SSM(1)

1、新建一个界面，使之访问testOne.do能够正常访问

```
@RequestMapping("test1.do")
public String TestOne(){
    return "testOne";
}
```

1）在web.xml中设置springmvc

web.xml

```
    <welcome-file>default.htm</welcome-file>
    <welcome-file>default.jsp</welcome-file>
  </welcome-file-list>

  <servlet>
    <servlet-name>springdemo</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <!-- 设置springmvc配置文件地址 【考虑后期需要加载多个配置文件 so 设置的地址为spring-*.xml】 -->
            <param-value>
                classpath:config/spring-*.xml
            </param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet-mapping>
    <servlet-name>springdemo</servlet-name>
    <url-pattern>*.do</url-pattern>
  </servlet-mapping>
```

2）编写springmvc的配置文件

spring-mvc.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:context="http://www.springframework.org/schema/context"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-4.1.xsd
http://www.springframework.org/schema/mvc http://www.springframework.org/schema/mvc/spring-mvc-4.1.xsd">
                                                                                    域名空间
    <mvc:annotation-driven/>
                开启spring注解
    <context:component-scan base-package="com.tx.controller" />
    <context:component-scan base-package="com.tx.service" />
                设置扫描的包【注解所在的包】

    <bean      允许创建请求地址url
        class="org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandlerMapping" />

    <bean       允许将请求地址绑定给对应的处理事件（方法）                                    开启springmvc的注解
        class="org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter" />


    <bean
        class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <!--JstlView表示JSP模板页面需要使用JSTL标签库，classpath中必须包含jstl的相关jar包 -->
        <property name="viewClass"
            value="org.springframework.web.servlet.view.JstlView" />
        <!-- 查找视图页面的前缀和后缀（前缀[逻辑视图名]后缀），比如传进来的逻辑 视图名为hello，则该jsp视图页面应该存放在“WEB-INF/jsp/hello.jsp” -->
        <property name="prefix" value="/page/" />
        <property name="suffix" value=".jsp" />
    </bean>                                          设置视图层的位置及视图层的类型


</beans>
```

3）编写控制层

MyController.java

```java
package com.tx.controller;

import java.util.List;

@Controller 控制层注解，声明这是一个控制器
public class MyController {

    @Autowired 使用注入的方式得到service层对象
    MyService service;

    @RequestMapping("testOne.do") 第一个测试案例的请求地址
    public String testOne(){
        System.out.println("进入第一个测试");
        return "testOne";
    }  处理完成后跳转的视图名称

```

2、第二第三方法能请求

```java
    @RequestMapping("test2.do")
    public String TestTwo(HttpServletRequest request,String username){
        String result = is.testTwo(username);
        request.setAttribute("testTwo", result);
        return "testTwo";
    }

    @RequestMapping("test3.do")
    public String TestThree(HttpServletRequest request,String username){
        String re = is.testThree(username);
        request.setAttribute("testTwo", re);
        return "testTwo";
    }
```

1）添加service层，编写service层处理方法

MyService.java

```java
package com.tx.service;

import java.util.List;

服务层注解
@Service
public class MyService {
                                    服务层处理方法
    public String test2(String name){
        if("admin".equals(name)){
            return "账号匹配正确";
        }else{
            return "账号错误";
        }
    }


    public String test3(String name){

        System.out.println(name.length());

        if("admin".equals(name)){
            return "账号匹配正确";
        }else{
            return "账号错误";
        }
    }


}

```

3、给service层下所有的类中所有的方法设置成切入点，创建切面类，设置处理方法

```xml
    <!-- 声明一个目标对象（通知对象） -->
    <bean id="aspectbean" class="com.tx.aspect.MyAspect" />

    <!-- proxy-target-class="true" 允许使用CGlib动态代理 -->
    <aop:config proxy-target-class="true">
        <!-- 设置切入点，并给切入点起别名 -->
        <aop:pointcut id="pointcut" expression="execution(* com.tx.service.*.*(..))" />
        <!-- 设置切入点所对应的目标对象（通知对象） -->
        <aop:aspect id="aspect" ref="aspectbean" order="1">
            <!-- 给目标对象指定处理方法 doRound -->
            <aop:around pointcut-ref="pointcut" method="doRound" />
        </aop:aspect>
    </aop:config>
```

4、显示所有用户信息，通过用户名获取用户权限列表

编号：1 用户名：admin 姓名：管理员11

编号：2 用户名：user 姓名：王书强

编号：3 用户名：user2 姓名：你说呢

编号：4 用户名：user3 姓名：王书强

编号：5 用户名：user4 姓名：我终于有名字了

编号：6 用户名：cao 姓名：王f

编号：7 用户名：cao2 姓名：王大锤222

编号：8 用户名：aaa 姓名：我是偷渡的

编号：9 用户名：123 姓名：tx1

编号：10 用户名：123 姓名：tx2

1）配置MyBatis

①配置c3p0连接池

```
    <!-- 1、配置c3p0 -->
    <bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource" destroy-method="close">
        <property name="driverClass">
            <value>com.mysql.jdbc.Driver</value>
        </property>
        <property name="jdbcUrl">
            <value>jdbc:mysql://localhost:3306/power?useUnicode=true&amp;characterEncoding=UTF-8</value>
        </property>
        <property name="user">
            <value>root</value>
        </property>
        <property name="password">
            <value>123</value>
        </property>
        <!--当连接池中的连接耗尽的时候c3p0一次同时获取的连接数。Default: 3 -->
        <property name="acquireIncrement">
            <value>3</value>
        </property>
        <!--初始化时获取三个连接。Default: 3 -->
        <property name="initialPoolSize">
            <value>3</value>
        </property>
        <property name="minPoolSize">
            <value>5</value>
        </property>
        <property name="maxPoolSize">
            <value>100</value>
        </property>
        <!--最大空闲时间,60秒内未使用则连接被丢弃。若为0则永不丢弃。Default: 0 -->
        <property name="maxIdleTime">
            <value>60</value>
        </property>
```

```
            <value>60</value>
        </property>
        <!--每60秒检查所有连接池中的空闲连接。Default: 0 -->
        <property name="idleConnectionTestPeriod">
            <value>60</value>
        </property>
        <!-- JDBC的标准参数，用以控制数据源内加载的PreparedStatements数量。但由于预缓存的statements 属于单个connection而不是整个连接池。所以设置这个参数需要考虑到多方面的因素。
            如果maxStatements与maxStatementsPerConnection均为0，则缓存被关闭。Default: 0 -->
        <property name="maxStatements">
            <value>100</value>
        </property>
        <!-- 通过多线程实现多个操作同时被执行。Default: 3 -->
        <property name="numHelperThreads">
            <value>10</value>
        </property>
    </bean>
```

②设置sqlsession工厂并进行配置

```

    <!-- 2、这是sqlsession工厂 -->
    <!-- 配置mybaits连接工厂类 -->
    <bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
        <property name="dataSource" ref="dataSource" />
        <!-- 指定mybatis配置文件路径 -->
        <property name="configLocation" value="classpath:config/SqlMapConfig.xml"></property>
    </bean>

    <!-- 构造方法形式注入工厂类至sqlSessionTemplate -->
    <bean id="sqlSessionTemplate" class="org.mybatis.spring.SqlSessionTemplate">
        <constructor-arg ref="sqlSessionFactory"></constructor-arg>
    </bean>
```

在sqlsession配置中指定mybatis的配置文件位置

③设置事务

```
        <!-- 3、配置事务 -->
    <!-- 配置事务驱动 -->
    <bean id="TransactionManager" name="txManager"
        class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource"></property>
    </bean>
    <tx:annotation-driven transaction-manager="txManager" />
```

④扫描对应的包

```
    <!-- 4、持久层扫描 -->
    <!-- 扫描basePackage下所有以@MyBatisRepository标识的 接口 -->
    <bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
        <!--扫描的包路径 -->
        <property name="basePackage" value="com.tx.dao" />
    </bean>
```

2）创建dao层接口

```
@Repository dao层注解
public interface UserDao {

    /**
     * 获取全部用户的信息
     * @return
     */
    public List<UserBean> getUserList();

}
```

3)创建对应的xml来实现UserDao接口

UserDao.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper
PUBLIC "-//ibatis.apache.org//DTD Mapper 3.0//EN"
"http://ibatis.apache.org/dtd/ibatis-3-mapper.dtd">
<mapper namespace="com.tx.dao.UserDao">

    <!-- 预设一个结果集，将model与数据库的数据进行匹配 column:代表数据库字段  property：class里面属性名称-->
    <resultMap type="com.tx.model.UserBean" id="user">
        <result column="id" property="id" />
        <result column="t_user_name" property="t_user_name"/>
        <result column="t_password" property="t_password"/>
        <result column="t_name" property="t_name"/>
        <result column="t_role_id" property="t_role_id"/>
        <result column="t_state" property="t_state"/>
    </resultMap>

    <select id="getUserList" resultMap="user" >
        select * from t_user
    </select>

</mapper>
```

编码规范

所对应的dao层接口中方法的名称

引用结果集

4）在MyBatis的配置文件中指定xml映射文件(例如：userdao.xml)的位置

SqlMapConfig.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE configuration
PUBLIC "-//ibatis.apache.org//DTD Config 3.0//EN"
"http://ibatis.apache.org/dtd/ibatis-3-config.dtd">
<configuration>
  <mappers>
    <mapper resource="com/tx/dao/UserDao.xml"/>
    <mapper resource="com/tx/dao/UserPowerDao.xml"/>
  </mappers>

</configuration>
```

编码规范

设置映射文件位置

5）完成service层，controller层逻辑

MyService.java

```java
@Service
public class MyService {

    @Autowired
    private UserDao userdao;

    @Autowired
    private UserPowerDao userpowerdao;

    public String test2(String name){


    public String test3(String name){

    public List<UserBean> test4(){
        return userdao.getUserList();
    }

    public List<PowerBean> test5(String name){
        return userpowerdao.getPowerListByName(name);
    }

}
```

查询全部用户的信息

通过用户名称查询用户的权限列表

MyController.java

```java
package com.tx.controller;

import java.util.List;

@Controller
public class MyController {

    @Autowired
    MyService service;

    public String testOne(){

    public String test2(HttpServletRequest request){

    public String test3(HttpServletRequest request){

    @RequestMapping("test4.do")
    public String test4(HttpServletRequest request){
        List<UserBean> list = service.test4();
        request.setAttribute("list", list);
        return "test4";
    }

    @RequestMapping("test5.do")
    public String test5(HttpServletRequest request){
        String name = request.getParameter("name");
        System.out.println("name:"+name);
        List<PowerBean> list = service.test5(name);
        request.setAttribute("powerlist", list);
        return "test5";
    }

}
```

设置第四次测试的请求地址

将查询得到的数据放到请求对象中

跳转到名称为test4的视图界面

从请求对象中获取参数name

调用service层的方法通过用户名获取用户的权限列表，并将权限列表放到请求对象中

执行完毕后，跳转到名称为test5的视图界面

5、统计用户数据的数量

1）编写dao层接口

UserDao.java

```java
package com.tx.dao;

import java.util.List;

@Repository
public interface UserDao {

    /**
     * 获取全部用户的信息
     * @return
     */
    public List<UserBean> getUserList();

    /**
     * 查询一个用户有几条数据
     */
    public int getCountByName(String name);

```

2）在xml映射文件中添加对应的方法实现

```xml
    <select id="getCountByName" parameterType="java.lang.String" resultType="int">
        select count(*) from t_user where t_user.t_name = #{name}
    </select>
```

3）完成service层和controller层逻辑

service层逻辑实现

```java
    public UserBean test7(User user){
        return userdao.getUserInfo(user);
    }
```

控制层逻辑实现

```java
    @RequestMapping("test6.do")
    public void test6(HttpServletRequest request,HttpServletResponse response){
        String name = request.getParameter("name");
        int i = service.test6(name);
        System.out.println("一共有"+i+"条数据");
         try {
            PrintWriter pw = response.getWriter();
            pw.println("一共有"+i+"条数据");
            pw.flush();
            pw.close();
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
```

7、根据账号密码查询用户数据，查询时对账号和密码做简单判断

1）完善dao层接口

```
    /**
     * 根据账号密码查询用户数据
     * @param user
     * @return
     */
    public UserBean getUserInfo(User user);
```

2）在对应的xml映射中实现

```
    <select id="getUserInfo" resultMap="user" parameterType="com.tx.model.User">
        select * from t_user
        <where>
            1 = 1
            <if test="username !='' and username != null ">
                and t_user.t_user_name = #{username}
            </if>
            <if test="password !='' and password != null ">
                and t_user.t_password = #{password}
            </if>
        </where>
    </select>
```

3）完善service层和控制层

```
    public UserBean test7(User user){
        return userdao.getUserInfo(user);
    }
```

service层实现

```
    @RequestMapping("test7.do")
    public String test7(User user,HttpServletRequest request){
        UserBean userbean = service.test7(user);

        request.setAttribute("userbean", userbean);

        return "test7";
    }
```

控制层实现

8、批量导入

1、）完善dao层接口，添加一个插入的方法

```
    public int insertData(List<UserBean> list);
```

2、）完善对应的xml映射文件

```
    <insert id="insertData" parameterType="java.util.List" >
        INSERT INTO T_USER(t_user_name,t_password,t_name,t_role_id,t_state) VALUES
            <foreach collection="list" item="user" index="index" separator=",">
                (#{user.t_user_name},#{user.t_password},#{user.t_name},1,'1')
            </foreach>
    </insert>
```

collection:引用的集合【list:集合的类型】 item：从集合中把数据依次取出并赋给item
separator:分隔符

3、）完善service层，controller层逻辑

```
60      public int test8(){
61          List<UserBean> list = new ArrayList<UserBean>();
62          for(int i = 0; i < 5;i++){
63              UserBean bean = new UserBean();
64              bean.setT_user_name("tx"+i);
65              bean.setT_password("tangxin"+i);
66              bean.setT_name("唐鑫"+i);
67              list.add(bean);
68          }
69          return userdao.insertData(list);
70      }
71
```

```
94      @RequestMapping("test8.do")
95      public void test8(HttpServletResponse response){
96          response.setCharacterEncoding("UTF-8");
97          int i = service.test8();
98          System.out.println("result::"+i);
99          try {
100             PrintWriter pw =  response.getWriter();
101             pw.println("插入成功的数量为："+i);
102             pw.flush();
103             pw.close();
104         } catch (IOException e) {
105             // TODO Auto-generated catch block
106             e.printStackTrace();
107         }
108     }
```

9、批量添加，事务回滚

添加大量数据，前几条数据已经添加成功，但是再次添加时发生了异常。现在需要将之前添加的数据全部回滚

1）、完善dao层接口

```
35
36      public void  insertUserBean(UserBean userbean);
```

2)、完善xml映射文件

```
45
46      <insert id="insertUserBean" parameterType="com.tx.model.UserBean">
47          INSERT INTO T_USER(t_user_name,t_password,t_name,t_role_id,t_state)
48              VALUES
49          (#{t_user_name},#{t_password},#{t_name},#{t_role_id},#{t_state})
50      </insert>
51
```

3）、完善service层和controller层

```java
    public void  test9(){
        List<UserBean> list = new ArrayList<UserBean>();
        for(int i = 0; i < 5;i++){
            UserBean bean = new UserBean();
            bean.setT_user_name("tx"+i);
            bean.setT_password("tangxin"+i);
            bean.setT_name("唐鑫"+i);
            bean.setT_role_id(String.valueOf(1));
            bean.setT_state("1");
            if(i==3){
                bean.setT_state("12313213132");
            }
            list.add(bean);
        }

        for (int i = 0; i < list.size(); i++) {
            userdao.insertUserBean(list.get(i));
        }

    }
```

当添加第四条数据时会发生异常

```java
    @RequestMapping("test9.do")
    public void test9(HttpServletResponse response){
        response.setContentType("text/html;charset=utf-8");
        service.test9();
        try {
            PrintWriter pw =  response.getWriter();
            pw.println("成功");
            pw.flush();
            pw.close();
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
```

4）、为了捕获异常设置回滚，完善aop.xml文件

```xml
在之前的aop配置文件中添加
    <!-- proxy-target-class="true" 允许使用CGlib动态代理 -->
    <aop:config proxy-target-class="true">
        <!-- 设置切入点，并给切入点起别名 -->
        <aop:pointcut id="pointcut" expression="execution(* com.tx.service.*.*(..))" />
        <aop:advisor advice-ref="rollback" pointcut-ref="pointcut"
            order="10" />
                                                给切入点添加异常回滚机制
        <!-- 设置切入点所对应的目标对象（通知对象） -->
        <aop:aspect id="aspect" ref="aspectbean" order="1">
            <!-- 给目标对象指定处理方法doRound -->
            <aop:around pointcut-ref="pointcut" method="doRound" />
        </aop:aspect>
    </aop:config>
                        引用

    <tx:advice id="rollback" transaction-manager="TransactionManager">
        <tx:attributes>
            <tx:method name="test9" propagation="REQUIRED"
                rollback-for="java.lang.Exception" />
                                                设置事务发生异常后，进行回滚处理
        </tx:attributes>
    </tx:advice>
```