将SSM项目转为springboot项目，在eclipse中实现

1.在eclipse中下载spring boot插件：

（1）在eclipse中选择Help,然后选择Eclipse Marketplace

（2）在显示出的界面选择Popularbiaoqian ,见到绿色的Spring Tools，就install（要在有网条件下）

（3）重启eclipse，新建一个工程时可以看到Spring Starter Project,选择它建工程

2.转移pom.xml

（1）有关于spring的依赖要删除，因为springboot已经帮我们自动装好了

（2）删除test依赖，springboot也自带了

（3）转移其他依赖后，发现标黄的版本信息部分就删除，选用springboot自带的版本

3.dao的转移

（1）将jdbc.properties的信息转到application.properties或application.yml中

#DataSource

//数据库驱动

jdbc.driver=com.mysql.jdbc.Drivaer

//数据库连接

jdbc.url=jdbc:mysql://39.107.100.210:3306/o2o?userUnicode=true&characterEncoding=utf8&useSSL=false

//数据库用户名（已加密）

jdbc.username=hGzKSsPSkBQ=

//数据库密码（已加密）

jdbc.password=msriOV2gJm8vWuqSFZA9oQ==

1. 将mybatis-config.xml和logback.xml移动到springboot下的resource中，在application.properties配置一些mybatis的相关属性

#mybatis

#全局mybatis-config.xml配置文件的路径

mybatis\_config\_file=mybatis-config.xml

#扫描sql配置文件:mapper需要的xml文件

mybatis\_path=/mapper/\*\*.xml

#扫描entity包

type\_alias\_packer=com.weilai.o2o.entity

1. 模仿以前的spring-dao.xml进行配置类的设置，主要是两个类，分别是DataSourceConfiguraton和SessionConfiguration.

@Configuration

//配置 mybatis mapper的扫描路径

@MapperScan("com.weilai.o2o.dao")

**public** **class** DataSourceConfiguration {

/\*\*

\* 通过这些@Value标签就可以在application.properties中读出数据库的配置信息

\*/

@Value("${jdbc.driver}")

**private** String jdbcDriver;

@Value("${jdbc.url}")

**private** String jdbcUrl;

@Value("${jdbc.username}")

**private** String jdbcUsername;

@Value("${jdbc.password}")

**private** String jdbcPassword;

/\*\*

\* 生成与spring-dao.xml对应的bean dataSource

\* <bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">

<!-- 配置连接池属性 -->

<property name="driverClass" value="${jdbc.driver}" />

<property name="jdbcUrl" value="${jdbc.url}" />

<property name="user" value="${jdbc.username}" />

<property name="password" value="${jdbc.password}" />

<!-- c3p0连接池的私有属性 -->

<property name="maxPoolSize" value="30" />

<property name="minPoolSize" value="10" />

<!-- 关闭连接后不自动commit -->

<property name="autoCommitOnClose" value="false" />

<!-- 获取连接超时时间 -->

<property name="checkoutTimeout" value="10000" />

<!-- 当获取连接失败重试次数 -->

<property name="acquireRetryAttempts" value="2" />

</bean>

\* **@return**

\* **@throws** PropertyVetoException

\*/

@Bean(name = "dataSource")

**public** ComboPooledDataSource dataSource() **throws** PropertyVetoException {

ComboPooledDataSource dataSource = **new** ComboPooledDataSource();

//设置驱动

dataSource.setDriverClass(jdbcDriver);

//设置连接数据库的URL

dataSource.setJdbcUrl(jdbcUrl);

//设置用户名,要解密

dataSource.setUser(DESUtil.*getDecryptString*(jdbcUsername));

//设置密码,要解密

dataSource.setPassword(DESUtil.*getDecryptString*(jdbcPassword));

//c3p0连接池的私有属性

dataSource.setMaxPoolSize(30);

dataSource.setMinPoolSize(10);

//关闭连接后不自动连接commit

dataSource.setAutoCommitOnClose(**false**);

//获取连接超时时间

dataSource.setCheckoutTimeout(10000);

//当连接失败重试次数

dataSource.setAcquireRetryAttempts(2);

**return** dataSource;

}

}

@Configuration

**public** **class** SessionConfiguration {

@Autowired

DataSource dataSource;

//mybatis-config.xml配置文件的路径

**private** **static** String *mybatisConfigFile*;

@Value("${mybatis\_config\_file}")

**public** **void** setMybatisConfigFile(String mybatisConfigFile) {

SessionConfiguration.*mybatisConfigFile* = mybatisConfigFile;

}

//mybatis mapper文件所在路径 即扫描sql配置文件:mapper需要的xml文件

**private** **static** String *mapperPath*;

@Value("${mybatis\_path}")

**public** **void** setMapperPath(String mapperPath) {

SessionConfiguration.*mapperPath* = mapperPath;

}

//扫描entity包 使用别名

@Value("${type\_alias\_packer}")

**private** String typeAliasPackage;

/\*\*

\* 创建SQLSessionFactoryBean实例

\* 设置configuration 设置mapper映射路径 设置dataSource数据源

\* **@return**

\* **@throws** IOException

\*/

@Bean(name = "sqlSessionFactory")

**public** SqlSessionFactoryBean createsqlSessionFactory() **throws** IOException {

SqlSessionFactoryBean sqlSessionFactoryBean = **new** SqlSessionFactoryBean();

//设置mybatis configuration 扫描路径

sqlSessionFactoryBean.setConfigLocation(**new** ClassPathResource(*mybatisConfigFile*));

//设置typeAlias 包扫描路径

sqlSessionFactoryBean.setTypeAliasesPackage(typeAliasPackage);

//添加mapper扫描路径

PathMatchingResourcePatternResolver pathMatchingResourcePatternResolver = **new** PathMatchingResourcePatternResolver();

String packageSearchPath = ResourcePatternResolver.***CLASSPATH\_ALL\_URL\_PREFIX***+*mapperPath*; sqlSessionFactoryBean.setMapperLocations(pathMatchingResourcePatternResolver.getResources(packageSearchPath));

//设置dataSource

sqlSessionFactoryBean.setDataSource(dataSource);

**return** sqlSessionFactoryBean;

}

}

1. 在DateSourceConfiguration设置数据库的属性，即jdbc.properties中的那4个属性，并用@Value将application.properties中的数据库值赋予进去，然后定义一个public ComboPooledDataSource dataSource()方法将原本SSM下的spring-dao.xml对应的bean dataSource下的属性设置进这个方法里。
2. 在SessionConfiguration中通过@Value方法从application.properties中获取 mybatis-config.xml配置文件，扫描sql配置文件:mapper需要的xml文件等的路径，然后定义一个public SqlSessionFactoryBean createsqlSessionFactory()方法，将原本SSM下得spring-dao.xml对应的bean sqlSessionFactory下的属性设置进这个方法中
3. 将dao、entity、test、mapper几个相关的包加进来，进行测试。
4. service的转移
5. 模仿以前的spring-service.xml进行配置类的设置，主要是配置TransactionManagementConfiguration。

/\*\*

\* 对标spring-service.xml里面的transactionManageer

\* 实现TransactionManagementConfigurer是因为开启annotation-driven

\* **@author** ASUS

\*

\*/

@Configuration

//首先使用注解@@EnableTransactionManagement开启事务支持后

//在service方法上添加注解@Transactional便可，相当于配置基于注解的声明式事务：<tx:annotation-driven transaction-manager="transactionManager" />

@EnableTransactionManagement

**public** **class** TransactionManagementConfiguration **implements** TransactionManagementConfigurer {

@Autowired

//注入DataSourceConfiguretion里边的dataSource，通过createDataSource()获取

DataSource dataSource;

/\*\*

\* 关于事务管理，需要返回PlatformTransactionManager的实现

\*/

@Override

**public** PlatformTransactionManager annotationDrivenTransactionManager() {

// **TODO** Auto-generated method stub

**return** **new** DataSourceTransactionManager(dataSource);

}

}

1. 对spring-redis.xml进行配置，首先引进了cache包下的两个Redis类，分别是JedisPoolWriper和JedisUtil，接着将原SSM下的Redis.properties的内容转移到application.properties中，然后在config包下的RedisConfiguration编写Redis的配置信息。最后引进service、dto、enum、test登报进行测试。

@Configuration

**public** **class** RedisConfiguration {

@Value("${redis.hostname}")

**private** String hostname;

@Value("${redis.port}")

**private** **int** port;

@Value("${redis.pool.maxActive}")

**private** **int** maxTotal;

@Value("${redis.pool.maxIdle}")

**private** **int** maxIdle;

@Value("${redis.pool.maxWait}")

**private** **long** maxWaitMillis;

@Value("${redis.pool.testOnBorrow}")

**private** **boolean** testOnBorrow;

//连接池的设置

@Autowired

**private** JedisPoolConfig jedisPoolConfig;

//创建连接池，并做相关配置

@Autowired

**private** JedisPoolWriper jedisWritePool;

//Redis的工具类

@Autowired

**private** JedisUtil jedisUtil;

/\*\*

\* 创建Redis连接池的配置

\* **@return**

\*/

@Bean(name ="jedisPoolConfig")

**public** JedisPoolConfig createJedisPoolConfig() {

JedisPoolConfig jedisPoolConfig = **new** JedisPoolConfig();

//控制一个pool可分配多少个jedis实例

jedisPoolConfig.setMaxTotal(maxTotal);

//连接池中最多可空闲maxIdle个连接，这里取值为20

//表示即使没有数据库连接时依然可以保持20空闲的连接

//而不清除，随时处于待命状态

jedisPoolConfig.setMaxIdle(maxIdle);

//最大等待时间，当没有可用连接时

//连接池等待连接被归还的最大时间（以毫秒技术），超过时间则跳出异常

jedisPoolConfig.setMaxWaitMillis(maxWaitMillis);

//在获取连接的时候检查有效性

jedisPoolConfig.setTestOnBorrow(testOnBorrow);

**return** jedisPoolConfig;

}

/\*\*

\* 创建Redis连接池，并做相关配置

\* **@return**

\*/

@Bean(name ="jedisWritePool")

**public** JedisPoolWriper createJedisPoolWriper() {

JedisPoolWriper jedisPoolWriper = **new** JedisPoolWriper(jedisPoolConfig, hostname, port);

**return** jedisPoolWriper;

}

/\*\*

\* 创建Redis工具类，封装好Redis的连接已进行相关操作

\* **@return**

\*/

@Bean(name ="jedisUtil")

**public** JedisUtil createJedisUtil() {

JedisUtil jedisUtil = **new** JedisUtil();

jedisUtil.setJedisPool(jedisWritePool);

**return** jedisUtil;

}

/\*\*

\* Redis的key操作

\* **@return**

\*/

@Bean(name ="jedisKeys")

**public** JedisUtil.Keys createKeys(){

JedisUtil.Keys jedisKeys = jedisUtil.**new** Keys();

**return** jedisKeys;

}

/\*\*

\* Redis的key操作

\* **@return**

\*/

@Bean(name ="jedisStrings")

**public** JedisUtil.Strings createStrings(){

JedisUtil.Strings jedisStrings = jedisUtil.**new** Strings();

**return** jedisStrings;

}

}

1. web的转移

（1）模仿以前的spring-web.xml进行配置类的设置，主要是MvcConfiguration的配置。

/\*\*

\* 开始MVC，自动注入spring容器，WebMvcConfigurerAdapter：配置视图解析器

\* 当一个类实现了这个接口（ApplicationContextAware）之后，这个类

\* 就方便获得ApplicationContext中所有的bean

\* **@author** ASUS

\*

\*/

@Configuration

//相当于<mvc:annotation-driven>，开启spring MVC注解模式

@EnableWebMvc

**public** **class** MvcConfiguration **extends** WebMvcConfigurerAdapter **implements** ApplicationContextAware {

//spring 容器

**private** ApplicationContext applicationContext;

//获取spring容器

@Override

**public** **void** setApplicationContext(ApplicationContext applicationContext) **throws** BeansException {

**this**.applicationContext = applicationContext;

}

/\*\*

\* 静态资源的配置

\* 静态资源默认servlet配置 加入对静态资源的处理：js,gif,png (2)允许使用"/"做整体映射

\* 即拦截 /resources/\*\*的请求，将其转移到类路径下的classpaht:/resources/\*\*

\* <mvc:resources mapping="/resources/\*\*" location="/resources/" />

\*/

**public** **void** addResourceHandlers(ResourceHandlerRegistry registry) {

registry.addResourceHandler("/resources/\*\*").addResourceLocations("classpaht:/resources/\*\*");

}

/\*\*

\* 定义默认的请求处理器

\*

\* <mvc:default-servlet-handler />

\* **@param** condiConfigurer

\*/

**public** **void** configureDefauilServletHandling(DefaultServletHandlerConfigurer condiConfigurer) {

condiConfigurer.enable();

}

/\*\*

\* 定义视图解析器

\* <bean id="viewResolver"

class="org.springframework.web.servlet.view.InternalResourceViewResolver">

<property name="prefix" value="/WEB-INF/html/"></property>

<property name="suffix" value=".html"></property>

</bean>

\* **@return**

\*/

@Bean(name = "viewResolver")

**public** ViewResolver createViewResolver() {

InternalResourceViewResolver viewResolver = **new** InternalResourceViewResolver();

//设置spring容器

viewResolver.setApplicationContext(**this**.applicationContext);

//取消缓存

viewResolver.setCache(**false**);

//设置解析的前缀

viewResolver.setPrefix("/WEB-INF/html/");

//设置解析的后缀

viewResolver.setSuffix(".html");

**return** viewResolver;

}

**public** CommonsMultipartResolver createMultipartResolver() {

CommonsMultipartResolver multipartResolver = **new** CommonsMultipartResolver();

multipartResolver.setDefaultEncoding("utf-8");

//1024\*1024\*20=20M

multipartResolver.setMaxUploadSize(20971520);

multipartResolver.setMaxInMemorySize(20971520);

**return** multipartResolver;

}

}

1. 将web层的类转移进来，在页面进行测试。