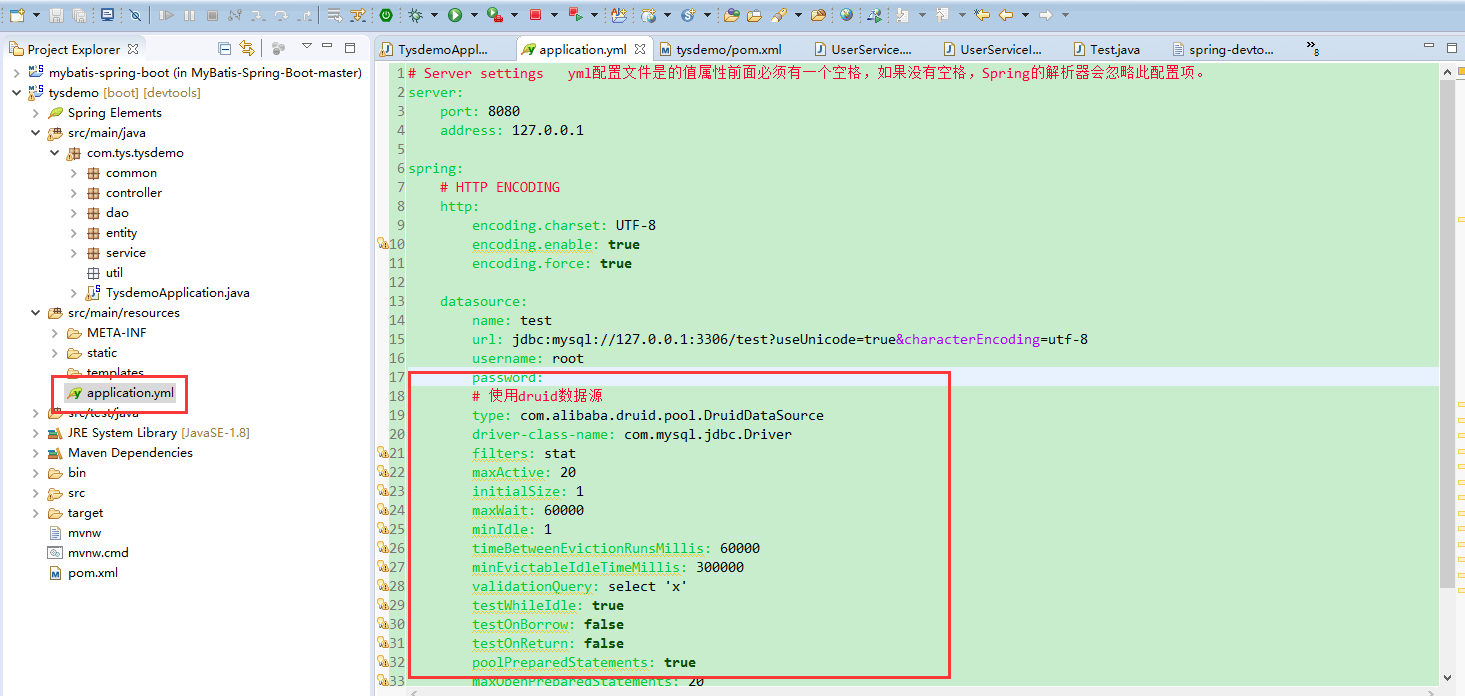
# SpringBoot

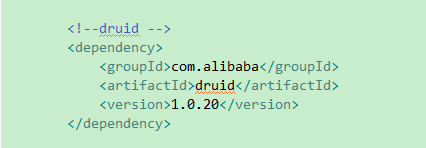
## 框架搭建

### 新建项目

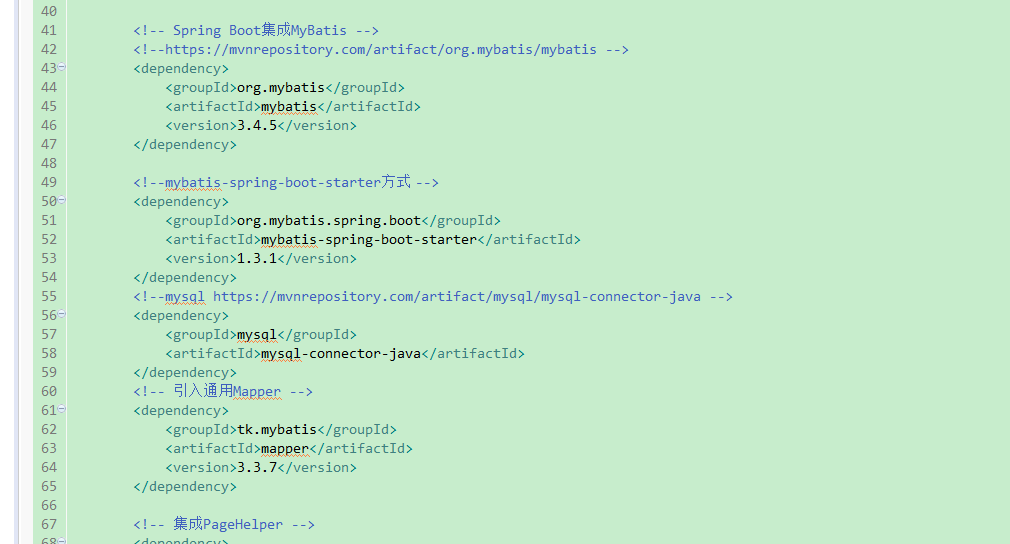
### Druid配置，后起之秀最好的数据库连接池！

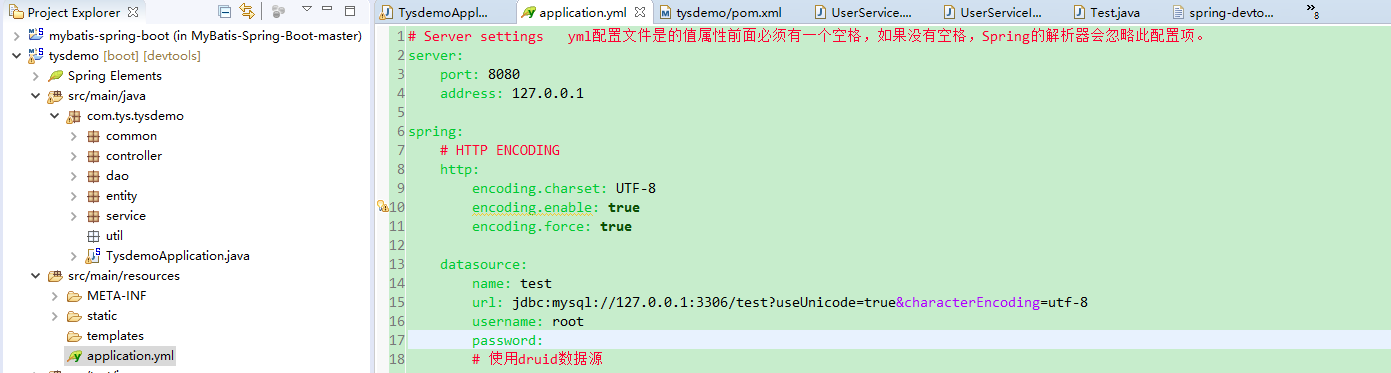
Druid提供了以交互方式访问数据的能力，并权衡了查询的灵活性和性能而采取了特殊的存储格式；



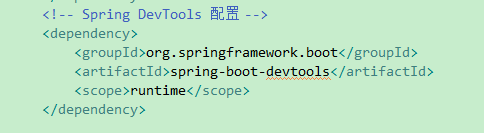


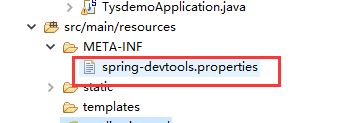
### Mybatis配置

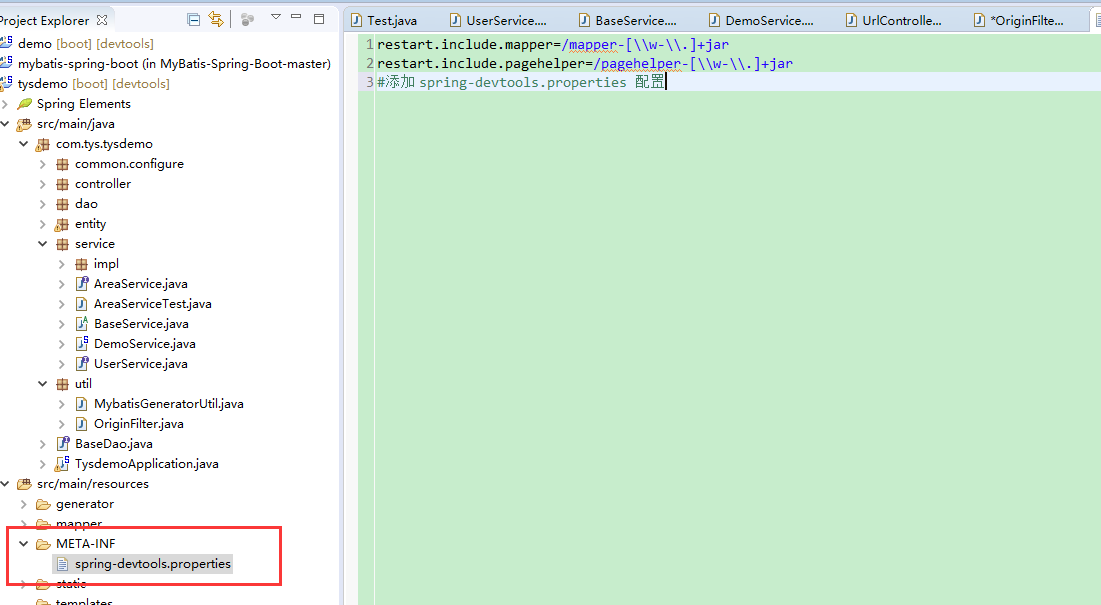




### Spring DevTools 配置



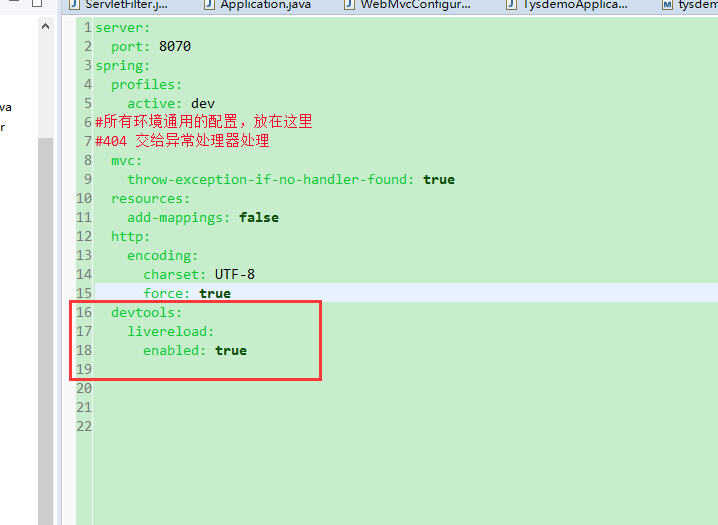




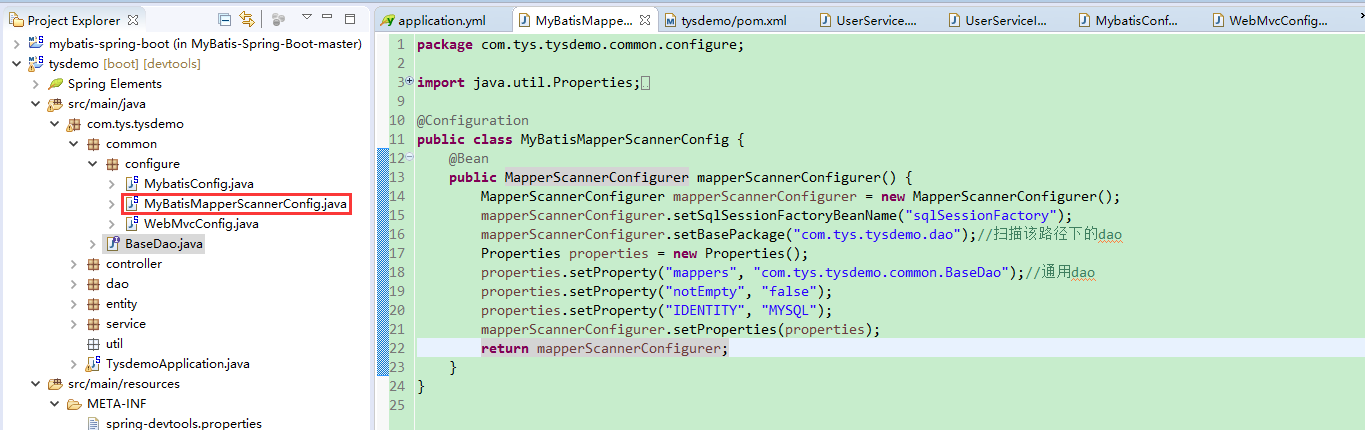
**热启动**

spring.devtools.\* 这些列出的配置都是 spring-boot-devtools 相关的配置。

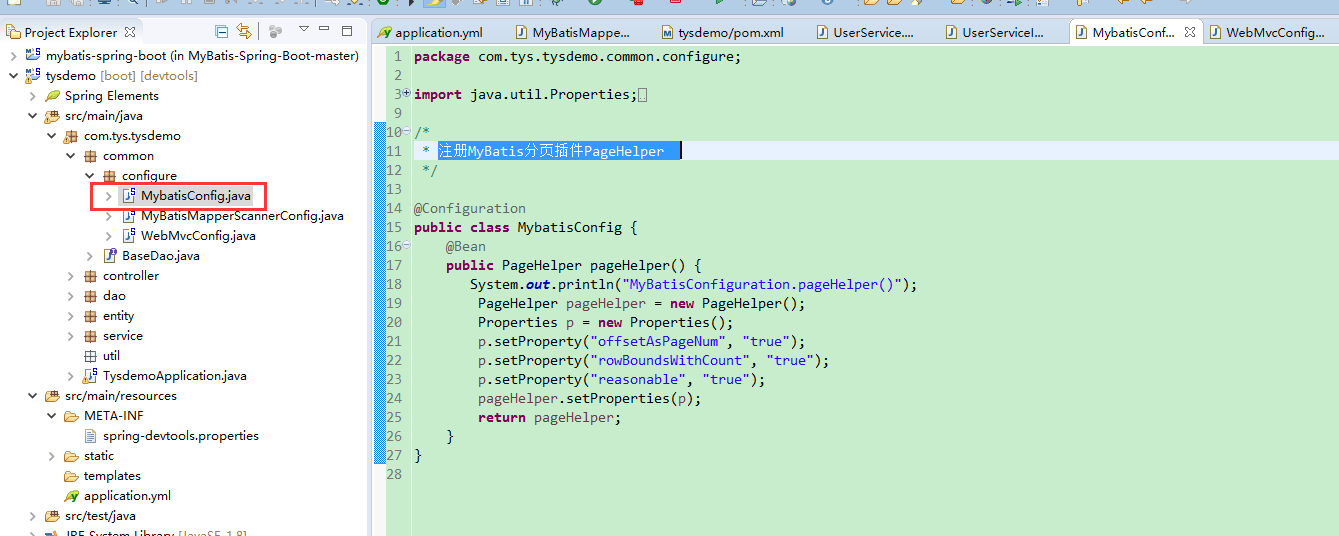
其中 spring.devtools.restart.enabled 表示是否开启自动启动



### 通用mapper配置

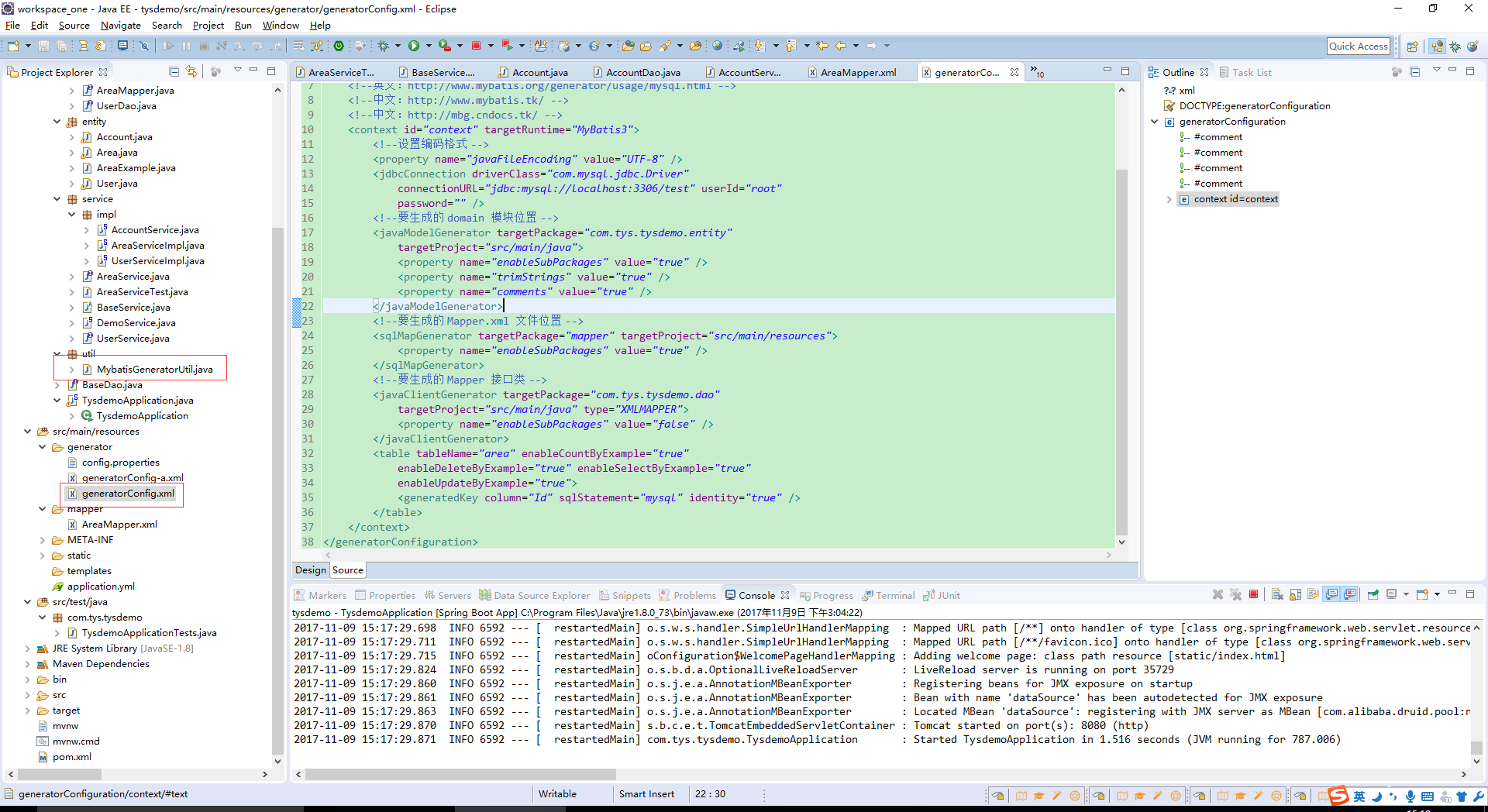


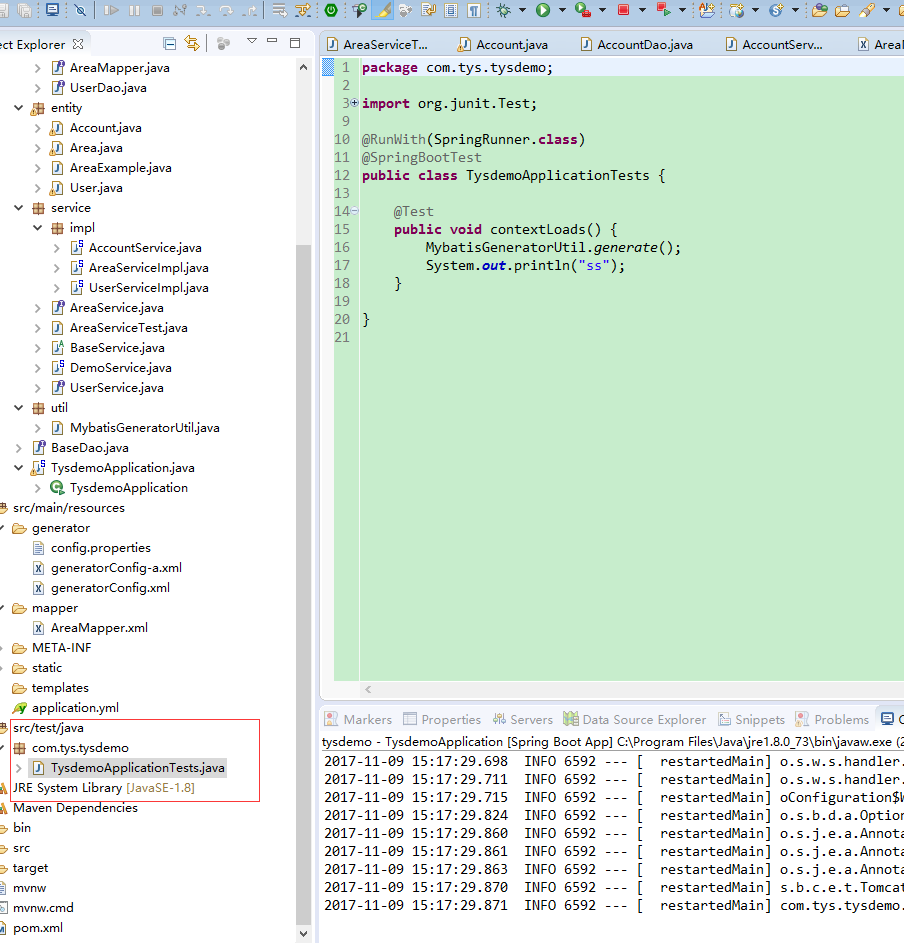
### 注册MyBatis分页插件PageHelper



### 使用Mapper专用的MyBatis Generator插件

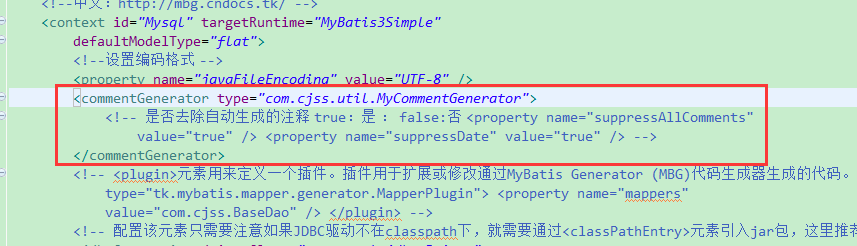
<https://gitee.com/free/Mapper/blob/master/wiki/mapper3/7.UseMBG.md>



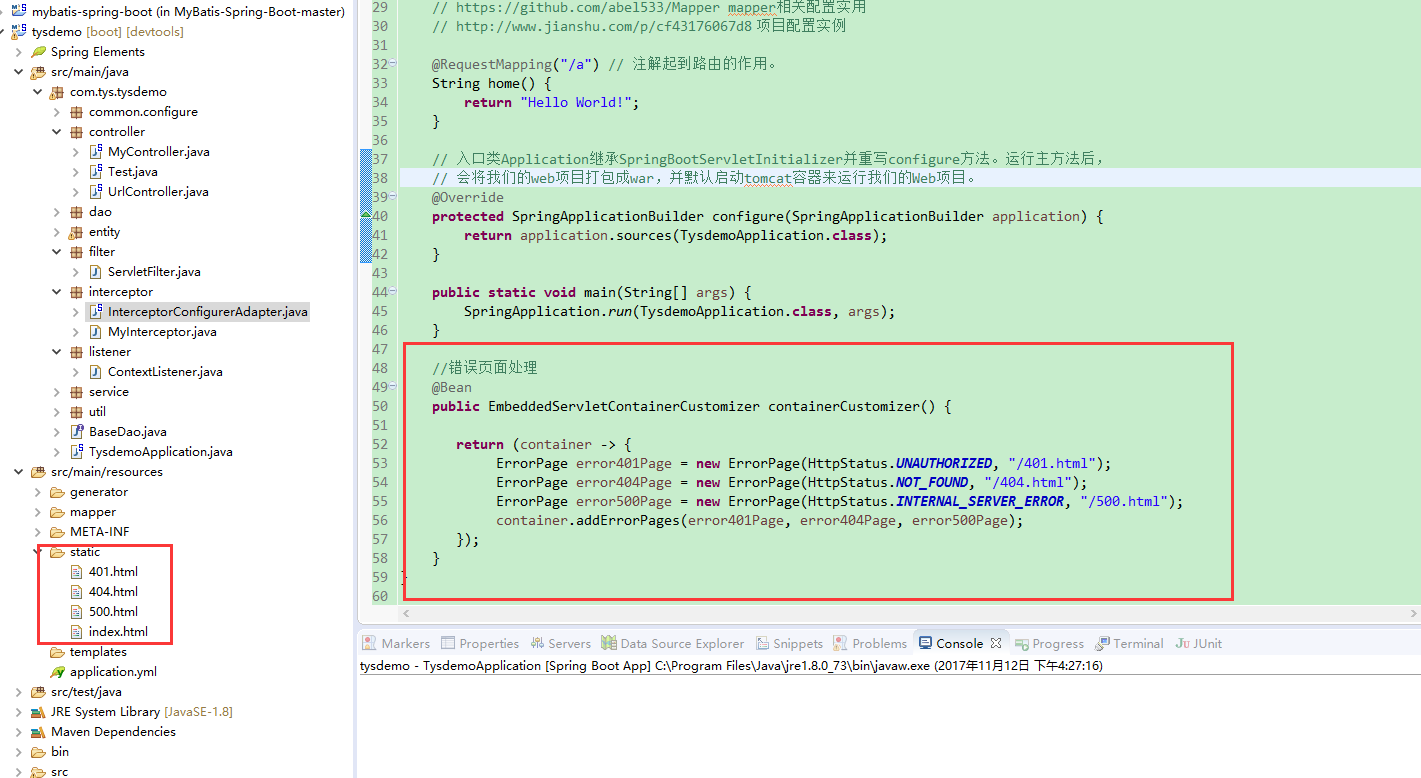


* 自定义注释

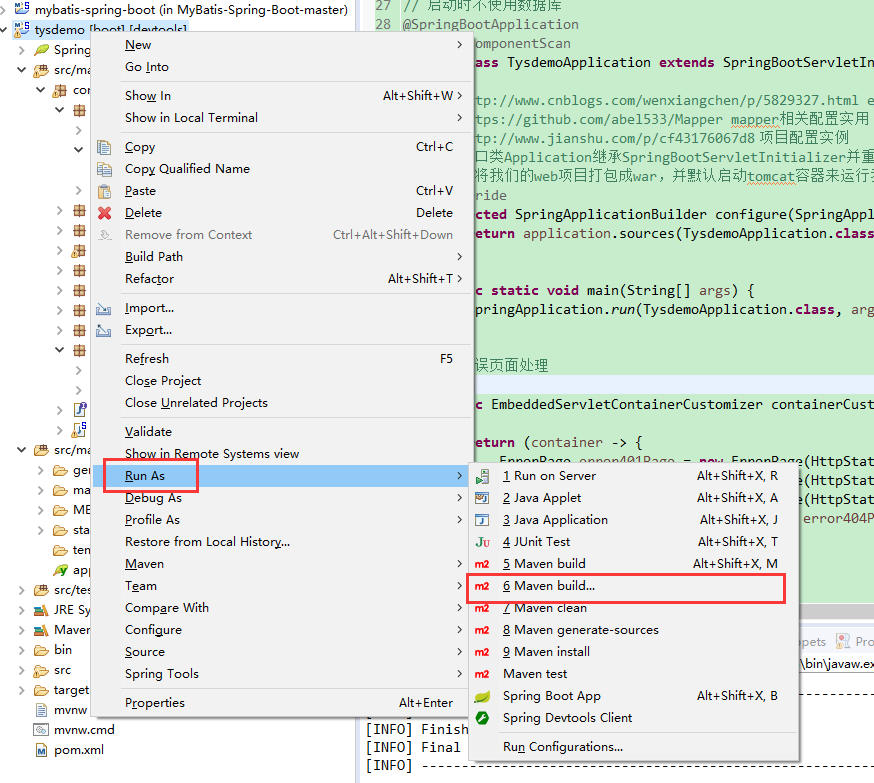


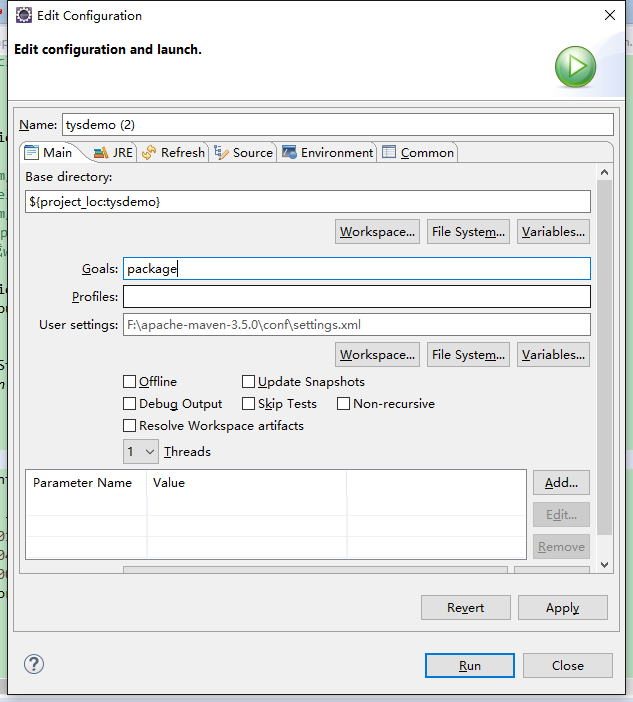


## 错误页面处理



## 打包运行





G:\java-eclipse(neno)\workspace\_one\zbw\target目录下就是打包的文件

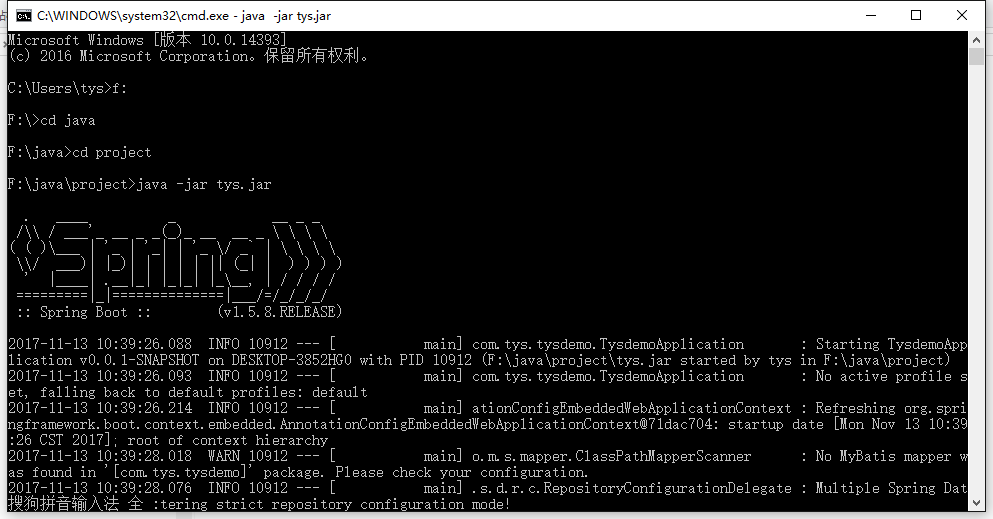
## 部署运行

java打jar包网站

<http://www.cnblogs.com/flashsun/p/7246260.html>

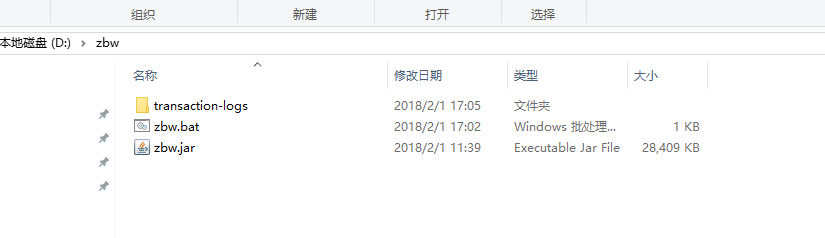
cd进入到jar包文件夹就ok

java -jar hello.jar



## Spring Boot 以 jar 包方式运行在后台

<https://www.cnblogs.com/kedarui/p/6518723.html>



Bat文件如下

java -jar zbw.jar

## 问题集合

1. Cannot determine embedded database driver class for database type NONE

原因是：springboot启动时会自动注入数据源和配置jpa

解决：在@SpringBootApplication或@EnableAutoConfiguration中排除其注入

@SpringBootApplication(exclude={DataSourceAutoConfiguration.class,HibernateJpaAutoConfiguration.class})

## Mapper的配置及其适用

<https://github.com/abel533/Mapper>

一旦继承了Mapper<T>,继承的Mapper就拥有了以下通用的方法:

*//根据实体类不为null的字段进行查询,条件全部使用=号and条件*

List**<**T**>** select**(**T record**);**

*//根据实体类不为null的字段查询总数,条件全部使用=号and条件*

**int** **selectCount**(T record**);**

*//根据主键进行查询,必须保证结果唯一*

*//单个字段做主键时,可以直接写主键的值*

*//联合主键时,key可以是实体类,也可以是Map*

T **selectByPrimaryKey**(Object key**);**

*//插入一条数据*

*//支持Oracle序列,UUID,类似Mysql的INDENTITY自动增长(自动回写)*

*//优先使用传入的参数值,参数值空时,才会使用序列、UUID,自动增长*

**int** **insert**(T record**);**

*//插入一条数据,只插入不为null的字段,不会影响有默认值的字段*

*//支持Oracle序列,UUID,类似Mysql的INDENTITY自动增长(自动回写)*

*//优先使用传入的参数值,参数值空时,才会使用序列、UUID,自动增长*

**int** **insertSelective**(T record**);**

*//根据实体类中字段不为null的条件进行删除,条件全部使用=号and条件*

**int** **delete**(T key**);**

*//通过主键进行删除,这里最多只会删除一条数据*

*//单个字段做主键时,可以直接写主键的值*

*//联合主键时,key可以是实体类,也可以是Map*

**int** **deleteByPrimaryKey**(Object key**);**

*//根据主键进行更新,这里最多只会更新一条数据*

*//参数为实体类*

**int** **updateByPrimaryKey**(T record**);**

*//根据主键进行更新*

*//只会更新不是null的数据*

**int** **updateByPrimaryKeySelective**(T record**);**

**4. 泛型(实体类)<T>的类型必须符合要求**

实体类按照如下规则和数据库表进行转换,注解全部是JPA中的注解:

1. 表名默认使用类名,驼峰转下划线,如UserInfo默认对应的表名为user\_info.
2. 表名可以使用@Table(name = "tableName")进行指定,对不符合第一条默认规则的可以通过这种方式指定表名.
3. 字段默认和@Column一样,都会作为表字段,表字段默认为Java对象的Field名字驼峰转下划线形式.
4. 可以使用@Column(name = "fieldName")指定不符合第3条规则的字段名
5. 使用@Transient注解可以忽略字段,添加该注解的字段不会作为表字段使用.
6. 建议一定是有一个@Id注解作为主键的字段,可以有多个@Id注解的字段作为联合主键.
7. 默认情况下,实体类中如果不存在包含@Id注解的字段,所有的字段都会作为主键字段进行使用(这种效率极低).
8. 实体类可以继承使用,可以参考测试代码中的com.github.abel533.model.UserLogin2类.
9. 由于基本类型,如int作为实体类字段时会有默认值0,而且无法消除,所以实体类中建议不要使用基本类型.

除了上面提到的这些,Mapper还提供了序列(支持Oracle)、UUID(任意数据库,字段长度32)、主键自增(类似Mysql,Hsqldb)三种方式，其中序列和UUID可以配置多个，主键自增只能配置一个。

这三种方式不能同时使用,同时存在时按照 序列>UUID>主键自增的优先级进行选择.下面是具体配置方法:

1. 使用序列可以添加如下的注解:
2. *//可以用于数字类型,字符串类型(需数据库支持自动转型)的字段*
3. @SequenceGenerator**(**name**=**"Any"**,**sequenceName**=**"seq\_userid"**)**
4. @Id
5. **private** Integer id**;**
6. 使用UUID时:
7. *//可以用于任意字符串类型长度超过32位的字段*
8. @GeneratedValue**(**generator **=** "UUID"**)**
9. **private** String countryname**;**
10. 使用主键自增:
11. *//不限于@Id注解的字段,但是一个实体类中只能存在一个(继承关系中也只能存在一个)*
12. @Id
13. @GeneratedValue**(**strategy **=** GenerationType**.**IDENTITY**)**

**private** Integer id**;**

## 使用 DevTools 时，通用Mapper经常会出现 class x.x.A cannot be cast to x.x.A。

一个类如果使用了不同的类加载器，就会产生这样的错误，所以解决方案就是让通用Mapper和实体类使用相同的类加载器即可。

DevTools 默认会对 IDE 中引入的所有项目使用 restart 类加载器，对于引入的 jar 包使用 base 类加载器，因此只要保证通用Mapper的jar包使用 restart 类加载器即可。

在 src/main/resources 中创建 META-INF 目录，在此目录下添加 spring-devtools.properties 配置，内容如下：

restart.include.mapper=/mapper-[\\w-\\.]+jar

restart.include.pagehelper=/pagehelper-[\\w-\\.]+jar

使用这个配置后，就会使用 restart 类加载加载 include 进去的 jar 包。

还有就是查看entity是否加了

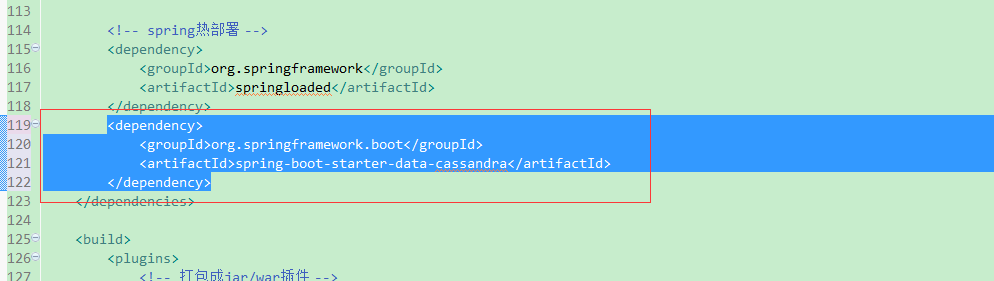
@Id

@Column(name = "id")//把主键值存入User param对象的id属性

@GeneratedValue(strategy = GenerationType.***IDENTITY***)//获取数据库生成的主键

1. org.springframework.beans.factory.BeanCreationException: Error creating bean with name 'session' defined in class path resource [org/springframework/boot/autoconfigure/data/cassandra/CassandraDataAutoConfiguration.class]: Invocation of init method failed; nested exception is com.datastax.driver.core.exceptions.NoHostAvailableException: All host(s) tried for query failed (tried: /C:\Users\tys\AppData\Local\Temp\%W@GJ$ACOF(TYDYECOKVDYB.png192.168.1.20:9042   
     
    (com.datastax.driver.core.exceptions.TransportException: [/C:\Users\tys\AppData\Local\Temp\%W@GJ$ACOF(TYDYECOKVDYB.png192.168.1.20:9042   
     
   ] Cannot connect))  
   at org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.initializeBean(AbstractAutowireCapableBeanFactory.java:1628) ~[spring-beans-4.3.12.RELEASE.jar:4.3.12.RELEASE]  
   at org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.doCreateBean(AbstractAutowireCapableBeanFactory.java:555) ~[spring-beans-4.3.12.RELEASE.jar:4.3.12.RELEASE]  
   at org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.createBean(AbstractAutowireCapableBeanFactory.java:483) ~[spring-beans-4.3.12.RELEASE.jar:4.3.12.RELEASE]  
   at org.springframework.beans.factory.support.AbstractBeanFactory$1.getObject(AbstractBeanFactory.java:306) ~[spring-beans-4.3.12.RELEASE.jar:4.3.12.RELEASE]

解决办法删除



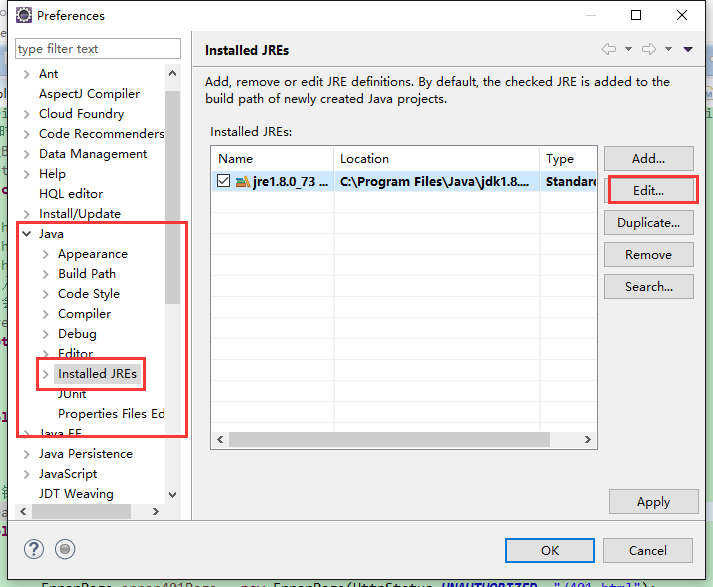
<dependency>

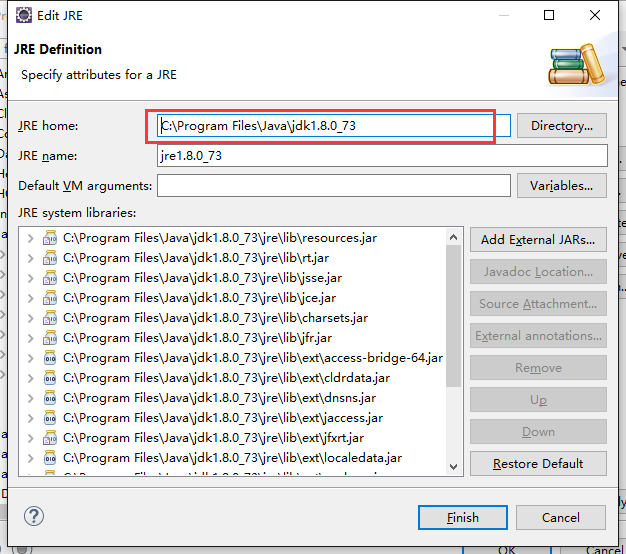
<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-cassandra</artifactId>

</dependency>

# No compiler is provided in this environment. Perhaps you are running on a JRE rather than a JDK





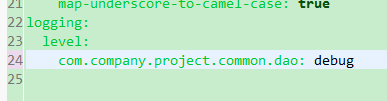
# 常见配置

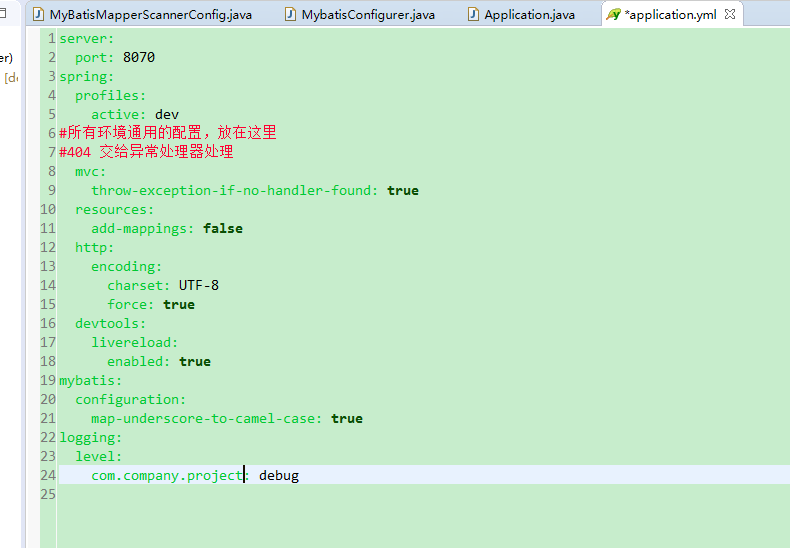
## 日志输出

logging:

level:

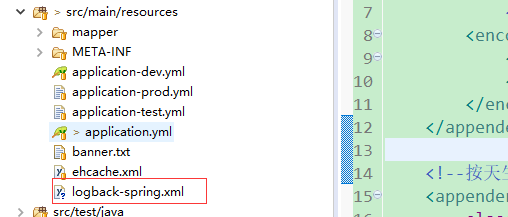
com.company.project: debug





## 日志输出到txt文件中

只需要在mainresource里引入



文件内容如下：

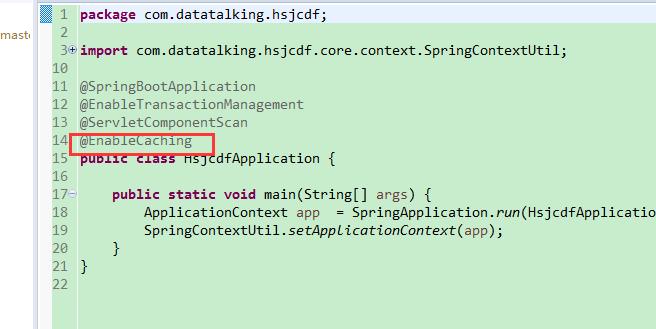


## 主配置文件配置实体类驼峰法不起作用

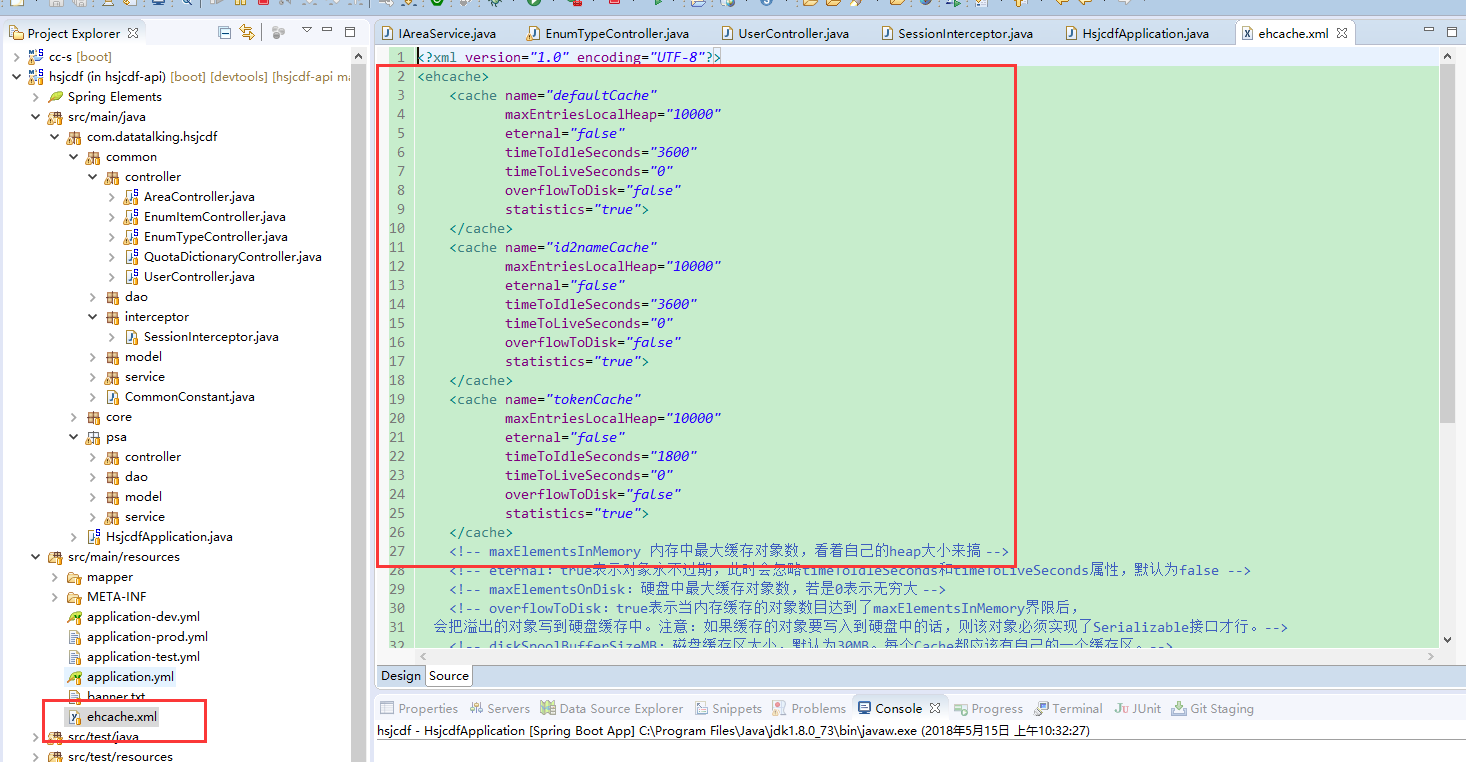
<https://www.cnblogs.com/flying607/p/8473075.html>



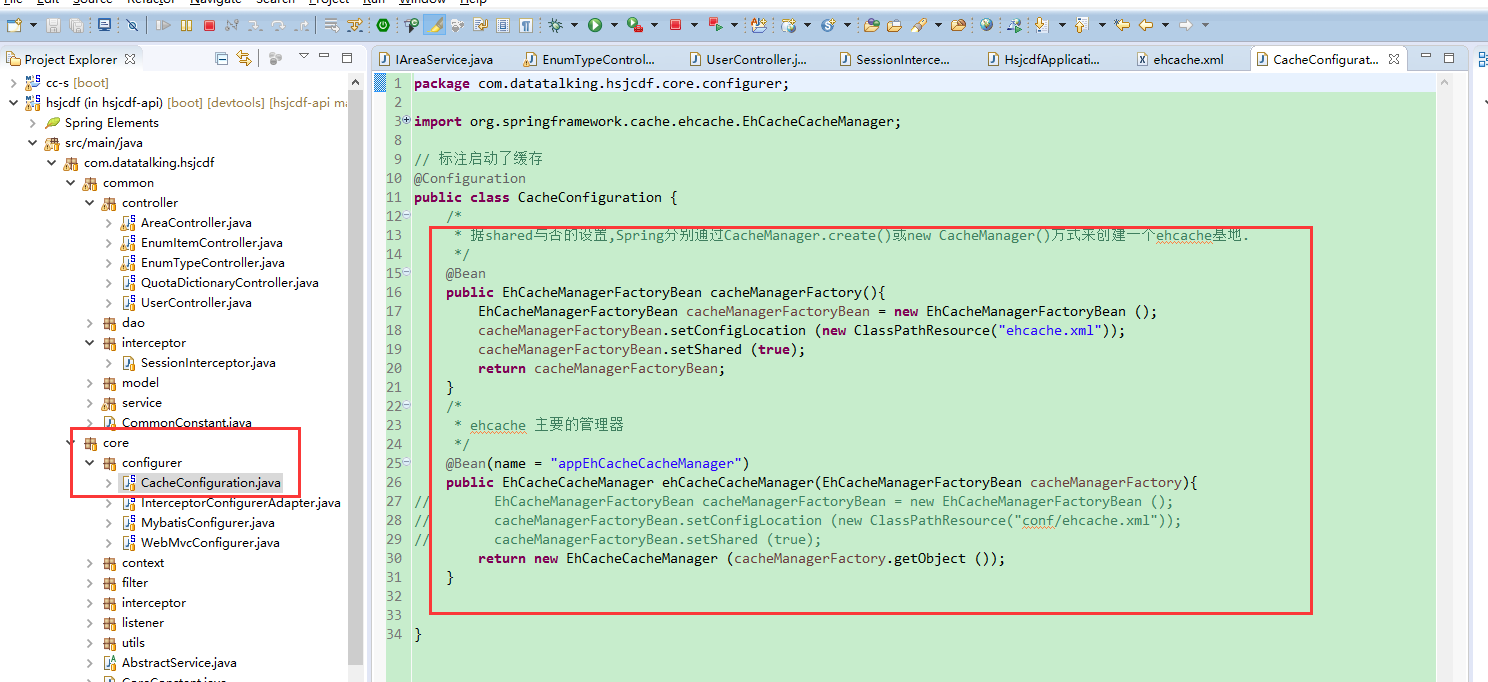
# 缓存



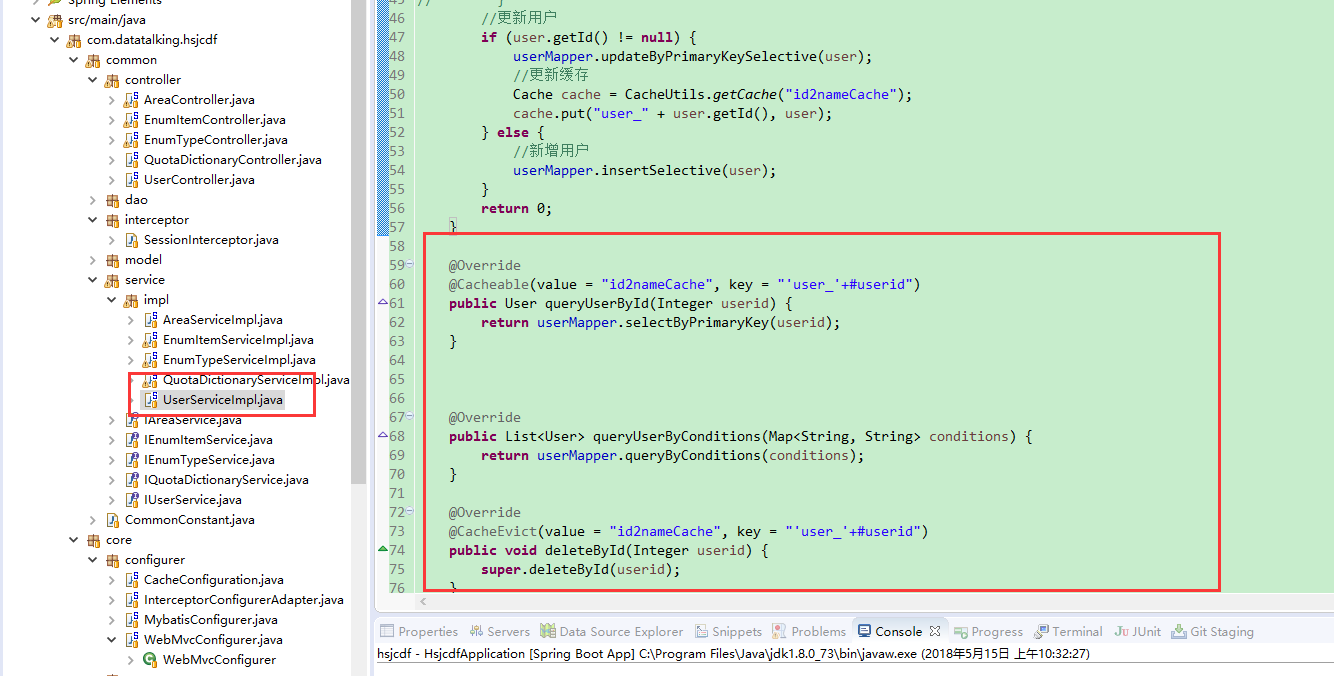
## 缓存定义



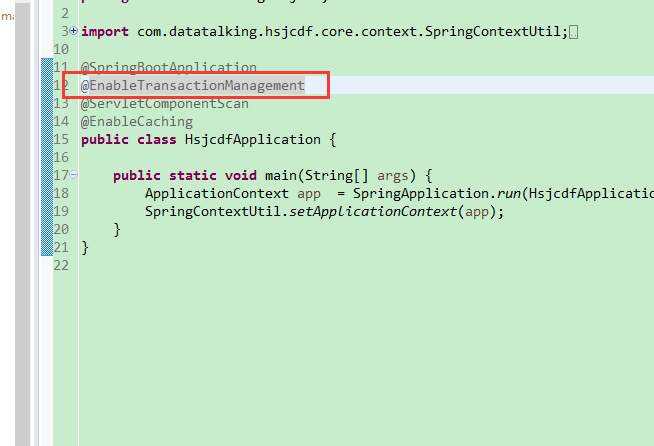
## 缓存注入

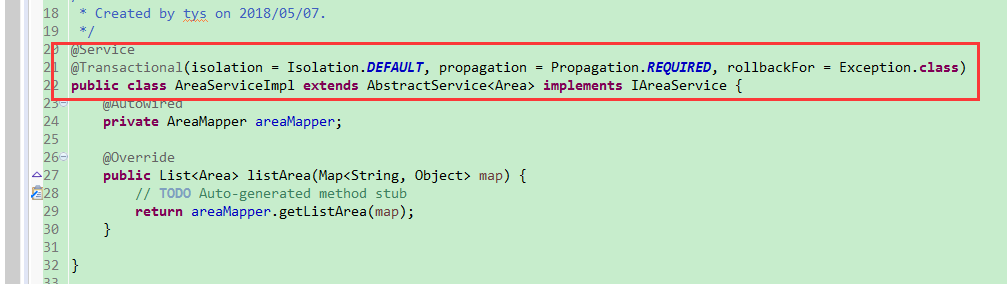


## 缓存使用



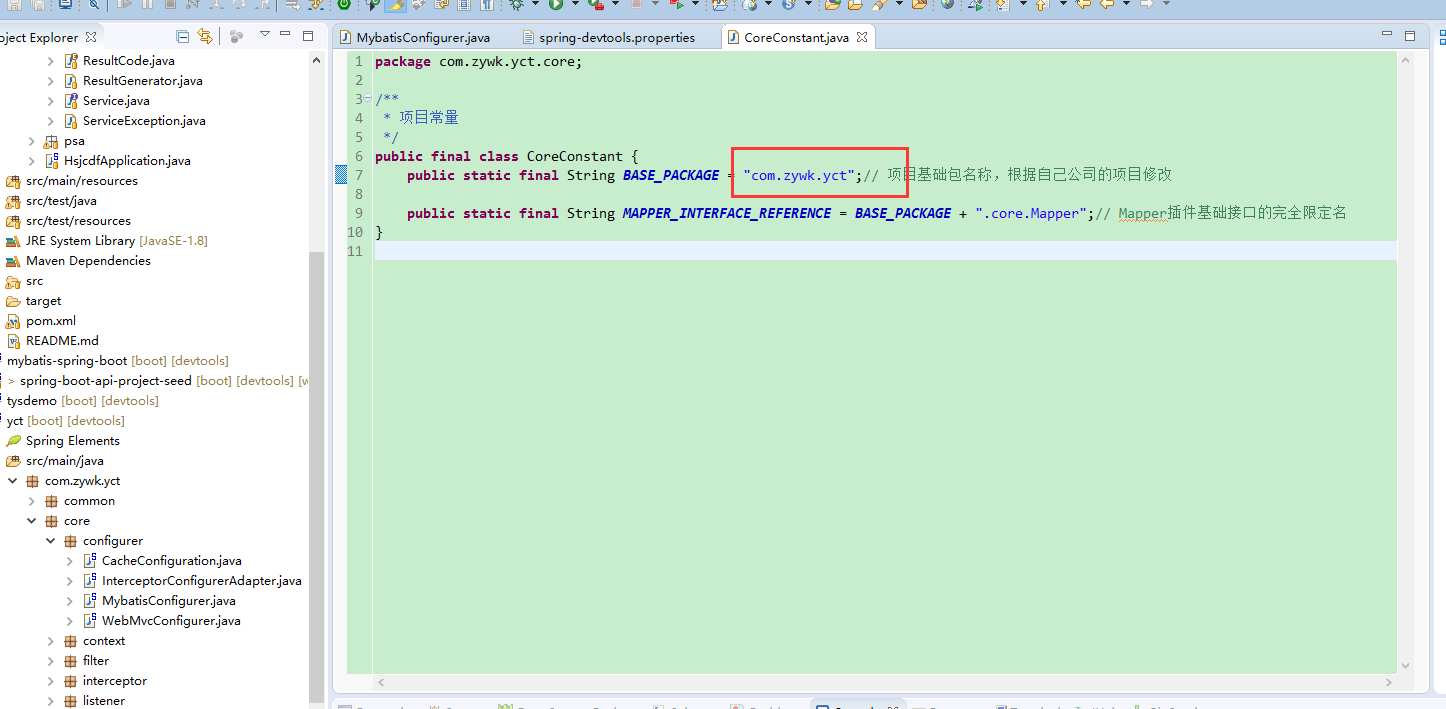
# 数据库事物





# 新建项目需要修改的地方

1、



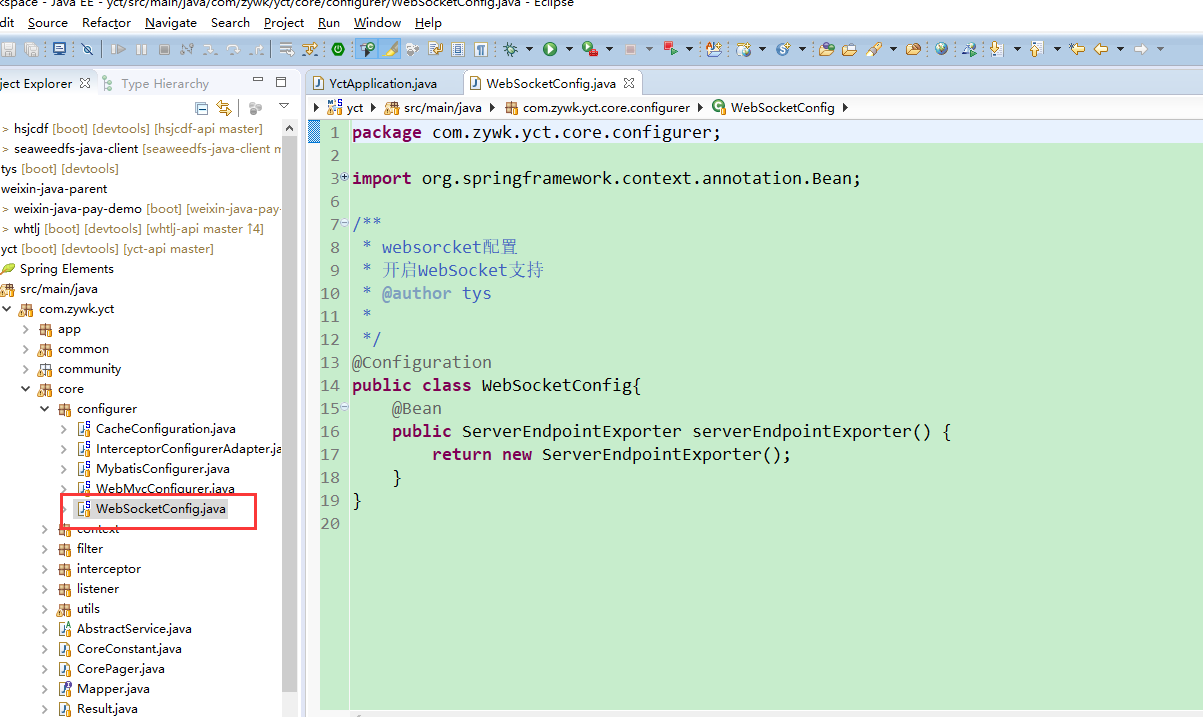
改成对应项目名

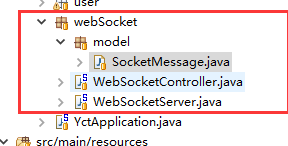
2、利用CodeGenerator生成一个对应文件

# 完全复制项目需要修改的地方

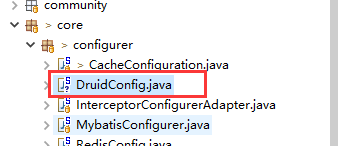
* 1. 常量coreconstant如果又对应mapper需要修改对应mapper
  2. 重命名main/java下的文件
  3. Pom.Xml文件修改

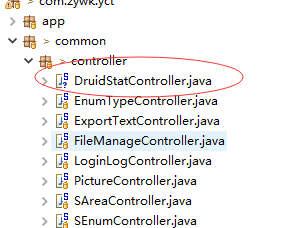
# Websocket

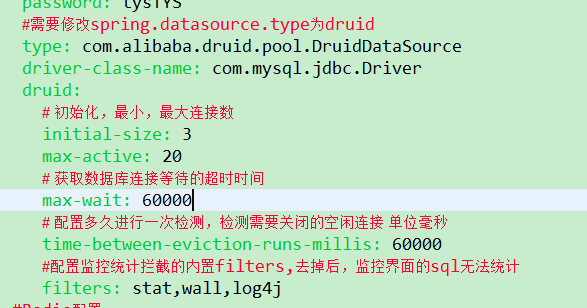




# Druid配置







访问：<http://localhost:8010/druid/sql.html>

# 緩存注解

<https://www.cnblogs.com/fashflying/p/6908028.html>

# 1       基于注解的支持

       Spring为我们提供了几个注解来支持Spring Cache。其核心主要是@Cacheable和@CacheEvict。使用@Cacheable标记的方法在执行后Spring Cache将缓存其返回结果，而使用@CacheEvict标记的方法会在方法执行前或者执行后移除Spring Cache中的某些元素。下面我们将来详细介绍一下Spring基于注解对Cache的支持所提供的几个注解。

## 1.1    @Cacheable

       @Cacheable可以标记在一个方法上，也可以标记在一个类上。当标记在一个方法上时表示该方法是支持缓存的，当标记在一个类上时则表示该类所有的方法都是支持缓存的。对于一个支持缓存的方法，Spring会在其被调用后将其返回值缓存起来，以保证下次利用同样的参数来执行该方法时可以直接从缓存中获取结果，而不需要再次执行该方法。Spring在缓存方法的返回值时是以键值对进行缓存的，值就是方法的返回结果，至于键的话，Spring又支持两种策略，默认策略和自定义策略，这个稍后会进行说明。需要注意的是当一个支持缓存的方法在对象内部被调用时是不会触发缓存功能的。@Cacheable可以指定三个属性，value、key和condition。

### 1.1.1  value属性指定Cache名称

       value属性是必须指定的，其表示当前方法的返回值是会被缓存在哪个Cache上的，对应Cache的名称。其可以是一个Cache也可以是多个Cache，当需要指定多个Cache时其是一个数组。

   @Cacheable("cache1")//Cache是发生在cache1上的

   public User find(Integer id) {

      returnnull;

   }

   @Cacheable({"cache1", "cache2"})//Cache是发生在cache1和cache2上的

   public User find(Integer id) {

      returnnull;

   }

### 1.1.2  使用key属性自定义key

       key属性是用来指定Spring缓存方法的返回结果时对应的key的。该属性支持SpringEL表达式。当我们没有指定该属性时，Spring将使用默认策略生成key。我们这里先来看看自定义策略，至于默认策略会在后文单独介绍。

       自定义策略是指我们可以通过Spring的EL表达式来指定我们的key。这里的EL表达式可以使用方法参数及它们对应的属性。使用方法参数时我们可以直接使用“#参数名”或者“#p参数index”。下面是几个使用参数作为key的示例。

   @Cacheable(value="users", key="#id")

   public User find(Integer id) {

      returnnull;

   }

   @Cacheable(value="users", key="#p0")

   public User find(Integer id) {

      returnnull;

   }

   @Cacheable(value="users", key="#user.id")

   public User find(User user) {

      returnnull;

   }

   @Cacheable(value="users", key="#p0.id")

   public User find(User user) {

      returnnull;

   }

       除了上述使用方法参数作为key之外，Spring还为我们提供了一个root对象可以用来生成key。通过该root对象我们可以获取到以下信息。

|  |  |  |
| --- | --- | --- |
| 属性名称 | 描述 | 示例 |
| methodName | 当前方法名 | #root.methodName |
| method | 当前方法 | #root.method.name |
| target | 当前被调用的对象 | #root.target |
| targetClass | 当前被调用的对象的class | #root.targetClass |
| args | 当前方法参数组成的数组 | #root.args[0] |
| caches | 当前被调用的方法使用的Cache | #root.caches[0].name |

       当我们要使用root对象的属性作为key时我们也可以将“#root”省略，因为Spring默认使用的就是root对象的属性。如：

   @Cacheable(value={"users", "xxx"}, key="caches[1].name")

   public User find(User user) {

      returnnull;

   }

### 1.1.3  condition属性指定发生的条件

       有的时候我们可能并不希望缓存一个方法所有的返回结果。通过condition属性可以实现这一功能。condition属性默认为空，表示将缓存所有的调用情形。其值是通过SpringEL表达式来指定的，当为true时表示进行缓存处理；当为false时表示不进行缓存处理，即每次调用该方法时该方法都会执行一次。如下示例表示只有当user的id为偶数时才会进行缓存。

   @Cacheable(value={"users"}, key="#user.id", condition="#user.id%2==0")

   public User find(User user) {

      System.out.println("find user by user " + user);

      return user;

   }

## 1.2     @CachePut

       在支持Spring Cache的环境下，对于使用@Cacheable标注的方法，Spring在每次执行前都会检查Cache中是否存在相同key的缓存元素，如果存在就不再执行该方法，而是直接从缓存中获取结果进行返回，否则才会执行并将返回结果存入指定的缓存中。@CachePut也可以声明一个方法支持缓存功能。与@Cacheable不同的是使用@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。

       @CachePut也可以标注在类上和方法上。使用@CachePut时我们可以指定的属性跟@Cacheable是一样的。

   @CachePut("users")//每次都会执行方法，并将结果存入指定的缓存中

   public User find(Integer id) {

      returnnull;

   }

## 1.3     @CacheEvict

       @CacheEvict是用来标注在需要清除缓存元素的方法或类上的。当标记在一个类上时表示其中所有的方法的执行都会触发缓存的清除操作。@CacheEvict可以指定的属性有value、key、condition、allEntries和beforeInvocation。其中value、key和condition的语义与@Cacheable对应的属性类似。即value表示清除操作是发生在哪些Cache上的（对应Cache的名称）；key表示需要清除的是哪个key，如未指定则会使用默认策略生成的key；condition表示清除操作发生的条件。下面我们来介绍一下新出现的两个属性allEntries和beforeInvocation。

### 1.3.1  allEntries属性

       allEntries是boolean类型，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，这比一个一个清除元素更有效率。

   @CacheEvict(value="users", allEntries=true)

   public void delete(Integer id) {

      System.out.println("delete user by id: " + id);

   }

### 1.3.2  beforeInvocation属性

       清除操作默认是在对应方法成功执行之后触发的，即方法如果因为抛出异常而未能成功返回时也不会触发清除操作。使用beforeInvocation可以改变触发清除操作的时间，当我们指定该属性值为true时，Spring会在调用该方法之前清除缓存中的指定元素。

   @CacheEvict(value="users", beforeInvocation=true)

   public void delete(Integer id) {

      System.out.println("delete user by id: " + id);

   }

       其实除了使用@CacheEvict清除缓存元素外，当我们使用Ehcache作为实现时，我们也可以配置Ehcache自身的驱除策略，其是通过Ehcache的配置文件来指定的。由于Ehcache不是本文描述的重点，这里就不多赘述了，想了解更多关于Ehcache的信息，请查看我关于Ehcache的专栏。

## 1.4     @Caching

       @Caching注解可以让我们在一个方法或者类上同时指定多个Spring Cache相关的注解。其拥有三个属性：cacheable、put和evict，分别用于指定@Cacheable、@CachePut和@CacheEvict。

   @Caching(cacheable = @Cacheable("users"), evict = { @CacheEvict("cache2"),

         @CacheEvict(value = "cache3", allEntries = true) })

   public User find(Integer id) {

      returnnull;

   }

## 1.5     使用自定义注解

       Spring允许我们在配置可缓存的方法时使用自定义的注解，前提是自定义的注解上必须使用对应的注解进行标注。如我们有如下这么一个使用@Cacheable进行标注的自定义注解。

@Target({ElementType.TYPE, ElementType.METHOD})

@Retention(RetentionPolicy.RUNTIME)

@Cacheable(value="users")

public @interface MyCacheable {

}

       那么在我们需要缓存的方法上使用@MyCacheable进行标注也可以达到同样的效果。

   @MyCacheable

   public User findById(Integer id) {

      System.out.println("find user by id: " + id);

      User user = new User();

      user.setId(id);

      user.setName("Name" + id);

      return user;

   }

# git上传

<https://blog.csdn.net/cnjy_/article/details/78916728>

**git init**

**git add .**

**git commit -m "first commit"**

**git remote add origin** [**https://github.com/cnjy/test.git**](https://github.com/cnjy/test.git)

可以通过如下命令进行代码合并 注：pull=fetch+merge]

**git pull --rebase origin master**

执行上面代码后可以看到本地代码库中多了README.md文件

此时再执行语句 **git push -u origin master**即可完成代码上传