# 总体架构

## 拦截器的总开关



## 登录拦截器的验证

原理：用户登录之后，系统会返回一个唯一的token，然后前端每次请求（header）携带token，调用方法之前会验证token，然后缓存用户的信息，当方法执行完毕后，销毁当前用户的缓存。

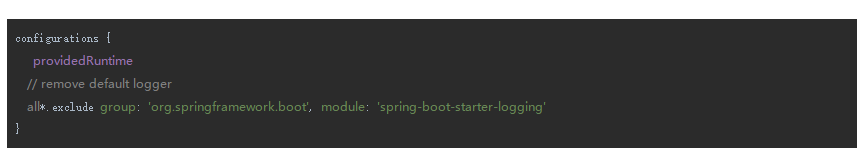


如上图所示，标注行表示该方法需要登录验证，具体操作如下图，首先要获取改方法的注解，判断是否有“api\_key”，如果有，就要获取到api\_key，并根据api\_key获取当前的登录用户，然后保存全局用户信息，等方法调用结束后，清除掉全局用户信息（图略）



## 日志管理（自定义内容到文件）

* 移除默认logger（logback）依赖



* 添加log4j依赖

compile 'org.springframework.boot:spring-boot-starter-log4j:1.3.8.RELEASE'

* yml配置log4j

logging:  
 config: classpath:log4j.properties

* log4j自定义消息



* 向文件内写入内容

Logger **logger** = LoggerFactory.*getLogger*(**"R"**);//名字与配置文件保持一致

**logger**.info(**"2134567"**);

## 跨域拦截及header参数设置



## 异常捕获

在调用接口时，如果出错，也是要将错误信息进行返回的，返回的格式为{code：401,msg:”token不能为空”}

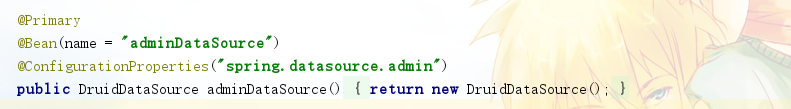


# Beetlsql

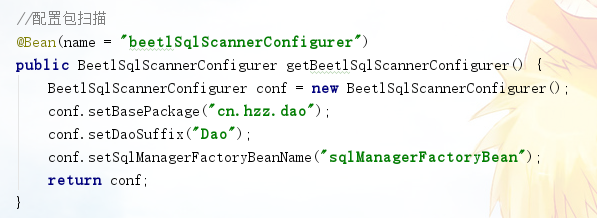
Beetlsql可以自定义sql语句查询，也可以使用内置sql和方法。

## 数据源配置

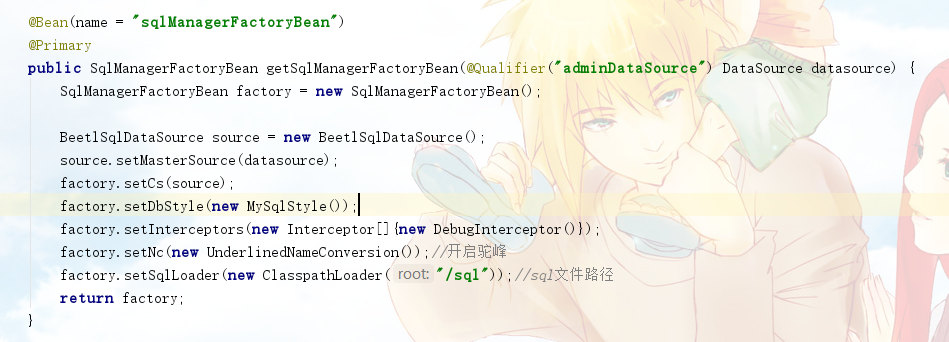
* 加载yml配置文件，配置数据库



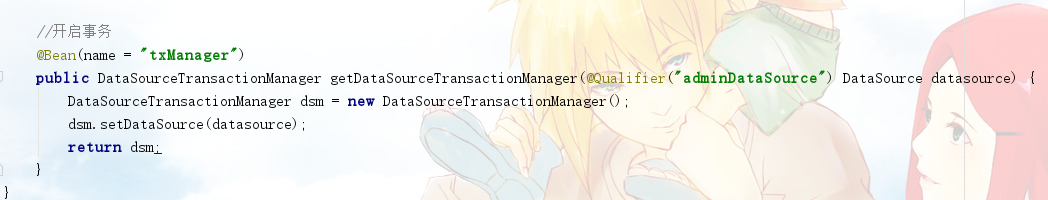
* 配置扫描包



* 配置sqlManagerFactoryBean



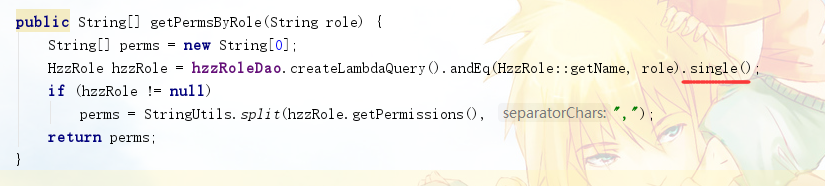
* 开启事务



**总结：多个数据源配置同样如此，只是访问了哪个目录下的md文件，自动调用相应的数据源**

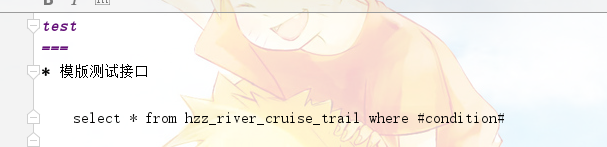
## 单表操作

增、删、改、查可以使用内置方法：insert、unique、updateById、deleteById，此外，条件查询也无需写sql语句，如下图。（如果查询的结果为多条数据，使用select()；如果结果仅为一条，使用single()）

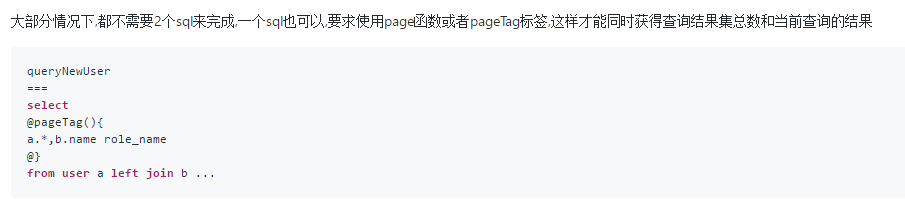


## 多表操作

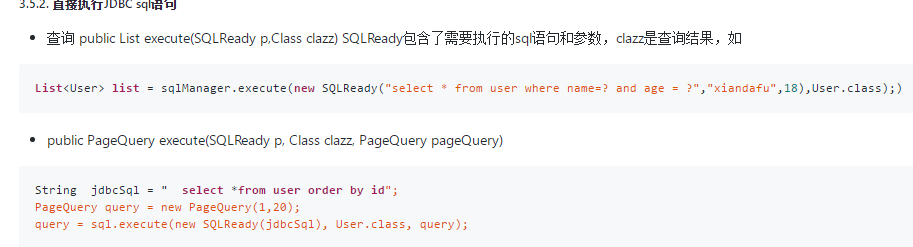
此处仅介绍分页查询，首先介绍beetlsql的使用方法，如下图所示，sql语句是写在md文件当中的。

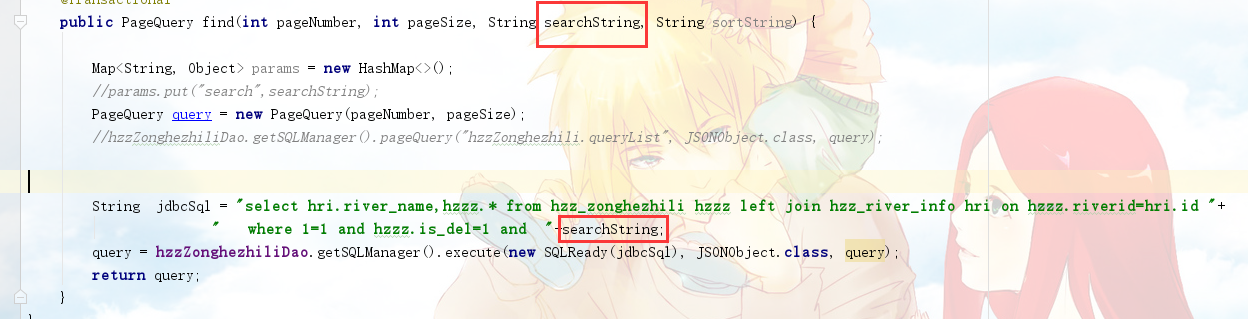


首先，这个语句是错误的，当传入的条件无论是什么值，都不会返回结果，且不报错，也就是md里不支持sql语句的拼接。



如图所示，这个是分页查询的sql语句，但是在查询的时候比较繁琐，上边说过不支持sql拼接，所以，有多少个条件查询就得传多少个参数，导致比较繁琐。我们可以采用如下形式，直接拼接sql语句（可以从前端拼接好传到后端，或者后台写方法获取所有的参数和值，然后拼接），然后执行sql语句来实现，如下图所示。





# Swagger-UI

## 引入依赖包

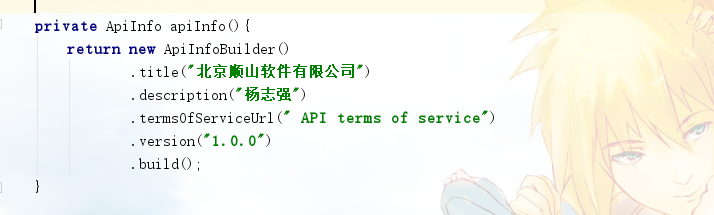
compile("io.springfox:springfox-swagger2:2.8.0")  
compile("io.springfox:springfox-swagger-ui:2.8.0")

## 配置启动类

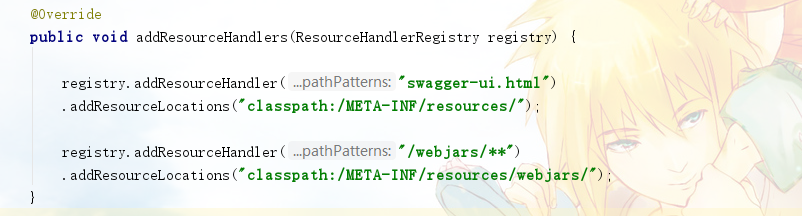
首先要创建Api，配置一些属性，例如：扫描哪些包、登录安全认证等



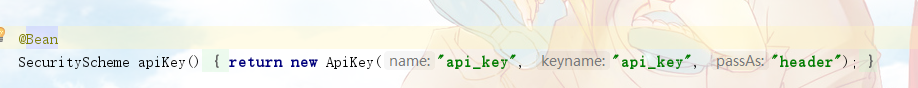
其次要配置主界面的标题、描述、版本、链接等



接着要释放jar包的资源，因为swagger-ui的界面都是在jar包中的，我们直接调用时访问不到的，所以要将jar包中的html等释放出来吗，允许我们访问



最后，加入安全认证，即在前台的header中要有api\_key的参数，呼应第一条中加入的安全认证，也就是第一条中的安全认证具体实现是这个方法

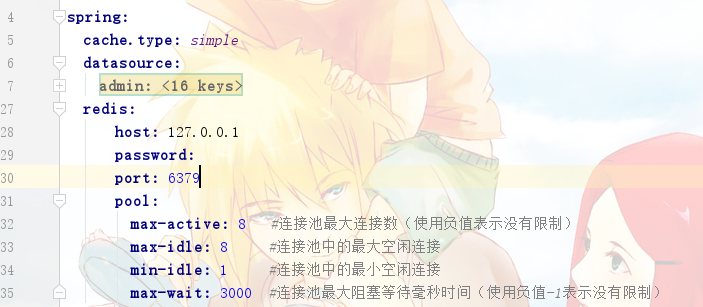


# Redis缓存

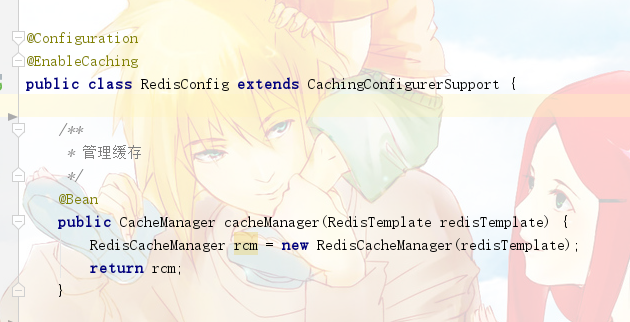
## 引入依赖包

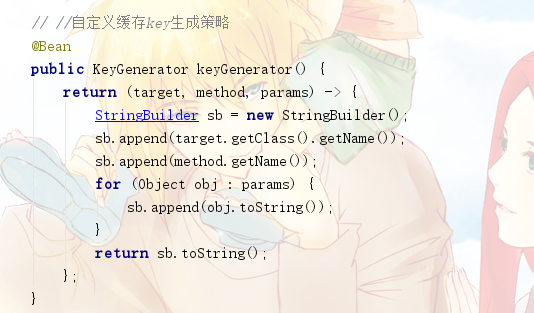
compile(**"org.springframework.boot:spring-boot-starter-data-redis"**)

## Redis链接



## Redis配置







## Redis使用

在service的类添加注解@CacheConfig然后方法名使@Cacheable、@CachePut、@ CacheEvict。本项目中使用@Cacheable（key=”#userId”）

**@Cacheable**是用来声明方法是可缓存的。将结果存储到缓存中以便后续使用相同参数调用时不需执行实际的方法。直接从缓存中取值

**@CachePut**如果缓存需要更新，且不干扰方法的执行,可以使用注解@CachePut。@CachePut标注的方法在执行前不会去检查缓存中是否存在之前执行过的结果，而是每次都会执行该方法，并将执行结果以键值对的形式存入指定的缓存中。

**@CacheEvict** spring cache不仅支持将数据缓存，还支持将缓存数据删除。此过程经常用于从缓存中清除过期或未使用的数据。 @CacheEvict要求指定一个或多个缓存，使之都受影响。此外，还提供了一个额外的参数allEntries 。表示是否需要清除缓存中的所有元素。默认为false，表示不需要。当指定了allEntries为true时，Spring Cache将忽略指定的key。有的时候我们需要Cache一下清除所有的元素。

# 项目打包

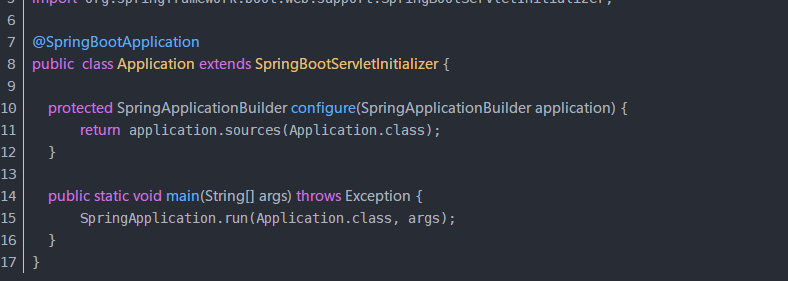
## 配置构建脚本

gradle 添加war插件：apply plugin: 'war'

## 去掉内置tomcat

gradle配置providedRuntime('org.springframework.boot:spring-boot-starter-tomcat')

## 启动类集成SpringBootServletInitializer



## gradle打包

在view->tool windows->gradle的build下，执行war