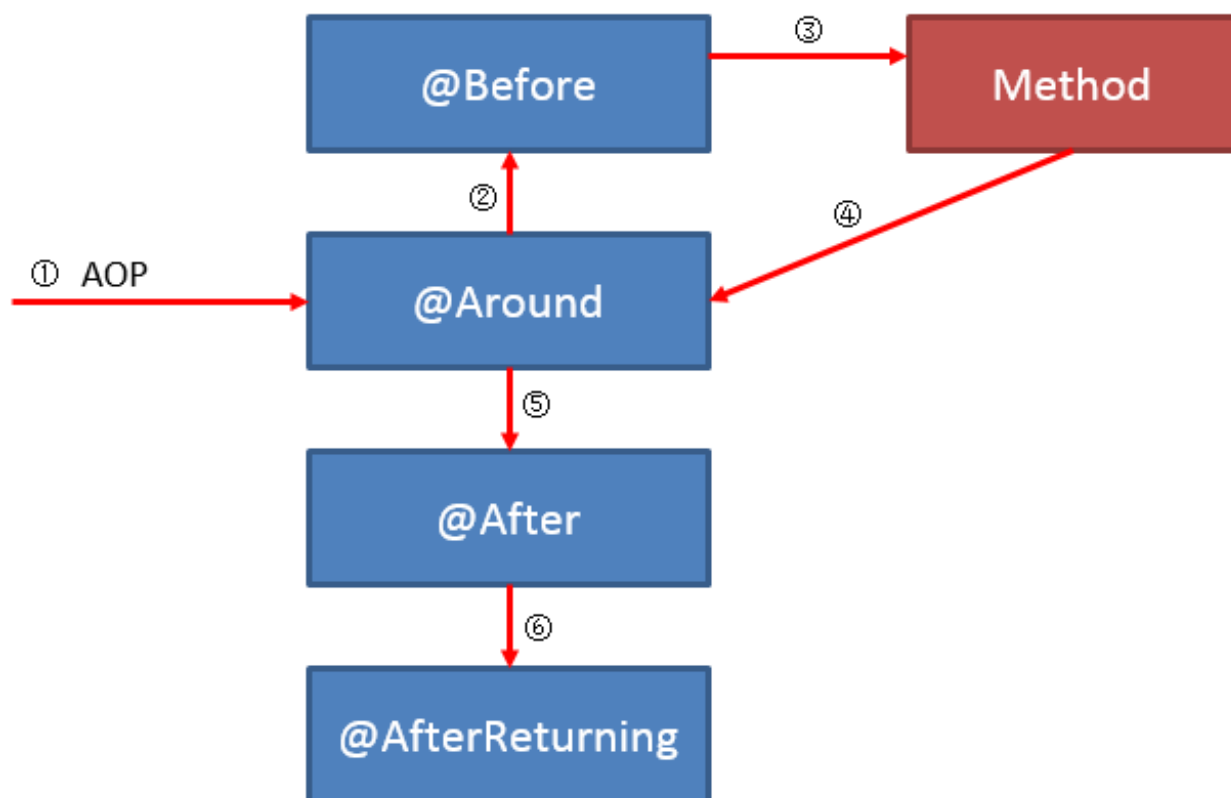
```

```

49.         //用户角色对应的部门ID列表
50.         List<Long> roleIdList = sysUserRoleService.queryRoleIdList(user
        .getUserId());
51.         if(roleIdList.size() > 0){
52.             List<Long> userDeptIdList =
        sysRoleDeptService.queryDeptIdList(roleIdList.toArray(new
        Long[roleIdList.size()]));
53.             deptIdList.addAll(userDeptIdList);
54.         }
55.
56.         //用户子部门ID列表
57.         if(dataFilter.subDept()){
58.             List<Long> subDeptIdList = sysDeptService.getSubDeptIdList(
        user.getDeptId());
59.             deptIdList.addAll(subDeptIdList);
60.         }
61.
62.         StringBuilder sqlFilter = new StringBuilder();
63.         sqlFilter.append(" ");
64.
65.         if(deptIdList.size() > 0){
66.             sqlFilter.append(tableAlias).append(dataFilter.deptId()).ap
        pend(" in(").append(StringUtils.join(deptIdList, ",")).append(")");
67.         }
68.
69.         //没有本部门数据权限，也能查询本人数据
70.         if(dataFilter.user()){
71.             if(deptIdList.size() > 0){
72.                 sqlFilter.append(" or ");
73.             }
74.             sqlFilter.append(tableAlias).append(dataFilter.userId()).ap
        pend("=").append(user.getUserId());
75.         }
76.
77.         sqlFilter.append(")");
78.
79.         return sqlFilter.toString();
80.     }
81. }

```

- 上面的实现类中，定义了一个切入点，只要方法上加了 `@DataFilter` 注解，执行该方法之前，会进入 `dataFilter()` 方法，因为使用了Spring AOP的 `@Before`，Spring AOP将按照如下图的顺序执行



- 接下来，我们来具体分析里面的代码，下面的这个方法规定了，第一个参数如果不是Map类型，就直接抛出异常，提示不能使用 `@DataFilter` 注解，如要使用，请使用Map类型的参数，且不能为NULL。

参数没问题了，就会判断是不是超级管理员，如果是超级管理员，则直接返回，不再进行下一步操作，也就是说，超级管理员拥有最高权限，能查看所有数据。如果不是超级管理员，则进行数据过滤。

```
1. public void dataFilter(JoinPoint point) throws Throwable {
2.     Object params = point.getArgs()[0];
3.     if(params != null && params instanceof Map){
4.         SysUserEntity user = ShiroUtils.getUserEntity();
5.
6.         //如果不是超级管理员，则进行数据过滤
7.         if(user.getUserId() != Constant.SUPER_ADMIN){
8.             Map map = (Map)params;
9.             map.put(Constant.SQL_FILTER, getSQLFilter(user, point));
10.        }
11.
12.        return ;
13.    }
14.
15.    throw new RRException("数据权限接口，只能是Map类型参数，且不能为NULL");
```

```
16.     }
```

- 上面的代码中，可以看到有这么一

行 `map.put(Constant.SQL_FILTER, getSQLFilter(user, point))`，这行的意思，就是把过滤的SQL条件，追加到map参数里，map参数对应的key为 `Constant.SQL_FILTER`，下面是常量定义的代码：

```
1.  public class Constant {
2.      /** 超级管理员ID */
3.      public static final int SUPER_ADMIN = 1;
4.      /** 数据权限过滤 */
5.      public static final String SQL_FILTER = "sql_filter";
6.  }
```

- 接下来，看看具体是怎么生成过滤的SQL条件，代码片段如下

```
1.  private String getSQLFilter(SysUserEntity user, JoinPoint point){
2.      MethodSignature signature = (MethodSignature) point.getSignature();
3.      DataFilter dataFilter = signature.getMethod().getAnnotation(DataFilter.class);
4.      //获取表的别名
5.      String tableAlias = dataFilter.tableAlias();
6.      if(StringUtils.isNotBlank(tableAlias)){
7.          tableAlias += ".";
8.      }
9.
10.     //部门ID列表
11.     Set<Long> deptIdList = new HashSet<>();
12.
13.     //用户角色对应的部门ID列表
14.     List<Long> roleIdList = sysUserRoleService.queryRoleIdList(user.getUserId());
15.     if(roleIdList.size() > 0){
16.         List<Long> userDeptIdList = sysRoleDeptService.queryDeptIdList(
17.             roleIdList.toArray(new Long[roleIdList.size()]));
18.         deptIdList.addAll(userDeptIdList);
19.     }
20.
21.     //用户子部门ID列表
22.     if(dataFilter.subDept()){
23.         List<Long> subDeptIdList = sysDeptService.getSubDeptIdList(user.getDeptId());
24.     }
```

```

23.         deptIdList.addAll(subDeptIdList);
24.     }
25.
26.     StringBuilder sqlFilter = new StringBuilder();
27.     sqlFilter.append(" ");
28.
29.     if(deptIdList.size() > 0){
30.         sqlFilter.append(tableAlias).append(dataFilter.deptId()).append
31.         (" in(").append(StringUtils.join(deptIdList, ",")).append(")");
32.     }
33.     //没有本部门数据权限，也能查询本人数据
34.     if(dataFilter.user()){
35.         if(deptIdList.size() > 0){
36.             sqlFilter.append(" or ");
37.         }
38.         sqlFilter.append(tableAlias).append(dataFilter.userId()).append
39.         ("=").append(user.getUserId());
40.     }
41.     sqlFilter.append(")");
42.
43.     return sqlFilter.toString();
44. }

```

1) 先判断有没有表的别名，如果有别名，就在别名后追加 `.`，多表关联查询及过滤数据时，会使用到别名，如下所示：

```

1.     select * from tb_sales_report t1, tb_product t2 where t1.product_id=t2.
2.     id
3.     #下面就是过滤的SQL条件，假设sql_filter对应的SQL条件为：t1.dept_id in(1,2)
4.     and ${sql_filter}

```

2) 接下来，就是获取用户的部门，及用户角色对应的部门

```

1.     //用户部门ID列表
2.     Set<Long> deptIdList = new HashSet<>();
3.
4.     //用户角色对应的部门ID列表
5.     List<Long> roleIdList =
        sysUserRoleService.queryRoleIdList(user.getUserId());

```



```

6.     if(roleIdList.size() > 0){
7.         List<Long> userDeptIdList = sysRoleDeptService.queryDeptIdList(role
            IdList.toArray(new Long[roleIdList.size()]));
8.         deptIdList.addAll(userDeptIdList);
9.     }

```

3) 如果subDept=true，则把用户所在部门的所有子部门查询出来

```

1.     //用户子部门ID列表
2.     if(dataFilter.subDept()){
3.         List<Long> subDeptIdList = sysDeptService.getSubDeptIdList(user.get
            DeptId());
4.         deptIdList.addAll(subDeptIdList);
5.     }

```

4) 如下代码片段，就是把数据过滤条件组装成dept_id in(1,2,3)

```

1.     StringBuilder sqlFilter = new StringBuilder();
2.     sqlFilter.append(" (");
3.
4.     if(deptIdList.size() > 0){
5.         sqlFilter.append(tableAlias).append(dataFilter.deptId()).append(" i
            n(").append(StringUtils.join(deptIdList, ",")).append(")");
6.     }

```

5) 如下代码片段，就是要实现用户换部门了，不能查看之前部门数据，但还能查看自己的数据

```

1.     if(dataFilter.user()){
2.         if(deptIdList.size() > 0){
3.             sqlFilter.append(" or ");
4.         }
5.         sqlFilter.append(tableAlias).append(dataFilter.userId()).append("="
            ).append(user.getUserId());
6.     }
7.     sqlFilter.append(")");

```

6) 上面的代码，就生成了数据过滤的SQL片段，我们只需把这个过滤SQL片段，追加到查询语句里，就可以了，如下所示：

```

1.  @DataFilter(subDept = true, user = false)
2.  public LayuiPage queryPage(Map<String, Object> params) {
3.      String username = (String)params.get("username");
4.
5.      Page<SysUserEntity> page = this.selectPage(
6.          new Query<SysUserEntity>(params).getPage(),
7.          new EntityWrapper<SysUserEntity>()
8.              .like(StringUtils.isNotBlank(username), "username", username)
9.              .addFilterIfNeed(params.get(Constant.SQL_FILTER) != null, (
10.                  String)params.get(Constant.SQL_FILTER))
11.          );
12.      return new LayuiPage(page.getRecords(), page.getTotal());
13.  }

```

7) 上面的代码，先判断Map里的 `Constant.SQL_FILTER` 是否为null，不为空则追加进去，这样就可以实现数据过滤了，也就是我们常说的数据权限

```

1.  .addFilterIfNeed(params.get(Constant.SQL_FILTER) != null, (String)params.get(Constant.SQL_FILTER))

```

4.3.3. 数据权限使用案例

如销售系统，销售经理，可以查看本部门所有销售情况，销售组长，可以查看本小组成员的销售情况，而普通销售，则只能查看自己的销售情况，要实现这样的数据权限，下面就是具体的实现步骤。

- 先创建好销售报表，如下所示：

```

1.  CREATE TABLE `tb_sales_report` (
2.      `id` bigint(20) NOT NULL AUTO_INCREMENT,
3.      `product_id` bigint(20) DEFAULT NULL COMMENT '产品ID',
4.      `product_name` varchar(50) DEFAULT NULL COMMENT '产品名称',
5.      `sale_id` bigint(20) DEFAULT NULL COMMENT '销售ID',
6.      `sale_name` varchar(20) DEFAULT NULL COMMENT '销售',
7.      `dept_id` bigint(20) DEFAULT NULL COMMENT '销售人员所属部门ID',
8.      `dept_name` varchar(20) DEFAULT NULL COMMENT '部门',
9.      `create_date` datetime DEFAULT NULL COMMENT '创建时间',

```

```
10.     PRIMARY KEY (`id`)
11. ) ENGINE=InnoDB CHARSET=utf8 COMMENT='销售报表';
```

- 数据权限是通过dept_id、user_id进行数据过滤的，所以销售表里，需要有这2个字段，不然数据权限就无法实现，当然，这2个字段名是可以修改的，下面就是把user_id修改成了sale_id

```
1.     @DataFilter(userId = "sale_id")
```

- 下面就是实现数据权限具体的代码，如下所示：

```
1.     @DataFilter(userId = "sale_id")
2.     public LayuiPage queryPage(Map<String, Object> params) {
3.         Page<SalesReportEntity> page = this.selectPage(
4.             new Query<SalesReportEntity>(params).getPage(),
5.             new EntityWrapper<SalesReportEntity>()
6.                 .addFilterIfNeed(params.get(Constant.SQL_FILTER) != null,
7.                     (String)params.get(Constant.SQL_FILTER))
8.         );
9.
10.         return new LayuiPage(page.getRecords(), page.getTotal());
11.     }
```

- 因为mybatis-plus不支持多表关联，如需多表关联，实现数据权限，则只能通过原生的MyBatis实现，如下所示：

```
1.     <select id="queryList"
2.         resultType="io.renren.modules.demo.entity.SalesReportEntity">
3.         select * from tb_sales_report t1, tb_product t2 where
4.         t1.product_id = t2.id
5.         <if test="sql_filter != null">
6.             and ${sql_filter}
7.         </if>
8.     </select>
```

- 完成以上代码，代码部分就完成了，接下来就只要配置角色的数据权限了，销售经理所属角色，拥有部门及小组的数据权限，如下：

后台首页

角色管理

刷新

常用操作

新增

角色名称

销售经理角色

所属部门

销售部

备注

销售经理

功能权限

☐ 系统管理

☒ 功能示例

☐ 文章管理

☒ 销售报表

数据权限

☐ 人人开源集团

☐ 长沙分公司

☐ 上海分公司

☒ 销售部

☒ 销售一组

☒ 销售二组

确定

- 销售一组组长所属角色，拥有销售一组的数据权限，如下：

后台首页

角色管理

刷新

常用操作

新增

角色名称

销售一组组长

所属部门

销售一组

备注

销售一组

功能权限

☐ 系统管理

☒ 功能示例

☐ 文章管理

☒ 销售报表

数据权限

☐ 人人开源集团

☐ 长沙分公司

☐ 上海分公司

☐ 销售部

☒ 销售一组

☐ 销售二组

确定

- 销售二组组长所属角色，拥有销售二组的数据权限，如下：

后台首页

角色管理

刷新

常用操作

新增

角色名称

销售二组组长

所属部门

销售二组

备注

销售二组

功能权限

☐

系统管理

☒

功能示例

☐

文章管理

☒

销售报表

数据权限

☐

人人开源集团

☐

长沙分公司

☐

上海分公司

☐

技术部

☐

销售部

☐

销售一组

☒

销售二组

确定

- 普通销售所属角色，只能参看本人数据权限，如下：

后台首页

角色管理

刷新

常用操作

新增

角色名称

普通销售角色

所属部门

销售一组

备注

普通销售

功能权限

☐

系统管理

☒

功能示例

☐

文章管理

☒

销售报表

数据权限

☐

人人开源集团

☐

长沙分公司

☐

上海分公司

☐

技术部

☐

销售部

☐

销售一组

☐

销售二组

确定

- 添加些测试数据，我们来看看数据权限，是否符合我们的需求

1) 我们用销售经理登录系统，可以查看所有数据，如下所示：

后台首页

销售报表

刷新

常用操作

请输入关键字

查询

新增

批量删除

导出

		产品ID	产品名称	销售	部门	创建时间	操作
<input type="checkbox"/>	1	1	MacBook	销售经理	销售部	2018-04-02 03:59:12	<div>编辑</div> <div>删除</div>
<input type="checkbox"/>	2	1	MacBook	销售一组组长	销售一组	2018-04-02 04:40:27	<div>编辑</div> <div>删除</div>
<input type="checkbox"/>	3	1	MacBook	销售二组组长	销售二组	2018-04-02 04:42:19	<div>编辑</div> <div>删除</div>
<input type="checkbox"/>	4	1	MacBook	普通销售(销售一组)	销售一组	2018-04-02 04:43:13	<div>编辑</div> <div>删除</div>

< 1 >

到第 1 页

确定

共 4 条

20 条/页

2) 我们用销售一组组长登录系统，可以查看销售一组数据，如下所示：

后台首页

销售报表

刷新

常用操作

请输入关键字

查询

新增

批量删除

导出

		产品ID	产品名称	销售	部门	创建时间	操作
<input type="checkbox"/>	2	1	MacBook	销售一组组长	销售一组	2018-04-02 04:40:27	<div>编辑</div> <div>删除</div>
<input type="checkbox"/>	4	1	MacBook	普通销售(销售一组)	销售一组	2018-04-02 04:43:13	<div>编辑</div> <div>删除</div>

< 1 >

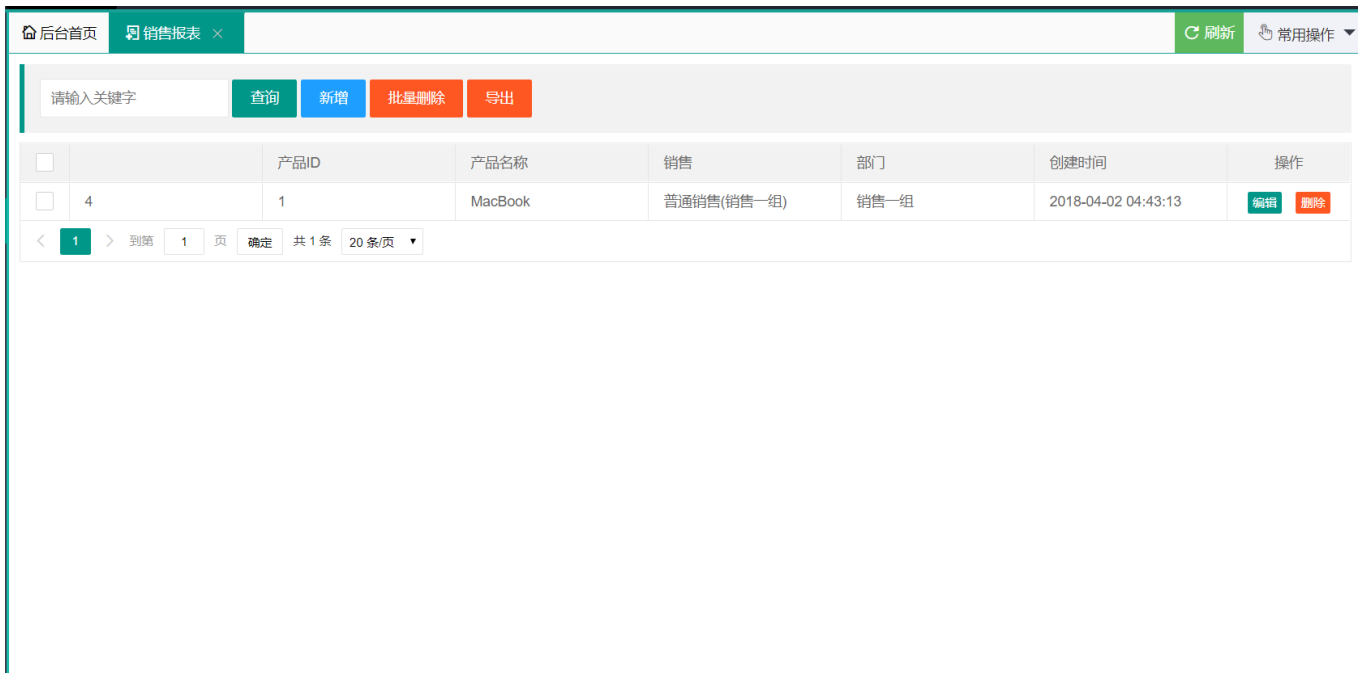
到第 1 页

确定

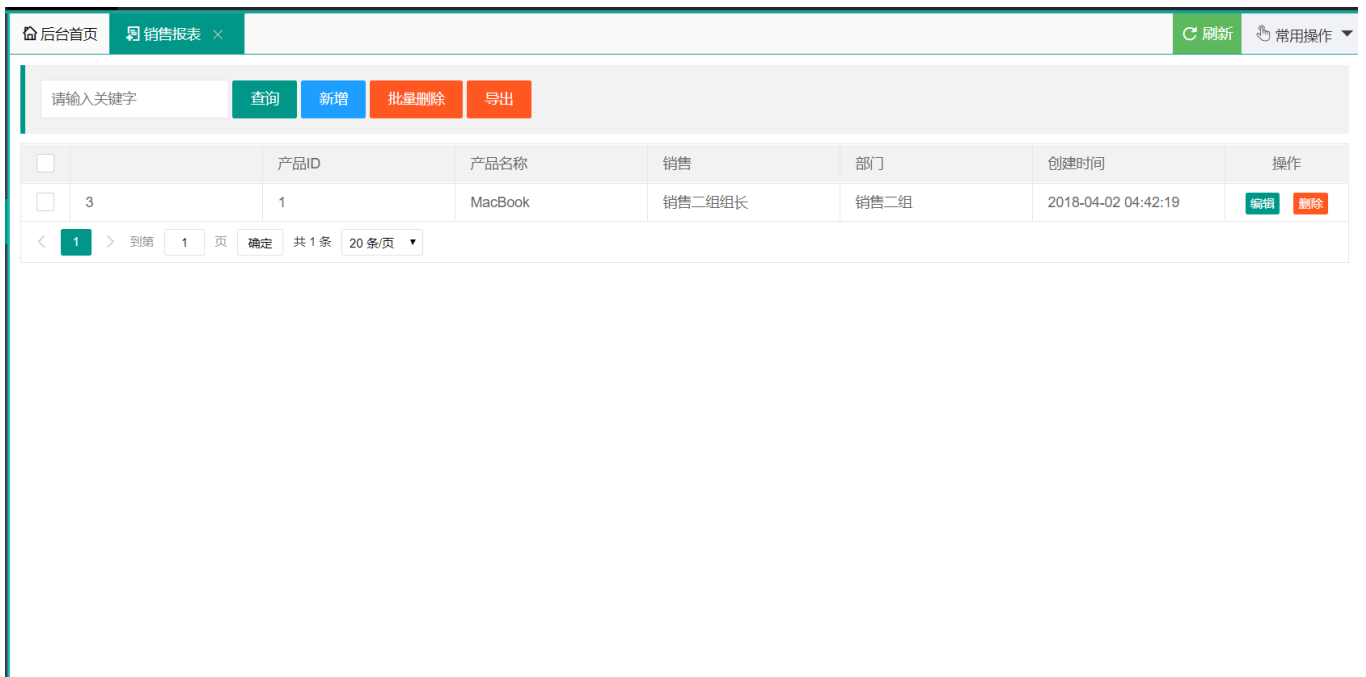
共 2 条

20 条/页

3) 我们用普通销售(销售一组)登录系统，只能查看本人数据，如下所示：



4) 我们用销售二组组长登录系统，可以查看销售二组数据，如下所示：



- 现在我们就把数据权限功能演示完了。其实，通过现有的系统，要实现数据权限，是一件挺简单的事情。

4.3.4. 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：

```
1.  public class XssFilter implements Filter {
2.
3.      @Override
4.      public void init(FilterConfig config) throws ServletException {
5.      }
6.
7.      public void doFilter(ServletRequest request, ServletResponse response,
8.                          FilterChain chain)
9.          throws IOException, ServletException {
10.         XssHttpServletRequestWrapper xssRequest = new
11.         XssHttpServletRequestWrapper (
12.             (HttpServletRequest) request);
13.         chain.doFilter(xssRequest, response);
14.     }
15.
16.     @Override
17.     public void destroy() {
18.     }
19.
20.     @Configuration
21.     public class FilterConfig {
22.         @Bean
23.         public FilterRegistrationBean xssFilterRegistration() {
24.             FilterRegistrationBean registration = new
25.             FilterRegistrationBean();
26.             registration.setDispatcherTypes(DispatcherType.REQUEST);
27.             registration.setFilter(new XssFilter());
28.             registration.addUrlPatterns("/");
29.             registration.setName("xssFilter");
30.             registration.setOrder(Integer.MAX_VALUE);
31.             return registration;
32.         }
33.     }
34. }
```

- 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：

```
1.  public class XssHttpServletRequestWrapper extends
    HttpServletRequestWrapper {
2.      //没被包装过的HttpServletRequest（特殊场景，需要自己过滤）
3.      HttpServletRequest orgRequest;
4.      //html过滤
5.      private final static HTMLFilter htmlFilter = new HTMLFilter();
6.
7.      public XssHttpServletRequestWrapper(HttpServletRequest request) {
8.          super(request);
9.          orgRequest = request;
10.     }
11.
12.     @Override
13.     public ServletInputStream getInputStream() throws IOException {
14.         //非json类型，直接返回
15.         if(!MediaType.APPLICATION_JSON_VALUE.equalsIgnoreCase(super.getHeader(HttpHeaders.CONTENT_TYPE))) {
16.             return super.getInputStream();
17.         }
18.
19.         //为空，直接返回
20.         String json = IOUtils.toString(super.getInputStream(), "utf-8");
21.
22.         if (StringUtils.isBlank(json)) {
23.             return super.getInputStream();
24.         }
25.
26.         //xss过滤
27.         json = xssEncode(json);
28.         final ByteArrayInputStream bis = new ByteArrayInputStream(json.getBytes("utf-8"));
29.         return new ServletInputStream() {
30.             @Override
31.             public boolean isFinished() {
32.                 return true;
33.             }
34.
35.             @Override
36.             public boolean isReady() {
37.                 return true;
38.             }
39.
40.             @Override
```

```
40.         public void setReadListener(ReadListener readListener) {
41.
42.         }
43.
44.         @Override
45.         public int read() throws IOException {
46.             return bis.read();
47.         }
48.     };
49. }
50.
51. @Override
52. public String getParameter(String name) {
53.     String value = super.getParameter(xssEncode(name));
54.     if (StringUtils.isNotBlank(value)) {
55.         value = xssEncode(value);
56.     }
57.     return value;
58. }
59.
60. @Override
61. public String[] getParameterValues(String name) {
62.     String[] parameters = super.getParameterValues(name);
63.     if (parameters == null || parameters.length == 0) {
64.         return null;
65.     }
66.
67.     for (int i = 0; i < parameters.length; i++) {
68.         parameters[i] = xssEncode(parameters[i]);
69.     }
70.     return parameters;
71. }
72.
73. @Override
74. public Map<String,String[]> getParameterMap() {
75.     Map<String,String[]> map = new LinkedHashMap<>();
76.     Map<String,String[]> parameters = super.getParameterMap();
77.     for (String key : parameters.keySet()) {
78.         String[] values = parameters.get(key);
79.         for (int i = 0; i < values.length; i++) {
80.             values[i] = xssEncode(values[i]);
81.         }
82.         map.put(key, values);
83.     }
84.     return map;
```

```

85.     }
86.
87.     @Override
88.     public String getHeader(String name) {
89.         String value = super.getHeader(xssEncode(name));
90.         if (StringUtils.isNotBlank(value)) {
91.             value = xssEncode(value);
92.         }
93.         return value;
94.     }
95.
96.     private String xssEncode(String input) {
97.         return htmlFilter.filter(input);
98.     }
99.
100.    /**
101.     * 获取最原始的request
102.     */
103.    public HttpServletRequest getOrgRequest() {
104.        return orgRequest;
105.    }
106.
107.    /**
108.     * 获取最原始的request
109.     */
110.    public static HttpServletRequest getOrgRequest(HttpServletRequest request) {
111.        if (request instanceof XssHttpServletRequestWrapper) {
112.            return ((XssHttpServletRequestWrapper) request).getOrgRequest();
113.        }
114.
115.        return request;
116.    }
117.
118.    }

```

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：

```

1.    public R data(HttpServletRequest request) {
2.        HttpServletRequest orgRequest = XssHttpServletRequestWrapper.getOrg

```

```
Request(request);
3.     String content = orgRequest.getParameter("content");
4.     //富文本数据
5.     System.out.println(content);
6.     return R.ok();
7. }
```

4.3.5. 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：

```
1.     public class SQLFilter {
2.
3.         /**
4.          * SQL注入过滤
5.          * @param str 待验证的字符串
6.          */
7.         public static String sqlInject(String str){
8.             if(StringUtils.isBlank(str)){
9.                 return null;
10.            }
11.            //去掉'|"||;\字符
12.            str = StringUtils.replace(str, "'", "");
13.            str = StringUtils.replace(str, "\"", "");
14.            str = StringUtils.replace(str, ";", "");
15.            str = StringUtils.replace(str, "\\\"", "");
16.
17.            //转换成小写
18.            str = str.toLowerCase();
19.
20.            //非法字符
21.            String[] keywords = {"master", "truncate", "insert", "select",
22.            "delete", "update", "declare", "alter", "drop"};
23.
24.            //判断是否包含非法字符
25.            for(String keyword : keywords){
26.                if(str.indexOf(keyword) != -1){
27.                    throw new RuntimeException("包含非法字符");
28.                }
29.            }
30.        }
31.    }
```

```
29.
30.         return str;
31.     }
32. }
```

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：

```
1.  public class Query extends LinkedHashMap<String, Object> {
2.      private static final long serialVersionUID = 1L;
3.      //当前页码
4.      private int page;
5.      //每页条数
6.      private int limit;
7.
8.      public Query(Map<String, Object> params) {
9.          this.putAll(params);
10.
11.          //分页参数
12.          this.page = Integer.parseInt(params.get("page").toString());
13.          this.limit = Integer.parseInt(params.get("limit").toString());
14.          this.put("offset", (page - 1) * limit);
15.          this.put("page", page);
16.          this.put("limit", limit);
17.
18.          //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）
19.          String idx = (String)params.get("sidx");
20.          String order = (String)params.get("order");
21.          if(StringUtils.isNotBlank(idx)) {
22.              this.put("sidx", SQLFilter.sqlInject(idx));
23.          }
24.          if(StringUtils.isNotBlank(order)) {
25.              this.put("order", SQLFilter.sqlInject(order));
26.          }
27.
28.      }
29.  }
```

4.3.6. 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：

```
1.  redis:
2.      open: false  # 是否开启redis缓存  true开启  false关闭
3.      database: 0
4.      host: localhost
5.      port: 6379
6.      password:      # 密码（默认为空）
7.      timeout: 6000  # 连接超时时长（毫秒）
8.      pool:
9.          max-active: 1000  # 连接池最大连接数（使用负值表示没有限制）
10.         max-wait: -1      # 连接池最大阻塞等待时间（使用负值表示没有限制）
11.         max-idle: 10      # 连接池中的最大空闲连接
12.         min-idle: 5       # 连接池中的最小空闲连接
```

本项目中，使用Redis服务的代码，如下所示：

```
1.  public class SysConfigServiceImpl implements SysConfigService {
2.      @Autowired
3.      private SysConfigDao sysConfigDao;
4.      @Autowired
5.      private SysConfigRedis sysConfigRedis;
6.
7.      @Override
8.      @Transactional
9.      public void save(SysConfigEntity config) {
10.         sysConfigDao.save(config);
11.         sysConfigRedis.saveOrUpdate(config);
12.     }
```

```

13.
14.     @Override
15.     @Transactional
16.     public void update(SysConfigEntity config) {
17.         sysConfigDao.update(config);
18.         sysConfigRedis.saveOrUpdate(config);
19.     }
20.
21.     @Override
22.     @Transactional
23.     public void updateValueByKey(String key, String value) {
24.         sysConfigDao.updateValueByKey(key, value);
25.         sysConfigRedis.delete(key);
26.     }
27.
28.     @Override
29.     @Transactional
30.     public void deleteBatch(Long[] ids) {
31.         sysConfigDao.deleteBatch(ids);
32.
33.         for(Long id : ids){
34.             SysConfigEntity config = queryObject(id);
35.             sysConfigRedis.delete(config.getKey());
36.         }
37.     }
38. }
39.
40.
41. -----
42.
43.
44. @Component
45. public class SysConfigRedis {
46.     @Autowired
47.     private RedisUtils redisUtils;
48.
49.     public void saveOrUpdate(SysConfigEntity config) {
50.         if(config == null){
51.             return ;
52.         }
53.         String key = RedisKeys.getSysConfigKey(config.getKey());
54.         redisUtils.set(key, config);
55.     }
56.
57.     public void delete(String configKey) {

```

```

58.         String key = RedisKeys.getSysConfigKey(configKey);
59.         redisUtils.delete(key);
60.     }
61.
62.     public SysConfigEntity get(String configKey){
63.         String key = RedisKeys.getSysConfigKey(configKey);
64.         return redisUtils.get(key, SysConfigEntity.class);
65.     }
66. }
67.
68.
69. -----
70.
71.
72. package io.renren.common.aspect;
73.
74. @Component
75. public class RedisUtils {
76.     @Autowired
77.     private RedisTemplate<String, Object> redisTemplate;
78.     @Autowired
79.     private ValueOperations<String, String> valueOperations;
80.     @Autowired
81.     private HashOperations<String, String, Object> hashOperations;
82.     @Autowired
83.     private ListOperations<String, Object> listOperations;
84.     @Autowired
85.     private SetOperations<String, Object> setOperations;
86.     @Autowired
87.     private ZSetOperations<String, Object> zSetOperations;
88.     /** 默认过期时长, 单位:秒 */
89.     public final static long DEFAULT_EXPIRE = 60 * 60 * 24;
90.     /** 不设置过期时长 */
91.     public final static long NOT_EXPIRE = -1;
92.     private final static Gson gson = new Gson();
93.
94.     public void set(String key, Object value, long expire){
95.         valueOperations.set(key, toJson(value));
96.         if(expire != NOT_EXPIRE){
97.             redisTemplate.expire(key, expire, TimeUnit.SECONDS);
98.         }
99.     }
100.
101.     public void set(String key, Object value){
102.         set(key, value, DEFAULT_EXPIRE);

```



```

103.     }
104.
105.     public <T> T get(String key, Class<T> clazz, long expire) {
106.         String value = valueOperations.get(key);
107.         if(expire != NOT_EXPIRE){
108.             redisTemplate.expire(key, expire, TimeUnit.SECONDS);
109.         }
110.         return value == null ? null : fromJson(value, clazz);
111.     }
112.
113.     public <T> T get(String key, Class<T> clazz) {
114.         return get(key, clazz, NOT_EXPIRE);
115.     }
116.
117.     public String get(String key, long expire) {
118.         String value = valueOperations.get(key);
119.         if(expire != NOT_EXPIRE){
120.             redisTemplate.expire(key, expire, TimeUnit.SECONDS);
121.         }
122.         return value;
123.     }
124.
125.     public String get(String key) {
126.         return get(key, NOT_EXPIRE);
127.     }
128.
129.     public void delete(String key) {
130.         redisTemplate.delete(key);
131.     }
132.
133.     /**
134.      * Object转成JSON数据
135.      */
136.     private String toJson(Object object){
137.         if(object instanceof Integer || object instanceof Long || objec
138.         t instanceof Float ||
139.             object instanceof Double || object instanceof Boolean |
140.             | object instanceof String){
141.             return String.valueOf(object);
142.         }
143.         return gson.toJson(object);
144.     }
145.
146.     /**
147.      * JSON数据, 转成Object

```

```

146.         */
147.         private <T> T fromJson(String json, Class<T> clazz){
148.             return gson.fromJson(json, clazz);
149.         }
150.     }

```

大家可能会有疑问，认为这个项目必须要配置Redis缓存，否则会报错，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：

```

1.  @Aspect
2.  @Configuration
3.  public class RedisAspect {
4.      private Logger logger = LoggerFactory.getLogger(getClass());
5.      //是否开启redis缓存 true开启 false关闭
6.      @Value("${spring.redis.open: false}")
7.      private boolean open;
8.
9.      @Around("execution(* io.renren.common.utils.RedisUtils.*(..))")
10.     public Object around(ProceedingJoinPoint point) throws Throwable {
11.         Object result = null;
12.         if(open){
13.             try{
14.                 result = point.proceed();
15.             }catch (Exception e){
16.                 logger.error("redis error", e);
17.                 throw new RRException("Redis服务异常");
18.             }
19.         }
20.         return result;
21.     }
22. }

```

4.3.7. 分布式支持

本系统支持分布式部署，分布式部署，无非就是解决session共享的问题，本系统是将session存储在redis中，从而解决分布式session共享问题。

- 本系统是使用的Shiro，作为权限框架，Shiro中的session默认是在容器（如：tomcat）内的；我们只要继承 `EnterpriseCacheSessionDAO` 类，并 `@Override` 里面的doCreate、

doReadSession、doUpdate、doDelete，就可以把session存储到redis中，代码如下所示：

```
1.  @Component
2.  public class RedisShiroSessionDAO extends EnterpriseCacheSessionDAO {
3.      @Autowired
4.      private RedisTemplate redisTemplate;
5.
6.      //创建session
7.      @Override
8.      protected Serializable doCreate(Session session) {
9.          Serializable sessionId = super.doCreate(session);
10.         final String key = RedisKeys.getShiroSessionKey(sessionId.toString());
11.         setShiroSession(key, session);
12.         return sessionId;
13.     }
14.
15.     //获取session
16.     @Override
17.     protected Session doReadSession(Serializable sessionId) {
18.         Session session = super.doReadSession(sessionId);
19.         if(session == null){
20.             final String key = RedisKeys.getShiroSessionKey(sessionId.toString());
21.             session = getShiroSession(key);
22.         }
23.         return session;
24.     }
25.
26.     //更新session
27.     @Override
28.     protected void doUpdate(Session session) {
29.         super.doUpdate(session);
30.         final String key = RedisKeys.getShiroSessionKey(session.getId().toString());
31.         setShiroSession(key, session);
32.     }
33.
34.     //删除session
35.     @Override
36.     protected void doDelete(Session session) {
37.         super.doDelete(session);
38.         final String key = RedisKeys.getShiroSessionKey(session.getId().toString());
```

```

39.         redisTemplate.delete(key);
40.     }
41.
42.     private Session getShiroSession(String key) {
43.         return (Session)redisTemplate.opsForValue().get(key);
44.     }
45.
46.     private void setShiroSession(String key, Session session){
47.         redisTemplate.opsForValue().set(key, session);
48.         //60分钟过期
49.         redisTemplate.expire(key, 60, TimeUnit.MINUTES);
50.     }
51.
52. }

```

- 上面代码，本系统已提供，如要开启分布式部署，只需打开以下配置即可（配置文件：application.yml）

```

1.   renren:
2.       redis:
3.           open: true # 是否开启redis缓存 true开启 false关闭
4.       shiro:
5.           redis: true # true表示shiro session存到redis里，需要开启redis，才会生效【分布式场景】

```

4.3.8. 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：

```

1.   public class RRException extends RuntimeException {
2.       private static final long serialVersionUID = 1L;
3.
4.       private String msg;
5.       private int code = 500;
6.
7.       public RRException(String msg) {

```

```

8.         super(msg);
9.         this.msg = msg;
10.    }
11.
12.    public RRException(String msg, Throwable e) {
13.        super(msg, e);
14.        this.msg = msg;
15.    }
16.
17.    public RRException(String msg, int code) {
18.        super(msg);
19.        this.msg = msg;
20.        this.code = code;
21.    }
22.
23.    public RRException(String msg, int code, Throwable e) {
24.        super(msg, e);
25.        this.msg = msg;
26.        this.code = code;
27.    }
28.
29.    public String getMsg() {
30.        return msg;
31.    }
32.
33.    public void setMsg(String msg) {
34.        this.msg = msg;
35.    }
36.
37.    public int getCode() {
38.        return code;
39.    }
40.
41.    public void setCode(int code) {
42.        this.code = code;
43.    }

```

如何处理抛出的异常呢，我们定义了RRExceptionHandler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：

```
1.  @RestControllerAdvice
2.  public class RREExceptionHandler {
3.      private Logger logger = LoggerFactory.getLogger(getClass());
4.
5.      /**
6.       * 处理自定义异常
7.       */
8.      @ExceptionHandler(RREException.class)
9.      public R handleRREException(RREException e) {
10.         R r = new R();
11.         r.put("code", e.getCode());
12.         r.put("msg", e.getMessage());
13.
14.         return r;
15.     }
16.
17.     @ExceptionHandler(DuplicateKeyException.class)
18.     public R handleDuplicateKeyException(DuplicateKeyException e) {
19.         logger.error(e.getMessage(), e);
20.         return R.error("数据库中已存在该记录");
21.     }
22.
23.     @ExceptionHandler(AuthorizationException.class)
24.     public R handleAuthorizationException(AuthorizationException e) {
25.         logger.error(e.getMessage(), e);
26.         return R.error("没有权限，请联系管理员授权");
27.     }
28.
29.     @ExceptionHandler(Exception.class)
30.     public R handleException(Exception e) {
31.         logger.error(e.getMessage(), e);
32.         return R.error();
33.     }
34. }
```

4.3.9. 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：

```
1.  public class ValidatorUtils {
2.      private static Validator validator;
3.
4.      static {
5.          validator = Validation.buildDefaultValidatorFactory().getValida
tor();
6.      }
7.
8.      /**
9.       * 校验对象
10.      * @param object      待校验对象
11.      * @param groups      待校验的组
12.      * @throws RRException 校验不通过，则报RRException异常
13.      */
14.      public static void validateEntity(Object object, Class<?>... groups
)
15.          throws RRException {
16.          Set<ConstraintViolation<Object>> constraintViolations = validat
or.validate(object, groups);
17.          if (!constraintViolations.isEmpty()) {
18.              StringBuilder msg = new StringBuilder();
19.              for (ConstraintViolation<Object> constraint:
constraintViolations) {
20.                  msg.append(constraint.getMessage()).append("<br>");
21.              }
22.              throw new RRException(msg.toString());
23.          }
24.      }
25.  }
```

使用案例：

```
1.  @RestController
2.  @RequestMapping("/sys/user")
3.  public class SysUserController extends AbstractController {
4.      /**
5.       * 保存用户
```

```

6.      */
7.      @SysLog("保存用户")
8.      @RequestMapping("/save")
9.      @RequiresPermissions("sys:user:save")
10.     public R save(@RequestBody SysUserEntity user) {
11.         //保存用户时，效验SysUserEntity里，带有AddGroup注解的属性
12.         ValidatorUtils.validateEntity(user, AddGroup.class);
13.
14.         user.setCreateUserId(getUserId());
15.         sysUserService.save(user);
16.
17.         return R.ok();
18.     }
19.
20.     /**
21.      * 修改用户
22.      */
23.     @SysLog("修改用户")
24.     @RequestMapping("/update")
25.     @RequiresPermissions("sys:user:update")
26.     public R update(@RequestBody SysUserEntity user) {
27.         //修改用户时，效验SysUserEntity里，带有UpdateGroup注解的属性
28.         ValidatorUtils.validateEntity(user, UpdateGroup.class);
29.
30.         user.setCreateUserId(getUserId());
31.         sysUserService.update(user);
32.
33.         return R.ok();
34.     }
35. }
36.
37.
38. -----
39.
40.
41. public class SysUserEntity implements Serializable {
42.     /**
43.      * 用户ID
44.      */
45.     private Long userId;
46.
47.     /**
48.      * 用户名
49.      */
50.     @NotBlank(message="用户名不能为空", groups = {AddGroup.class,

```



```
UpdateGroup.class}))
51.     private String username;
52.
53.     /**
54.      * 密码
55.      */
56.     @NotBlank(message="密码不能为空", groups = AddGroup.class)
57.     private String password;
58.
59.     /**
60.      * 盐
61.      */
62.     private String salt;
63.
64.     /**
65.      * 邮箱
66.      */
67.     @NotBlank(message="邮箱不能为空", groups = {AddGroup.class, UpdateGro
up.class})
68.     @Email(message="邮箱格式不正确", groups = {AddGroup.class,
UpdateGroup.class})
69.     private String email;
70.
71.     /**
72.      * 手机号
73.      */
74.     private String mobile;
75.
76.     /**
77.      * 状态  0:禁用    1:正常
78.      */
79.     private Integer status;
80.
81.     /**
82.      * 角色ID列表
83.      */
84.     private List<Long> roleIdList;
85.
86.     /**
87.      * 创建者ID
88.      */
89.     private Long createUserId;
90.
91.     /**
92.      * 创建时间
```

```
93.         */
94.         private Date createTime;
95.     }
```

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。

4.3.10. 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：

```
1.  @Target(ElementType.METHOD)
2.  @Retention(RetentionPolicy.RUNTIME)
3.  @Documented
4.  public @interface SysLog {
5.
6.      String value() default "";
7.  }
```

下面是自定义注解@SysLog的使用方式，如下所示：

```
1.  @RestController
2.  @RequestMapping("/sys/user")
3.  public class SysUserController extends AbstractController {
4.
5.      @SysLog("保存用户")
6.      @RequestMapping("/save")
7.      @RequiresPermissions("sys:user:save")
8.      public R save(@RequestBody SysUserEntity user) {
9.          ValidatorUtils.validateEntity(user, AddGroup.class);
10.
11.          user.setCreateUserId(getUserId());
12.          sysUserService.save(user);
13.      }
```

```
14.         return R.ok();
15.     }
16. }
```

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：

```
1.  /**
2.   * 系统日志，切面处理类
3.   *
4.   * @author chenshun
5.   * @email sunlightcs@gmail.com
6.   * @date 2017年3月8日 上午11:07:35
7.   */
8.  @Aspect
9.  @Component
10. public class SysLogAspect {
11.     @Autowired
12.     private SysLogService sysLogService;
13.
14.     @Pointcut("@annotation(io.renren.common.annotation.SysLog)")
15.     public void logPointCut() {
16.
17.     }
18.
19.     @Around("logPointCut()")
20.     public Object around(ProceedingJoinPoint point) throws Throwable {
21.         long beginTime = System.currentTimeMillis();
22.         //执行方法
23.         Object result = point.proceed();
24.         //执行时长(毫秒)
25.         long time = System.currentTimeMillis() - beginTime;
26.
27.         //保存日志
28.         saveSysLog(point, time);
29.
30.         return result;
31.     }
32.
33.     private void saveSysLog(ProceedingJoinPoint joinPoint, long time) {
```

```

34.         MethodSignature signature = (MethodSignature) joinPoint.getSignature();
35.         Method method = signature.getMethod();
36.
37.         SysLogEntity sysLog = new SysLogEntity();
38.         SysLog syslog = method.getAnnotation(SysLog.class);
39.         if (syslog != null) {
40.             //注解上的描述
41.             syslog.setOperation(syslog.value());
42.         }
43.
44.         //请求的方法名
45.         String className = joinPoint.getTarget().getClass().getName();
46.         String methodName = signature.getName();
47.         syslog.setMethod(className + "." + methodName + "()");
48.
49.         //请求的参数
50.         Object[] args = joinPoint.getArgs();
51.         try {
52.             String params = new Gson().toJson(args[0]);
53.             syslog.setParams(params);
54.         } catch (Exception e) {
55.
56.         }
57.
58.         //获取request
59.         HttpServletRequest request =
HttpContextUtils.getHttpServletRequest();
60.         //设置IP地址
61.         syslog.setIp(IPUtils.getIpAddr(request));
62.
63.         //用户名
64.         String username = ((SysUserEntity) SecurityUtils.getSubject().getPrincipal()).getUsername();
65.         syslog.setUsername(username);
66.
67.         syslog.setTime(time);
68.         syslog.setCreateDate(new Date());
69.         //保存系统日志
70.         syslogService.save(syslog);
71.     }
72. }

```

`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。

4.3.11. 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。

人人权限系统

系统管理

功能示例

清除缓存

主题设置

退出

您好! 欢迎登录

后台首页

系统用户管理

系统菜单

菜单管理

图标管理

系统设置

系统监控

后台首页

菜单管理

修改

类型

菜单名称

上级菜单

菜单URL

授权标识

排序号

图标

确定

刷新

常用操作

参数管理

系统设置

modules/sys/config.html

sys:config:list,sys:config:info,sys:config:save,sys:config:update,sys:config:delete

2

larry-xitong-pressed

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：

```
1. @RestController
2. @RequestMapping("/sys/config")
3. public class SysConfigController extends AbstractController {
4.
5.     @RequestMapping("/list")
6.     @RequiresPermissions("sys:config:list")
7.     public R list(@RequestParam Map<String, Object> params){
8.
9.     }
10.
11.     @RequestMapping("/info/{id}")
12.     @RequiresPermissions("sys:config:info")
13.     public R info(@PathVariable("id") Long id){
14.
```

```
15.     }
16.
17.     @RequestMapping("/save")
18.     @RequiresPermissions("sys:config:save")
19.     public R save(@RequestBody SysConfigEntity config){
20.
21.     }
22.
23.     @RequestMapping("/update")
24.     @RequiresPermissions("sys:config:update")
25.     public R update(@RequestBody SysConfigEntity config){
26.
27.     }
28.
29.     @RequestMapping("/delete")
30.     @RequiresPermissions("sys:config:delete")
31.     public R delete(@RequestBody Long[] ids){
32.
33.     }
34.
35. }
```

4.3.12. 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：

人人权限系统

系统管理功能示例

清除缓存主题设置退出

您好! 欢迎登录

后台首页

系统用户管理

用户管理

角色管理

部门管理

系统菜单

系统设置

系统监控

角色管理

新增

角色名称开发角色

所属部门上海分公司

备注开发人员

功能权限

系统管理

系统设置

系统用户管理

系统监控

系统菜单

功能示例

文章管理

销售报表

数据权限

人人开源集团

长沙分公司

上海分公司

技术部

销售部

销售一组

销售二组

确定

4.3.13. 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：

人人权限系统

系统管理功能示例

清除缓存主题设置退出

您好! 欢迎登录

后台首页

系统用户管理

用户管理

角色管理

部门管理

系统菜单

系统设置

系统监控

用户管理

新增

用户名zhangsang

所属部门人人开源集团

密码

邮箱test@163.com

手机号13512345678

角色

销售经理角色

销售一组组长

销售二组组长

普通销售角色

开发角色

状态

正常

确定

4.4. 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。

4.4.1. 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：

```
1.  @Component("testTask")
2.  public class TestTask {
3.      private Logger logger = LoggerFactory.getLogger(getClass());
4.
5.      @Autowired
6.      private SysUserService sysUserService;
7.
8.      //定时任务只能接受一个参数；如果有多个参数，使用json数据即可
9.      public void test(String params){
10.         logger.info("我是带参数的test方法，正在被执行，参数为：" + params);
11.
12.         try {
13.             Thread.sleep(1000L);
14.         } catch (InterruptedException e) {
15.             e.printStackTrace();
16.         }
17.
18.         SysUserEntity user = sysUserService.queryObject(1L);
19.         System.out.println(ToStringBuilder.reflectionToString(user));
20.
21.     }
22.
23.
24.     public void test2(){
25.         logger.info("我是不带参数的test2方法，正在被执行");
26.     }
27. }
```

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：

人人权限系统

系统管理功能示例

清除缓存主题设置退出

您好! 欢迎登录

后台首页系统用户管理系统菜单系统设置定时任务参数管理字典管理文件上传系统监控

后台首页定时任务

修改

bean名称testTask

方法名称test

参数renren

cron表达式0 0/30 * * * ?

备注有参数测试

确定

人人权限系统

系统管理功能示例

清除缓存主题设置退出

您好! 欢迎登录

后台首页系统用户管理系统菜单系统设置定时任务参数管理字典管理文件上传系统监控

后台首页定时任务

bean名称查询新增批量删除日志列表

<input type="checkbox"/>	任务ID	bean名称	方法名称	参数	cron表达式	备注	状态	操作
<input type="checkbox"/>	1	testTask	test	renren	0 0/30 * * * ?	有参数测试	正常	编辑暂停恢复执行删除

<1>

到第1页

确定

共1条

20条/页

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。

4.4.2. 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：

```
1. public class ScheduleUtils {
```

```

2.     private final static String JOB_NAME = "TASK_";
3.
4.     /**
5.      * 获取触发器key
6.      */
7.     private static TriggerKey getTriggerKey(Long jobId) {
8.         return TriggerKey.triggerKey(JOB_NAME + jobId);
9.     }
10.
11.    /**
12.     * 获取jobKey
13.     */
14.    private static JobKey getJobKey(Long jobId) {
15.        return JobKey.jobKey(JOB_NAME + jobId);
16.    }
17.
18.    /**
19.     * 获取表达式触发器
20.     */
21.    public static CronTrigger getCronTrigger(Scheduler scheduler, Long
jobId) {
22.        try {
23.            return (CronTrigger) scheduler.getTrigger(getTriggerKey(job
Id));
24.        } catch (SchedulerException e) {
25.            throw new RRException("getCronTrigger异常, 请检查qrtz开头的表,
是否有脏数据", e);
26.        }
27.    }
28.
29.    /**
30.     * 创建定时任务
31.     */
32.    public static void createScheduleJob(Scheduler scheduler, ScheduleJ
obEntity scheduleJob) {
33.        try {
34.            //构建job信息
35.            JobDetail jobDetail = JobBuilder.newJob(ScheduleJob.class).
withIdentity(getJobKey(scheduleJob.getId())).build();
36.
37.            //表达式调度构建器
38.            CronScheduleBuilder scheduleBuilder = CronScheduleBuilder.c
ronSchedule(scheduleJob.getCronExpression())
39.                .withMisfireHandlingInstructionDoNothing();
40.

```

```

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

43.
44.
45.         //放入参数，运行时的方法可以获取
46.         jobDetail.getJobDataMap().put(ScheduleJobEntity.JOB_PARAM_KEY, new Gson().toJson(scheduleJob));
47.
48.         scheduler.scheduleJob(jobDetail, trigger);
49.
50.         //暂停任务
51.         if (scheduleJob.getStatus() == ScheduleStatus.PAUSE.getValue()) {
52.             pauseJob(scheduler, scheduleJob.getJobId());
53.         }
54.     } catch (SchedulerException e) {
55.         throw new RuntimeException("创建定时任务失败", e);
56.     }
57. }
58.
59. /**
60.  * 更新定时任务
61.  */
62. public static void updateScheduleJob(Scheduler scheduler, ScheduleJobEntity scheduleJob) {
63.     try {
64.         TriggerKey triggerKey = getTriggerKey(scheduleJob.getJobId());
65.
66.         //表达式调度构建器
67.         CronScheduleBuilder scheduleBuilder = CronScheduleBuilder.cronSchedule(scheduleJob.getCronExpression())
.withMisfireHandlingInstructionDoNothing();
68.
69.
70.         CronTrigger trigger = getCronTrigger(scheduler, scheduleJob.getJobId());
71.
72.         //按新的cronExpression表达式重新构建trigger
73.         trigger = trigger.getTriggerBuilder().withIdentity(triggerKey).withSchedule(scheduleBuilder).build();
74.
75.         //参数
76.         trigger.getJobDataMap().put(ScheduleJobEntity.JOB_PARAM_KEY

```

```

77.         , new Gson().toJson(scheduleJob));
78.
79.         scheduler.rescheduleJob(triggerKey, trigger);
80.
81.         //暂停任务
82.         if(scheduleJob.getStatus() == ScheduleStatus.PAUSE.getValue
83.         ()){
84.
85.             pauseJob(scheduler, scheduleJob.getJobId());
86.         }
87.
88.     } catch (SchedulerException e) {
89.         throw new RuntimeException("更新定时任务失败", e);
90.     }
91.
92. /**
93.  * 立即执行任务
94.  */
95.     public static void run(Scheduler scheduler, ScheduleJobEntity sched
96.     uleJob) {
97.         try {
98.             //参数
99.             JobDataMap dataMap = new JobDataMap();
100.            dataMap.put(ScheduleJobEntity.JOB_PARAM_KEY, new Gson().toJ
101.            son(scheduleJob));
102.
103.            scheduler.triggerJob(getJobKey(scheduleJob.getJobId()), dat
104.            aMap);
105.        } catch (SchedulerException e) {
106.            throw new RuntimeException("立即执行定时任务失败", e);
107.        }
108.    }
109.
110. /**
111.  * 暂停任务
112.  */
113.     public static void pauseJob(Scheduler scheduler, Long jobId) {
114.         try {
115.             scheduler.pauseJob(getJobKey(jobId));
116.         } catch (SchedulerException e) {
117.             throw new RuntimeException("暂停定时任务失败", e);
118.         }
119.     }
120.
121. /**

```

```

117.      * 恢复任务
118.      */
119.      public static void resumeJob(Scheduler scheduler, Long jobId) {
120.          try {
121.              scheduler.resumeJob(getJobKey(jobId));
122.          } catch (SchedulerException e) {
123.              throw new RRException("暂停定时任务失败", e);
124.          }
125.      }
126.
127.      /**
128.       * 删除定时任务
129.       */
130.      public static void deleteScheduleJob(Scheduler scheduler, Long jobId) {
131.          try {
132.              scheduler.deleteJob(getJobKey(jobId));
133.          } catch (SchedulerException e) {
134.              throw new RRException("删除定时任务失败", e);
135.          }
136.      }
137.  }

```

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：

```

1.      @Service("scheduleJobService")

```

```

2. public class ScheduleJobServiceImpl implements ScheduleJobService {
3.     /**
4.      * 项目启动时，初始化定时器
5.      */
6.     @PostConstruct
7.     public void init(){
8.         List<ScheduleJobEntity> scheduleJobList = schedulerJobDao.query
List(new HashMap<>());
9.         for(ScheduleJobEntity scheduleJob : scheduleJobList){
10.             CronTrigger cronTrigger = ScheduleUtils.getCronTrigger(sche
duler, scheduleJob.getJobId());
11.             //如果不存在，则创建
12.             if(cronTrigger == null) {
13.                 ScheduleUtils.createScheduleJob(scheduler, scheduleJob)
;
14.             }else {
15.                 ScheduleUtils.updateScheduleJob(scheduler, scheduleJob)
;
16.             }
17.         }
18.     }
19.
20. }

```

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：

```

1. //构建一个新的定时任务，JobBuilder.newJob() 只能接受Job类型的参数
2. //把ScheduleJob.class作为参数传进去，ScheduleJob继承QuartzJobBean，而Quartz
JobBean实现了Job接口
3. JobDetail jobDetail =
JobBuilder.newJob(ScheduleJob.class).withIdentity(getJobKey(scheduleJob
.getJobId())).build();
4.
5. //构建cron，定时任务的周期
6. CronScheduleBuilder scheduleBuilder = CronScheduleBuilder.cronSchedule(
scheduleJob.getCronExpression())
7.     .withMisfireHandlingInstructionDoNothing();
8.
9. //根据cron，构建一个CronTrigger
10. CronTrigger trigger = TriggerBuilder.newTrigger().withIdentity(getTrigg
erKey(scheduleJob.getJobId())).

```

```

11.         withSchedule(scheduleBuilder).build();
12.
13.         //放入参数, 运行时的方法可以获取
14.         jobDetail.getJobDataMap().put(ScheduleJobEntity.JOB_PARAM_KEY, new
            Gson().toJson(scheduleJob));
15.
16.         //把任务添加到Quartz中
17.         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:

```

1.  public class ScheduleJob extends QuartzJobBean {
2.      private Logger logger = LoggerFactory.getLogger(getClass());
3.      private ExecutorService service = Executors.newSingleThreadExecutor
        ();
4.
5.      @Override
6.      protected void executeInternal(JobExecutionContext context) throws
        JobExecutionException {
7.          //获取job里的参数, 创建job时, 传进去的ScheduleJobEntity对象
8.          String jsonJob = context.getMergedJobDataMap().getString(Schedu
        leJobEntity.JOB_PARAM_KEY);
9.          ScheduleJobEntity scheduleJob = new Gson().fromJson(jsonJob, Sc
        heduleJobEntity.class);
10.
11.         //获取scheduleJobLogService
12.         ScheduleJobLogService scheduleJobLogService =
            (ScheduleJobLogService)
            SpringContextUtils.getBean("scheduleJobLogService");
13.
14.         //数据库保存执行记录
15.         ScheduleJobLogEntity log = new ScheduleJobLogEntity();
16.         log.setJobId(scheduleJob.getJobId());
17.         log.setBeanName(scheduleJob.getBeanName());
18.         log.setMethodName(scheduleJob.getMethodName());
19.         log.setParams(scheduleJob.getParams());
20.         log.setCreateTime(new Date());
21.
22.         //任务开始时间
23.         long startTime = System.currentTimeMillis();
24.
25.         try {

```

```

26.         //执行任务，这步是关键
27.         logger.info("任务准备执行，任务ID：" + scheduleJob.getJobId())
28.         ;
29.         ScheduleRunnable task = new ScheduleRunnable(scheduleJob.get
30. tBeanName(),
31.                 scheduleJob.getMethodName(), scheduleJob.getParams (
32. ));
33.         Future<?> future = service.submit(task);
34.         future.get();
35.
36.         //任务执行总时长
37.         long times = System.currentTimeMillis() - startTime;
38.         log.setTimes((int)times);
39.         //任务状态    0：成功    1：失败
40.         log.setStatus(0);
41.
42.         logger.info("任务执行完毕，任务ID：" + scheduleJob.getJobId()
43. + "    总共耗时：" + times + "毫秒");
44.     } catch (Exception e) {
45.         logger.error("任务执行失败，任务ID：" + scheduleJob.getJobId()
46. , e);
47.
48.         //任务执行总时长
49.         long times = System.currentTimeMillis() - startTime;
50.         log.setTimes((int)times);
51.
52.         //任务状态    0：成功    1：失败
53.         log.setStatus(1);
54.         log.setError(StringUtils.substring(e.toString(), 0, 2000));
55.     }finally {
56.         scheduleJobLogService.save(log);
57.     }
58. }
59. }

```

我们搞了一个线程，用来执行定时任务。具体执行是在ScheduleRunnable类里，通过Java反射，执行对应方法的，如下所示：

```

1. public class ScheduleRunnable implements Runnable {
2.     private Object target;
3.     private Method method;
4.     private String params;
5.

```



```

6.     public ScheduleRunnable(String beanName, String methodName, String
params) throws NoSuchMethodException, SecurityException {
7.         //获取spring bean
8.         this.target = SpringContextUtils.getBean(beanName);
9.         this.params = params;
10.
11.         if(StringUtils.isNotBlank(params)){
12.             this.method = target.getClass().getDeclaredMethod(methodName, String.class);
13.         }else{
14.             this.method = target.getClass().getDeclaredMethod(methodName);
15.         }
16.     }
17.
18.     @Override
19.     public void run() {
20.         try {
21.             ReflectionUtils.makeAccessible(method);
22.             if(StringUtils.isNotBlank(params)){
23.                 method.invoke(target, params);
24.             }else{
25.                 method.invoke(target);
26.             }
27.         } catch (Exception e) {
28.             throw new RuntimeException("执行定时任务失败", e);
29.         }
30.     }
31. }

```

4.5. 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。

4.5.1. 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，并登录后，再创建七牛空间，如下图：

产品列表

数据统计 文档中心 工单 站内信 个人面板

资源主页 个人中心 财务统计

对象存储

融合 CDN SSL 证书服务 数据处理 直播云服务

新建存储空间

搜索存储空间

存储空间列表

华东 renren 跨区域同步管理

存储空间名称

存储空间名称作为唯一的 Bucket 识别符，遇到冲突请更换名称。名称由 4 ~ 63 个字符组成，可包含字母、数字、中划线。

ios-app

存储区域

北美区域尚未支持自定义数据处理服务，一旦创建区域无法修改，请谨慎选择。

华东 华北 华南 北美

访问控制

公开和私有仅对 Bucket 的读文件生效，修改、删除、写入等对 Bucket 的操作均需要拥有者的授权才能进行操作。

公开空间 私有空间

取消 确定创建

2. 进入管理后端，填写七牛配置信息，如下图：

人人权限系统

系统管理 功能示例

清除缓存 主题设置 退出

后台首页 文件上传

云存储配置

存储类型 七牛 阿里云 腾讯云

域名 http://7xllj2.com1.z0.glb.clouddn.com

路径前缀 upload

AccessKey

SecretKey

空间名 ios-app

确定

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：



4.5.2. 阿里云的配置

- 进入管理后端，填写阿里云配置信息，如下图：

系统管理

功能示例

后台首页

文件上传 ×

云存储配置

存储类型

☐ 七牛 ☒ 阿里云 ☐ 腾讯云

域名

阿里云绑定的域名

路径前缀

不设置默认为空

EndPoint

阿里云EndPoint

AccessKeyId

阿里云AccessKeyId

AccessKeySecret

阿里云AccessKeySecret

BucketName

阿里云BucketName

确定

- 进去阿里云管理后台，并创建Bucket，如下图：

管理控制台

产品与服务

搜索

223

费用

工单

备案

支持

hi317****@aliyun.com

简体中文

对象存储 OSS

新建 Bucket

命名

renren-oss

Bucket 命名规范：

- 只能包含小写字母，数字和短横线
- 必须以小写字母和数字开头和结尾
- 长度限制在 3-63 之间

所属地域

华北 2

相同地域内的产品内网可以互通；订购后不支持更换地域，请谨慎选择

EndPoint

oss-cn-beijing.aliyuncs.com

存储类型

标准存储

- 标准存储类型：高可靠、高可用、高性能，数据会经常被访问到
- 低频访问类型：数据长期存储、较少访问，存储单价低于标准类型
- 归档存储类型：数据长期存储、基本不访问，存储单价低于低频访问型

读写权限

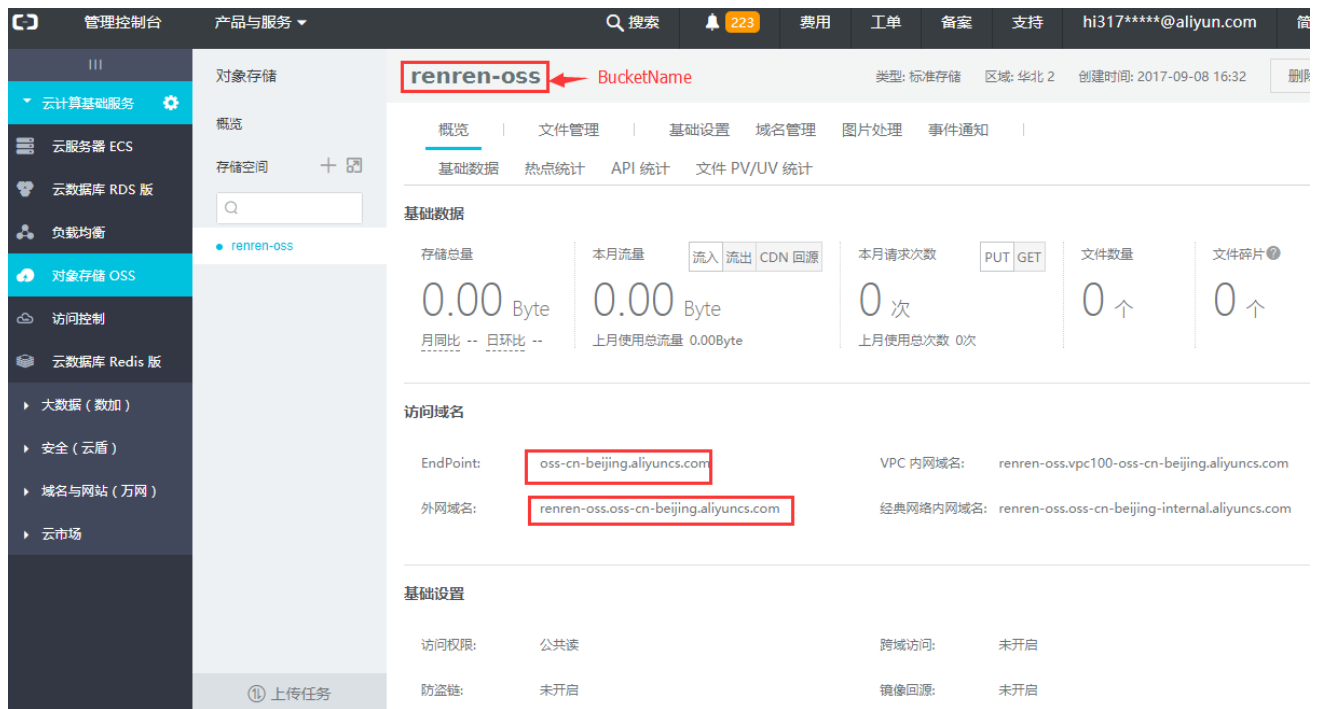
公共读

- 私有：对文件的所有访问操作需要进行身份验证
- 公共读：对文件写操作需要进行身份验证；可以对文件进行匿名读
- 公共读写：所有人都可以对文件进行读写操作

取消

确定

- 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



4.5.3. 腾讯云的配置

- 进入管理后端，填写腾讯云配置信息，如下图：

系统管理

功能示例

后台首页

文件上传

云存储配置

存储类型

☐ 七牛

☐ 阿里云

☒ 腾讯云

域名

腾讯云绑定的域名

路径前缀

不设置默认为空

AppId

腾讯云AppId

SecretId

腾讯云SecretId

SecretKey

腾讯云SecretKey

BucketName

腾讯云BucketName

Bucket所属地区

如：sh（可选值，华南：gz 华北：tj 华东：sh）

确定

- 进去腾讯云管理后台，并创建Bucket，如下图：

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列表

任务管理

监控报表

密钥管理

Bucket

创建Bucket

所属项目

默认项目

* 名称

renren

仅支持小写字母、数字的组合，不能超过40字符。

地域

上海(华东)

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](#)

共 0 项

确定

取消

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



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



4.5.4. 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：

```
1. <dependency>
2.     <groupId>com.qiniu</groupId>
3.     <artifactId>qiniu-java-sdk</artifactId>
4.     <version>${qiniu.version}</version>
5. </dependency>
6. <dependency>
7.     <groupId>com.aliyun.oss</groupId>
8.     <artifactId>aliyun-sdk-oss</artifactId>
9.     <version>${aliyun.oss.version}</version>
10. </dependency>
11. <dependency>
12.     <groupId>com.qcloud</groupId>
13.     <artifactId>cos_api</artifactId>
14.     <version>${qcloud.cos.version}</version>
15.     <exclusions>
16.         <exclusion>
17.             <groupId>org.slf4j</groupId>
18.             <artifactId>slf4j-log4j12</artifactId>
19.         </exclusion>
20.     </exclusions>
21. </dependency>
```

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

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



```

4.
5.     /**
6.      * 文件路径
7.      * @param prefix 前缀
8.      * @return 返回上传路径
9.      */
10.    public String getPath(String prefix) {
11.        //生成uuid
12.        String uuid = UUID.randomUUID().toString().replaceAll("-", "");
13.        //文件路径
14.        String path = DateUtils.format(new Date(), "yyyyMMdd") + "/" +
uuid;

15.
16.        if (StringUtils.isNotBlank(prefix)) {
17.            path = prefix + "/" + path;
18.        }

19.
20.        return path;
21.    }
22.
23.    /**
24.     * 文件上传
25.     * @param data      文件字节数组
26.     * @param path      文件路径, 包含文件名
27.     * @return          返回http地址
28.     */
29.    public abstract String upload(byte[] data, String path);
30.
31.    /**
32.     * 文件上传
33.     * @param data      文件字节数组
34.     * @return          返回http地址
35.     */
36.    public abstract String upload(byte[] data);
37.
38.    /**
39.     * 文件上传
40.     * @param inputStream 字节流
41.     * @param path        文件路径, 包含文件名
42.     * @return            返回http地址
43.     */
44.    public abstract String upload(InputStream inputStream, String path)
;

45.
46.    /**

```

```

47.      * 文件上传
48.      * @param inputStream 字节流
49.      * @return 返回http地址
50.      */
51.      public abstract String upload(InputStream inputStream);
52.
53.  }

```

- 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：

```

1.  import com.qiniu.common.Zone;
2.  import com.qiniu.http.Response;
3.  import com.qiniu.storage.Configuration;
4.  import com.qiniu.storage.UploadManager;
5.  import com.qiniu.util.Auth;
6.  import io.renren.common.exception.RRException;
7.  import org.apache.commons.io.IOUtils;
8.
9.  public class QiniuCloudStorageService extends CloudStorageService{
10.      private UploadManager uploadManager;
11.      private String token;
12.
13.      public QiniuCloudStorageService(CloudStorageConfig config){
14.          this.config = config;
15.
16.          //初始化
17.          init();
18.      }
19.
20.      private void init(){
21.          uploadManager = new UploadManager(new Configuration(Zone.autoZone()));
22.          token = Auth.create(config.getQiniuAccessKey(), config.getQiniuSecretKey()).
23.              uploadToken(config.getQiniuBucketName());
24.      }
25.
26.      @Override
27.      public String upload(byte[] data, String path) {
28.          try {
29.              Response res = uploadManager.put(data, path, token);
30.              if (!res.isOK()) {
31.                  throw new RuntimeException("上传七牛出错：" + res.toString());

```

```

32.         }
33.     } catch (Exception e) {
34.         throw new RuntimeException("上传文件失败, 请核对七牛配置信息", e);
35.     }
36.
37.     return config.getQiniuDomain() + "/" + path;
38. }
39.
40. @Override
41. public String upload(InputStream inputStream, String path) {
42.     try {
43.         byte[] data = IOUtils.toByteArray(inputStream);
44.         return this.upload(data, path);
45.     } catch (IOException e) {
46.         throw new RuntimeException("上传文件失败", e);
47.     }
48. }
49.
50. @Override
51. public String upload(byte[] data) {
52.     return upload(data, getPath(config.getQiniuPrefix()));
53. }
54.
55. @Override
56. public String upload(InputStream inputStream) {
57.     return upload(inputStream, getPath(config.getQiniuPrefix()));
58. }
59. }

```

- 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：

```

1.  import com.aliyun.oss.OSSClient;
2.  import java.io.ByteArrayInputStream;
3.  import java.io.InputStream;
4.
5.  public class AliyunCloudStorageService extends CloudStorageService{
6.      private OSSClient client;
7.
8.      public AliyunCloudStorageService(CloudStorageConfig config){
9.          this.config = config;
10.
11.          //初始化
12.          init();

```

```

13.     }
14.
15.     private void init(){
16.         client = new OSSClient(config.getAliyunEndPoint(), config.getAl
iyunAccessKeyId(),
17.             config.getAliyunAccessKeySecret());
18.     }
19.
20.     @Override
21.     public String upload(byte[] data, String path) {
22.         return upload(new ByteArrayInputStream(data), path);
23.     }
24.
25.     @Override
26.     public String upload(InputStream inputStream, String path) {
27.         try {
28.             client.putObject(config.getAliyunBucketName(), path, inputS
tream);
29.         } catch (Exception e){
30.             throw new RuntimeException("上传文件失败, 请检查配置信息", e);
31.         }
32.
33.         return config.getAliyunDomain() + "/" + path;
34.     }
35.
36.     @Override
37.     public String upload(byte[] data) {
38.         return upload(data, getPath(config.getAliyunPrefix()));
39.     }
40.
41.     @Override
42.     public String upload(InputStream inputStream) {
43.         return upload(inputStream, getPath(config.getAliyunPrefix()));
44.     }
45. }

```

- 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：

```

1. import com.qcloud.cos.COSClient;
2. import com.qcloud.cos.ClientConfig;
3. import com.qcloud.cos.request.UploadFileRequest;
4. import com.qcloud.cos.sign.Credentials;
5. import net.sf.json.JSONObject;

```

```

6.  import org.apache.commons.io.IOUtils;
7.
8.  public class QcloudCloudStorageService extends CloudStorageService{
9.      private COSClient client;
10.
11.      public QcloudCloudStorageService(CloudStorageConfig config){
12.          this.config = config;
13.
14.          //初始化
15.          init();
16.      }
17.
18.      private void init(){
19.          Credentials credentials = new Credentials(config.getQcloudAppId
20.          (), config.getQcloudSecretId(),
21.              config.getQcloudSecretKey());
22.
23.          //初始化客户端配置
24.          ClientConfig clientConfig = new ClientConfig();
25.          //设置bucket所在的区域, 华南:gz 华北:tj 华东:sh
26.          clientConfig.setRegion(config.getQcloudRegion());
27.
28.          client = new COSClient(clientConfig, credentials);
29.
30.          @Override
31.          public String upload(byte[] data, String path) {
32.              //腾讯云必需要以"/"开头
33.              if(!path.startsWith("/")) {
34.                  path = "/" + path;
35.              }
36.
37.              //上传到腾讯云
38.              UploadFileRequest request = new UploadFileRequest(config.getQcl
39.              oudBucketName(), path, data);
40.              String response = client.uploadFile(request);
41.
42.              JSONObject jsonObject = JSONObject.fromObject(response);
43.              if(jsonObject.getInt("code") != 0) {
44.                  throw new RuntimeException("文件上传失败, " + jsonObject.getString(
45.                  "message"));
46.              }
47.
48.              return config.getQcloudDomain() + path;
49.          }

```

```

48.
49.     @Override
50.     public String upload(InputStream inputStream, String path) {
51.         try {
52.             byte[] data = IOUtils.toByteArray(inputStream);
53.             return this.upload(data, path);
54.         } catch (IOException e) {
55.             throw new RuntimeException("上传文件失败", e);
56.         }
57.     }
58.
59.     @Override
60.     public String upload(byte[] data) {
61.         return upload(data, getPath(config.getQcloudPrefix()));
62.     }
63.
64.     @Override
65.     public String upload(InputStream inputStream) {
66.         return upload(inputStream, getPath(config.getQcloudPrefix()));
67.     }
68. }

```

- 对外提供了OSSFactory工厂，可方便业务的调用，如下所示：

```

1.     public final class OSSFactory {
2.         private static SysConfigService sysConfigService;
3.
4.         static {
5.             OSSFactory.sysConfigService = (SysConfigService)
6.             SpringContextUtils.getBean("sysConfigService");
7.         }
8.
9.         public static CloudStorageService build(){
10.             //获取云存储配置信息
11.             CloudStorageConfig config = sysConfigService.getConfigObject(Con
12.             nfigConstant.CLOUD_STORAGE_CONFIG_KEY, CloudStorageConfig.class);
13.
14.             if(config.getType() == Constant.CloudService.QINIU.getValue()){
15.                 return new QiniuCloudStorageService(config);
16.             }else if(config.getType() == Constant.CloudService.ALIYUN.getVa
17.             lue()){
18.                 return new AliyunCloudStorageService(config);
19.             }else if(config.getType() == Constant.CloudService.QCLOUD.getVa
20.             lue()){

```

```

17.         return new QcloudCloudStorageService(config);
18.     }
19.
20.     return null;
21. }
22.
23. }

```

- 文件上传的例子，如下：

```

1.  @RequestMapping("/upload")
2.  public R upload(@RequestParam("file") MultipartFile file) throws Except
    ion {
3.      if (file.isEmpty()) {
4.          throw new RRException("上传文件不能为空");
5.      }
6.
7.      //上传文件，并返回文件的http地址
8.      String url = OSSFactory.build().upload(file.getBytes());
9.  }

```

4.6. API模块

APP模块，主要是简化APP开发，如：为微信小程序、IOS、Android提供接口，拥有一套单独的用户体系，没有与renren-admin用户表共用，因为renren-admin用户表里存放的是企业内部人员账号，具有后台管理员权限，可以登录后台管理系统，而renren-api用户表里存放的是我们的真实用户，不具备登录后台管理系统的权限。renren-api主要是实现了用户注册、登录、接口权限认证、获取登录用户等功能，为APP接口的安全调用，提供一套优雅的解决方案，从而简化APP接口开发。

4.6.1. API的使用

API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——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。

```
1.  import io.renren.annotation.Login;
2.  import io.renren.annotation.LoginUser;
3.
4.  @RestController
5.  @RequestMapping("/api")
6.  @Api(tags="测试接口")
7.  public class ApiTestController {
8.
9.      @Login
10.     @GetMapping("userInfo")
11.     @ApiOperation(value="获取用户信息", response=UserEntity.class)
12.     public R userInfo(@ApiIgnore @LoginUser UserEntity user){
13.         return R.ok().put("user", user);
14.     }
15.
16.     @Login
17.     @GetMapping("userId")
18.     @ApiOperation("获取用户ID")
19.     public R userInfo(@ApiIgnore @RequestAttribute("userId") Integer userId){
20.         return R.ok().put("userId", userId);
21.     }
22.
23.     @GetMapping("notToken")
24.     @ApiOperation("忽略Token验证测试")
25.     public R notToken(){
26.         return R.ok().put("msg", "无需token也能访问。。。");
27.     }
28. }
```

4.6.2. 源码分析

- 我们先来看看，API用户登录的时候，都干了那些事情，如下所示：

```
1.  @RestController
2.  @RequestMapping("/api")
3.  @Api(tags="登录接口")
4.  public class ApiLoginController {
5.      @Autowired
```



```

6.     private UserService userService;
7.     @Autowired
8.     private TokenService tokenService;
9.
10.
11.     @PostMapping("login")
12.     @ApiOperation("登录")
13.     public R login(@RequestBody LoginForm form) {
14.         //表单校验
15.         ValidatorUtils.validateEntity(form);
16.
17.         //用户登录
18.         Map<String, Object> map = userService.login(form);
19.
20.         return R.ok(map);
21.     }
22.
23.     @Login
24.     @PostMapping("logout")
25.     @ApiOperation("退出")
26.     public R logout(@ApiIgnore @RequestAttribute("userId") long userId)
27.     {
28.         tokenService.expireToken(userId);
29.         return R.ok();
30.     }
31. }
32.
33.
34. -----
35.
36.
37. /**
38.  * jwt工具类
39.  */
40. @ConfigurationProperties(prefix = "renren.jwt")
41. @Component
42. public class JwtUtils {
43.     private Logger logger = LoggerFactory.getLogger(getClass());
44.
45.     private String secret;
46.     private long expire;
47.     private String header;
48.
49.     /**

```

```
50.      * 生成jwt token
51.      */
52.      public String generateToken(long userId) {
53.          Date nowDate = new Date();
54.          //过期时间
55.          Date expireDate = new Date(nowDate.getTime() + expire * 1000);
56.
57.          return Jwts.builder()
58.              .setHeaderParam("typ", "JWT")
59.              .setSubject(userId+"")
60.              .setIssuedAt(nowDate)
61.              .setExpiration(expireDate)
62.              .signWith(SignatureAlgorithm.HS512, secret)
63.              .compact();
64.      }
65.
66.      public Claims getClaimByToken(String token) {
67.          try {
68.              return Jwts.parser()
69.                  .setSigningKey(secret)
70.                  .parseClaimsJws(token)
71.                  .getBody();
72.          } catch (Exception e) {
73.              logger.debug("validate is token error ", e);
74.              return null;
75.          }
76.      }
77.
78.      /**
79.       * token是否过期
80.       * @return true: 过期
81.       */
82.      public boolean isTokenExpired(Date expiration) {
83.          return expiration.before(new Date());
84.      }
85.
86.      public String getSecret() {
87.          return secret;
88.      }
89.
90.      public void setSecret(String secret) {
91.          this.secret = secret;
92.      }
93.
94.      public long getExpire() {
```

```

95.         return expire;
96.     }
97.
98.     public void setExpire(long expire) {
99.         this.expire = expire;
100.    }
101.
102.    public String getHeader() {
103.        return header;
104.    }
105.
106.    public void setHeader(String header) {
107.        this.header = header;
108.    }
109. }

```

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：

```

1.  import io.renren.modules.api.interceptor.AuthorizationInterceptor;
2.  import
   io.renren.modules.api.resolver.LoginUserHandlerMethodArgumentResolver;
3.  import org.springframework.beans.factory.annotation.Autowired;
4.  import org.springframework.context.annotation.Configuration;
5.  import
   org.springframework.web.method.support.HandlerMethodArgumentResolver;
6.  import org.springframework.web.servlet.config.annotation.InterceptorReg
   istry;
7.  import org.springframework.web.servlet.config.annotation.WebMvcConfigur
   erAdapter;
8.
9.  @Configuration
10. public class WebMvcConfig extends WebMvcConfigurerAdapter {
11.     @Autowired
12.     private AuthorizationInterceptor authorizationInterceptor;
13.     @Autowired
14.     private LoginUserHandlerMethodArgumentResolver
loginUserHandlerMethodArgumentResolver;
15.
16.     @Override
17.     public void addInterceptors(InterceptorRegistry registry) {
18.         registry.addInterceptor(authorizationInterceptor).addPathPatter

```

```

18.         ns("/api/**");
19.     }
20.
21.     @Override
22.     public void addArgumentResolvers(List<HandlerMethodArgumentResolver
23. > argumentResolvers) {
24.         argumentResolvers.add(loginUserHandlerMethodArgumentResolver);
25.     }

```

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

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

```

1.
2.     import io.jsonwebtoken.Claims;
3.     import io.renren.common.exception.RRException;
4.     import io.renren.modules.app.utils.JwtUtils;
5.     import io.renren.modules.app.annotation.Login;
6.     import org.apache.commons.lang.StringUtils;
7.     import org.springframework.beans.factory.annotation.Autowired;
8.     import org.springframework.http.HttpStatus;
9.     import org.springframework.stereotype.Component;
10.    import org.springframework.web.method.HandlerMethod;
11.    import
12.    org.springframework.web.servlet.handler.HandlerInterceptorAdapter;
13.
14.    import javax.servlet.http.HttpServletRequest;
15.    import javax.servlet.http.HttpServletResponse;
16.
17.    /**
18.     * 权限(Token)验证
19.     */
20.    @Component
21.    public class AuthorizationInterceptor extends
22.    HandlerInterceptorAdapter {
23.
24.        @Autowired
25.        private JwtUtils jwtUtils;

```

```

23.
24.     public static final String USER_KEY = "userId";
25.
26.     @Override
27.     public boolean preHandle(HttpServletRequest request, HttpServletResponse
ponse response, Object handler) throws Exception {
28.         Login annotation;
29.         if(handler instanceof HandlerMethod) {
30.             annotation = ((HandlerMethod) handler).getMethodAnnotation(
Login.class);
31.         }else{
32.             return true;
33.         }
34.
35.         if(annotation == null){
36.             return true;
37.         }
38.
39.         //获取用户凭证
40.         String token = request.getHeader(HeaderConstants.getHeader());
41.         if(StringUtils.isBlank(token)){
42.             token = request.getParameter(HeaderConstants.getHeader());
43.         }
44.
45.         //凭证为空
46.         if(StringUtils.isBlank(token)){
47.             throw new RuntimeException(HeaderConstants.getHeader() + "不能为空", Http
Status.UNAUTHORIZED.value());
48.         }
49.
50.         Claims claims = HeaderConstants.getHeaderByToken(token);
51.         if(claims == null || HeaderConstants.isTokenExpired(claims.getExpirati
on())){
52.             throw new RuntimeException(HeaderConstants.getHeader() + "失效，请重新登录
", HttpStatus.UNAUTHORIZED.value());
53.         }
54.
55.         //设置userId到request里，后续根据userId，获取用户信息
56.         request.setAttribute(USER_KEY, Long.parseLong(claims.getSubject
()));
57.
58.         return true;
59.     }
60. }

```

我们可以发现，进入 `/api` 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？

```
1.  /**
2.   * 获取用户信息
3.   */
4.  @GetMapping("userInfo")
5.  public R userInfo(@LoginUser UserEntity user) {
6.      return R.ok().put("user", user);
7.  }
```

- 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：

```
1.  import io.renren.modules.api.annotation.LoginUser;
2.  import io.renren.modules.api.entity.UserEntity;
3.  import io.renren.modules.api.interceptor.AuthorizationInterceptor;
4.  import io.renren.modules.api.service.UserService;
5.  import org.springframework.beans.factory.annotation.Autowired;
6.  import org.springframework.core.MethodParameter;
7.  import org.springframework.stereotype.Component;
8.  import org.springframework.web.bind.support.WebDataBinderFactory;
9.  import org.springframework.web.context.request.NativeWebRequest;
10. import org.springframework.web.context.request.RequestAttributes;
11. import
12.     org.springframework.web.method.support.HandlerMethodArgumentResolver;
13. import org.springframework.web.method.support.ModelAndViewContainer;
14.
15. @Component
16. public class LoginUserHandlerMethodArgumentResolver implements
17.     HandlerMethodArgumentResolver {
18.
19.     @Autowired
20.     private UserService userService;
21.
22.     @Override
23.     public boolean supportsParameter(MethodParameter parameter) {
24.         //如果方法的参数是UserEntity，且参数前面有@LoginUser注解，则进入resolv
```

eArgument方法, 进行处理

```
22.         return parameter.getParameterType().isAssignableFrom(UserEntity
23.         .class) && parameter.hasParameterAnnotation(LoginUser.class);
24.     }
25.     @Override
26.     public Object resolveArgument(MethodParameter parameter, ModelAndView
27.     ewContainer container,
28.         NativeWebRequest request,
29.     WebDataBinderFactory factory) throws Exception {
30.         //获取用户ID, 之前设置进去的, 还有印象吧
31.         Object object = request.getAttribute(AuthorizationInterceptor.U
32.     SER_KEY, RequestAttributes.SCOPE_REQUEST);
33.         if(object == null){
34.             return null;
35.         }
36.
37.         //通过userId, 获取用户信息
38.         UserEntity user = userService.queryObject((Long)object);
39.
40.         //把当前用户信息, 设置到UserEntity参数的user对象里
41.         return user;
42.     }
43. }
```

4.6.3. Swagger生成接口文档

本项目, 支持Swagger注解的方式, 生成接口文档, 简化接口文档的维护工作

- 如要支持Swagger, 需要引入对应的Jar包, 我们使用的版本是2.7.0, 如下所示:

```
1.     <properties>
2.         <swagger.version>2.7.0</swagger.version>
3.     </properties>
4.
5.     <dependencies>
6.         <dependency>
7.             <groupId>io.springfox</groupId>
8.             <artifactId>springfox-swagger2</artifactId>
9.             <version>${swagger.version}</version>
10.        </dependency>
11.        <dependency>
```

```

12.         <groupId>io.springfox</groupId>
13.         <artifactId>springfox-swagger-ui</artifactId>
14.         <version>${swagger.version}</version>
15.     </dependency>
16. </dependencies>

```

- 怎么生成Swagger接口文档，很简单，只要方法加了 `@ApiOperation` 注解，就会自动生成接口文档，如下所示，其中， `response=UserEntity.class` 是返回的数据对象

```

1.  @Login
2.  @GetMapping("userInfo")
3.  @ApiOperation(value="获取用户信息", response=UserEntity.class)
4.  public R userInfo(@ApiIgnore @LoginUser UserEntity user) {
5.      return R.ok().put("user", user);
6.  }

```

- 我们再来看下Swagger的具体配置，其中配置了 `securitySchemes(security())`，目的就是为把登录的token，放到header头里，以免不能调用，需要认证的接口，如下所示：

```

1.  @Configuration
2.  @EnableSwagger2
3.  public class SwaggerConfig {
4.      @Bean
5.      public Docket createRestApi() {
6.          return new Docket(DocumentationType.SWAGGER_2)
7.              .apiInfo(apiInfo())
8.              .select()
9.              //加了ApiOperation注解的类，才生成接口文档
10.             .apis(RequestHandlerSelectors.withMethodAnnotation(ApiOperation.class))
11.             //包下的类，才生成接口文档
12.
13.             // .apis(RequestHandlerSelectors.basePackage("io.renren.controller"))
14.             .paths(PathSelectors.any())
15.             .build()
16.             .securitySchemes(security());
17.     }
18.
19.     private ApiInfo apiInfo() {
20.         return new ApiInfoBuilder()
21.             .title("人人开源")

```



```

21.         .description("renren-api模块接口文档")
22.         .termsOfServiceUrl("http://www.renren.io")
23.         .version("1.0")
24.         .build();
25.     }
26.
27.     private List<ApiKey> security() {
28.         return new ArrayList<
29.             new ApiKey("token", "token", "header")
30.         );
31.     }
32.
33. }

```

- 我们再来看一下，生成的接口文档，以及如何通过界面，调用接口的，如下所示：

Swagger UI interface for renren-api module. The interface shows three controllers: Api Register Controller, Api Test Controller, and Api Login Controller. Red boxes and arrows highlight specific steps:

- 步骤1、注册账号**: Points to the **POST /api/register** endpoint under the **注册接口 : Api Register Controller**.
- 步骤2、登录系统，获取token**: Points to the **POST /api/login** endpoint under the **登录接口 : Api Login Controller**.
- 步骤3、输入获取的token**: Points to the **Authorize** button in the top right corner.
- 步骤4、调用需要权限访问的接口**: Points to the **GET /api/userInfo** endpoint under the **测试接口 : Api Test Controller**.

Other endpoints visible include **POST /api/logout** (退出), **GET /api/notToken** (忽略Token验证测试), and **GET /api/userId** (获取用户ID).

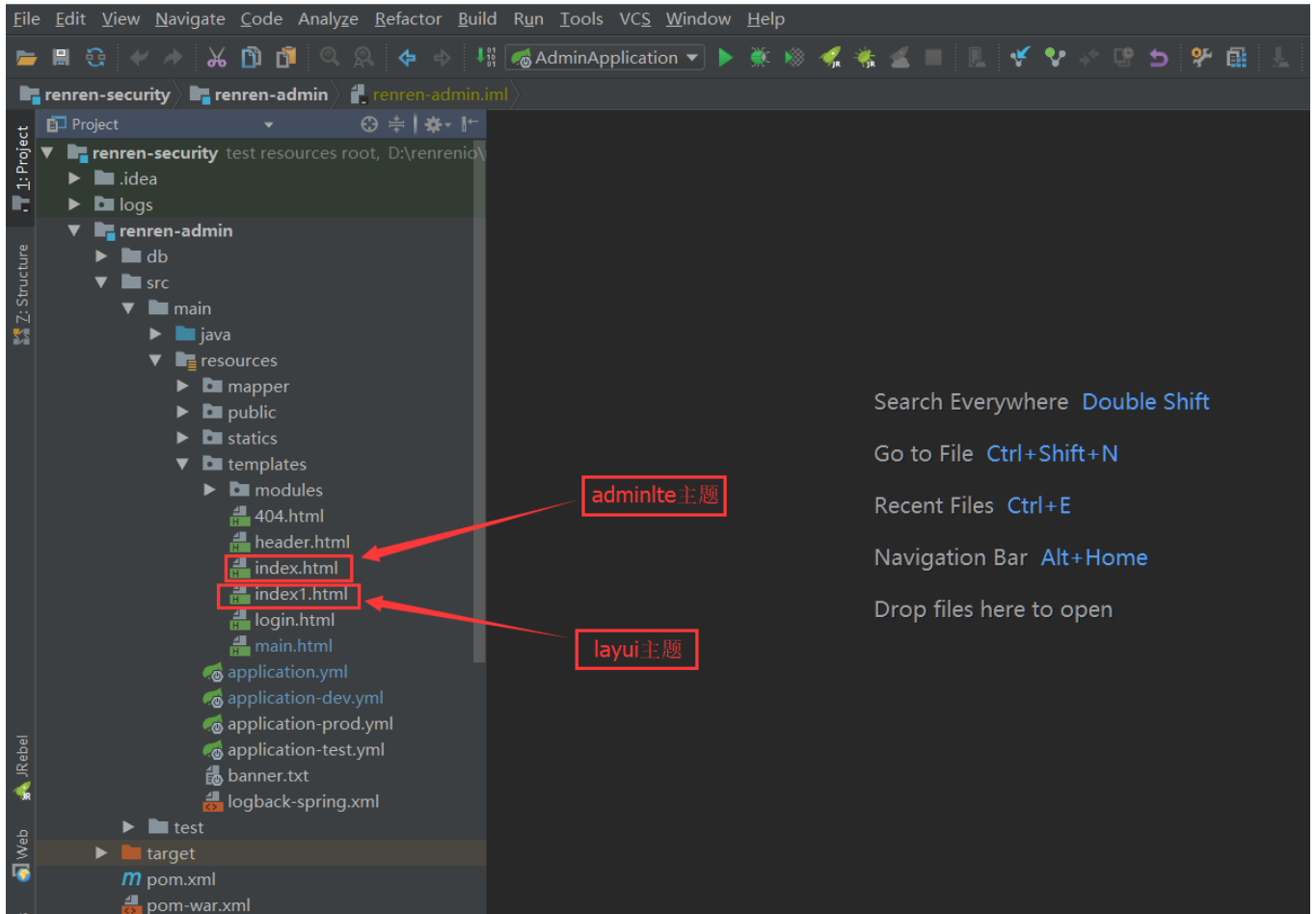
[BASE URL: /renren-api , API VERSION: 1.0]

5. 前端源码分析

前端提供了2套主题，adminlte主题及layui主题，默认使用adminlte主题。

5.1. 使用layui风格主题

- 系统提供了2套主题，其中，index.html为adminlte主题，index1.html为layui主题，如下所示：



- 默认是adminlte主题，如要修改成layui主题，只需删除index.html文件，并把index1.html文件名修改成index.html即可。

5.2. 数据列表、查询分析

- 下面分析 **参数管理** 相关的前端代码，包括数据列表、查询、新增、编辑、删除等功能，掌握了这部分内容，再开发其他前端页面，就会得心应手，界面如下：

<div> 首页 > 参数管理 </div>					
<input type="text" value="参数名"/>		<input type="button" value="查询"/>	<input type="button" value="+ 新增"/>	<input type="button" value="✎ 修改"/>	<input type="button" value="🗑 删除"/>
	<input type="checkbox"/>	ID	参数名	参数值	备注
1	<input type="checkbox"/>	2	system_version	v3.2.0	系统版本号
<div> <div> <div>⏮</div> <div>⏪</div> <div>1</div> <div>共 1 页</div> <div>⏩</div> <div>⏭</div> </div> <div> <div>10</div> <div>▼</div> </div> </div>					

- 下面的这段代码，用来展示数据列表的，定义了数据列表ID为jqGrid，分页ID为jqGridPager

```

1. <table id="jqGrid"></table>
2. <div id="jqGridPager"></div>

```

- 下面的代码，就是具体的数据列表，jqGrid、jqGridPager就是上面定义的id

```

1. $(function () {
2.     $("#jqGrid").jqGrid({
3.         url: baseUrl + 'sys/config/list',
4.         datatype: "json",
5.         colModel: [
6.             { label: 'ID', name: 'id', width: 30, key: true },
7.             { label: '参数名', name: 'key', sortable: false, width: 60
8.         },
9.             { label: '参数值', name: 'value', width: 100 },
10.            { label: '备注', name: 'remark', width: 80 }
11.        ],
12.        viewrecords: true,
13.        height: 385,
14.        rowNum: 10,
15.        rowList : [10,30,50],
16.        rownumbers: true,

```

```

16.         rownumWidth: 25,
17.         autowidth:true,
18.         multiselect: true,
19.         pager: "#jqGridPager",
20.         jsonReader : {
21.             root: "page.list",
22.             page: "page.currPage",
23.             total: "page.totalPage",
24.             records: "page.totalCount"
25.         },
26.         prmNames : {
27.             page:"page",
28.             rows:"limit",
29.             order: "order"
30.         },
31.         gridComplete:function(){
32.             //隐藏grid底部滚动条
33.             $("#jqGrid").closest(".ui-jqgrid-bdiv").css({ "overflow-x" : "
hidden" });
34.         }
35.     });
36. });

```

- 上面这些代码，就实现了数据列表的功能，下面就来看看，查询的实现，页面代码如下：

```

1.     <div v-show="showList">
2.         <div class="grid-btn">
3.             <div class="form-group col-sm-2">
4.                 <input type="text" class="form-control" v-model="q.key" @ke
yup.enter="query" placeholder="参数名">
5.             </div>
6.             <a class="btn btn-default" @click="query">查询</a>
7.         </div>
8.         <table id="jqGrid"></table>
9.         <div id="jqGridPager"></div>
10.    </div>

```

- 我们定义了查询参数key，点击查询，就会调用vue的query方法，其中一定要定义key变量，不然页面会报错，代码如下：

```

1.     var vm = new Vue({
2.         el: '#rrapp',
3.         data:{

```

```

4.         q:{
5.             key: null
6.         },
7.     },
8.     methods: {
9.         query: function () {
10.             vm.reload();
11.         },
12.         reload: function (event) {
13.             vm.showList = true;
14.             var page = $("#jqGrid").jqGrid('getGridParam','page');
15.             $("#jqGrid").jqGrid('setGridParam',{
16.                 postData:{'key': vm.q.key},
17.                 page:page
18.             }).trigger("reloadGrid");
19.         }
20.     }
21. });

```

5.2. 新增、编辑、删除功能

- 页面代码如下，下面具体分析

```

1.     <div id="rrapp" v-cloak>
2.         <div v-show="showList">
3.             <div class="grid-btn">
4.                 <a class="btn btn-primary" @click="add"><i class="fa fa-plu
5. s"></i>&nbsp;<新增</a>
6.                 <a class="btn btn-primary" @click="update"><i class="fa fa-
7. pencil-square-o"></i>&nbsp;<修改</a>
8.                 <a class="btn btn-primary" @click="del"><i class="fa fa-tra
9. sh-o"></i>&nbsp;<删除</a>
10.             </div>
11.         </div>
12.
13.         <div v-show="!showList" class="panel panel-default">
14.             <div class="panel-heading">{{title}}</div>
15.             <form class="form-horizontal">
16.                 <div class="form-group">
17.                     <div class="col-sm-2 control-label">参数名</div>
18.                     <div class="col-sm-10">

```

```

16.         <input type="text" class="form-control" v-
model="config.key" placeholder="参数名"/>
17.     </div>
18. </div>
19.     <div class="form-group">
20.         <div class="col-sm-2 control-label">参数值</div>
21.         <div class="col-sm-10">
22.             <input type="text" class="form-control" v-
model="config.value" placeholder="参数值"/>
23.         </div>
24.     </div>
25.     <div class="form-group">
26.         <div class="col-sm-2 control-label">备注</div>
27.         <div class="col-sm-10">
28.             <input type="text" class="form-control" v-
model="config.remark" placeholder="备注"/>
29.         </div>
30.     </div>
31.     <div class="form-group">
32.         <div class="col-sm-2 control-label"></div>
33.         <input type="button" class="btn btn-primary" @click="sa
veOrUpdate" value="确定"/>
34.         &nbsp;&nbsp;&nbsp;<input type="button" class="btn btn-warning
" @click="reload" value="返回"/>
35.     </div>
36. </form>
37. </div>
38. </div>

```

- 点击新增按钮，就会调用vue的add方法，用户填写表单后，点击保存，就调用saveOrUpdate方法，把数据提交到服务端，如下所示

```

1.  var vm = new Vue({
2.      el: '#rrapp',
3.      data: {
4.          showList: true,
5.          title: null,
6.          config: {}
7.      },
8.      methods: {
9.          add: function() {
10.              vm.showList = false;
11.              vm.title = "新增";
12.              vm.config = {};

```

```

13.         },
14.         saveOrUpdate: function (event) {
15.             var url = vm.config.id == null ? "sys/config/save" :
"sys/config/update";
16.             $.ajax({
17.                 type: "POST",
18.                 url: baseUrl + url,
19.                 contentType: "application/json",
20.                 data: JSON.stringify(vm.config),
21.                 success: function(r) {
22.                     if(r.code === 0) {
23.                         alert('操作成功', function(index) {
24.                             vm.reload();
25.                         });
26.                     }else{
27.                         alert(r.msg);
28.                     }
29.                 }
30.             });
31.         }
32.     }
33. });

```

- 点击修改按钮，就会调用vue的update方法，用户填写表单后，点击保存，就调用saveOrUpdate方法，把数据提交到服务端，如下所示

```

1.     var vm = new Vue({
2.         el: '#rrapp',
3.         data: {
4.             showList: true,
5.             title: null,
6.             config: {}
7.         },
8.         methods: {
9.             update: function () {
10.                 var id = getSelectedRow();
11.                 if(id == null){
12.                     return ;
13.                 }
14.
15.                 $.get(baseUrl + "sys/config/info/"+id, function(r) {
16.                     vm.showList = false;
17.                     vm.title = "修改";
18.                     vm.config = r.config;

```

```

19.         });
20.     },
21.     saveOrUpdate: function (event) {
22.         var url = vm.config.id == null ? "sys/config/save" :
23.         "sys/config/update";
24.         $.ajax({
25.             type: "POST",
26.             url: baseUrl + url,
27.             contentType: "application/json",
28.             data: JSON.stringify(vm.config),
29.             success: function(r) {
30.                 if(r.code === 0) {
31.                     alert('操作成功', function(index) {
32.                         vm.reload();
33.                     });
34.                 }else{
35.                     alert(r.msg);
36.                 }
37.             });
38.         }
39.     }
40. });

```

- 点击删除按钮，就会调用vue的del方法，询问用户 **确定要删除选中的记录**，如果确定，就会调用删除接口，删除数据，如下所示

```

1.     var vm = new Vue({
2.         el: '#rrapp',
3.         data: {
4.             showList: true,
5.             title: null,
6.             config: {}
7.         },
8.         methods: {
9.             del: function (event) {
10.                 var ids = getSelectedRows();
11.                 if(ids == null){
12.                     return ;
13.                 }
14.
15.                 confirm('确定要删除选中的记录?', function() {
16.                     $.ajax({
17.                         type: "POST",

```



```
18.         url: baseUrl + "sys/config/delete",
19.         contentType: "application/json",
20.         data: JSON.stringify(ids),
21.         success: function(r) {
22.             if (r.code == 0) {
23.                 alert('操作成功', function(index) {
24.                     vm.reload();
25.                 });
26.             } else {
27.                 alert(r.msg);
28.             }
29.         }
30.     });
31. });
32. }
33. }
34. });
```

6. 生产环境部署

6.1. 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：

```
1.     server:
2.         tomcat:
3.             uri-encoding: UTF-8
4.             max-threads: 1000
5.             min-spare-threads: 20
6.         connection-timeout: 5000
7.         port: 80
8.         context-path: /renren-admin
9.     ssl:
10.         key-store: classpath:..keystore
11.         key-store-type: JKS
12.         key-password: 123456
```

```
13.      key-alias: tomcat
```

- 当然，还可以指定jvm的内存大小，如下所示：

```
1.      java -Xms4g -Xmx4g -Xmn1g -server -jar renren-admin.jar
```

- 在windows下部署，只需打开cmd窗口，输入如下命令：

```
1.      java -jar renren-admin.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，还可以放到/etc/rc.local里，每次重启Linux时，项目都会自动起来：

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

6.2. war包部署

war包的部署，也很方便，只是不推荐这种方式。

- 在项目的子模块里，如：renren-admin目录下，执行【mvn clean package -f pom-war.xml】命令，就可以把renren-admin打成renren-admin.war包了
- 把生成的war包，放在tomcat【8.5+】的webapps目录下面，再启动tomcat即可

6.3. docker部署

- 安装docker环境

```
1.      #安装docker
2.      [root@mark ~]# curl -sSL https://get.docker.com/ | sh
3.
4.      #启动docker
5.      [root@mark ~]# service docker start
```

```
6.
7. #查看docker版本信息
8. [root@mark ~]# docker version
9. Client:
10.  Version:      17.07.0-ce
11.  API version:   1.31
12.  Go version:    go1.8.3
13.  Git commit:    8784753
14.  Built:         Tue Aug 29 17:42:01 2017
15.  OS/Arch:       linux/amd64
16.
17. Server:
18.  Version:      17.07.0-ce
19.  API version:   1.31 (minimum version 1.12)
20.  Go version:    go1.8.3
21.  Git commit:    8784753
22.  Built:         Tue Aug 29 17:43:23 2017
23.  OS/Arch:       linux/amd64
24.  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

```
1. #打包并构建项目镜像
2. [root@mark renren-admin]# mvn clean package docker:build
3. #省略打包log...
4. [INFO] Building image renren/admin
5. Step 1/6 : FROM java:8
6. ----> d23bdf5b1b1b
7. Step 2/6 : EXPOSE 8080
8. ----> Using cache
9. ----> 8e33aadb2c18
10. Step 3/6 : VOLUME /tmp
11. ----> Using cache
12. ----> c5dc0c509062
13. Step 4/6 : ADD renren-admin.jar /app.jar
14. ----> 831bc3ca84bc
15. Step 5/6 : RUN bash -c 'touch /app.jar'
16. ----> Running in fe3ef9343e4c
17. ----> b3d6dd6fc297
18. Removing intermediate container fe3ef9343e4c
19. Step 6/6 : ENTRYPOINT java -jar /app.jar
20. ----> Running in 89adce4ae167
21. ----> a4ae60970a77
```

```

22. Removing intermediate container 89adce4ae167
23. ProgressMessage{id=null, status=null, stream=null, error=null,
    progress=null, progressDetail=null}
24. Successfully built a4ae60970a77
25. Successfully tagged renren/admin:latest
26.
27. #查看镜像
28. [root@mark renren-admin]# docker images
29. REPOSITORY          TAG                 IMAGE ID            CREATED
    SIZE
30. renren/admin         latest             a4ae60970a77       14 seconds
    ago               714MB
31. java                 8                 d23bdf5b1b1b       7 months ago
    643MB

```

- 安装docker-compose , 用来管理容器

```

1. #下载地址:https://github.com/docker/compose/releases
2.
3. #下载docker-compose
4. [root@mark renren-admin]# curl -L
    https://github.com/docker/compose/releases/download/1.16.1/docker-
    compose-`uname -s`-`uname -m` > /usr/local/bin/docker-compose
5.
6. #增加可执行权限
7. [root@mark renren-admin]# chmod +x /usr/local/bin/docker-compose
8.
9. #查看版本信息
10. [root@mark renren-admin]# docker-compose version
11. docker-compose version 1.16.1, build 6dlac21
12. docker-py version: 2.5.1
13. CPython version: 2.7.13
14. 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:

```

1. #启动项目
2. [root@mark renren-admin]# docker-compose up -d
3. Creating network "renrenadmin_default" with the default driver

```

```
4. Creating renrenadmin_campus_1 ...
5. Creating renrenadmin_campus_1 ... done
6.
7. #查看启动的容器
8. [root@mark renren-admin]# docker ps
9. CONTAINER ID          IMAGE                COMMAND                  CREATED
   STATUS                PORTS               NAMES
10. f4e3fcdd8dd4          renren/admin        "java -jar /app.jar"    55 seco
   nds ago              Up 3 seconds        0.0.0.0:8080->8080/tcp
   renrenadmin_renren-admin_1
11.
12. #停掉并删除, docker-compose管理的容器
13. [root@mark renren-admin]# docker-compose down
14. Stopping renrenadmin_renren-admin_1 ... done
15. Removing renrenadmin_renren-admin_1 ... done
16. Removing network renrenadmin_default
```
