renren-fast开发文档2.0_完整版

1. 介绍

1.1. 项目描述

- renren-fast是一个轻量级的`Spring Boot`快速开发平台，能快速开发项目并交付【接私活利器】
- 完善的`XSS`防范及脚本过滤，彻底杜绝`XSS`攻击
- 实现前后端分离，通过`token`进行数据交互
- 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】

1.2. 获取帮助

- 后台地址：https://gitee.com/renrenio/renren-fast
- element-ui地址：https://github.com/daxiongYang/renren-fast-vue
- adminlte地址：https://gitee.com/renrenio/renren-fast-adminlte
- 代码生成器：https://gitee.com/renrenio/renren-generator
- 官方社区：http://www.renren.io
- 如需关注项目最新动态或担心以后找不到项目，可以Watch、Star项目，同时也是对项目最好的支持

1.3. 官方QQ群

- 高级群：324780204(大牛云集，跟大牛学习新技能)
- 普通群：145799952(学习交流，互相解答各种疑问)

---

2. 入门

2.1. 快速开始

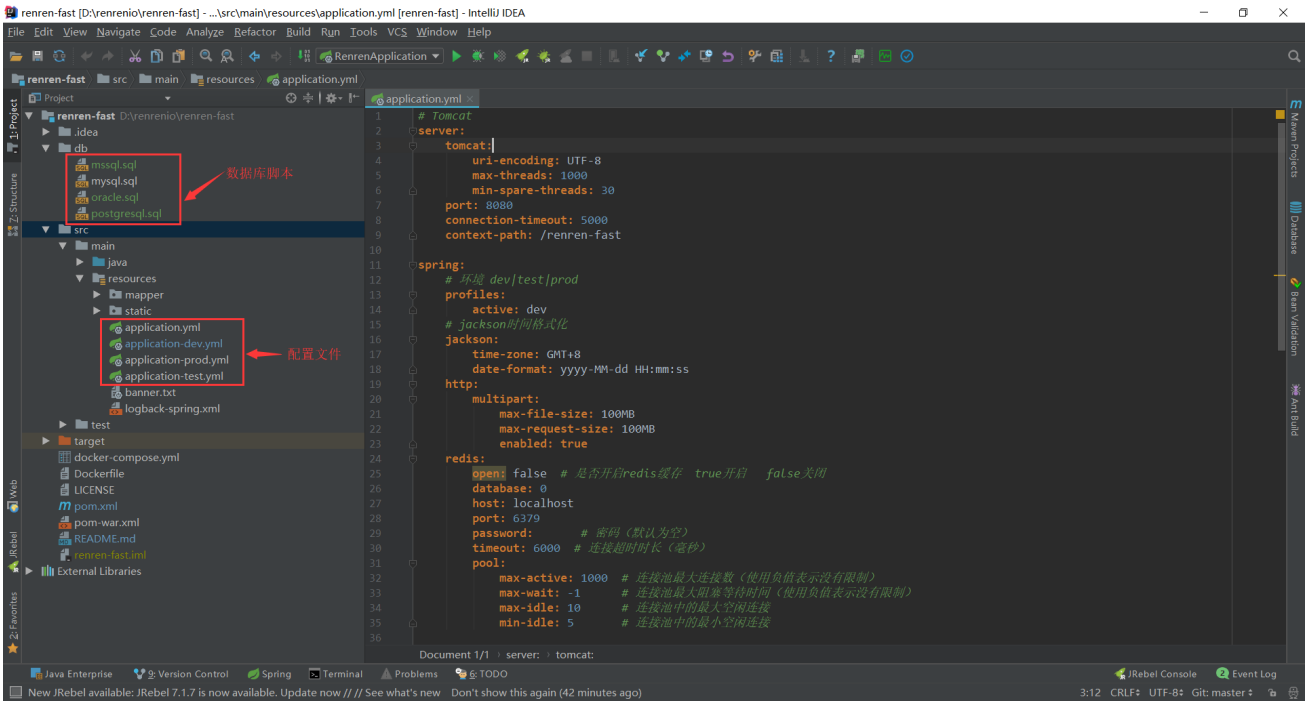
> 本项目是前后端分离的，需要先部署好后端，再部署前端页面，才能看到项目的页面效果。

2.1.1. 后端部署

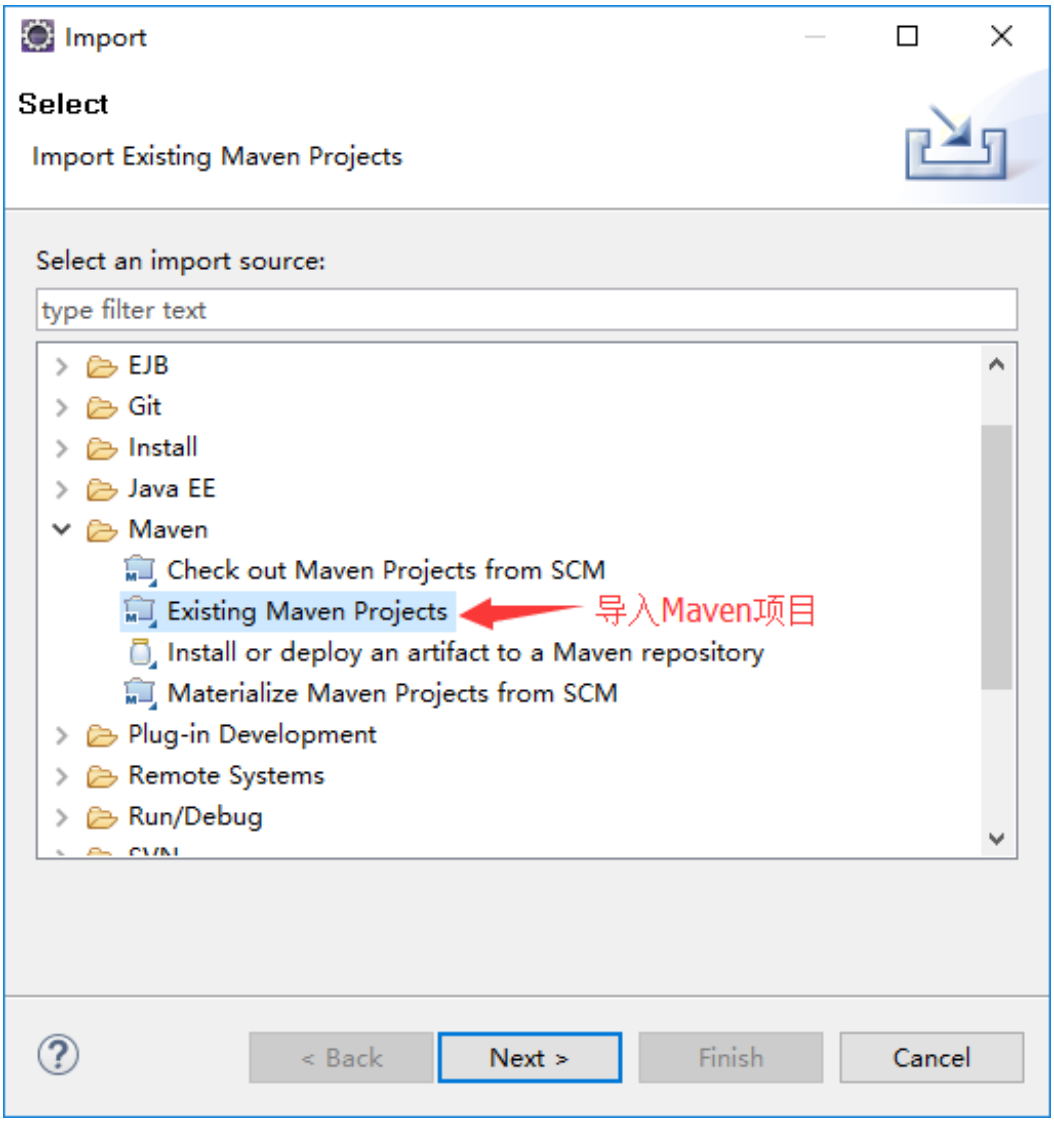
- 环境要求`JDK1.8`、`Tomcat8.0+`、`MySQL5.5+`
- 通过`git`，下载renren-fast源码，如下：

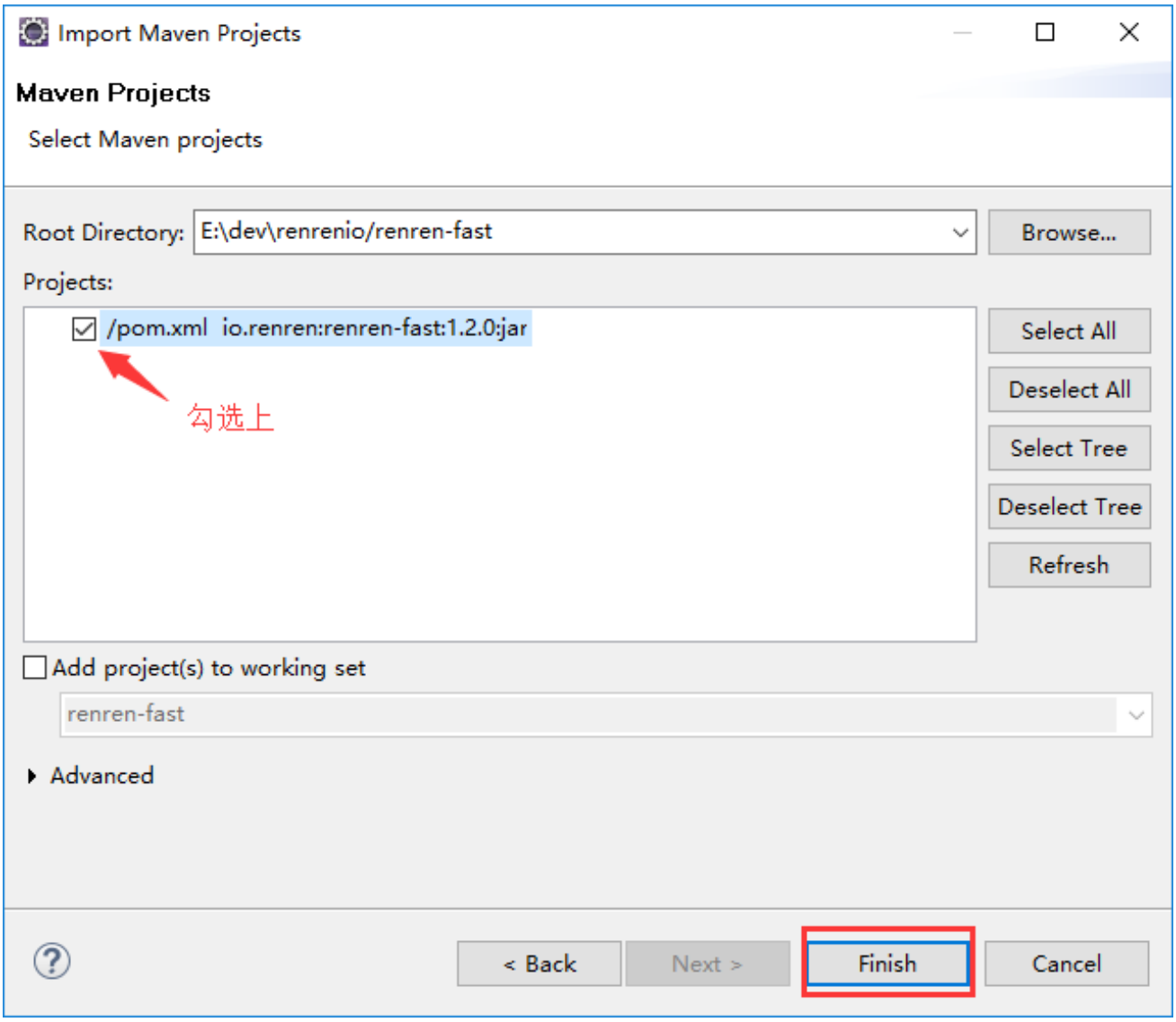
```
git clone https://gitee.com/renrenio/renren-fast.git
```

- 用`idea`打开项目，`File -> Open`如下图：



- 用`eclipse`打开项目，如下图：


Import
Select
Import Existing Maven Projects
Select an import source:
type filter text
EJB
Git
Install
Java EE
Maven
Check out Maven Projects from SCM
Existing Maven Projects
导入Maven项目
Install or deploy an artifact to a Maven repository
Materialize Maven Projects from SCM
Plug-in Development
Remote Systems
Run/Debug
< Back
Next >
Finish
Cancel



- 创建数据库`renren_fast`，数据库编码为`UTF-8`
- 执行`db/mysql.sql`文件，初始化数据（默认支持MySQL）
- 修改`application-dev.yml`，更新MySQL账号和密码
- 运行`io.renren.RenrenApplication.java`的`main`方法，则可启动项目
- Swagger路径：http://localhost:8080/renren-fast/swagger/index.html
- Swagger注解路径：http://localhost:8080/renren-fast/swagger-ui.html

---

2.1.2. 前端部署

> 本项目提供了element-ui及adminlte两套主题，推荐使用element-ui主题【基于vue、

> element-ui构建开发】
> 欢迎star或fork前端Git库，方便日后寻找，及二次开发

- element-ui主题

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

# 安装依赖
npm install

# 启动服务
npm run dev
```

- adminlte主题

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

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

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

# 启动Nginx后，访问如下路径即可
http://localhost
```

- 登录的账号密码：admin/admin

2.1.3. 配置文件

```
# Tomcat
server:
    tomcat:
```

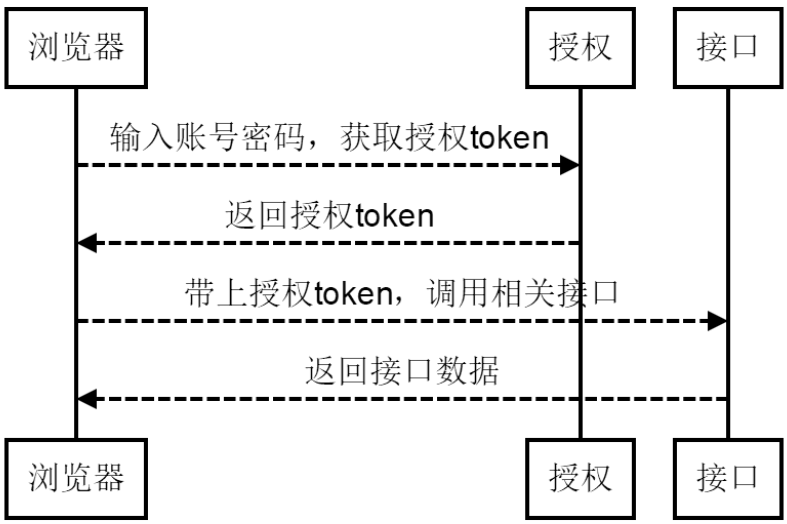
```yaml
        uri-encoding: UTF-8
        max-threads: 1000
        min-spare-threads: 30
    port: 8080  #tomcat端口号
    context-path: /renren-fast

spring:
    # 环境 dev|test|prod
    profiles:
        active: dev
    # jackson时间格式化
    jackson:
        time-zone: GMT+8
        date-format: yyyy-MM-dd HH:mm:ss
    http:
        multipart:
            max-file-size: 100MB
            max-request-size: 100MB
            enabled: true
    redis:
        open: false  # 是否开启redis缓存  true开启   false关闭
        database: 0
        host: localhost
        port: 6379
        password:        # 密码（默认为空）
        timeout: 6000  # 连接超时时长（毫秒）
        pool:
            max-active: 1000  # 连接池最大连接数（使用负值表示没有限制）
            max-wait: -1      # 连接池最大阻塞等待时间（使用负值表示没有限制）
            max-idle: 10      # 连接池中的最大空闲连接
            min-idle: 5       # 连接池中的最小空闲连接

#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: 0
    #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

renren:
    # APP模块，是通过jwt认证的，如果要使用APP模块，则需要修改【加密秘钥】
    jwt:
        # 加密秘钥
        secret: f4e2e52034348f86b67cde581c0f9eb5[www.renren.io]
        # token有效时长，7天，单位秒
        expire: 604800
        header: token

# 数据库配置
spring:
    datasource:
        type: com.alibaba.druid.pool.DruidDataSource
        driverClassName: com.mysql.jdbc.Driver
        druid:
            first:  #数据源1
                url: jdbc:mysql://localhost:3306/renren_fast?allowMultiQueries=true&useUnicode=true&characterEncoding=UTF-8
                username: renren
                password: 123456
            second:  #数据源2
                url: jdbc:mysql://localhost:3306/bdshop?allowMultiQueries=true&useUnicode=true&characterEncoding=UTF-8
                username: renren
                password: 123456
            initial-size: 10
            max-active: 100
```

```
91.                 min-idle: 10
92.                 max-wait: 60000
93.
```

2.2. 数据交互

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



3. 项目实战

3.1. 功能描述

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

- 我们先建一个商品表tb_goods，表结构如下所示：

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

- 接下来，我们利用代码生成器，帮我们生成基础代码，可以大大的节省重复工作量，代码生成器Git地址：https://gitee.com/renrenio/renren-generator

---

3.2. 使用代码生成器

- 代码生成器是Spring Boot开发的项目，通过git clone把项目下载到本地后，直接运行main方法，就可以通过http://localhost打开，我们先来看看配置文件，如下所示：

```
#配置文件：generator.properties

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

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

```
bit=Boolean

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

date=Date
datetime=Date
timestamp=Date
```

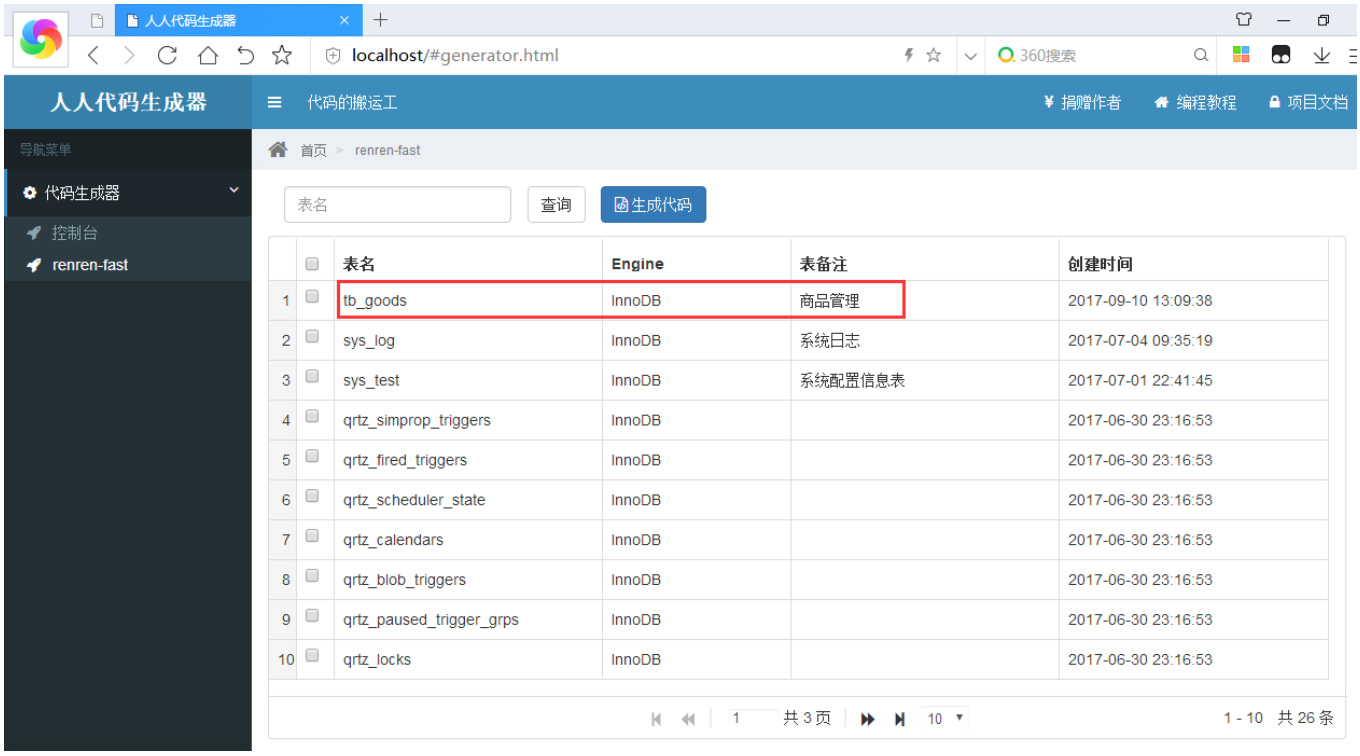
上面的配置文件，可以配置包名、作者信息、表前缀、类型转换。其中，类型转换是指，MySQL中的类型与JavaBean中的类型，是怎么一个对应关系。如果有缺少的类型，可自行在generator.properties文件中补充。

- 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。

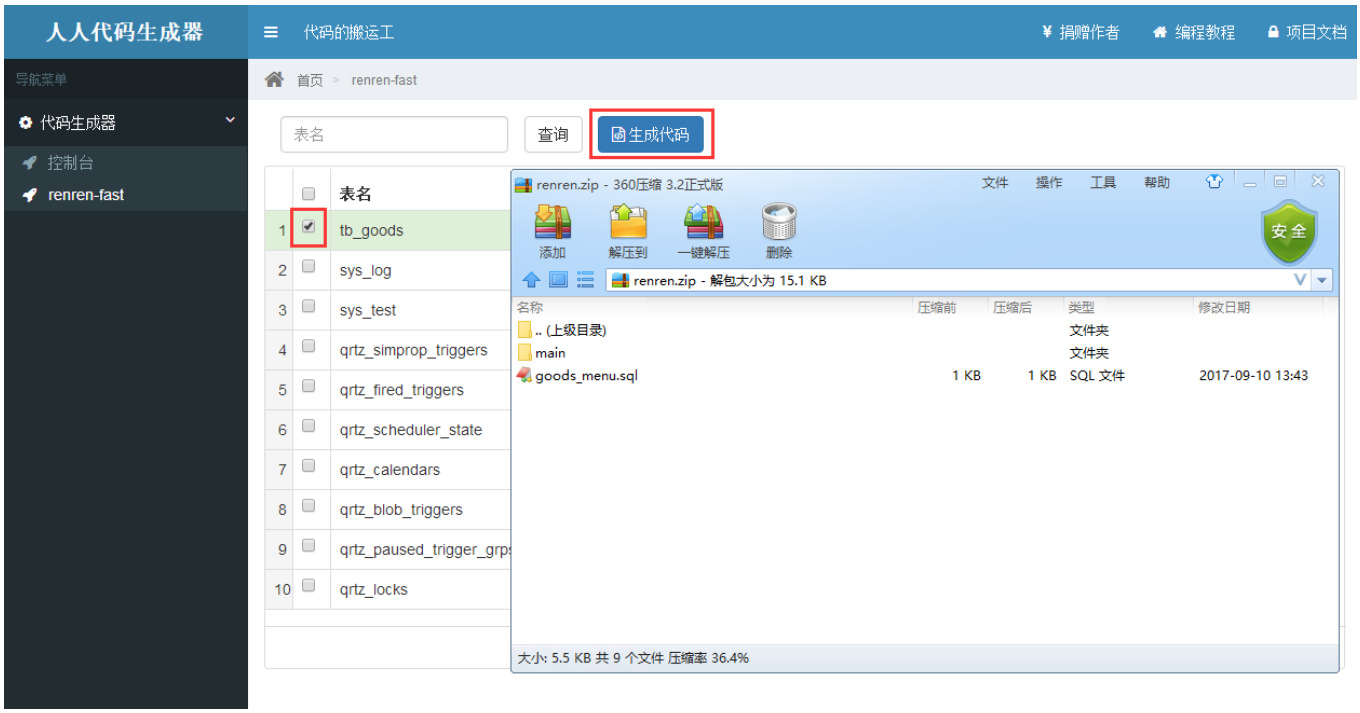
```
# Tomcat
server:
    tomcat:
        max-threads: 10
        min-spare-threads: 10
    port: 80

# mysql
spring:
    datasource:
        type: com.alibaba.druid.pool.DruidDataSource
        driverClassName: com.mysql.jdbc.Driver
        url: jdbc:mysql://localhost:3306/renren_fast?useUnicode=true&characterEncoding=UTF-8
        username: renren
        password: 123456
    jackson:
        time-zone: GMT+8
        date-format: yyyy-MM-dd HH:mm:ss
    resources:
            static-locations: classpath:/static/,classpath:/views/

# Mybatis配置
```

```
23.  mybatis:
24.      mapperLocations: classpath:mapper/**/*.xml
```

- 在数据库renren_fast中，执行建表语句，创建tb_goods表，再启动renren-generator项目(运行RenrenApplication.java的main方法即可)，如下所示：



- 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：

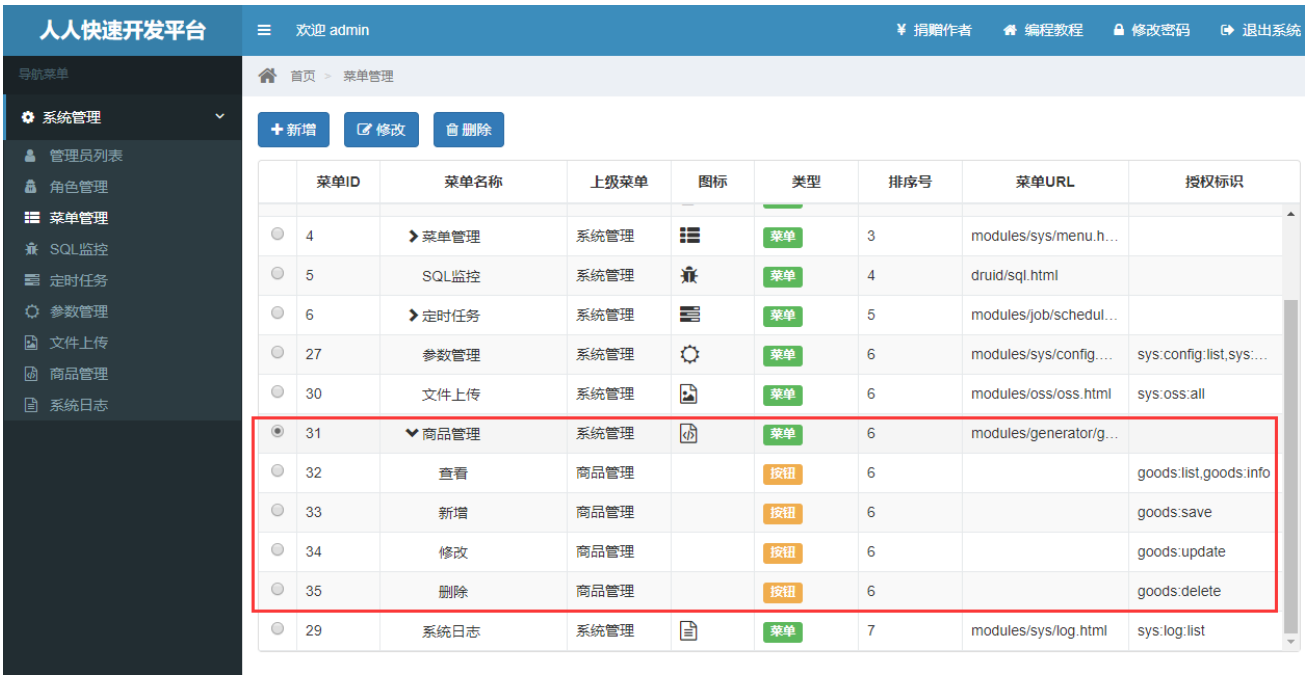


3.3. 测试项目

- 我们先在renren_fast库中，执行goods_menu.sql语句，这个SQL是生成菜单的，SQL语句如下所示：

```
-- 菜单SQL
INSERT INTO `sys_menu` (`parent_id`, `name`, `url`, `perms`, `type`, `icon`, `order_num`)
    VALUES ('1', '商品管理', 'modules/generator/goods.html', NULL, '1', 'fa fa-file-code-o', '6');

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

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

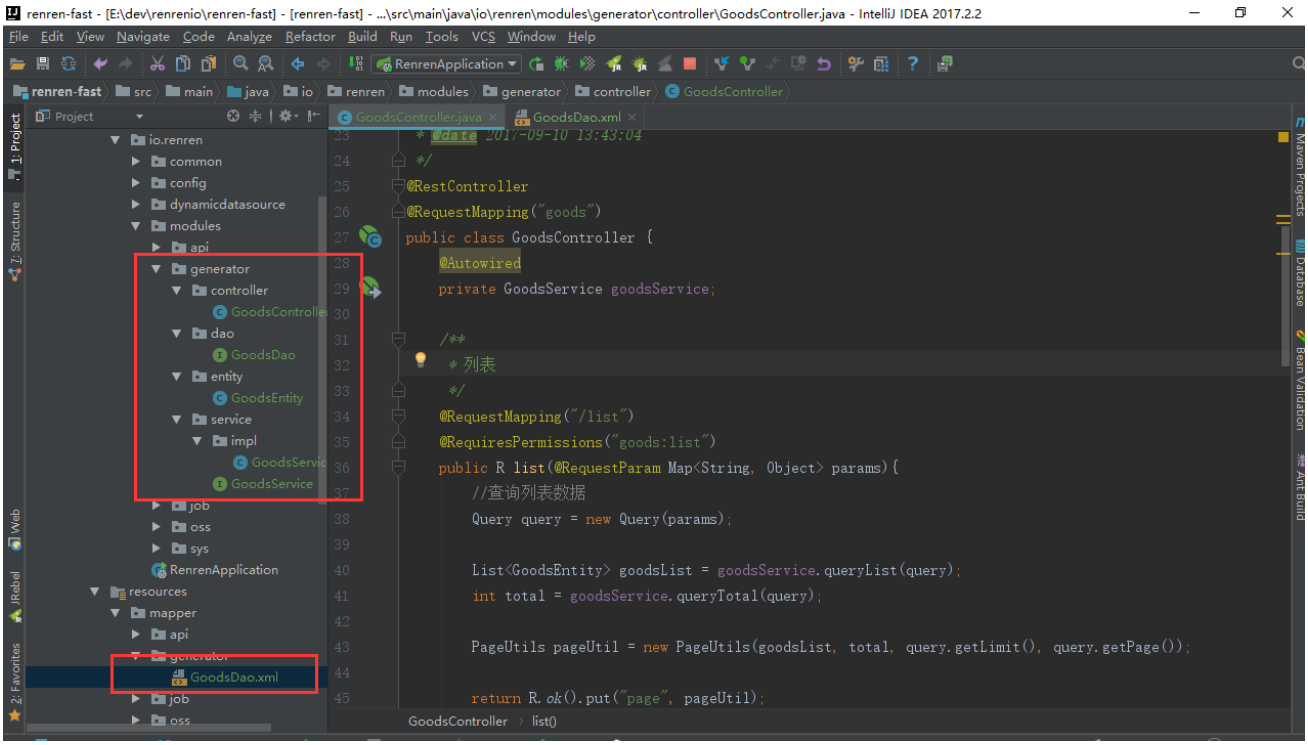
- 然后把生成的main目录覆盖renren-fast的main目录，再重启renren-fast，效果如下所示：


人人快速开发平台
欢迎 admin
捐赠作者
编程教程
修改密码
退出系统
导航菜单
首页 > 菜单管理
系统管理
管理员列表
角色管理
菜单管理
SQL监控
定时任务
参数管理
文件上传
商品管理
系统日志
新增
修改
删除
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
4 菜单管理 系统管理 菜单 3 modules/sys/menu.h...
5 SQL监控 系统管理 菜单 4 druid/sql.html
6 定时任务 系统管理 菜单 5 modules/job/schedul...
27 参数管理 系统管理 菜单 6 modules/sys/config.... sys:config:list,sys:...
30 文件上传 系统管理 菜单 6 modules/oss/oss.html sys:oss:all
31 商品管理 系统管理 菜单 6 modules/generator/g...
32 查看 商品管理 按钮 6 goods:list,goods:info
33 新增 商品管理 按钮 6 goods:save
34 修改 商品管理 按钮 6 goods:update
35 删除 商品管理 按钮 6 goods:delete
29 系统日志 系统管理 菜单 7 modules/sys/log.html sys:log:list


人人快速开发平台
欢迎 admin
捐赠作者
编程教程
修改密码
退出系统
导航菜单
首页 > 商品管理
系统管理
管理员列表
角色管理
菜单管理
SQL监控
定时任务
参数管理
文件上传
商品管理
系统日志
新增
修改
删除
goodsId
商品名
介绍
价格
数量


人人快速开发平台
欢迎 admin
捐赠作者
导航菜单
首页 > 商品管理
系统管理
管理员列表
角色管理
菜单管理
SQL监控
定时任务
参数管理
文件上传
商品管理
系统日志
新增
商品名
测试商品
介绍
测试商品介绍
价格
9.99
数量
100
确定
返回

- 我们就操作了这几步，就把查询、新增、修改、删除就完成了，是不是很快啊，下面就是我们才生成的代码，如下所示：



4. 源码分析

4.1. 多数据源

> 多数据源的应用场景，主要针对跨多个MySQL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。

4.1.1. 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：

```
spring:
    datasource:
        type: com.alibaba.druid.pool.DruidDataSource
        driverClassName: com.mysql.jdbc.Driver
        druid:
            first:  #数据源1
                url: jdbc:mysql://192.168.1.10:3306/renren_fast?allowMultiQueries=true&useUnicode=true&characterEncoding=UTF-8
                username: renren
                password: 123456
            second:  #数据源2
                url: jdbc:mysql://192.168.1.11:3306/order?allowMultiQueries=true&useUnicode=true&characterEncoding=UTF-8
                username: root
                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：

```
public class DynamicDataSource extends AbstractRoutingDataSource {
    private static final ThreadLocal<String> contextHolder = new ThreadLocal<>();

    public DynamicDataSource(DataSource defaultTargetDataSource, Map<String, DataSource> targetDataSources) {
        //设置默认数据源
        super.setDefaultTargetDataSource(defaultTargetDataSource);
        super.setTargetDataSources(new HashMap<>(targetDataSources));
        super.afterPropertiesSet();
    }

    @Override
    protected Object determineCurrentLookupKey() {
        //获取数据源，没有指定，则为默认数据源
        return getDataSource();
    }
```

```
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
    @Bean
    @Primary
    public DynamicDataSource dataSource(DataSource firstDataSource, DataSource secondDataSource) {
        Map<String, DataSource> targetDataSources = new HashMap<>();
        targetDataSources.put(DataSourceNames.FIRST, firstDataSource);
        targetDataSources.put(DataSourceNames.SECOND, secondDataSource);
        return new DynamicDataSource(firstDataSource, targetDataSources);
    }
}
```

- 步骤4，通过注解，实现多数据源，如下所示：

```
/**
 * 多数据源注解
 */
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface DataSource {
    String name() default "";
}


/**
 * 多数据源，切面处理类
 */
@Aspect
@Component
public class DataSourceAspect implements Ordered {
    protected Logger logger = LoggerFactory.getLogger(getClass());


@Pointcut("@annotation(io.renren.datasources.annotation.DataSource)")
    public void dataSourcePointCut() {

    }

    @Around("dataSourcePointCut()")
    public Object around(ProceedingJoinPoint point) throws Throwable {
        MethodSignature signature = (MethodSignature) point.getSignatur
```

```
e();
        Method method = signature.getMethod();

        DataSource ds = method.getAnnotation(DataSource.class);
        if(ds == null){
            DynamicDataSource.setDataSource(DataSourceNames.FIRST);
            logger.debug("set datasource is " + DataSourceNames.FIRST);
        }else {
            DynamicDataSource.setDataSource(ds.name());
            logger.debug("set datasource is " + ds.name());
        }

        try {
            return point.proceed();
        } finally {
            DynamicDataSource.clearDataSource();
            logger.debug("clean datasource");
        }
    }

    @Override
    public int getOrder() {
        return 1;
    }
}
```

4.1.2. 测试多数据源

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class DynamicDataSourceTest {
    @Autowired
    private DataSourceTestService dataSourceTestService;

    @Test
    public void test(){
        //数据源1
        UserEntity user = dataSourceTestService.queryObject(1L);
        System.out.println(ToStringBuilder.reflectionToString(user));

        //数据源2
        UserEntity user2 = dataSourceTestService.queryObject2(1L);
        System.out.println(ToStringBuilder.reflectionToString(user2));
```

```
16.
17.         //数据源1
18.         UserEntity user3 = dataSourceTestService.queryObject(1L);
19.         System.out.println(ToStringBuilder.reflectionToString(user3));
20.     }
21. }
22.
23.
24.
25. @Service
26. public class DataSourceTestService {
27.     @Autowired
28.     private UserService userService;
29.
30.     public UserEntity queryObject(Long userId){
31.         return userService.queryObject(userId);
32.     }
33.
34.     @DataSource(name = DataSourceNames.SECOND)
35.     public UserEntity queryObject2(Long userId){
36.         return userService.queryObject(userId);
37.     }
38. }
```

4.1.3. 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_fast?allowMu
    ltiQueries=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
                username: root
                password: root

            #数据源4、5、6......
```

```
public interface DataSourceNames {
    String FIRST = "first";
    String SECOND = "second";
    String THIRD = "third";
}


@Configuration
public class DynamicDataSourceConfig {

    @Bean
    @ConfigurationProperties("spring.datasource.druid.first")
    public DataSource firstDataSource(){
        return DruidDataSourceBuilder.create().build();
    }

    @Bean
    @ConfigurationProperties("spring.datasource.druid.second")
    public DataSource secondDataSource(){
        return DruidDataSourceBuilder.create().build();
    }

    @Bean
    @ConfigurationProperties("spring.datasource.druid.third")
    public DataSource thirdDataSource(){
        return DruidDataSourceBuilder.create().build();
    }

    @Bean
    @Primary
    public DynamicDataSource dataSource(DataSource firstDataSource, DataSource secondDataSource, DataSource thirdDataSource) {
        Map<String, DataSource> targetDataSources = new HashMap<>();
        targetDataSources.put(DataSourceNames.FIRST, firstDataSource);
        targetDataSources.put(DataSourceNames.SECOND, secondDataSource);
        targetDataSources.put(DataSourceNames.THIRD, thirdDataSource);
        return new DynamicDataSource(firstDataSource, targetDataSources
```

```
        );
37.         }
38.     }
```

4.1.4. 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_fast?allowMultiQuer
    ies=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

4.2. 核心模块

4.2.1. 前后端分离

> 要实现前后端分离，需要考虑以下2个问题：
> 1. 项目不再基于session了，如何知道访问者是谁？
> 2. 如何确认访问者的权限？

- 前后端分离，一般都是通过token实现，本项目也是一样；用户登录时，生成token及token过期时间，token与用户是一一对应关系，调用接口的时候，把token放到header或请求参数中，服务端就知道是谁在调用接口，登录如下所示：

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

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

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

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

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

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

    //账号锁定
    if(user.getStatus() == 0){
        return R.error("账号已被锁定,请联系管理员");
    }

    //生成token，并保存到数据库
```

```
        R r = sysUserTokenService.createToken(user.getUserId());
        return r;
    }


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

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

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

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

            //更新token
            update(tokenEntity);
        }

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

        return r;
    }
```

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

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

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

步骤1，所有请求全部拒绝访问

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

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

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

        return false;
    }

    return executeLogin(request, response);
}

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

    //如果header中不存在token，则从参数中获取token
    if(StringUtils.isBlank(token)){
```

```
            token = httpRequest.getParameter("token");
        }

        return token;
    }
```

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

```
//AuthenticatingFilter类中的方法
protected boolean executeLogin(ServletRequest request, ServletResponse response) throws Exception {
        AuthenticationToken token = createToken(request, response);
        if (token == null) {
            String msg = "createToken method implementation returned null. A valid non-null AuthenticationToken " +
                    "must be created in order to execute a login attempt.";
            throw new IllegalStateException(msg);
        }
        try {
            Subject subject = getSubject(request, response);
            subject.login(token);
            return onLoginSuccess(token, subject, request, response);
        } catch (AuthenticationException e) {
            return onLoginFailure(token, e, request, response);
        }
    }


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

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

    return new OAuth2Token(token);
```

```
}

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

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

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

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

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

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

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

    //查询用户信息
    SysUserEntity user =
shiroService.queryUser(tokenEntity.getUserId());
    //账号锁定
```

```
        if(user.getStatus() == 0){
            throw new LockedAccountException("账号已被锁定,请联系管理员");
        }

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

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

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

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

        }

        return false;
    }
```

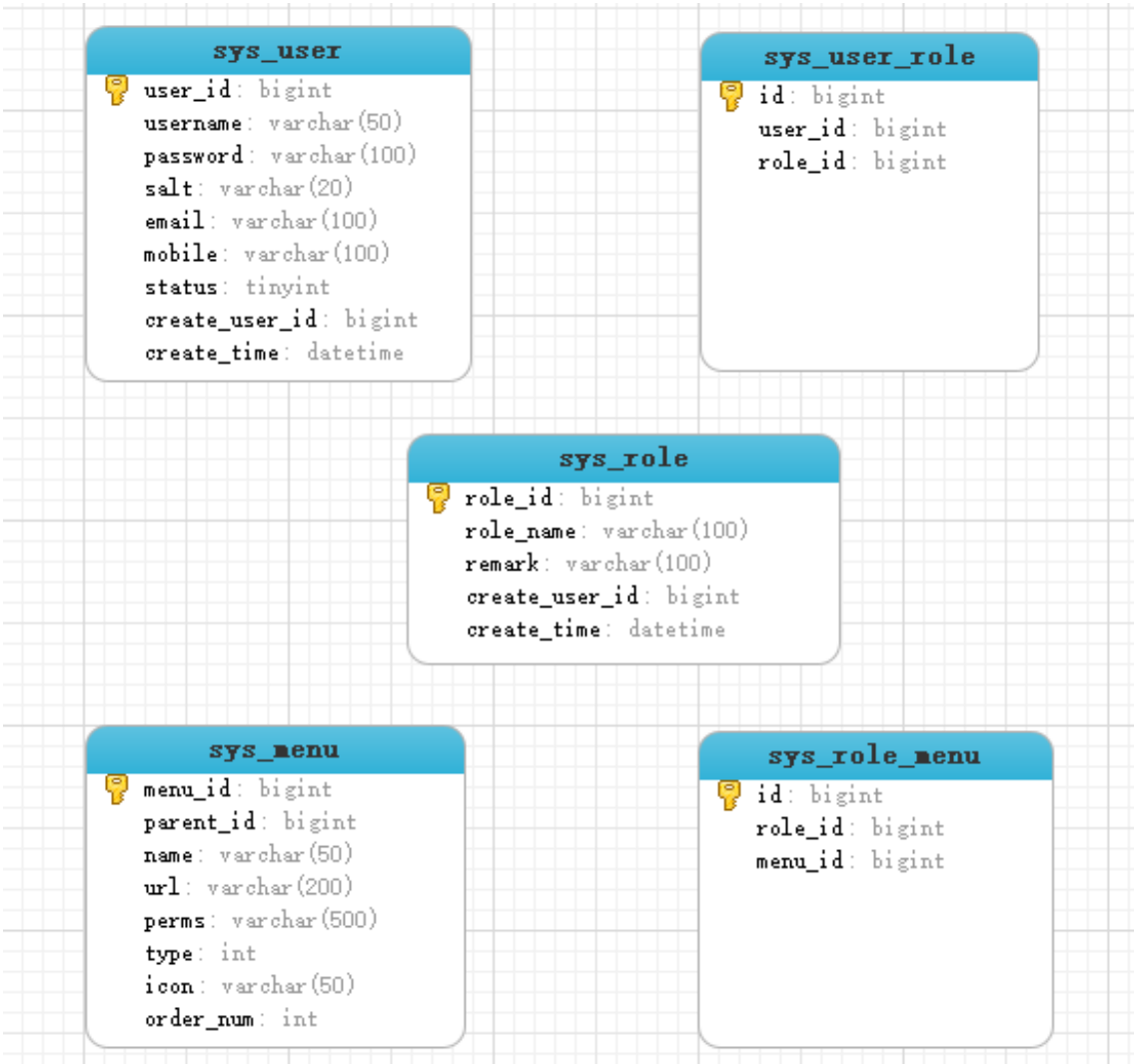
步骤6，登录成功后，则调用doGetAuthorizationInfo方法，查询用户的权限，再调用具体的接口，整个流程结束

```
    @Override
    protected AuthorizationInfo doGetAuthorizationInfo(PrincipalCollection principals) {
        SysUserEntity user = (SysUserEntity)principals.getPrimaryPrincipal();
        Long userId = user.getUserId();

```

```
        //用户权限列表
        Set<String> permsSet = shiroService.getUserPermissions(userId);

        SimpleAuthorizationInfo info = new SimpleAuthorizationInfo();
        info.setStringPermissions(permsSet);
        return info;
    }
```

4.2.2. 权限设计思路

> 权限相关的表结构，如下图所示：



1. sys_user[用户]表，保存用户相关数据，通过sys_user_role[用户与角色关联]表，与sys_role[角色]表关联；sys_menu[菜单]表通过sys_role_menu[菜单与角色关联]表，与

sys_role[角色]表关联

2. sys_menu表，保存菜单相关数据，并在perms字段里，保存了shiro的权限标识，也就是说，拥有此菜单，就拥有perms字段里的所有权限，比如，某用户拥有的菜单权限标识`sys:user:info`，就可以访问下面的方法

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

}
```

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

```
@Configuration
public class ShiroConfig {

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

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

        return securityManager;
    }

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

```
();
        shiroFilter.setSecurityManager(securityManager);

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

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

        return shiroFilter;
    }

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

    @Bean
    public DefaultAdvisorAutoProxyCreator
defaultAdvisorAutoProxyCreator() {
        DefaultAdvisorAutoProxyCreator proxyCreator = new DefaultAdviso
rAutoProxyCreator();
        proxyCreator.setProxyTargetClass(true);
        return proxyCreator;
    }

    @Bean
    public AuthorizationAttributeSourceAdvisor
authorizationAttributeSourceAdvisor(SecurityManager securityManager) {
        AuthorizationAttributeSourceAdvisor advisor = new Authorization
AttributeSourceAdvisor();
        advisor.setSecurityManager(securityManager);
        return advisor;
```

```
    }

}
```

4.2.3. XSS脚本过滤

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

```
public class XssFilter implements Filter {

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

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

    @Override
    public void destroy() {
    }

}

@Configuration
public class FilterConfig {
    @Bean
    public FilterRegistrationBean xssFilterRegistration() {
        FilterRegistrationBean registration = new FilterRegistrationBean();
        registration.setDispatcherTypes(DispatcherType.REQUEST);
```

```
            registration.setFilter(new XssFilter());
            registration.addUrlPatterns("/*");
            registration.setName("xssFilter");
            registration.setOrder(Integer.MAX_VALUE);
            return registration;
        }
    }
```

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

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

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

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

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

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

```
                public boolean isFinished() {
                    return true;
                }

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

                @Override
                public void setReadListener(ReadListener readListener) {

                }

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

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

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

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

        @Override
        public Map<String,String[]> getParameterMap() {
```

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

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

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

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

    /**
     * 获取最原始的request
     */
    public static HttpServletRequest getOrgRequest(HttpServletRequest request) {
        if (request instanceof XssHttpServletRequestWrapper) {
            return ((XssHttpServletRequestWrapper) request).getOrgRequest();
        }

        return request;
    }

```

```
}
```

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

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

4.2.4. SQL注入

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

```
public class SQLFilter {

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

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

```

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

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

            return str;
        }
    }
```

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

```
public class Query extends LinkedHashMap<String, Object> {
    private static final long serialVersionUID = 1L;
    //当前页码
    private int page;
    //每页条数
    private int limit;

    public Query(Map<String, Object> params){
        this.putAll(params);

        //分页参数
        this.page = Integer.parseInt(params.get("page").toString());
        this.limit = Integer.parseInt(params.get("limit").toString());
        this.put("offset", (page - 1) * limit);
        this.put("page", page);
        this.put("limit", limit);

        //防止SQL注入（因为sidx、order是通过拼接SQL实现排序的，会有SQL注入风险
    )
        String sidx = (String)params.get("sidx");
        String order = (String)params.get("order");
        if(StringUtils.isNotBlank(sidx)){
            this.put("sidx", SQLFilter.sqlInject(sidx));
        }
        if(StringUtils.isNotBlank(order)){
            this.put("order", SQLFilter.sqlInject(order));
```

```
            }

        }
    }
```

4.2.5. Redis缓存

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

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

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

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

本项目中，使用Redis服务的代码，如下所示：

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

```
    private SysConfigRedis sysConfigRedis;

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

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

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

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

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


-------------------------------------------------


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

    public void saveOrUpdate(SysConfigEntity config) {
```

```
            if(config == null){
                return ;
            }
            String key = RedisKeys.getSysConfigKey(config.getKey());
            redisUtils.set(key, config);
        }

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

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


    ----------------------------------------------------


    package io.renren.common.aspect;

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

        public void set(String key, Object value, long expire){
```

```
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
    t instanceof Float ||
138.                    object instanceof Double || object instanceof Boolean |
```

```
    | object instanceof String){
139.             return String.valueOf(object);
140.         }
141.         return gson.toJson(object);
142.     }
143. 
144.     /**
145.      * 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，不然会报错，因为有操作Redis的代码，其实不然，通过Spring AOP，我们可以控制，是否真的使用Redis，代码如下：

```
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. }
```

4.2.6. 异常处理机制

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

RRException代码如下所示：

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

    private String msg;
    private int code = 500;

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

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

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

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

    public String getMsg() {
        return msg;
    }

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

    public int getCode() {
```

```
        return code;
    }

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

> 如何处理抛出的异常呢，我们定义了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。

RRExceptionHandler代码如下所示：

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

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

        return r;
    }

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

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

```

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

4.2.7. 后端效验机制

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

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

ValidatorUtils代码如下所示：

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

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

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

```
                msg.append(constraint.getMessage()).append("<br>");
            }
            throw new RRException(msg.toString());
        }
    }
}
```

使用案例：

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

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

        return R.ok();
    }

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

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

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

```
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, UpdateG
     roup.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
```

```
     */
    private Integer status;

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

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

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

通过分析上面的代码，我们来理解Hibernate Validator校验框架的使用。
其中，username属性，表示保存或修改用户时，都会效验username属性； 而password属性，表示只有保存用户时，才会效验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)进行效验。

4.2.8. 系统日志

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

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

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

下面是自定义注解`@SysLog`的使用方式，如下所示：

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

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

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

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

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

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

```
/**
 * 系统日志，切面处理类
 *
 * @author chenshun
 * @email sunlightcs@gmail.com
 * @date 2017年3月8日 上午11:07:35
 */
@Aspect
@Component
public class SysLogAspect {
    @Autowired
    private SysLogService sysLogService;

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

    }
```

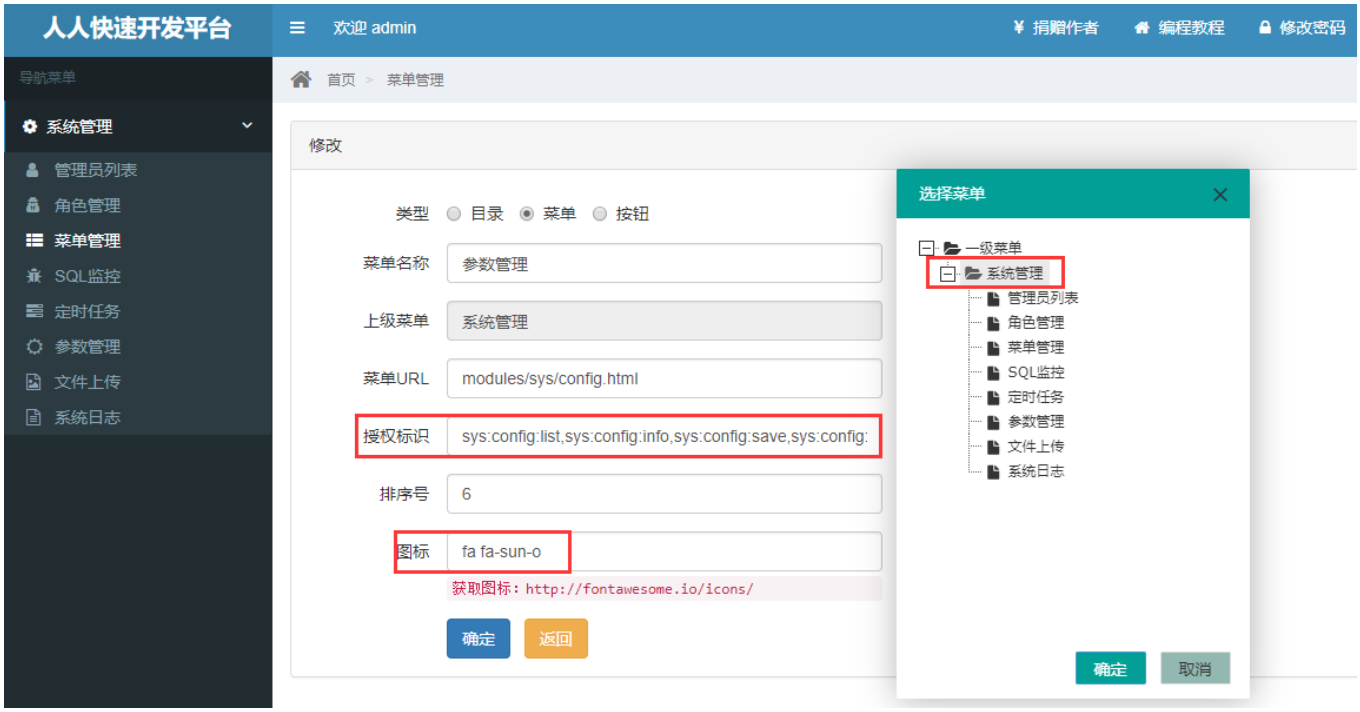
```
18.
19.      @Around("logPointCut()")
20.      public Object around(ProceedingJoinPoint point) throws Throwable {
21.          long beginTime = System.currentTimeMillis();
22.          //执行方法
23.          Object result = point.proceed();
24.          //执行时长(毫秒)
25.          long time = System.currentTimeMillis() - 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.getSign
     ature();
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
```

```
            sysLog.setIp(IPUtils.getIpAddr(request));

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

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

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

4.2.9. 添加菜单

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



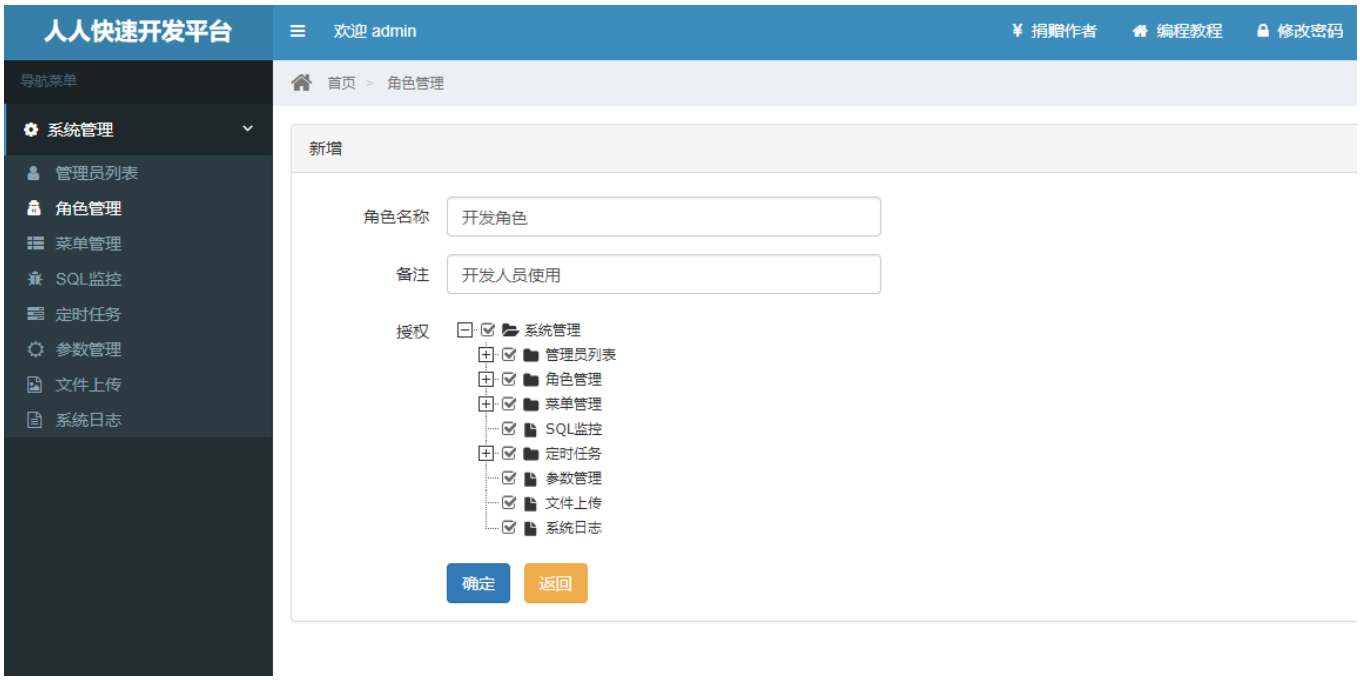
上图是拿现有的菜单进行讲解。其中，授权标识与shiro中的注解@RequiresPermissions，定义的授权标识是一一对应的，如下所示：

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

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

    }

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

    }

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

    }

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

    }

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

    }

}
```

4.2.10. 添加角色

> 管理员权限是通过角色进行管理的，给管理员分配权限时，要先创建好角色。

下面创建了一个【开发角色】，如下图所示：



4.2.11. 添加管理员

> 本系统默认就创建了admin账号，无需分配任何角色，就拥有最高权限。
> 一个管理员是可以拥有多个角色的。
> 非admin账号，只能查看及管理自己创建的账号。

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

4.3. 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。

4.3.1. 新增定时任务

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

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

    @Autowired
    private SysUserService sysUserService;

    //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
    public void test(String params){
        logger.info("我是带参数的test方法，正在被执行，参数为：" + params);

        try {
            Thread.sleep(1000L);
```

```
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

            SysUserEntity user = sysUserService.queryObject(1L);
            System.out.println(ToStringBuilder.reflectionToString(user));

        }


        public void test2(){
            logger.info("我是不带参数的test2方法，正在被执行");
        }
    }
```

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：

刚才配置的定时任务，每隔10秒，就会调用TestTask的test方法了，是不是很简单啊。

4.3.2. 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 {
```

```
            return (CronTrigger) scheduler.getTrigger(getTriggerKey(jobId));
        } catch (SchedulerException e) {
            throw new RRException("getCronTrigger异常，请检查qrtz开头的表，是否有脏数据", e);
        }
    }

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

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

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

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

            scheduler.scheduleJob(jobDetail, trigger);

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

    /**
```

```
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.c
    ronSchedule(scheduleJob.getCronExpression())
68.                     .withMisfireHandlingInstructionDoNothing();
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(triggerK
    ey).withSchedule(scheduleBuilder).build();
74.
75.             //参数
76.             trigger.getJobDataMap().put(ScheduleJobEntity.JOB_PARAM_KEY
    , new Gson().toJson(scheduleJob));
77.
78.             scheduler.rescheduleJob(triggerKey, trigger);
79.
80.             //暂停任务
81.             if(scheduleJob.getStatus() == ScheduleStatus.PAUSE.getValue
    ()){
82.                 pauseJob(scheduler, scheduleJob.getJobId());
83.             }
84.
85.         } catch (SchedulerException e) {
86.             throw new RRException("更新定时任务失败", e);
87.         }
88.     }
89.
90.     /**
91.      * 立即执行任务
92.      */
93.     public static void run(Scheduler scheduler, ScheduleJobEntity
    scheduleJob) {
94.         try {
95.             //参数
96.             JobDataMap dataMap = new JobDataMap();
```

```
97.                 dataMap.put(ScheduleJobEntity.JOB_PARAM_KEY, new Gson().toJ
    son(scheduleJob));
98.
99.                 scheduler.triggerJob(getJobKey(scheduleJob.getJobId()), dat
    aMap);
100.            } catch (SchedulerException e) {
101.                throw new RRException("立即执行定时任务失败", e);
102.            }
103.        }
104.
105.        /**
106.         * 暂停任务
107.         */
108.        public static void pauseJob(Scheduler scheduler, Long jobId) {
109.            try {
110.                scheduler.pauseJob(getJobKey(jobId));
111.            } catch (SchedulerException e) {
112.                throw new RRException("暂停定时任务失败", e);
113.            }
114.        }
115.
116.        /**
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：

```
@Service("scheduleJobService")
public class ScheduleJobServiceImpl implements ScheduleJobService {
    /**
     * 项目启动时，初始化定时器
     */
    @PostConstruct
    public void init(){
        List<ScheduleJobEntity> scheduleJobList = schedulerJobDao.queryList(new HashMap<>());
        for(ScheduleJobEntity scheduleJob : scheduleJobList){
            CronTrigger cronTrigger = ScheduleUtils.getCronTrigger(scheduler, scheduleJob.getJobId());
            //如果不存在，则创建
            if(cronTrigger == null) {
                ScheduleUtils.createScheduleJob(scheduler, scheduleJob);
            }else {
                ScheduleUtils.updateScheduleJob(scheduler, scheduleJob);
            }
        }
    }

}
```

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

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

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

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

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

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

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

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

    @Override
    protected void executeInternal(JobExecutionContext context) throws JobExecutionException {
        //获取job里的参数，创建job时，传进去的ScheduleJobEntity对象
        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 = (ScheduleJobLogSe
rvice) 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.             ScheduleRunnable task = new ScheduleRunnable(scheduleJob.ge
tBeanName(),
29.                     scheduleJob.getMethodName(), scheduleJob.getParams(
));
30.             Future<?> future = service.submit(task);
31.
32.             future.get();
33.
34.             //任务执行总时长
35.             long times = System.currentTimeMillis() - startTime;
36.             log.setTimes((int)times);
37.             //任务状态    0：成功    1：失败
38.             log.setStatus(0);
39.
40.             logger.info("任务执行完毕，任务ID：" + scheduleJob.getJobId()
+ "  总共耗时：" + times + "毫秒");
41.         } catch (Exception e) {
42.             logger.error("任务执行失败，任务ID：" + scheduleJob.getJobId()
, e);
43.
44.             //任务执行总时长
45.             long times = System.currentTimeMillis() - startTime;
```

```
                log.setTimes((int)times);

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

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：

```
public class ScheduleRunnable implements Runnable {
    private Object target;
    private Method method;
    private String params;

    public ScheduleRunnable(String beanName, String methodName, String params) throws NoSuchMethodException, SecurityException {
        //获取spring bean
        this.target = SpringContextUtils.getBean(beanName);
        this.params = params;

        if(StringUtils.isNotBlank(params)){
            this.method = target.getClass().getDeclaredMethod(methodName, String.class);
        }else{
            this.method = target.getClass().getDeclaredMethod(methodName);
        }
    }

    @Override
    public void run() {
        try {
            ReflectionUtils.makeAccessible(method);
            if(StringUtils.isNotBlank(params)){
                method.invoke(target, params);
            }else{
                method.invoke(target);
            }
        }catch (Exception e) {
```

```
                throw new RRException("执行定时任务失败", e);
            }
        }
    }
```

4.4. 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。

4.4.1. 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：

2. 进入管理后端，填写七牛配置信息，如下图：

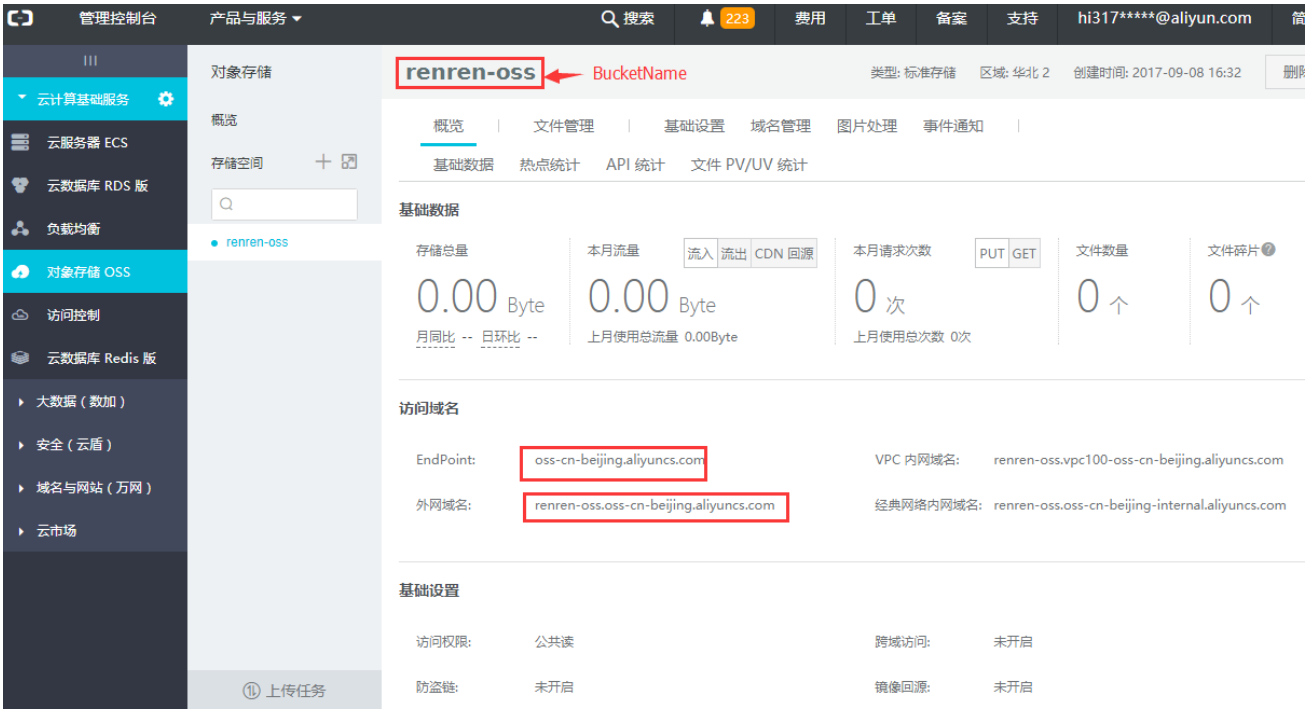
必填项有域名、AccessKey、SecretKey、空间名。其中，空间名就是才创建的空间名`ios-app`，填进去就可以了。域名、AccessKey、SecretKey可以通过下图找到：

4.4.2. 阿里云的配置

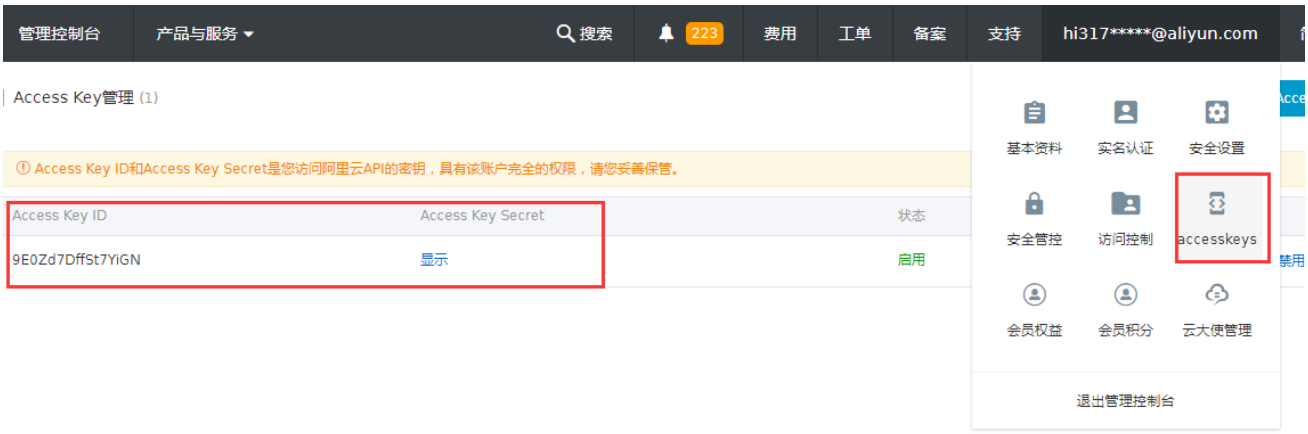
- 进入管理后端，填写阿里云配置信息，如下图：

- 进去阿里云管理后台，并创建Bucket，如下图：

- 通过下面的界面，可以找到域名、BucketName、EndPoint

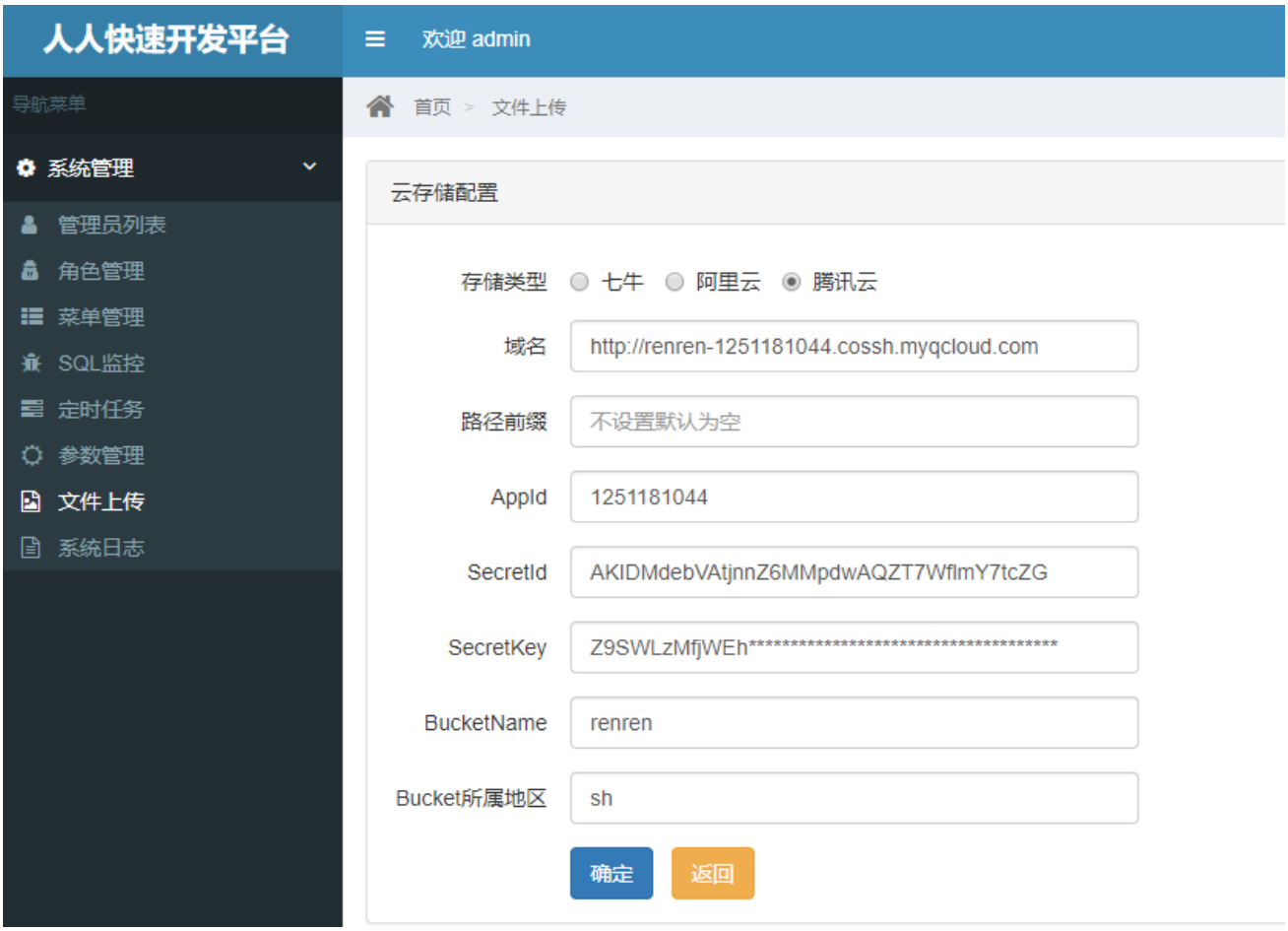


- 通过下面的界面，可以找到AccessKeyId、AccessKeySecret

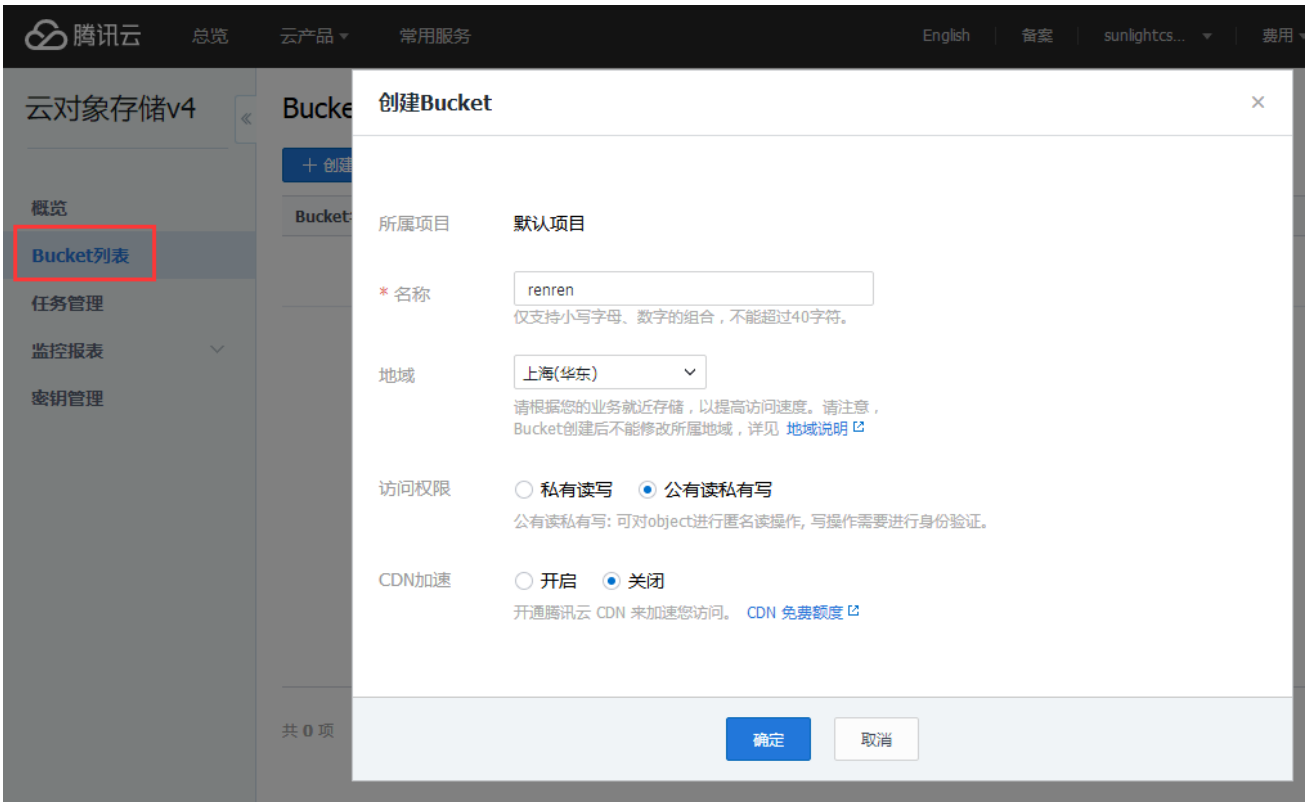


4.4.3. 腾讯云的配置

- 进入管理后端，填写腾讯云配置信息，如下图：



- 进去腾讯云管理后台，并创建Bucket，如下图：



- 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

4.4.4. 源码分析

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

```
<dependency>
    <groupId>com.qiniu</groupId>
    <artifactId>qiniu-java-sdk</artifactId>
    <version>${qiniu.version}</version>
</dependency>
```

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

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

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

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

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

        return path;
    }

    /**
     * 文件上传
```

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

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

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

    /**
     * 文件上传
     * @param inputStream  字节流
     * @return             返回http地址
     */
    public abstract String upload(InputStream inputStream);

}
```

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

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

public class QiniuCloudStorageService extends CloudStorageService{
    private UploadManager uploadManager;
    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.autoZo
    ne()));
22.         token = Auth.create(config.getQiniuAccessKey(), config.getQiniu
    SecretKey()).
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 RRException("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 RRException("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 RRException("上传文件失败，请检查配置信息", e);
31.         }
32.
```

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

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

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

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

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

public class QcloudCloudStorageService extends CloudStorageService{
    private COSClient client;

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

        //初始化
        init();
    }

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

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

```

```
            client = new COSClient(clientConfig, credentials);
        }

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

            //上传到腾讯云
            UploadFileRequest request = new UploadFileRequest(config.getQcl
oudBucketName(), path, data);
            String response = client.uploadFile(request);

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

            return config.getQcloudDomain() + path;
        }

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

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

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

- 对外提供了OSSFactory工厂，可方便业务的调用，如下所示：

```
public final class OSSFactory {
    private static SysConfigService sysConfigService;

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

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

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

        return null;
    }

}
```

- 文件上传的例子，如下：

```
@RequestMapping("/upload")
public R upload(@RequestParam("file") MultipartFile file) throws Exception {
    if (file.isEmpty()) {
        throw new RRException("上传文件不能为空");
    }

    //上传文件，并返回文件的http地址
    String url = OSSFactory.build().upload(file.getBytes());
}
```

4.5. APP模块

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

4.5.1. APP的使用

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

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

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

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

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

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

    /**
```

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

}
```

4.5.2. 源码分析

- 我们先来看看，APP用户登录的时候，都干了那些事情，如下所示：

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

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

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

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

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

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

```
}

-------------------------------------------------------------

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

    private String secret;
    private long expire;
    private String header;

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

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

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

```
            }
        }

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

        public String getSecret() {
            return secret;
        }

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

        public long getExpire() {
            return expire;
        }

        public void setExpire(long expire) {
            this.expire = expire;
        }

        public String getHeader() {
            return header;
        }

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

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

- APP模块的核心配置，如下所示：

```
import io.renren.modules.api.interceptor.AuthorizationInterceptor;
import
io.renren.modules.api.resolver.LoginUserHandlerMethodArgumentResolver;
```

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Configuration;
import
org.springframework.web.method.support.HandlerMethodArgumentResolver;
import
org.springframework.web.servlet.config.annotation.InterceptorRegistry;
import
org.springframework.web.servlet.config.annotation.WebMvcConfigurerAdapt
er;

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

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

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

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

温馨提示，别忘了配置shiro，不然会被shiro拦截掉的，如下所示：

```
@Configuration
public class ShiroConfig {
    @Bean("shiroFilter")
```

```
    public ShiroFilterFactoryBean shirFilter(SecurityManager securityManager) {

        //部分代码省略...

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

        //部分代码省略...

        shiroFilter.setFilterChainDefinitionMap(filterMap);

        return shiroFilter;
    }
}
```

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

```

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

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

/**
 * 权限(Token)验证
 */
@Component
public class AuthorizationInterceptor extends HandlerInterceptorAdapter {
    @Autowired
    private JwtUtils jwtUtils;
```

```
23.
24.     public static final String USER_KEY = "userId";
25.
26.     @Override
27.     public boolean preHandle(HttpServletRequest request, HttpServletRes
    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(jwtUtils.getHeader());
41.         if(StringUtils.isBlank(token)){
42.             token = request.getParameter(jwtUtils.getHeader());
43.         }
44.
45.         //凭证为空
46.         if(StringUtils.isBlank(token)){
47.             throw new RRException(jwtUtils.getHeader() + "不能为空",
    HttpStatus.UNAUTHORIZED.value());
48.         }
49.
50.         Claims claims = jwtUtils.getClaimByToken(token);
51.         if(claims == null || jwtUtils.isTokenExpired(claims.getExpirati
    on())){
52.             throw new RRException(jwtUtils.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`/app`请求的接口之前，会判断请求的接口，是否加了@Login注解(需要token认证)，如果没有@Login注解，则不验证token，可以直接访问接口。如果有@Login注解，则需要验证token的正确性，并把userId放到request的USER_KEY里，后续会用到。

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

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

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

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

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

    @Override
    public boolean supportsParameter(MethodParameter parameter) {
        //如果方法的参数是UserEntity，且参数前面有@LoginUser注解，则进入
```

```
resolveArgument方法，进行处理
        return parameter.getParameterType().isAssignableFrom(UserEntity.class) && parameter.hasParameterAnnotation(LoginUser.class);
    }

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

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

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

---

5. 生产环境部署

5.1. 常规部署

> Spring Boot项目，推荐打成jar包的方式，部署到服务器上。

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

```
server:
  tomcat:
    uri-encoding: UTF-8
    max-threads: 1000
    min-spare-threads: 20
```

```
    connection-timeout: 5000
    port: 80
    context-path: /renren-fast
    ssl:
      key-store: classpath:.keystore
      key-store-type: JKS
      key-password: 123456
      key-alias: tomcat
```

- 当然，还可以指定jvm的内存大小，如下所示：

```
java -Xms4g -Xmx4g -Xmn1g -server -jar renren-fast-1.2.0.jar
```

- 在windows下部署，只需打开cmd窗口，输入如下命令：

```
java -jar renren-fast-1.2.0.jar --spring.profiles.active=pro
```

- 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：

```
nohup java -jar renren-fast-1.2.0.jar --spring.profiles.active=pro > renren.log &
```

5.2. war包部署

> war包的部署，也很方便，只是不推荐这种方式。

- 在项目的根目录下，执行【mvn clean package -f pom-war.xml】命令，打成war包
- 把生成的war包，放在tomcat【8.0+】的webapps目录下面，再启动tomcat即可

5.3. docker部署

- 安装docker环境

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

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

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

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

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

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

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

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

- 安装docker-compose，用来管理容器

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

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

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

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

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

- 通过docker-compose，启动项目，如下所示：

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

#查看启动的容器
[root@mark renren-fast]# docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED
STATUS              PORTS                    NAMES
f4e3fcdd8dd4        renren/fast         "java -jar /app.jar"   55 secon
ds ago      Up 3 seconds        0.0.0.0:8080->8080/tcp
renrenfast_renren-fast_1

#停掉并删除，docker-compose管理的容器
[root@mark renren-fast]# docker-compose down
Stopping renrenfast_renren-fast_1 ... done
Removing renrenfast_renren-fast_1 ... done
Removing network renrenfast_default
```